

Head First Android Development

A Brain-Friendly Guide

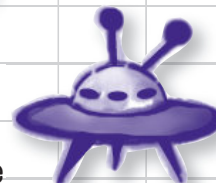


Put fragments
under the
microscope



Avoid
embarrassing
activities

Learn how
Constraint
Layouts can
change your life



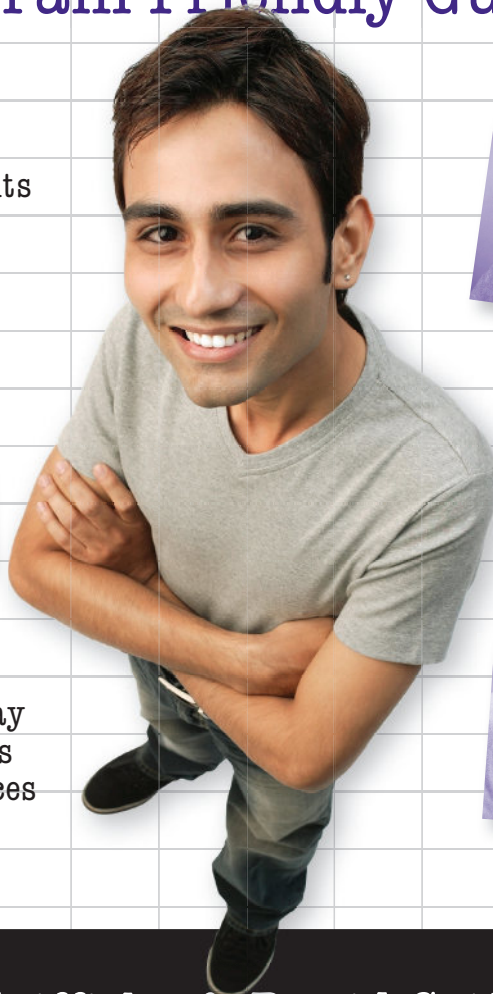
Create
out-of-this-world
services



Find your way
with Android's
Location Services



Fool around
in the Design
Support Library

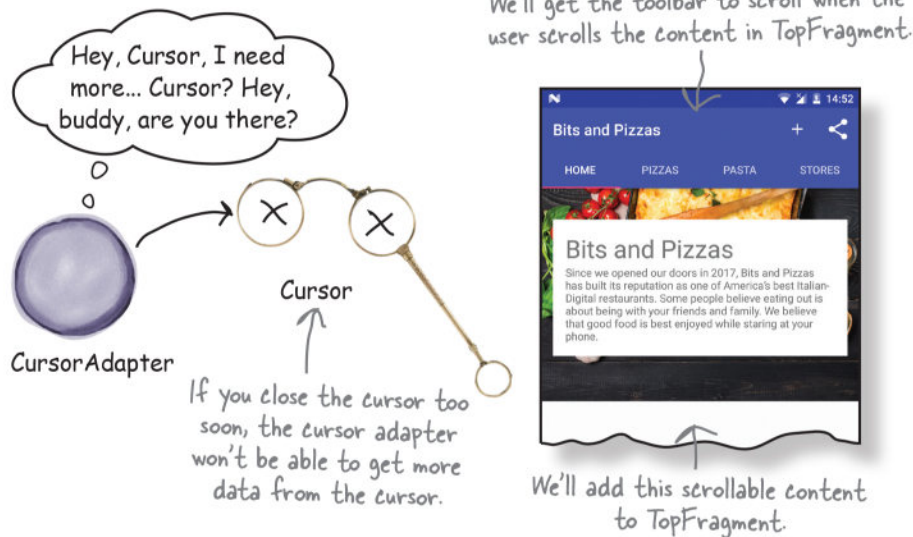


Dawn Griffiths & David Griffiths

Android Development

What will you learn from this book?

If you have an idea for a killer Android app, this fully revised and updated edition will help you build your first working application in a jiffy. You'll learn hands-on how to structure your app, design flexible and interactive interfaces, run services in the background, make your app work on various smartphones and tablets, and much more. It's like having an experienced Android developer sitting right next to you! All you need to get started is some Java know-how.



Why does this book look so different?

Based on the latest research in cognitive science and learning theory, *Head First Android Development* uses a visually rich format to engage your mind, rather than a text-heavy approach that puts you to sleep. Why waste your time struggling with new concepts? This multi-sensory learning experience is designed for the way your brain really works.

“If you’re starting out in mobile development, this is the book for you. It’s quite simply the best book on Android development out there.”

—Andy Parker
Lead Software Developer
at Next plc

“This is, without a doubt, the best available book for learning Android development. If you can get only one, make it this one.”

—Kenneth Kousen
President, Kousen IT Inc.,
and JavaOne Rock Star

“Become an able Android developer applying up-to-date patterns and create that next killer app. *Head First Android Development* will be your friendly, accurate, and fun-to-be-with master craftsman on that path.”

—Ingo Krotzky
Android Learner

US \$69.99

CAN \$92.99

ISBN: 978-1-491-97405-6



9



Twitter: @oreillymedia
facebook.com/oreilly

Head First Android Development

Wouldn't it be dreamy if there were a book on developing Android apps that was easier to understand than the space shuttle flight manual? I guess it's just a fantasy...



Dawn Griffiths
David Griffiths

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

Head First Android Development

by Dawn Griffiths and David Griffiths

Copyright © 2017 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/safari>). For more information, contact our corporate/institutional sales department: (800) 998-9938 or corporate@oreilly.com.

Series Creators:	Kathy Sierra, Bert Bates
Editor:	Dawn Schanafelt
Cover Designer:	Karen Montgomery
Production Editor:	Kristen Brown
Proofreader:	Rachel Monaghan
Indexer:	Angela Howard
Page Viewers:	Mum and Dad, Rob and Lorraine

Printing History:

June 2015: First Edition.

August 2017: Second Edition

Mum and Dad →



← Rob and Lorraine

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: 978-1-491-97405-6

[M]

To our friends and family. Thank you so
much for all your love and support.

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.

Before writing *Head First Android Development*, Dawn wrote three other Head First books (*Head First Statistics*, *Head First 2D Geometry*, and *Head First C*). She also created the video course *The Agile Sketchpad* with her husband, David, to teach key concepts and techniques in a way that keeps your brain active and engaged.

When Dawn's not working on Head First books or creating videos, 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. After studying pure mathematics at university, he began writing code for computers and magazine articles for humans. He's worked as an Agile coach, a developer, and a garage attendant, but not in that order. He can write code in over 10 languages and prose in just one, and when not writing, coding, or coaching, he spends much of his spare time traveling with his lovely wife—and coauthor—Dawn.

Before writing *Head First Android Development*, David wrote three other Head First books—*Head First Rails*, *Head First Programming*, and *Head First C*—and created *The Agile Sketchpad* video course with Dawn.

You can follow us on Twitter at <https://twitter.com/HeadFirstDroid> and visit the book's website at <https://tinyurl.com/HeadFirstAndroid>.

Table of Contents (Summary)

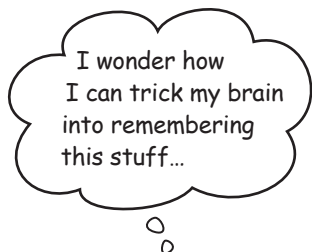
	Intro	xxix
1	Getting Started: <i>Diving in</i>	1
2	Building Interactive Apps: <i>Apps that do something</i>	37
3	Multiple Activities and Intents: <i>State your intent</i>	77
4	The Activity Lifecycle: <i>Being an activity</i>	119
5	Views and View Groups: <i>Enjoy the view</i>	169
6	Constraint Layouts: <i>Put things in their place</i>	221
7	List views and Adapters: <i>Getting organized</i>	247
8	Support Libraries and App Bars: <i>Taking shortcuts</i>	289
9	Fragments: <i>Make it modular</i>	339
10	Fragments for Larger Interfaces: <i>Different size, different interface</i>	393
11	Dynamic Fragments: <i>Nesting fragments</i>	433
12	Design Support Library: <i>Swipe right</i>	481
13	Recycler Views and Card Views: <i>Get recycling</i>	537
14	Navigation Drawers: <i>Going places</i>	579
15	SQLite Databases: <i>Fire up the database</i>	621
16	Basic cursors: <i>Getting data out</i>	657
17	Cursors and AsyncTasks: <i>Staying in the background</i>	693
18	Started Services: <i>At your service</i>	739
19	Bound Services and Permissions: <i>Bound together</i>	767
i	Relative and Grid Layouts: <i>Meet the relatives</i>	817
ii	Gradle: <i>The Gradle build tool</i>	833
iii	ART: <i>The Android Runtime</i>	841
iv	ADB: <i>The Android debug bridge</i>	849
v	The Android Emulator: <i>Speeding things up</i>	857
vi	Leftovers: <i>The top ten things (we didn't cover)</i>	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?

Authors of Head First Android Development	iv
Who is this book for?	xxx
We know what you're thinking	xxxix
We know what your <i>brain</i> is thinking	xxxix
Metacognition: thinking about thinking	xxxix
Here's what WE did	xxxix
Read me	xxxix
The technical review team	xxxix
Acknowledgments	xxxix
Safari® Books Online	xl



getting started

1

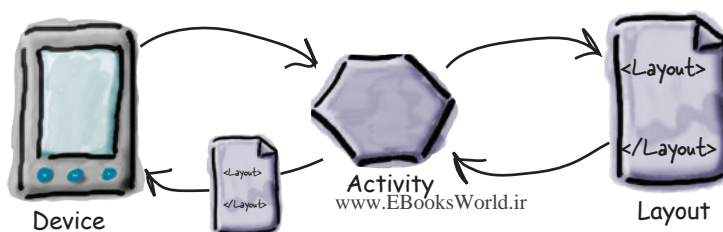
Diving In

Android has taken the world by storm.

Everybody wants a smartphone or tablet, and Android devices are hugely popular. In this book, we'll teach you how to **develop your own apps**, and we'll start by getting you to build a basic app and run it on an Android Virtual Device. Along the way, you'll meet some of the basic components of all Android apps, such as **activities** and **layouts**. **All you need is a little Java know-how...**



Welcome to Androidville	2
The Android platform dissected	3
Here's what we're going to do	4
Your development environment	5
Install Android Studio	6
Build a basic app	7
How to build the app	8
Activities and layouts from 50,000 feet	12
How to build the app (continued)	13
You've just created your first Android app	15
Android Studio creates a complete folder structure for you	16
Useful files in your project	17
Edit code with the Android Studio editors	18
Run the app in the Android emulator	23
Creating an Android Virtual Device	24
Run the app in the emulator	27
You can watch progress in the console	28
What just happened?	30
Refining the app	31
What's in the layout?	32
activity_main.xml has two elements	33
Update the text displayed in the layout	34
Take the app for a test drive	35
Your Android Toolbox	36



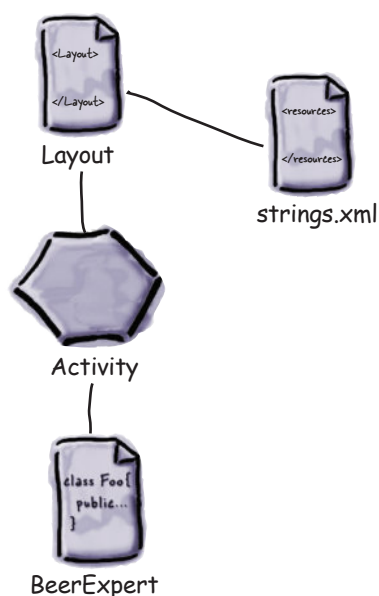
building interactive apps

Apps That Do Something

2

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

In this chapter, you'll see how you can make your apps **a bit more interactive**. You'll learn how to get your app to **do** something in response to the user, and **how to get your activity and layout talking to each other** like best buddies. Along the way, we'll take you a bit **deeper into how Android actually works** by introducing you to **R**, the hidden gem that glues everything together.



Let's build a Beer Adviser app	38
Create the project	40
We've created a default activity and layout	41
A closer look at the design editor	42
Add a button using the design editor	43
activity_find_beer.xml has a new button	44
A closer look at the layout code	45
Let's take the app for a test drive	49
Hardcoding text makes localization hard	50
Create the String resource	51
Use the String resource in your layout	52
The code for activity_find_beer.xml	53
Add values to the spinner	56
Add the string-array to strings.xml	57
Test drive the spinner	58
We need to make the button do something	59
Make the button call a method	60
What activity code looks like	61
Add an <code>onClickFindBeer()</code> method to the activity	62
<code>onClickFindBeer()</code> needs to do something	63
Once you have a View, you can access its methods	64
Update the activity code	65
The first version of the activity	67
What the code does	68
Build the custom Java class	70
What happens when you run the code	74
Test drive your app	75
Your Android Toolbox	76

multiple activities and intents

State Your Intent

3

Most apps need more than one activity.

So far we've just looked at single-activity apps, which is fine for simple apps. But when things get more complicated, just having the one activity won't cut it. We're going to show you **how to build apps with multiple activities**, and how you can get your apps talking to each other using **intents**. We'll also look at how you can use intents to **go beyond the boundaries of your app and make activities in other apps on your device perform actions**. Things are about to get a whole lot more powerful...

Apps can contain more than one activity	78
Here's the app structure	79
Get started: create the project	79
Update the layout	80
Create the second activity and layout	82
Welcome to the Android manifest file	84
An intent is a type of message	86
What happens when you run the app	88
Pass text to a second activity	90
Update the text view properties	91
putExtra() puts extra info in an intent	92
Update the CreateMessageActivity code	95
Get ReceiveMessageActivity to use the information in the intent	96
What happens when the user clicks the Send Message button	97
We can change the app to send messages to other people	98
How Android apps work	99
Create an intent that specifies an action	101
Change the intent to use an action	102
How Android uses the intent filter	106
What if you ALWAYS want your users to choose an activity?	112
What happens when you call createChooser()	113
Change the code to create a chooser	115
Your Android Toolbox	118

Intent



To: AnotherActivity



CreateMessageActivity

Hey, user. Which activity do you want to use this time?



Android



User www.EBooksWorld.ir

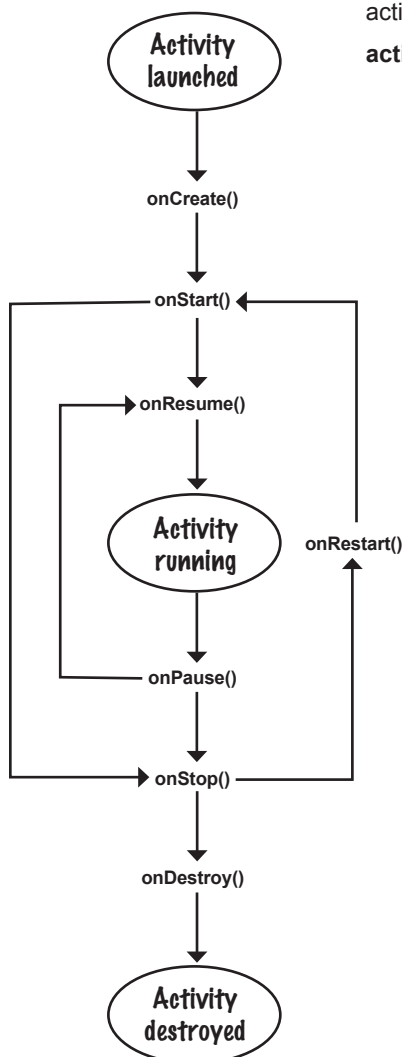
the activity lifecycle

Being an Activity

4

Activities form the foundation of every Android app.

So far you've seen how to create activities, and made one activity start another using an intent. But *what's really going on beneath the hood?* In this chapter, we're going to dig a little deeper into **the activity lifecycle**. What happens when an activity is **created** and **destroyed**? Which methods get called when an activity is **made visible and appears in the foreground**, and which get called when the activity **loses the focus and is hidden**? And **how do you save and restore your activity's state**? Read on to find out.



How do activities really work?	120
The Stopwatch app	122
Add String resources	123
How the activity code will work	125
Add code for the buttons	126
The runTimer() method	127
The full runTimer() code	129
The full StopwatchActivity code	130
Rotating the screen changes the device configuration	136
The states of an activity	137
The activity lifecycle: from create to destroy	138
The updated StopwatchActivity code	142
What happens when you run the app	143
There's more to an activity's life than create and destroy	146
The updated StopwatchActivity code	151
What happens when you run the app	152
What if an app is only partially visible?	154
The activity lifecycle: the foreground lifetime	155
Stop the stopwatch if the activity's paused	158
Implement the onPause() and onResume() methods	159
The complete StopwatchActivity code	160
What happens when you run the app	163
Your handy guide to the lifecycle methods	167
Your Android Toolbox	168

views and view groups

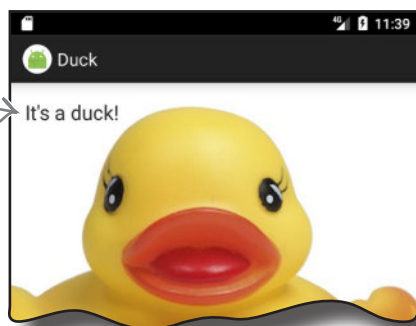
5

Enjoy the View

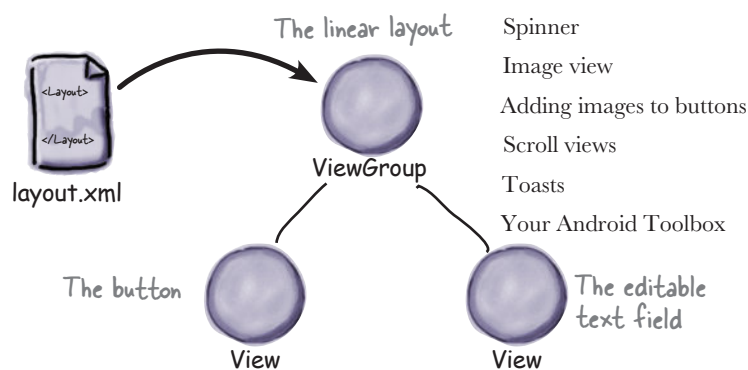
You've seen how to arrange GUI components using a linear layout, but so far we've only scratched the surface.

In this chapter we'll **look a little deeper** and show you how linear layouts *really work*. We'll introduce you to the **frame layout**, a simple layout used to stack views, and we'll also take a tour of the **main GUI components** and **how you use them**. By the end of the chapter, you'll see that even though they all look a little different, all layouts and GUI components have *more in common than you might think*.

Frame layouts let your views overlap one another. This is useful for displaying text on top of images, for example.



Your user interface is made up of layouts and GUI components	170
LinearLayout displays views in a single row or column	171
Add a dimension resource file for consistent padding across layouts	174
Use margins to add distance between views	176
Let's change up a basic linear layout	177
Make a view stretch by adding weight	179
Values you can use with the android:gravity attribute	183
The full linear layout code	186
Frame layouts stack their views	188
Add an image to your project	189
The full code to nest a layout	192
FrameLayout: a summary	193
Playing with views	201
Editable text view	202
Toggle button	204
Switch	205
Checkboxes	206
Radio buttons	208
Spinner	210
Image view	211
Adding images to buttons	213
Scroll views	215
Toasts	216
Your Android Toolbox	220



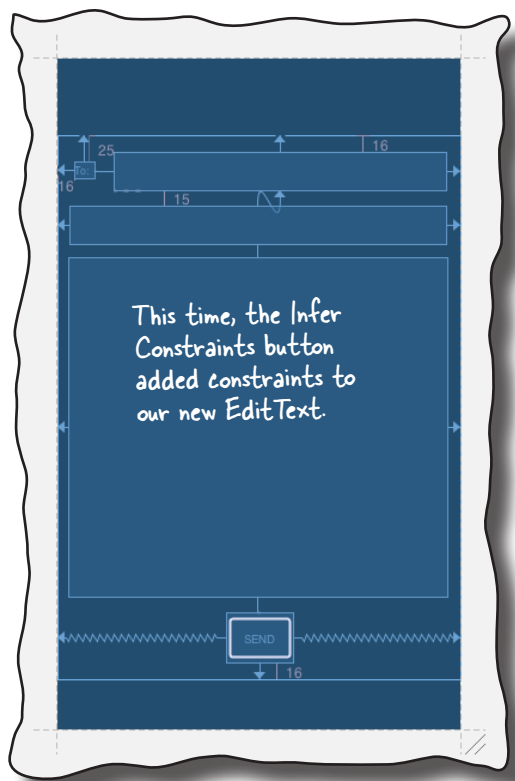
constraint layouts

Put Things in Their Place

6

Let's face it, you need to know how to create great layouts.

If you're building apps you want people to *use*, you need to make sure they *look exactly the way you want*. So far you've seen how to use linear and frame layouts, but *what if your design is more complex?* To deal with this, we'll introduce you to Android's new **constraint layout**, a type of layout you **build visually using a blueprint**. We'll show you how **constraints** let you position and size your views, *irrespective of screen size and orientation*. Finally, you'll find out how to save time by making Android Studio **infer and add constraints** on your behalf.



Nested layouts can be inefficient	222
Introducing the Constraint Layout	223
Make sure your project includes the Constraint Layout Library	224
Add the String resources to strings.xml	225
Use the blueprint tool	226
Position views using constraints	227
Add a vertical constraint	228
Changes to the blueprint are reflected in the XML	229
How to center views	230
Adjust a view's position by updating its bias	231
How to change a view's size	232
How to align views	238
Let's build a real layout	239
First, add the top line of views	240
The Infer Constraints feature guesses which constraints to add	241
Add the next line to the blueprint...	242
Finally, add a view for the message	243
Test drive the app	244
Your Android Toolbox	245

list views and adapters

7

Getting Organized**Want to know how best to structure your Android app?**

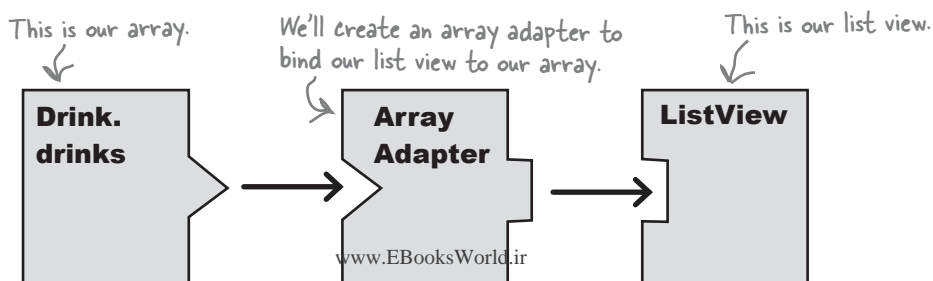
You've learned about some of the basic building blocks that are used to create apps, and now **it's time to get organized**. In this chapter, we'll show you how you can take a bunch of ideas and **structure them into an awesome app**. You'll learn how **lists of data** can form the core part of your app design, and how **linking them together** can create a **powerful and easy-to-use app**. Along the way, you'll get your first glimpse of using **event listeners** and **adapters** to make your app more dynamic.

Display a start screen with a list of options.

Display a list of the drinks we sell.

Show details of each drink.

Every app starts with ideas	248
Use list views to navigate to data	251
The drink detail activity	253
The Starbuzz app structure	254
The Drink class	256
The top-level layout contains an image and a list	258
The full top-level layout code	260
Get list views to respond to clicks with a listener	261
Set the listener to the list view	262
A category activity displays the data for a single category	267
Update activity_drink_category.xml	268
For nonstatic data, use an adapter	269
Connect list views to arrays with an array adapter	270
Add the array adapter to DrinkCategoryActivity	271
App review: where we are	274
How we handled clicks in TopLevelActivity	276
The full DrinkCategoryActivity code	278
Update the views with the data	281
The DrinkActivity code	283
What happens when you run the app	284
Your Android Toolbox	288



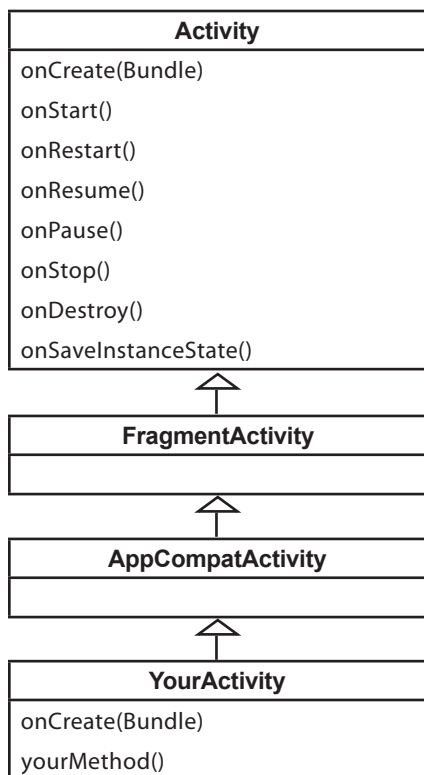
support libraries and app bars

8

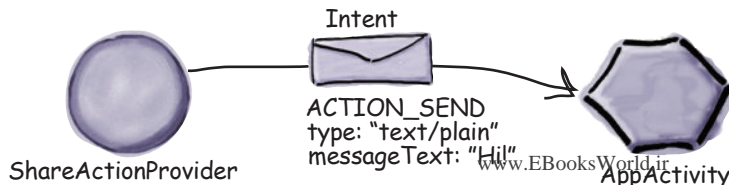
Taking Shortcuts

Everybody likes a shortcut.

And in this chapter you'll see how to add shortcuts to your apps using **app bars**. We'll show you how to start activities by *adding actions* to your app bar, how to share content with other apps using the *share action provider*, and how to navigate up your app's hierarchy by implementing *the app bar's Up button*. Along the way we'll introduce you to the powerful **Android Support Libraries**, which are key to making your apps look fresh on older versions of Android.



Great apps have a clear structure	290
Different types of navigation	291
Add an app bar by applying a theme	293
Create the Pizza app	295
Add the v7 AppCompat Support Library	296
AndroidManifest.xml can change your app bar's appearance	299
How to apply a theme	300
Define styles in a style resource file	301
Customize the look of your app	303
Define colors in a color resource file	304
The code for activity_main.xml	305
ActionBar vs. Toolbar	306
Include the toolbar in the activity's layout	312
Add actions to the app bar	315
Change the app bar text by adding a label	318
The code for AndroidManifest.xml	319
Control the action's appearance	322
The full MainActivity.java code	325
Enable Up navigation	327
Share content on the app bar	331
Add a share action provider to menu_main.xml	332
Specify the content with an intent	333
The full MainActivity.java code	334
Your Android Toolbox	337



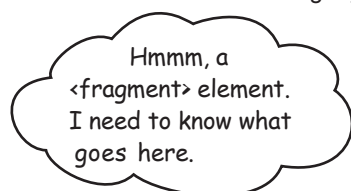
fragments

Make It Modular

9

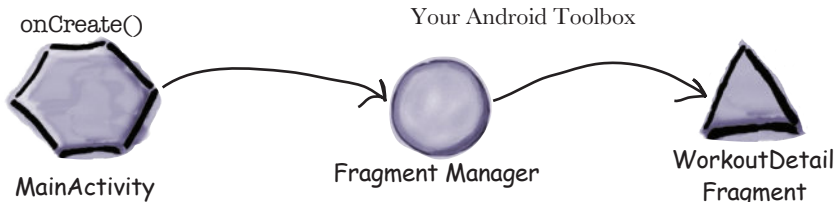
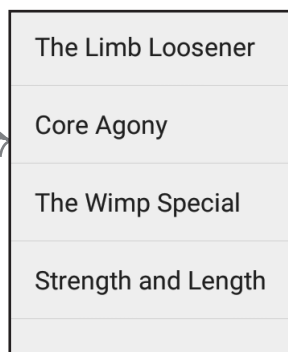
You've seen how to create apps that work in the same way no matter what device they're running on.

But what if you want your app to look and behave differently depending on whether it's running on a *phone* or a *tablet*? In this case you need **fragments**, modular code components that can be **reused by different activities**. We'll show you how to create **basic fragments** and **list fragments**, how to **add them to your activities**, and how to get your fragments and activities to **communicate** with one another.



activity_detail

A list fragment comes complete with its own list view so you don't need to add it yourself. You just need to provide the list fragment with data.



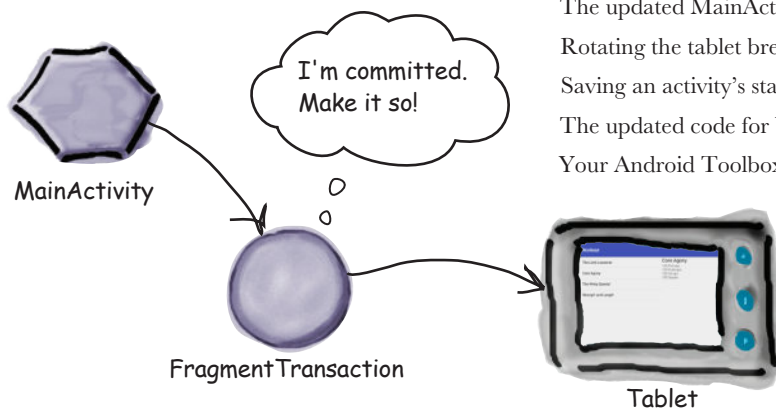
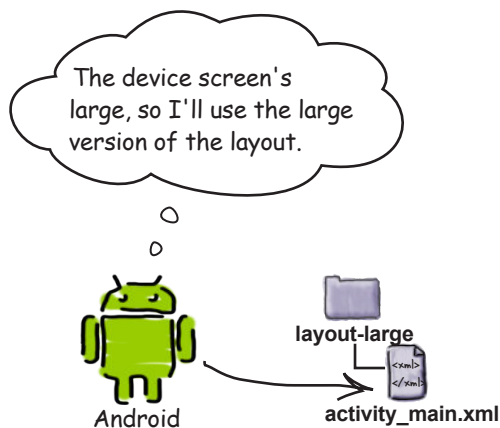
Your app needs to look great on ALL devices	340
Your app may need to behave differently too	341
Fragments allow you to reuse code	342
The phone version of the app	343
Create the project and activities	345
Add a button to MainActivity's layout	346
How to add a fragment to your project	348
The fragment's onCreateView() method	350
Add a fragment to an activity's layout	352
Get the fragment and activity to interact	359
The Workout class	360
Pass the workout ID to the fragment	361
Get the activity to set the workout ID	363
The fragment lifecycle	365
Set the view's values in the fragment's onStart() method	367
How to create a list fragment	374
The updated WorkoutListFragment code	377
The code for activity_main.xml	381
Connect the list to the detail	384
The code for WorkoutListFragment.java	387
MainActivity needs to implement the interface	388
DetailActivity needs to pass the ID to WorkoutDetailFragment	389
Your Android Toolbox	392

10

fragments for larger interfaces

Different Size, Different Interface

So far we've only run our apps on devices with a small screen. But what if your users have tablets? In this chapter you'll see how to create **flexible user interfaces** by making your app **look and behave differently** depending on the device it's running on. We'll show you how to control the behavior of your app when you press the Back button by introducing you to the **back stack** and **fragment transactions**. Finally, you'll find out how to **save and restore the state** of your fragment.



The Workout app looks the same on a phone and a tablet	394
Designing for larger interfaces	395
The phone version of the app	396
The tablet version of the app	397
Create a tablet AVD	399
Put screen-specific resources in screen-specific folders	402
The different folder options	403
Tablets use layouts in the layout-large folder	408
What the updated code does	410
We need to change the itemClicked() code	412
You want fragments to work with the Back button	413
Welcome to the back stack	414
Back stack transactions don't have to be activities	415
Use a frame layout to replace fragments programmatically	416
Use layout differences to tell which layout the device is using	417
The revised MainActivity code	418
Using fragment transactions	419
The updated MainActivity code	423
Rotating the tablet breaks the app	427
Saving an activity's state (revisited)	428
The updated code for WorkoutDetailFragment.java	430
Your Android Toolbox	432

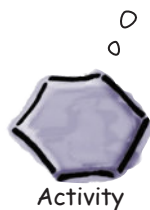
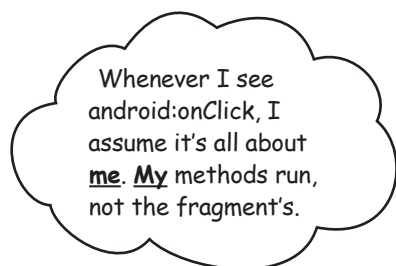
11

dynamic fragments

Nesting Fragments

So far you've seen how to create and use **static fragments**. But what if you want your fragments to be more **dynamic**?

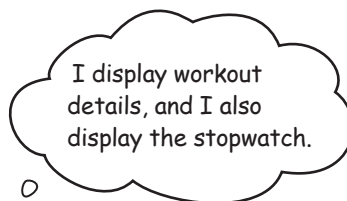
Dynamic fragments have a lot in common with dynamic activities, but there are crucial differences you need to be able to deal with. In this chapter you'll see how to **convert dynamic activities** into **working dynamic fragments**. You'll find out how to use **fragment transactions** to help **maintain your fragment state**. Finally, you'll discover how to **nest one fragment inside another**, and how the **child fragment manager** helps you control unruly back stack behavior.



Activity

Adding dynamic fragments	434
The new version of the app	436
Create TempActivity	437
TempActivity needs to extend AppCompatActivity	438
The StopwatchFragment.java code	444
The StopwatchFragment layout	447
Add StopwatchFragment to TempActivity's layout	449
The onClick attribute calls methods in the activity, not the fragment	452
Attach the OnClickListener to the buttons	457
The StopwatchFragment code	458
Rotating the device resets the stopwatch	462
Use <fragment> for static fragments...	463
Change activity_temp.xml to use a FrameLayout	464
The full code for TempActivity.java	467
Add the stopwatch to WorkoutDetailFragment	469
The full WorkoutDetailFragment.java code	476
Your Android Toolbox	480

The transaction to add StopwatchFragment is nested inside the transaction to add WorkoutDetailFragment.



design support library

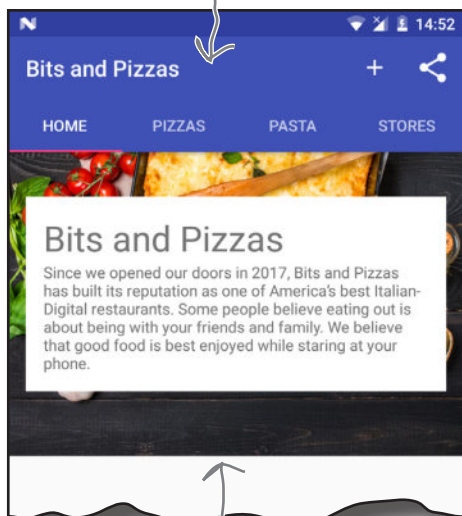
Swipe Right

12

Ever wondered how to develop apps with a rich, slick UI?

With the release of the **Android Design Support Library**, it became much easier to create apps with an intuitive UI. In this chapter, we'll show you around some of the highlights. You'll see how to add **tabs** so that your users can *navigate around your app more easily*. You'll discover how to **animate your toolbars** so that they can *collapse or scroll on a whim*. You'll find out how to add **floating action buttons** for common user actions. Finally, we'll introduce you to **snackbars**, a way of displaying short, informative messages to the user that they can interact with.

We'll get the toolbar to scroll when the user scrolls the content in TopFragment.



We'll add this scrollable content to TopFragment.

The Bits and Pizzas app revisited	482
The app structure	483
Use a view pager to swipe through fragments	489
Add a view pager to MainActivity's layout	490
Tell a view pager about its pages using a fragment pager adapter	491
The code for our fragment pager adapter	492
The full code for MainActivity.java	494
Add tab navigation to MainActivity	498
How to add tabs to your layout	499
Link the tab layout to the view pager	501
The full code for MainActivity.java	502
The Design Support Library helps you implement material design	506
Make the toolbar respond to scrolls	508
Add a coordinator layout to MainActivity's layout	509
How to coordinate scroll behavior	510
Add scrollable content to TopFragment	512
The full code for fragment_top.xml	515
Add a collapsing toolbar to OrderActivity	517
How to create a plain collapsing toolbar	518
How to add an image to a collapsing toolbar	523
The updated code for activity_order.xml	524
FABs and snackbars	526
The updated code for activity_order.xml	528
The full code for OrderActivity.java	533
Your Android Toolbox	535

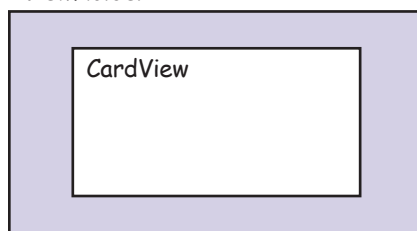
recycler views and card views

13

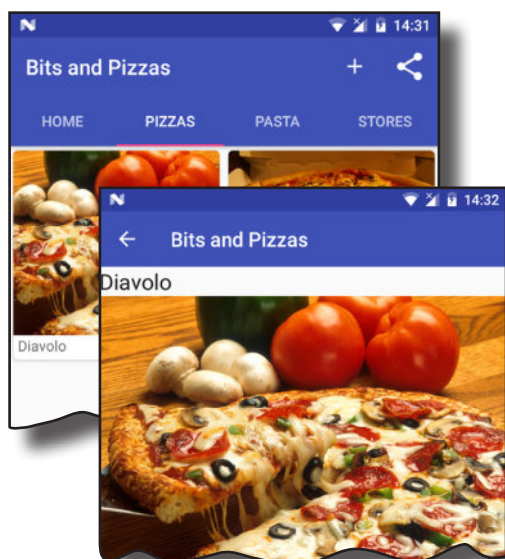
Get Recycling

You've already seen how the humble list view is a key part of most apps. But compared to some of the *material design* components we've seen, it's somewhat plain. In this chapter, we'll introduce you to the **recycler view**, a more advanced type of list that gives you *loads more flexibility* and *fits in with the material design ethos*. You'll see how to create **adapters** tailored to your data, and how to completely change the look of your list with *just two lines of code*. We'll also show you how to use **card views** to give your data a *3D material design* appearance.

ViewHolder



Each of our ViewHolders will contain a CardView. We created the layout for this CardView earlier in the chapter.



There's still work to do on the Bits and Pizzas app	538
Recycler views from 10,000 feet	539
Add the pizza data	541
Display the pizza data in a card	542
How to create a card view	543
The full card_captioned_image.xml code	544
Add a recycler view adapter	546
Define the adapter's view holder	548
Override the onCreateViewHolder() method	549
Add the data to the card views	550
The full code for CaptionedImagesAdapter.java	551
Create the recycler view	553
Add the RecyclerView to PizzaFragment's layout	554
The full PizzaFragment.java code	555
A recycler view uses a layout manager to arrange its views	556
Specify the layout manager	557
The full PizzaFragment.java code	558
Make the recycler view respond to clicks	566
Create PizzaDetailActivity	567
The code for PizzaDetailActivity.java	569
Get a recycler view to respond to clicks	570
You can listen for view events from the adapter	571
Keep your adapters reusable	572
Add the interface to the adapter	573
Implement the listener in PizzaFragment.java	575
Your Android Toolbox	578

14

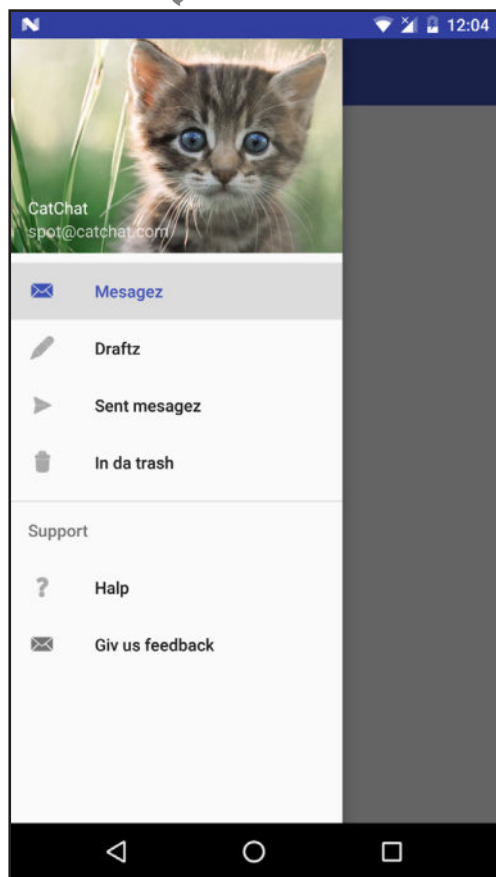
navigation drawers

Going Places

You've already seen how tabs help users navigate your apps.

But if you need a *large number* of them, or want to *split them into sections*, the **navigation drawer** is your new BFF. In this chapter, we'll show you how to create a navigation drawer that *slides out from the side of your activity at a single touch*. You'll learn how to give it a header using a **navigation view**, and provide it with a **structured set of menu items** to take the user to all the major hubs of your app. Finally, you'll discover how to set up a **navigation view listener** so that the drawer *responds to the slightest touch and swipe*.

This is the CatChat app.



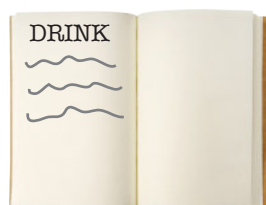
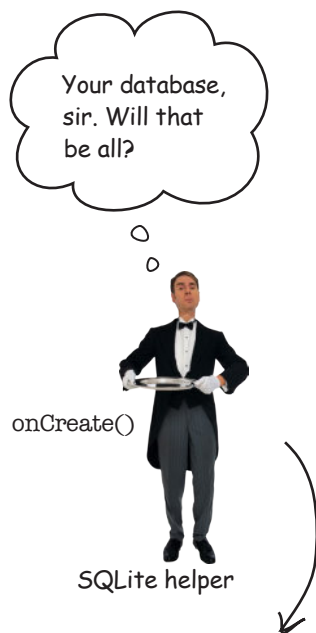
Tab layouts allow easy navigation...	580
We're going to create a navigation drawer for a new email app	581
Navigation drawers deconstructed	582
Create the CatChat project	584
Create InboxFragment	585
Create DraftsFragment	586
Create SentItemsFragment	587
Create TrashFragment	588
Create a toolbar layout	589
Update the app's theme	590
Create HelpActivity	591
Create FeedbackActivity	592
Create the navigation drawer's header	594
The full nav_header.xml code	595
How to group items together	598
Add the support section as a submenu	600
The full menu_nav.xml code	601
How to create a navigation drawer	602
The full code for activity_main.xml	603
Add InboxFragment to MainActivity's frame layout	604
Add a drawer toggle	607
Respond to the user clicking items in the drawer	608
Implement the onNavigationItemSelectedListener() method	609
Close the drawer when the user presses the Back button	614
The full MainActivity.java code	615
Your Android Toolbox	619

15

SQLite databases

Fire Up the Database

If you're recording high scores or saving tweets, your app will need to store data. And on Android you usually keep your data safe inside a **SQLite database**. In this chapter, we'll show you how to **create a database, add tables to it, and prepopulate it with data**, all with the help of the friendly **SQLite helper**. You'll then see how you can cleanly roll out **upgrades** to your database structure, and how to **downgrade** it if you need to undo any changes.



SQLite database

Name: "starbuzz"
Version: 1

Back to Starbuzz	622
Android uses SQLite databases to persist data	623
Android comes with SQLite classes	624
The current Starbuzz app structure	625
Let's change the app to use a database	626
The SQLite helper manages your database	627
Create the SQLite helper	628
Inside a SQLite database	630
You create tables using Structured Query Language (SQL)	631
Insert data using the insert() method	632
Insert multiple records	633
The StarbuzzDatabaseHelper code	634
What the SQLite helper code does	635
What if you need to make changes to the database?	636
SQLite databases have a version number	637
What happens when you change the version number	638
Upgrade your database with onUpgrade()	640
Downgrade your database with onDowngrade()	641
Let's upgrade the database	642
Upgrade an existing database	645
Update records with the update() method	646
Apply conditions to multiple columns	647
Change the database structure	649
Delete tables by dropping them	650
The full SQLite helper code	651
Your Android Toolbox	656

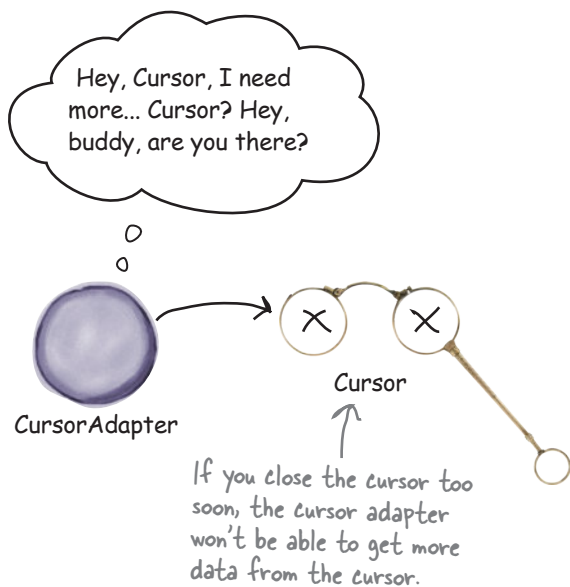
basic cursors

16

Getting Data Out

So how do you connect your app to a SQLite database?

So far you've seen how to create a SQLite database using a SQLite helper. The next step is to get your activities to access it. In this chapter, we'll focus on how you read data from a database. You'll find out **how to use cursors to get data from the database**. You'll see **how to navigate cursors**, and **how to get access to their data**. Finally, you'll discover how to use **cursor adapters** to bind cursors to list views.



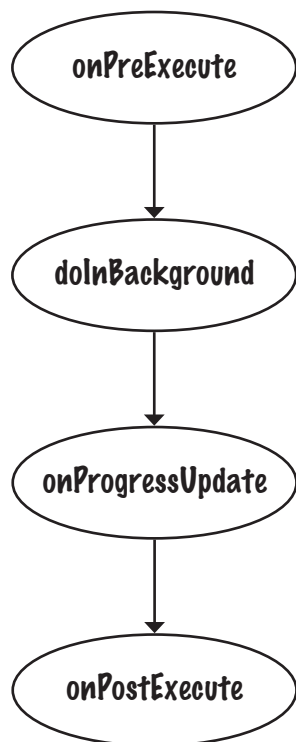
The story so far...	658
The new Starbuzz app structure	659
What we'll do to change DrinkActivity to use the Starbuzz database	660
The current DrinkActivity code	661
Get a reference to the database	662
Get data from the database with a cursor	663
Return all the records from a table	664
Return records in a particular order	665
Return selected records	666
The DrinkActivity code so far	669
To read a record from a cursor, you first need to navigate to it	670
Navigate cursors	671
Get cursor values	672
The DrinkActivity code	673
What we've done so far	675
The current DrinkCategoryActivity code	677
Get a reference to the Starbuzz database...	678
How do we replace the array data in the list view?	679
A simple cursor adapter maps cursor data to views	680
How to use a simple cursor adapter	681
Close the cursor and database	682
The story continues	683
The revised code for DrinkCategoryActivity	688
The DrinkCategoryActivity code (continued)	689
Your Android Toolbox	691

cursors and asyncTasks

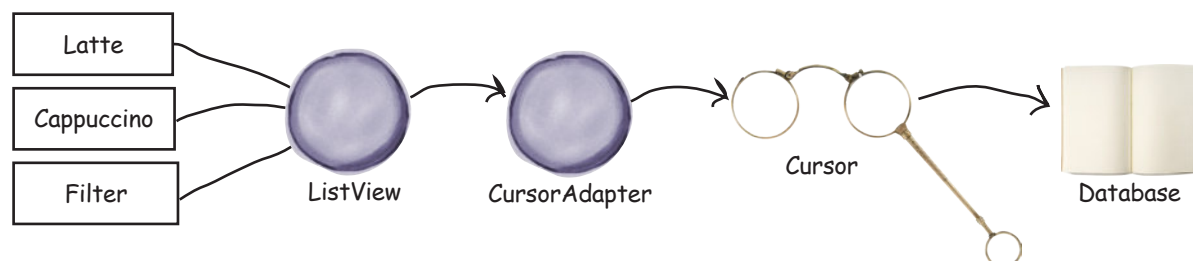
Staying in the Background**17**

In most apps, you'll need your app to update its data.

So far you've seen how to create apps that read data from a SQLite database. But what if you want to update the app's data? In this chapter you'll see how to get your app to **respond to user input** and **update values in the database**. You'll also find out how to **refresh the data that's displayed** once it's been updated. Finally, you'll see how writing efficient **multithreaded code** with **AsyncTasks** will keep your app speedy.



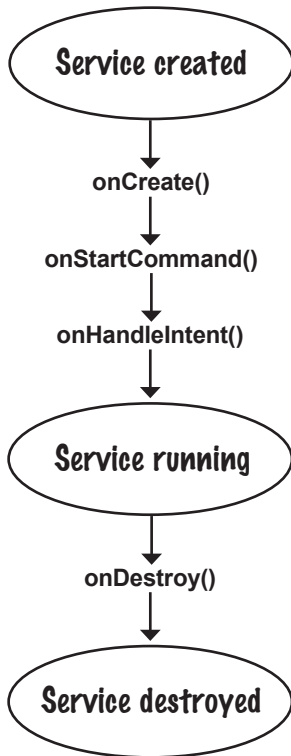
We want our Starbuzz app to update database data	694
Add a checkbox to DrinkActivity's layout	696
Display the value of the FAVORITE column	697
Respond to clicks to update the database	698
The full DrinkActivity.java code	701
Display favorites in TopLevelActivity	705
Refactor TopLevelActivity.java	707
The new TopLevelActivity.java code	710
Change the cursor with changeCursor()	715
What code goes on which thread?	723
AsyncTask performs asynchronous tasks	724
The onPreExecute() method	725
The doInBackground() method	726
The onProgressUpdate() method	727
The onPostExecute() method	728
The AsyncTask class parameters	729
The full UpdateDrinkTask class	730
The full DrinkActivity.java code	732
A summary of the AsyncTask steps	737
Your Android Toolbox	737



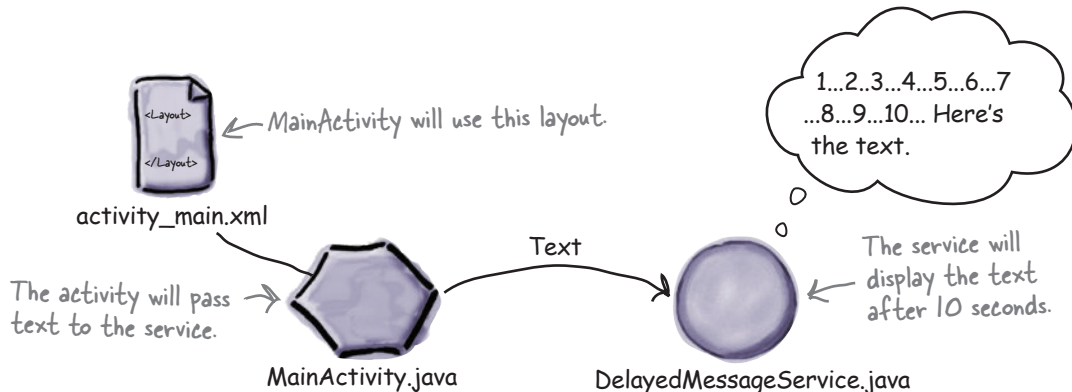
18

started services At Your Service

There are some operations you want to keep on running, irrespective of which app has the focus. If you start downloading a file, for instance, *you don't want the download to stop when you switch to another app*. In this chapter we'll introduce you to **started services**, components that *run operations in the background*. You'll see how to create a started service using the `IntentService` class, and find out how its lifecycle fits in with that of an activity. Along the way, you'll discover how to **log messages**, and *keep users informed* using Android's built-in **notification service**.



Services work in the background	740
We'll create a STARTED service	741
Use the <code>IntentService</code> class to create a basic started service	742
How to log messages	743
The full <code>DelayedMessageService</code> code	744
You declare services in <code>AndroidManifest.xml</code>	745
Add a button to <code>activity_main.xml</code>	746
You start a service using <code>startService()</code>	747
The states of a started service	750
The started service lifecycle: from create to destroy	751
Your service inherits the lifecycle methods	752
Android has a built-in notification service	755
We'll use notifications from the AppCompat Support Library	756
First create a notification builder	757
Issue the notification using the built-in notification service	759
The full code for <code>DelayedMessageService.java</code>	760
Your Android Toolbox	765



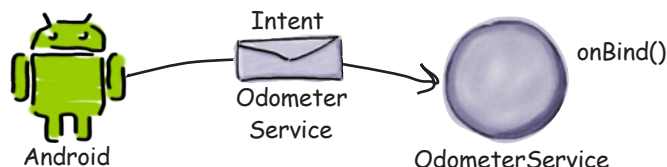
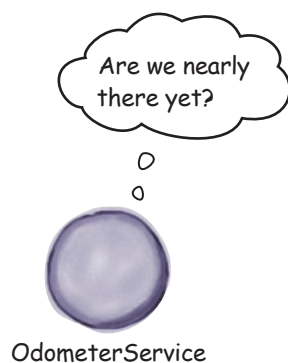
19

bound services and permissions

Bound Together

Started services are great for background operations, but what if you need a service that's more interactive? In this chapter you'll discover how to create a **bound service**, a *type of service your activity can interact with*. You'll see how to **bind** to the service when you need it, and how to **unbind** from it when you're done to save resources. You'll find out how to use **Android's Location Services** to get *location updates from your device GPS*. Finally, you'll discover how to use **Android's permission model**, including *handling runtime permission requests*.

Bound services are bound to other components	768
Create a new service	770
Implement a binder	771
Add a <code>getDistance()</code> method to the service	772
Update MainActivity's layout	773
Create a ServiceConnection	775
Use <code>bindService()</code> to bind the service	778
Use <code>unbindService()</code> to unbind from the service	779
Call OdometerService's <code>getDistance()</code> method	780
The full MainActivity.java code	781
The states of a bound service	787
Add the AppCompat Support Library	790
Add a location listener to OdometerService	792
Here's the updated OdometerService code	795
Calculate the distance traveled	796
The full OdometerService.java code	798
Get the app to request permission	802
Check the user's response to the permission request	805
Add notification code to <code>onRequestPermissionsResult()</code>	809
The full code for MainActivity.java	811
Your Android Toolbox	815
It's been great having you here in Androidville	816



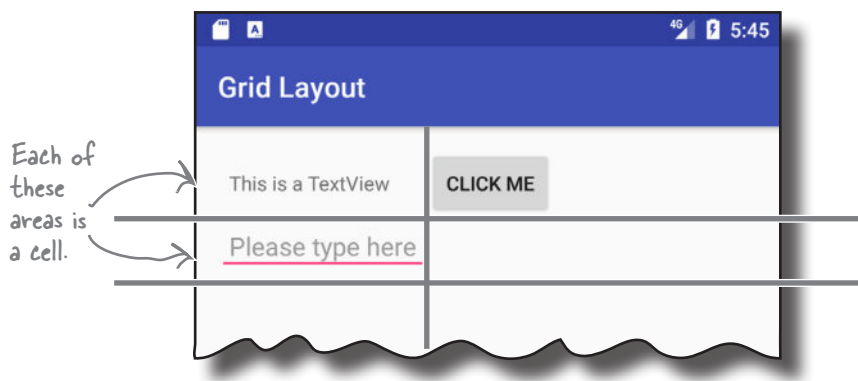
relative and grid layouts

Meet the Relatives



There are two more layouts you will often meet in **Androidville**.

In this book we've concentrated on using simple *linear and frame layouts*, and introduced you to *Android's new constraint layout*. But there are two more layouts we'd like you to know about: **relative layouts** and **grid layouts**. They've largely been superseded by the constraint layout, but we have a soft spot for them, and we think they'll stay around for a few more years.



gradle

The Gradle Build Tool



Most Android apps are created using a build tool called **Gradle**.

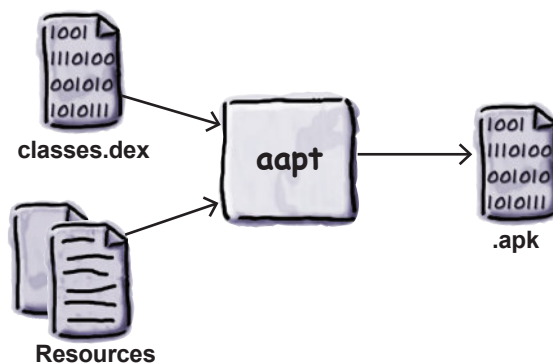
Gradle works behind the scenes to find and download libraries, compile and deploy your code, run tests, clean the grouting, and so on. Most of the time you *might not even realize it's there* because Android Studio provides a graphical interface to it. Sometimes, however, it's helpful to dive into Gradle and **hack it manually**. In this appendix we'll introduce you to some of Gradle's many talents.

art



The Android Runtime

Ever wonder how Android apps can run on so many kinds of devices? Android apps run in a virtual machine called the **Android runtime (ART)**, not the Oracle Java Virtual Machine (JVM). This means that your apps are quicker to start on small, low-powered devices and run more efficiently. In this appendix, we'll look at how ART works.

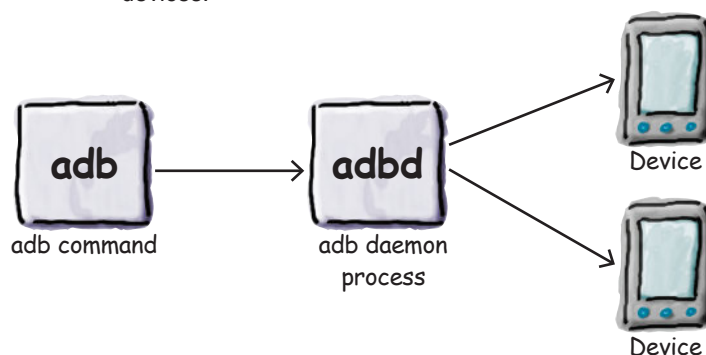


adb



The Android Debug Bridge

In this book, we've focused on using an IDE for all your Android needs. But there are times when using a command-line tool can be plain useful, like those times when Android Studio can't see your Android device but you just *know* it's there. In this chapter, we'll introduce you to the **Android Debug Bridge (or adb)**, a command-line tool you can use to communicate with the emulator or Android devices.

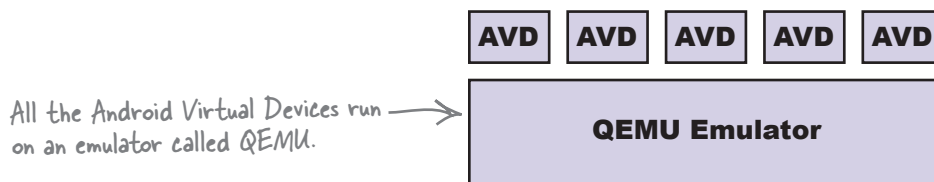


the android emulator

Speeding Things Up

V

Ever felt like you were spending all your time waiting for the emulator? There's no doubt that using the Android emulator is useful. It allows you to see how your app will run on devices other than the physical ones you have access to. But at times it can feel a little...sluggish. In this appendix, we'll explain why the emulator can seem slow. Even better, we'll give you a few tips we've learned for speeding it up.



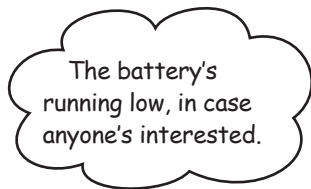
leftovers

The Top Ten Things (we didn't cover)

vi

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.**

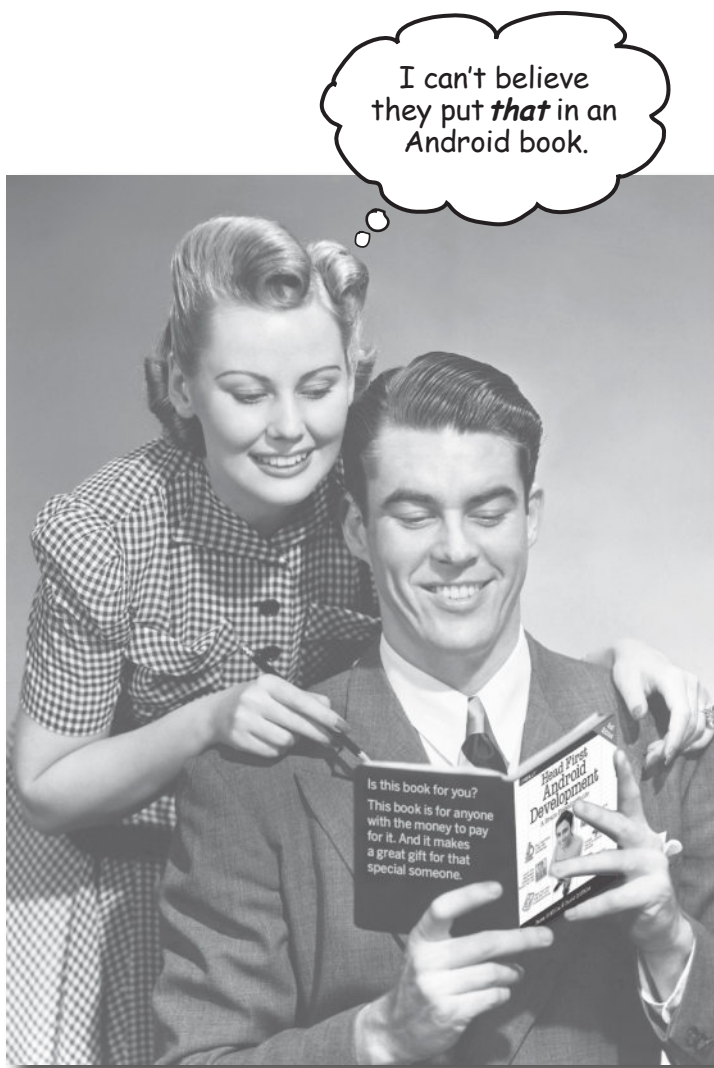


Android

- | | |
|--------------------------|-----|
| 1. Distributing your app | 862 |
| 2. Content providers | 863 |
| 3. Loaders | 864 |
| 4. Sync adapters | 864 |
| 5. Broadcasts | 865 |
| 6. The WebView class | 866 |
| 7. Settings | 867 |
| 8. Animation | 868 |
| 9. App widgets | 869 |
| 10. Automated testing | 870 |

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 Java?
- 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?

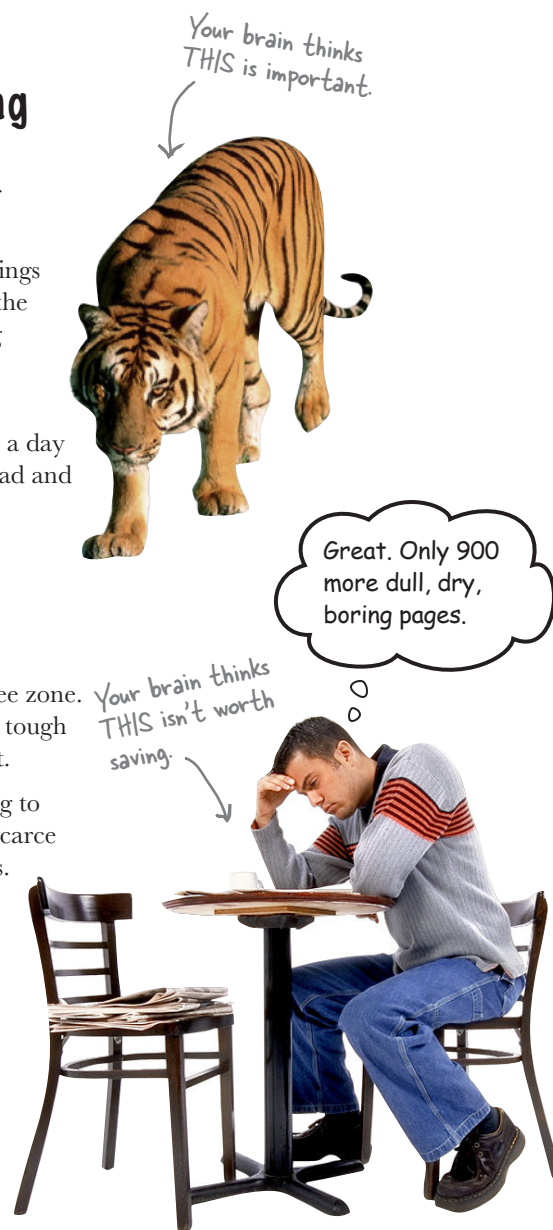
Neurons fire. Emotions crank up. *Chemicals surge.*

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 your Facebook page. 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 make 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.

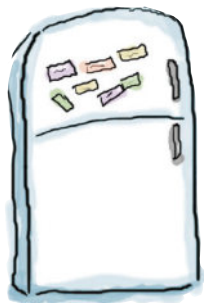
We used **multiple learning styles**, because *you* might prefer step-by-step procedures, while someone else wants to understand the big picture first, and someone else just wants to see an example. But regardless of your own learning preference, *everyone* benefits from seeing the same content represented 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, and the like, because, well, *you're* a person. And your brain pays more attention to *people* than it does to *things*.



Cut this out and stick it on your refrigerator.

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.

-
- 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 Java.

We're going to be building Android apps using a combination of Java and XML. We assume that you're familiar with the Java programming language. If you've never done any Java programming *at all*, then you might want to read *Head First Java* 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. You'll also 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. The source code for all the apps is here: <https://tinyurl.com/HeadFirstAndroid>.

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.

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 technical review team

Andy



Jacqui



Technical reviewers:

Andy Parker is currently working as a development manager, but has been a research physicist, teacher, designer, reviewer, and team leader at various points in his career. Through all of his roles, he has never lost his passion for creating top quality, well-designed, and well-engineered software. Nowadays, he spends most of his time managing great Agile teams and passing on his wide range of experience to the next generation of developers.

Jacqui Cope started coding to avoid school netball practice. Since then she has gathered 30 years' experience working with a variety of financial software systems, from coding in COBOL to test management. Recently she has gained her MSc in Computer Security and has moved into software 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.

Acknowledgments

Our editor:

Heartfelt thanks to our wonderful editor **Dawn Schanafelt** for picking up the reins on the second edition. She has truly been amazing, and a delight to work with. She made us feel valued and supported every step of the way, and gave us invaluable feedback and insight exactly when we needed it. We've appreciated all the many times she told us our sentences had all the right words, but not necessarily in the right order.

Thanks also to **Bert Bates** for teaching us to throw away the old rulebook and for letting us into his brain.

Dawn Schanafelt



The O'Reilly team:

A big thank you goes to **Mike Hendrickson** for having confidence in us and asking us to write the first edition of the book; **Heather Scherer** for her behind-the-scenes organization skills and herding; the **early release team** for making early versions of the book available for download; and the **design team** for all their extra help. 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 a rollercoaster of a ride, even when it's a second edition, and this one's been no exception. We've truly valued the kindness and support of our family and friends along the way. Special thanks go to **Ian, Steve, Colin, Angela, Paul B, Chris, Michael, Mum, Dad, Carl, Rob,** and **Lorraine**.

The without-whom list:

Our technical review team did a great job of keeping us straight, and making sure that what we covered was spot on. We're also grateful to **Ingo Krotzky** for his valuable feedback on the early release of this book, and all the people who gave us feedback on the first edition. We think the book's much, much better as a result.

Finally, our thanks to **Kathy Sierra** and **Bert Bates** for creating this extraordinary series of books.

O'Reilly Safari®

Safari (formerly Safari Books Online) is a membership-based training and reference platform for enterprise, government, educators, and individuals.

Members have access to thousands of books, training videos, Learning Paths, interactive tutorials, and curated playlists from over 250 publishers, including O'Reilly Media, Harvard Business Review, Prentice Hall Professional, Addison-Wesley Professional, Microsoft Press, Sams, Que, Peachpit Press, Adobe, Focal Press, Cisco Press, John Wiley & Sons, Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett, and Course Technology, among others.

For more information, please visit <http://oreilly.com/safari>.

1 getting started

✦ *Diving In* ✦



Android has taken the world by storm.

Everybody wants a smartphone or tablet, and Android devices are hugely popular. In this book, we'll teach you how to **develop your own apps**, and we'll start by getting you to build a basic app and run it on an Android Virtual Device. Along the way, you'll meet some of the basic components of all Android apps, such as **activities** and **layouts**. **All you need is a little Java know-how...**

Welcome to Androidville

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

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 using a mix of Java and XML. 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?

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** to define its appearance. Layouts are usually defined in XML, and can include GUI components such as buttons, text fields, and labels.

Activities define what the app does

Layouts only define the *appearance* of the app. You define what the app *does* using one or more **activities**. An activity is a special Java class that decides which layout to use and tells the app how to respond to the user. As an example, if a layout includes a button, you need to write Java code in the activity to define what the button should do when you press it.

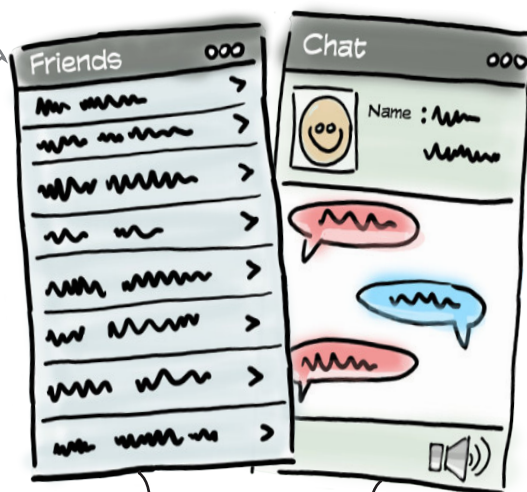
Sometimes extra resources are needed too

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.

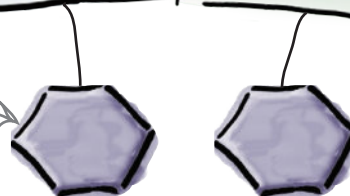
Android apps are really just a bunch of files in particular directories. When you build your app, all of these files get bundled together, giving you an app you can run on your device.

We're going to build our Android apps using a mixture of Java and XML. We'll explain things along the way, but you'll need to have a fair understanding of Java to get the most out of this book.

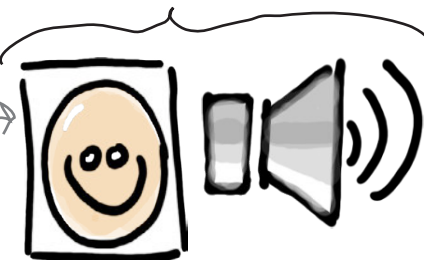
Layouts tell Android what the screens in your app look like.



Activities define what the app should do.



Resources can include sound and image files.



The Android platform dissected

The Android platform is made up of a number of different components. It includes core applications such as Contacts, a set of APIs to help you control what your app looks like and how it behaves, and a whole load of supporting files and libraries. Here's a quick look at how they all fit together:



Don't worry if this seems like a lot to take in.

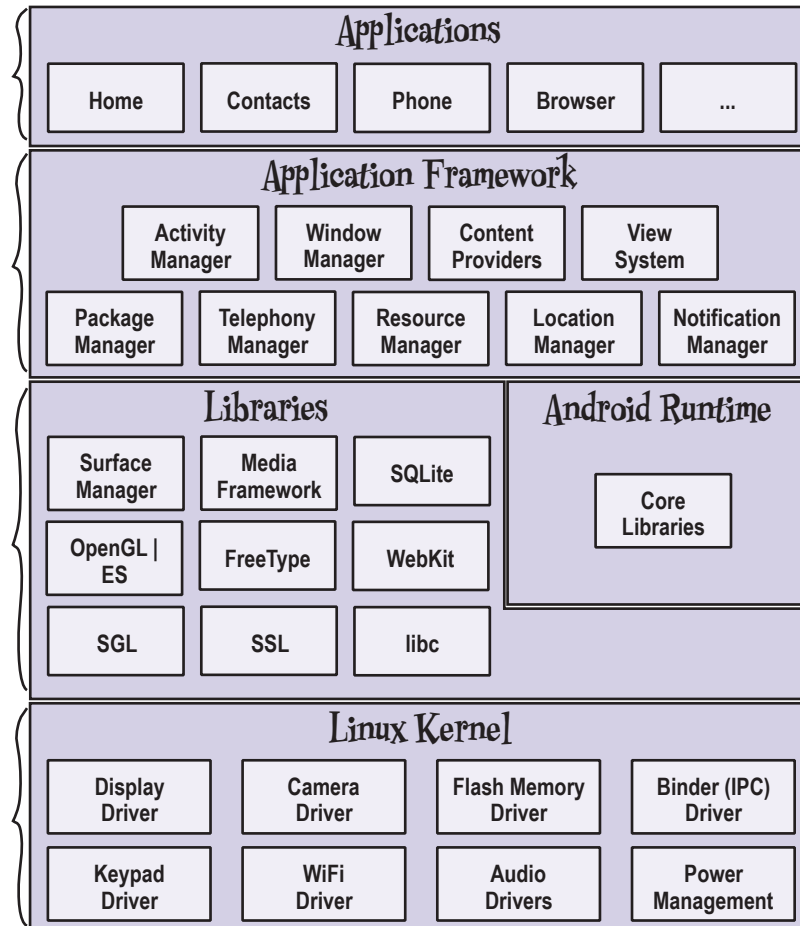
We're just giving you an overview of what's included in the Android platform. We'll explain the different components in more detail as and when we need to.

Android comes with a set of core applications such as Contacts, Phone, Calendar, and a browser.

When you build apps, you have access to the same APIs used by the core applications. You use these APIs to control what your app looks like and how it behaves.

Underneath the application framework lies a set of C and C++ libraries. These libraries get exposed to you through the framework APIs.

Underneath everything else lies the Linux kernel. Android relies on the kernel for drivers, and also core services such as security and memory management.



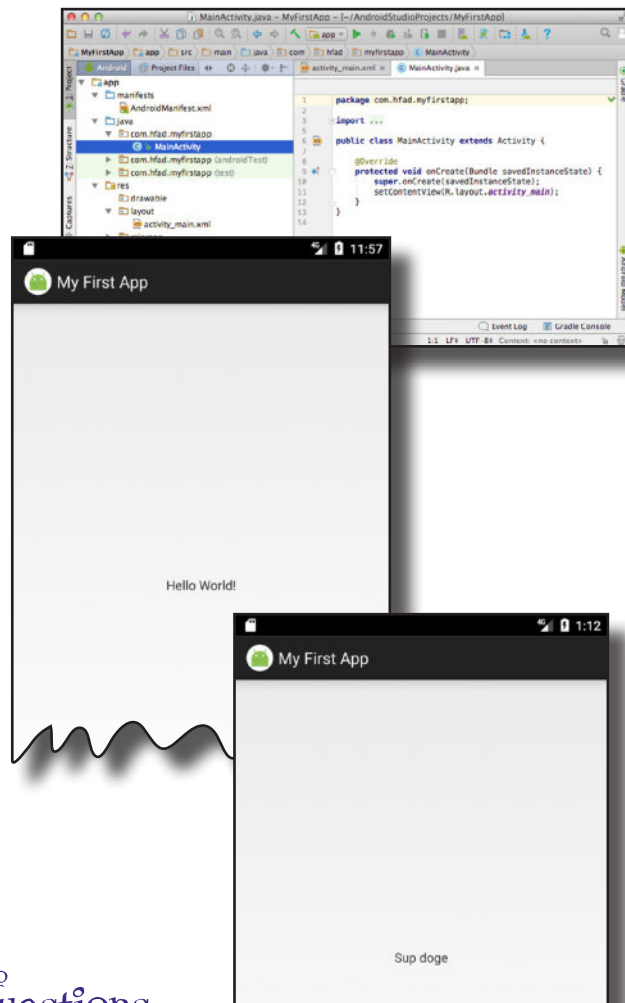
The Android runtime comes with a set of core libraries that implement most of the Java programming language. Each Android app runs in its own process.

The great news is that all of the powerful Android libraries are exposed through the APIs in the application framework, and it's these APIs that you use to create great Android apps. All you need to begin is some Java knowledge and a great idea for an app.

Here's what we're going to do

So let's dive in and create a basic 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 in the Android emulator.**
We'll use the built-in emulator to see the app up and running.
- 4 **Change the app.**
Finally, we'll make a few tweaks to the app we created in step 2, and run it again.



there are no Dumb Questions

Q: Are all Android apps developed in Java?

A: You can develop Android apps in other languages, too. Most developers use Java, so that's what we're covering in this book.

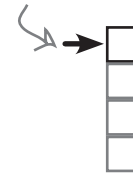
Q: How much Java do I need to know for Android app development?

A: You really need experience with Java SE (Standard Edition). If you're feeling rusty, we suggest getting a copy of *Head First Java* by Kathy Sierra and Bert Bates.

Q: Do I need to know about Swing and AWT?

A: Android doesn't use Swing or AWT, so don't worry if you don't have Java desktop GUI experience.

You are here.



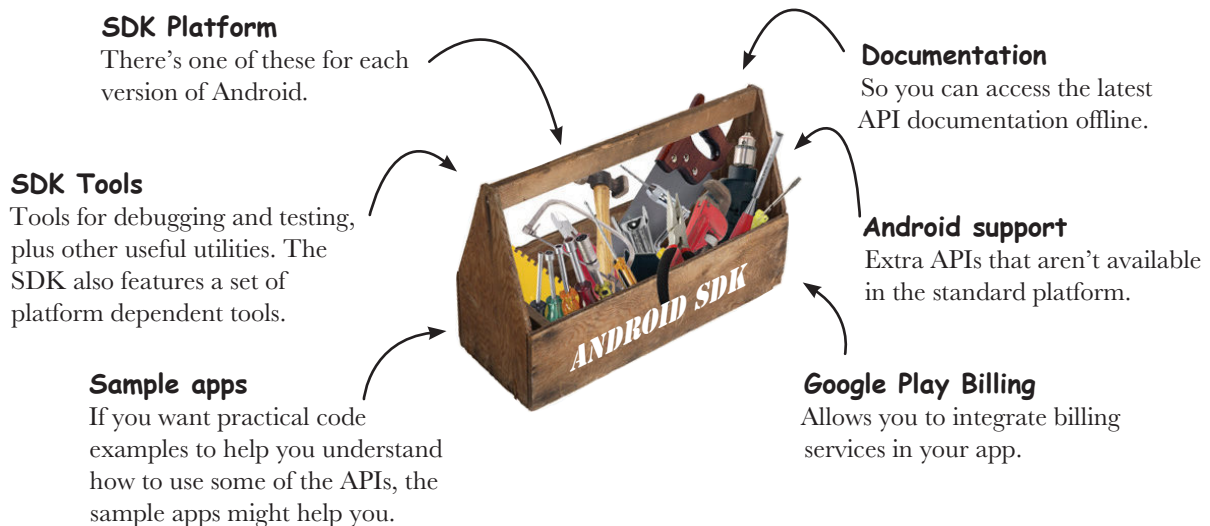
Your development environment

Java is the most popular language used to develop Android applications. Android devices don't run `.class` and `.jar` files. Instead, to improve speed and battery performance, Android devices use their own optimized formats for compiled code. That means that you can't use an ordinary Java development environment—you also need special tools to convert your compiled code into an Android format, to deploy them to an Android device, and to let you debug the app once it's running.

All of these come as part of the **Android SDK**. Let's take a look at what's included.

The Android SDK

The Android Software Development Kit contains the libraries and tools you need to develop Android apps. Here are some of the main points:



Android Studio is a special version of IntelliJ IDEA

IntelliJ IDEA is one of the most popular IDEs for Java development. Android Studio is a version of IDEA that includes a version of the Android SDK and extra GUI tools to help you with your app development.

In addition to providing you with an editor and access to the tools and libraries in the Android SDK, Android Studio gives you templates you can use to help you create new apps and classes, and it makes it easy to do things such as package your apps and run them.

Install Android Studio

Before we go any further, you need to install Android Studio on your machine. We're not including the installation instructions in this book 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:

<http://developer.android.com/sdk/index.html#Requirements>

Then follow the Android Studio installation instructions here:

<https://developer.android.com/sdk/installing/index.html?pkg=studio>

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

When you're done, you should see the Android Studio welcome screen. You're now ready to build your first Android app.

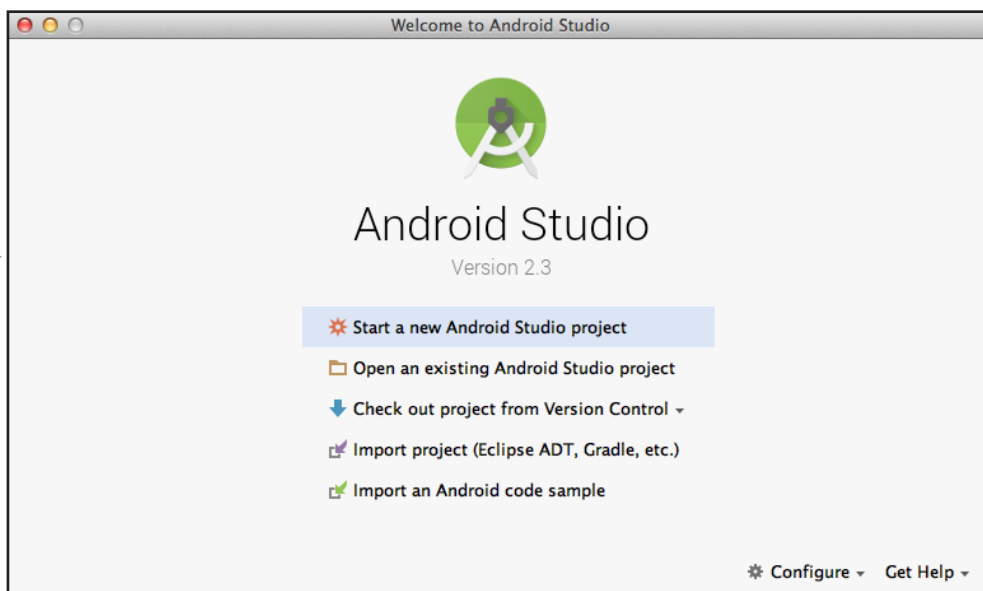


- Set up environment
- Build app
- Run app
- Change app

We're using Android Studio version 2.3. You'll need to use this version or above to get the most out of this book.

Google sometimes changes their URLs. If these URLs don't work, search for Android Studio and you should find them.

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



there are no Dumb Questions

Q: You say we're going to use Android Studio to build the Android apps. Do I have to?

A: Strictly speaking, you don't *have* to use Android Studio to build Android apps. All you need is a tool that will let you write and compile Java code, plus a few other tools to convert the compiled code into a form that Android devices can run.

Android Studio is the official Android IDE, and the Android team recommends using it. But quite a lot of people use IntelliJ IDEA instead.

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

A: It's possible, but it's more work. Most Android apps are now created using a build tool called *Gradle*. Gradle projects can be created and built using a text editor and a command line.

Q: A build tool? So is gradle like ANT?

A: It's similar, but Gradle is much more powerful than ANT. Gradle can compile and deploy code, just like ANT, but it also uses Maven to download any third-party libraries your code needs. Gradle also uses Groovy as a scripting language, which means you can easily create quite complex builds with Gradle.

Q: Most apps are built using Gradle? I thought you said most developers use Android Studio.

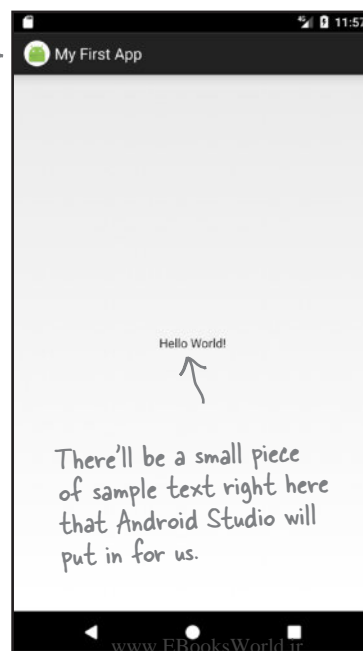
A: Android Studio provides a graphical interface to Gradle, and also to other tools for creating layouts, reading logs, and debugging.

You can find out more about Gradle in Appendix II.

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 the app will look like:

This is the name of the application.



This is a very simple app, but that's all you need for your very first Android app.

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



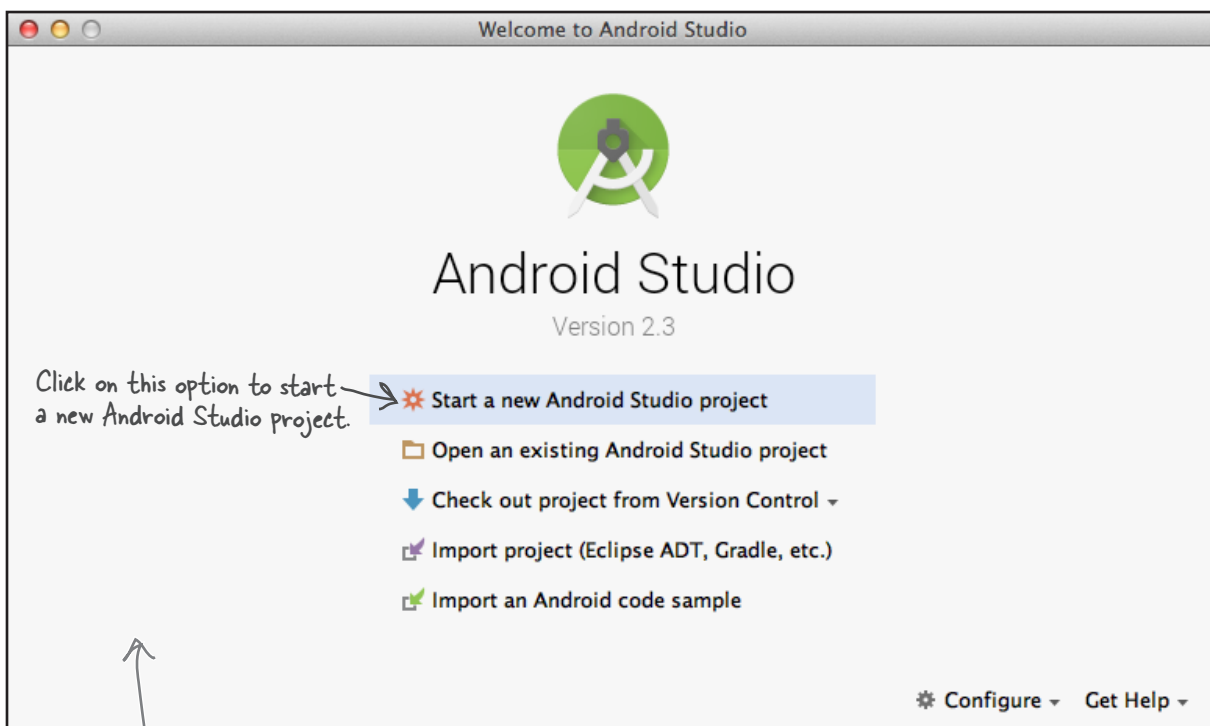
- Set up environment
- Build app**
- Run app
- Change app

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.

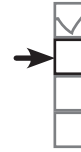
1. Create a new project

The Android Studio welcome screen gives you a number of options. We want to create a new project, so click on the option for “Start a new Android Studio project.”



Click on this option to start a new Android Studio project.

Any projects you create will appear here. This is our first project, so this area is currently empty.



How to build the app (continued)

2. Configure the project

You now need to configure the app by telling Android Studio what you want to call it, what company domain to use, and where you would like to store the files.

Android Studio uses the company domain and application name to form the name of the package that will be used for your app. As an example, if you give your app a name of “My First App” and use a company domain of “hfad.com”, Android Studio will derive a package name of `com.hfad.myfirstapp`. The package name is really important in Android, as it’s used by Android devices to uniquely identify your app.

Enter an application name of “My First App”, enter a company domain of “hfad.com”, uncheck the option to include C++ support, and accept the default project location. Then click on the Next button.



Watch it!

The package name must stay the same for the lifetime of your app.

It's a unique identifier for your app and used to manage multiple versions of the same app.

Some versions of Android Studio may have an extra option asking if you want to include Kotlin support. Uncheck this option if it's there.

The wizard forms the package name by combining the application name and the company domain.

Configure your new project

The application name is shown in the Google Play Store and various other places, too.

Application name:

Company domain:

Package name: [Edit](#)

Include C++ support

Use a company domain of hfad.com.

Uncheck the option to include C++ support. If prompted, also uncheck the option to include Kotlin support.

Project location: ...

All of the files for your project will be stored here.

Cancel Previous **Next** Finish

How to build the app (continued)



Set up environment
Build app
Run app
Change app

3. Specify the minimum SDK

You now need to indicate the minimum SDK of Android your app will use. API levels increase with every new version of Android. Unless you only want your app to run on the very newest devices, you'll probably want to specify one of the older APIs.

Here, we're choosing a minimum SDK of API level 19, which means it will be able to run on most devices. Also, we're only going to create a version of our app to run on phones and tablets, so we'll leave the other options unchecked.

← There's more about the different API levels on the next page.

When you've done this, click on the Next button.

The minimum required SDK is the lowest version your app will support. Your app will run on devices with this level API or higher. It won't run on devices with a lower API.

Create New Project

Target Android Devices

Select the form factors your app will run on

Different platforms may require separate SDKs

Phone and Tablet

Minimum SDK

Lower API levels target more devices, but have fewer features available. By targeting API 19 and later, your app will run on approximately 73.9% of the devices that are active on the Google Play Store.

[Help me choose](#)

Wear

Minimum SDK

TV

Minimum SDK

Android Auto

Glass

Minimum SDK

Cancel Previous **Next** Finish

Android Versions Up Close



You've probably heard a lot of things about Android that sound tasty, like Jelly Bean, KitKat, Lollipop, and Nougat. So what's with all the confectionary?

Android versions have a version number and a codename. The version number gives the precise version of Android (e.g., 7.0), while the codename is a more generic "friendly" name that may cover several versions of Android (e.g., Nougat). The API level refers to the version of the APIs used by applications. As an example, the equivalent API level for Android version 7.1.1 is 25.

Version	Codename	API level
1.0		1
1.1		2
1.5	Cupcake	3
1.6	Donut	4
2.0–2.1	Eclair	5–7
2.2.x	Froyo	8
2.3–2.3.7	Gingerbread	9–10
3.0 - 3.2	Honeycomb	11–13
4.0–4.0.4	Ice Cream Sandwich	14–15
4.1 - 4.3	Jelly Bean	16–18
4.4	KitKat	19–20
5.0–5.1	Lollipop	21–22
6.0	Marshmallow	23
7.0	Nougat	24
7.1–7.1.2	Nougat	25

Hardly anyone uses these versions anymore.

Most devices use one of these APIs.

When you develop Android apps, you really need to consider which versions of Android you want your app to be compatible with. If you specify that your 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 the percentage of devices running particular versions here: <https://developer.android.com/about/dashboards/index.html>.

Activities and layouts from 50,000 feet

The next thing you'll be prompted to do is add an activity to your project. Every Android app is a collection of screens, and each screen is composed of an activity and a layout.

An **activity** is a **single, defined thing that your user can do**. You might have an activity to compose an email, take a photo, or find a contact. Activities are usually associated with one screen, and they're written in Java.

A **layout** describes **the appearance of the screen**. Layouts are written as XML files and they tell Android how the different screen elements are arranged.

Let's look in more detail at how activities and layouts work together to create a user interface:



Set up environment
Build app
Run app
Change app

Layouts define how the user interface is presented.

Activities define actions.

1 The device launches your app and creates an activity object.

2 The activity object specifies a layout.

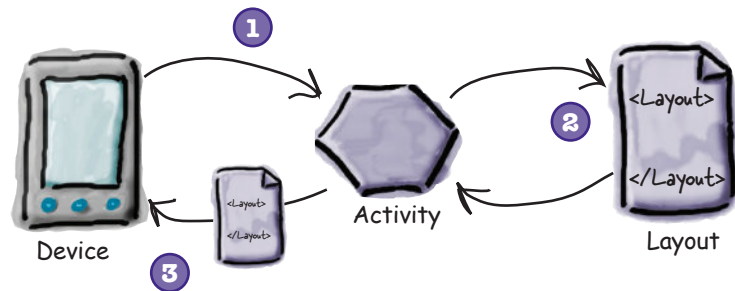
3 The activity tells Android to display the layout onscreen.

4 The user interacts with the layout that's displayed on the device.

5 The activity responds to these interactions by running application code.

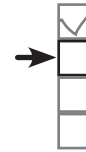
6 The activity updates the display...

7 ...which the user sees on the device.



Now that you know a bit more about what activities and layouts are, let's go through the last couple of steps in the Create New Project wizard and get it to create an activity and layout.

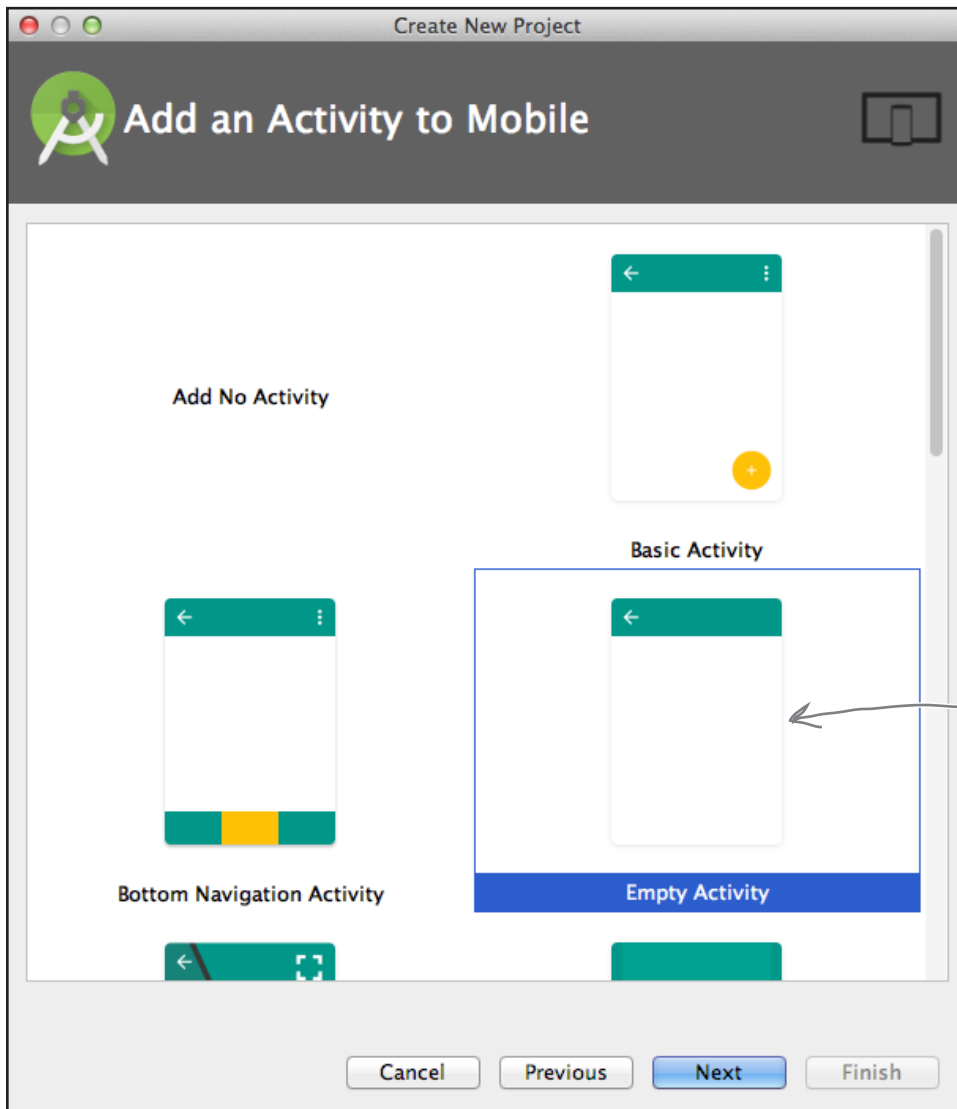
How to build the app (continued)



- getting started
- Set up environment
- Build app**
- Run app
- Change app

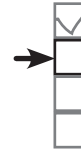
4. Add an activity

The next screen lets you choose among a series of templates you can use to create an activity and layout. We're going to create an app with an empty activity and layout, so choose the Empty Activity option and click the Next button.



There are other types of activity you can choose from, but for this exercise make sure you select the Empty Activity option.

How to build the app (continued)

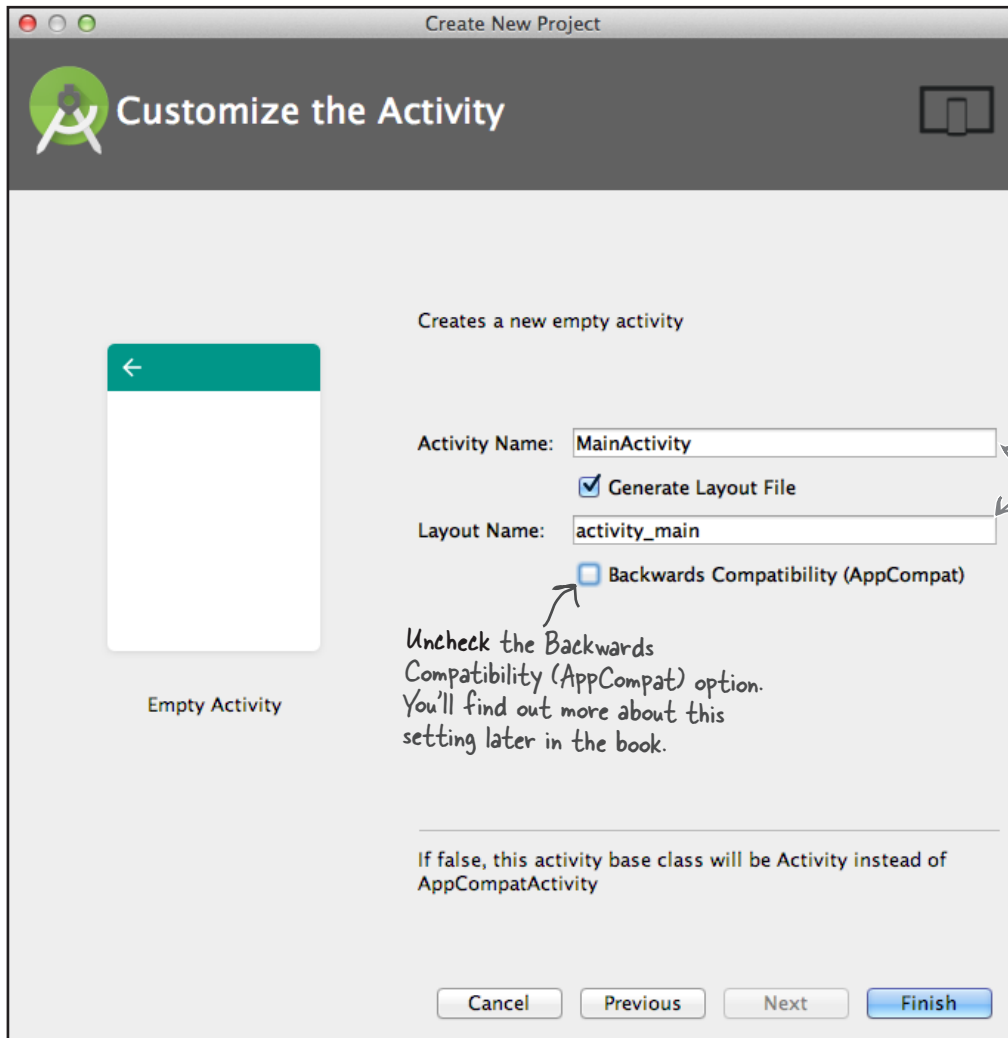


- Set up environment
- Build app**
- Run app
- Change app

5. Customize the activity

You will now be asked what you want to call the screen’s activity and layout. Enter an activity name of “MainActivity”, make sure the option to generate a layout file is checked, enter a layout name of “activity_main”, and then uncheck the Backwards Compatibility (AppCompat) option. The activity is a Java class, and the layout is an XML file, so the names we’ve given here will create a Java class file called *MainActivity.java* and an XML file called *activity_main.xml*.

When you click on the Finish button, Android Studio will build your app.



Name the activity “MainActivity” and the layout “activity_main”. Also make sure the option to generate the layout is checked.

Uncheck the Backwards Compatibility (AppCompat) option. You’ll find out more about this setting later in the book.

You've just created your first Android app



getting started
Set up environment
Build app
Run app
Change app

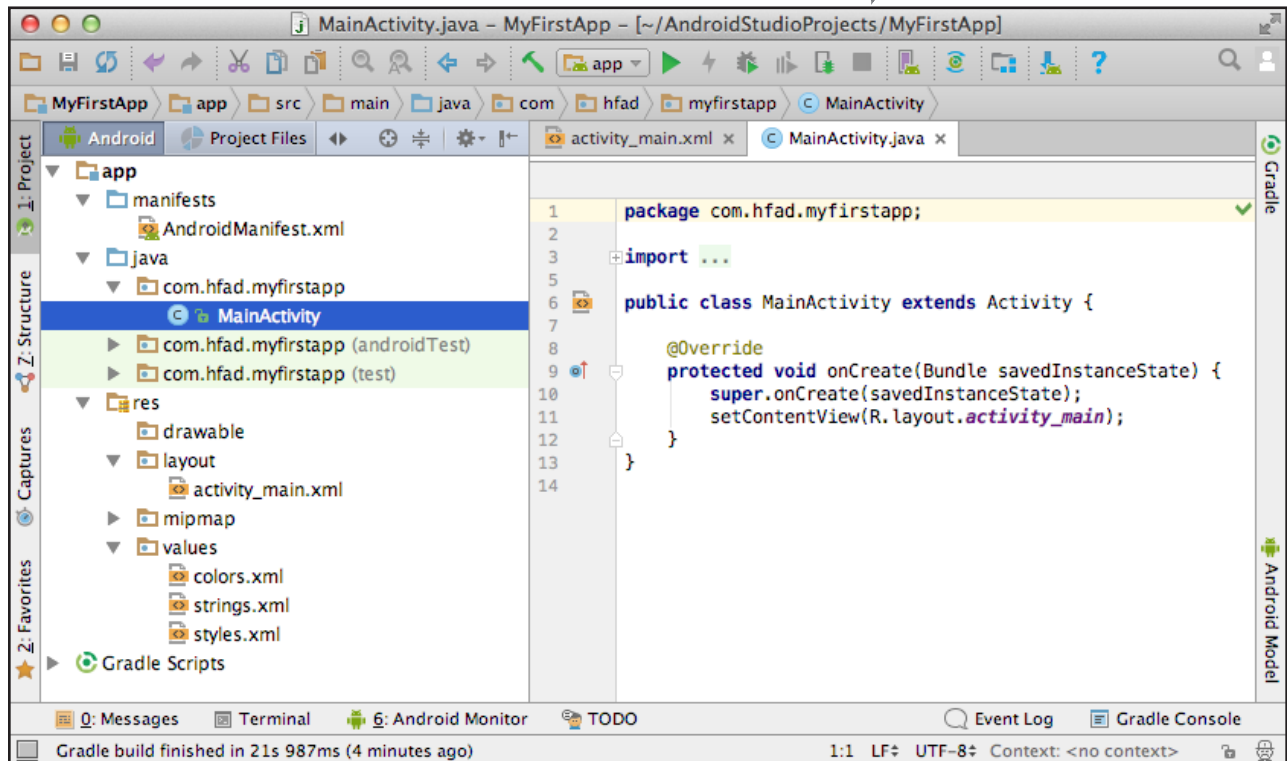
So what just happened?

- ★ **The Create New Project wizard created a project for your app, configured to your specifications.**
You defined which versions of Android the app should be compatible with, and the wizard created all of the files and folders needed for a basic valid app.
- ★ **The wizard created an activity and layout with template code.**
The template code includes layout XML and activity Java code, with sample “Hello World!” text in the layout.

When you finish creating your project by going through the wizard, Android Studio automatically displays the project 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):

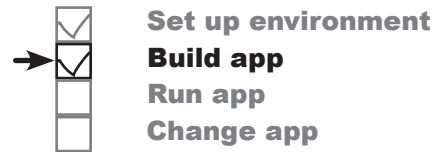
This is the project in Android Studio.



Android Studio creates a complete folder structure for you

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 with the explorer in the leftmost column of Android Studio.

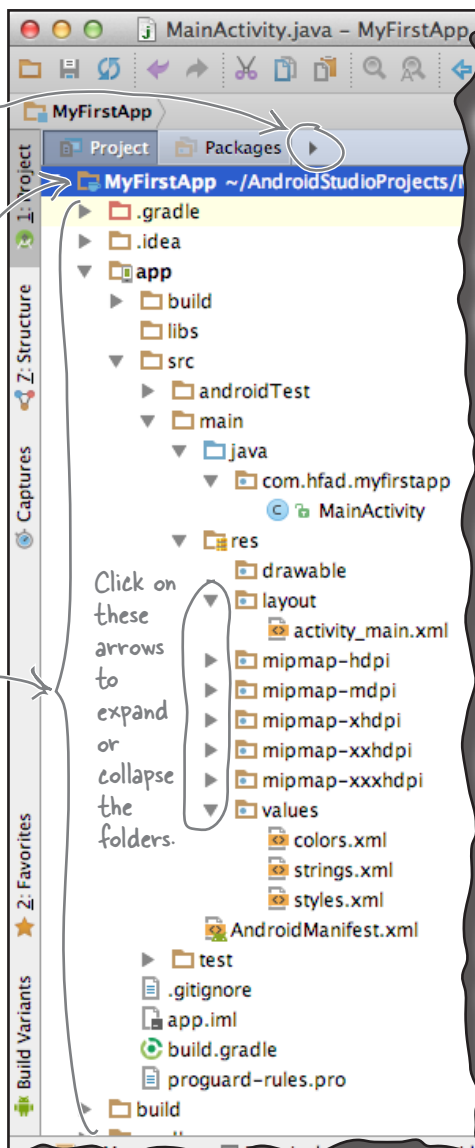
The explorer contains all of the projects that you currently have open. To expand or collapse folders, just click on the arrows to the left of the folder icons.



Click on the arrow here and choose the Project option to see the files and folders that make up your project.

This is the name of the project.

These files and folders are all included in your project.



Click on these arrows to expand or collapse the folders.

The folder structure includes different types of files

If you browse through the folder structure, you'll see that the wizard has created various types of files and folders for you:

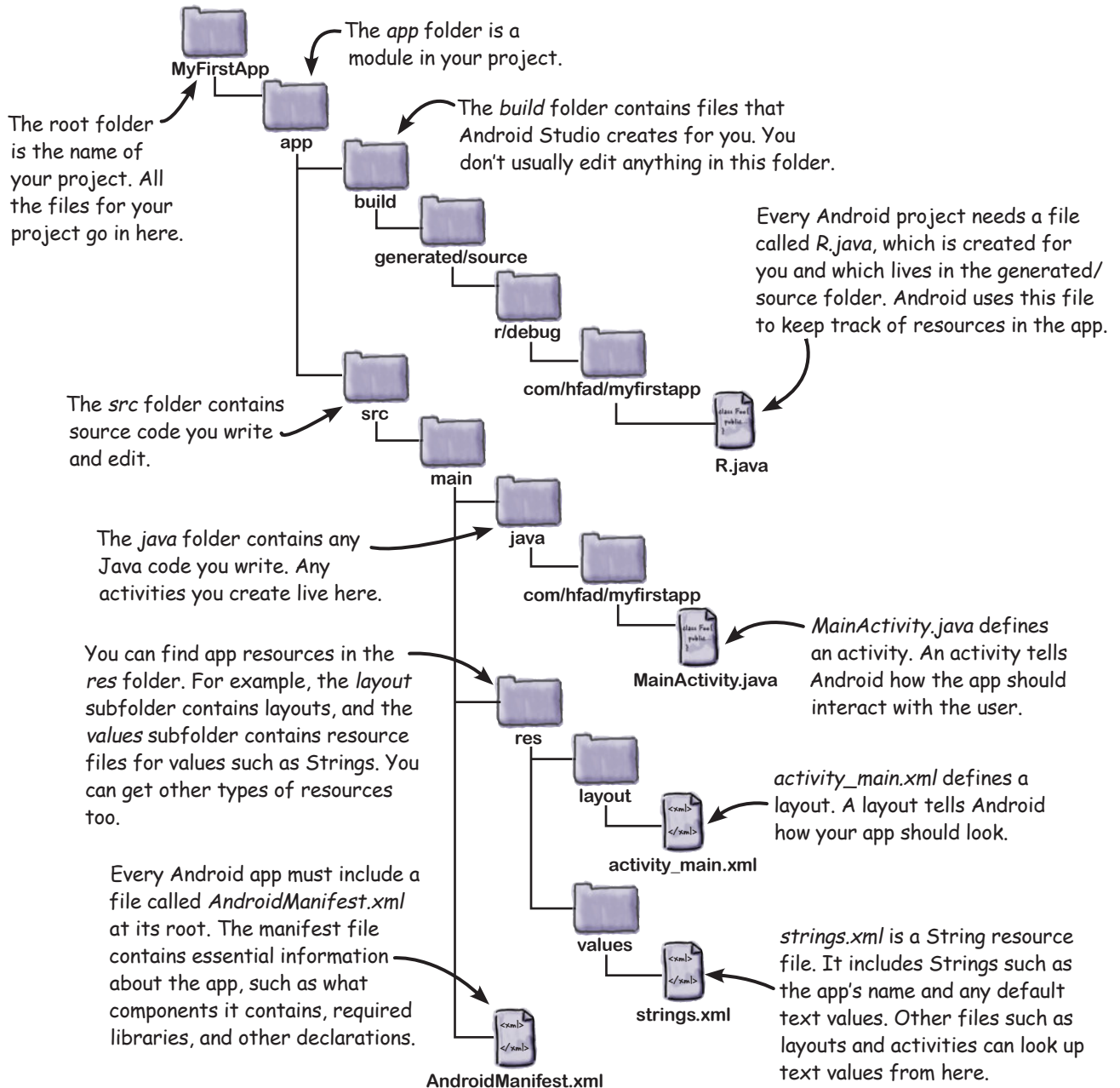
- ★ **Java and XML source files**
 These are the activity and layout files for your app.
- ★ **Android-generated Java files**
 There are some extra Java files you don't need to touch that Android Studio generates for you automatically.
- ★ **Resource files**
 These include default image files for icons, styles your app might use, and any common String values your app might want to look up.
- ★ **Android libraries**
 In the wizard, you specified the minimum SDK version you want your app to be compatible with. Android Studio makes sure your app includes the relevant Android libraries for that version.
- ★ **Configuration files**
 The configuration files tell Android what's actually in the app and how it should run.

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



Useful files in your project

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



Edit code with the Android Studio editors



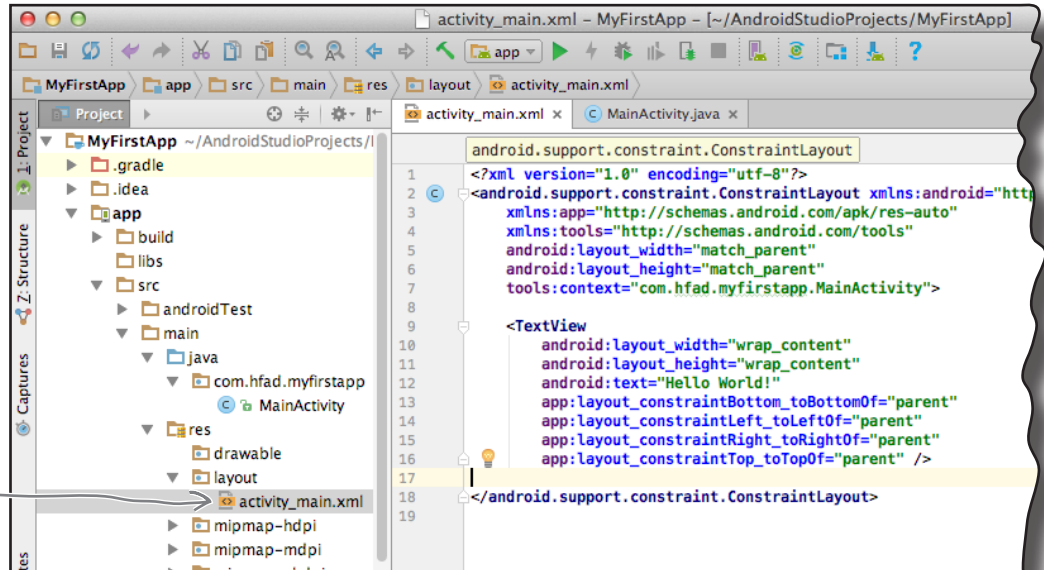
- Set up environment
- Build app
- Run app
- Change app

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 color coding and code checking.

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

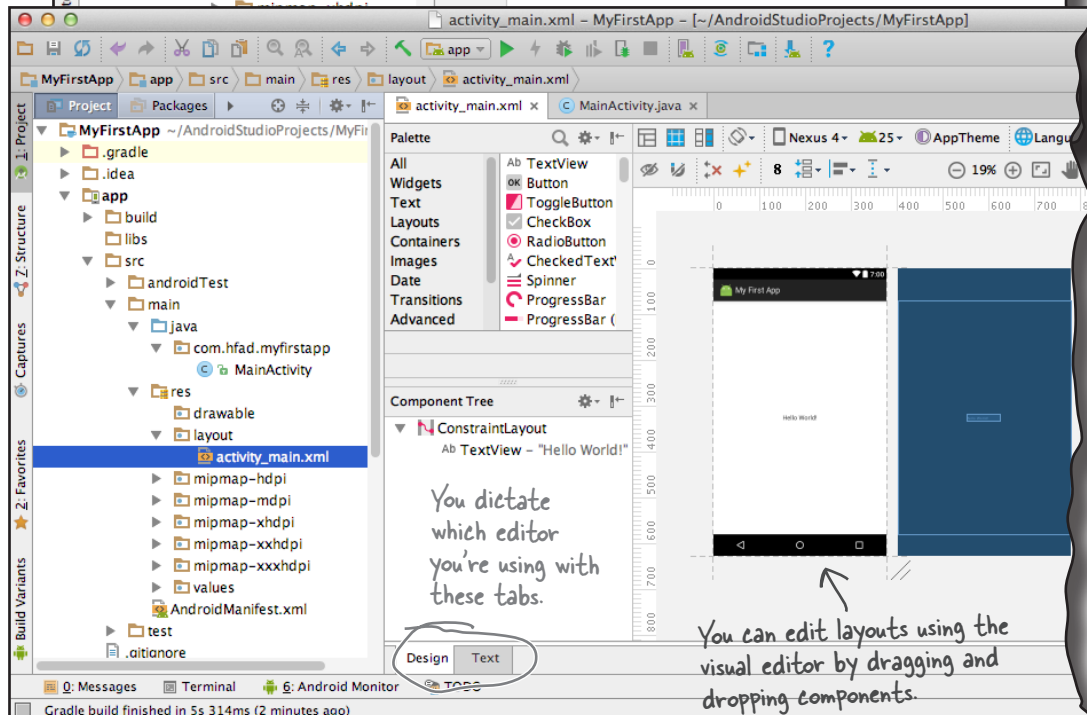


The design editor

If you're editing a layout, you have an extra option. Rather than edit the XML (such as that shown on the next page), you can use the design editor, which allows you to drag GUI components onto your 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 dictate which editor you're using with these tabs.

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



WHAT'S MY PURPOSE?

Here's the code from an example layout file (**not the one Android Studio generated for us**). We know you've not seen layout code before, but just 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.

activity_main.xml

```
<?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:paddingLeft="16dp"
    android:paddingRight="16dp"
    android:paddingTop="16dp"
    android:paddingBottom="16dp"
    android:orientation="vertical"
    tools:context="com.hfad.myfirstapp.MainActivity">

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

Add padding to the screen margins.

Include a `<TextView>` GUI component for displaying text.

Make the GUI component just large enough for its content.

Display the String "Hello World!"

Make the layout the same width and height as the screen size on the device.

WHAT'S MY PURPOSE? SOLUTION

Here's the code from an example layout file (**not the one Android Studio generated for us**). We know you've not seen layout code before, but just 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.

activity_main.xml

```
<?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:paddingLeft="16dp"
    android:paddingRight="16dp"
    android:paddingTop="16dp"
    android:paddingBottom="16dp"
    android:orientation="vertical"
    tools:context="com.hfad.myfirstapp.MainActivity">
    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hello World!" />
</LinearLayout>
```

Add padding to the screen margins.

Display the String "Hello World!"

Include a `<TextView>` GUI component for displaying text.

Make the layout the same width and height as the screen size on the device.

Make the GUI component just large enough for its content.

WHAT'S MY PURPOSE?

Now let's see if you can do the same thing for some activity code. **This is example code, and not necessarily the code that Android Studio will have generated for you.** Match the descriptions below to the correct lines of code.

MainActivity.java

```
package com.hfad.myfirstapp;

import android.os.Bundle;
import android.app.Activity;

public class MainActivity extends Activity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        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 Activity class. This method is called when the activity is first created.

MainActivity extends the Android class android.app.Activity.

WHAT'S MY PURPOSE? SOLUTION

Now let's see if you can do the same thing for some activity code. **This is example code, and not necessarily the code that Android Studio will have generated for you.** Match the descriptions below to the correct lines of code.

MainActivity.java

```
package com.hfad.myfirstapp;

import android.os.Bundle;
import android.app.Activity;

public class MainActivity extends Activity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        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 Activity class. This method is called when the activity is first created.

MainActivity extends the Android class android.app.Activity.



Run the app in the Android emulator

So far you've seen what your Android app looks like in Android Studio and got a feel for how it hangs together. But what you *really* want to do is see it running, right?

You have a couple of options when it comes to running your apps. The first option is to run them on a physical device. But what if you don't have one with you, or you want to see how your app looks on a type of device you don't have?

In that case, you can use the **Android emulator** that's built into the Android SDK. The emulator enables you to set up one or more **Android virtual devices** (AVDs) and then run your app in the emulator *as though it's running on a physical device*.

So what does the emulator look like?

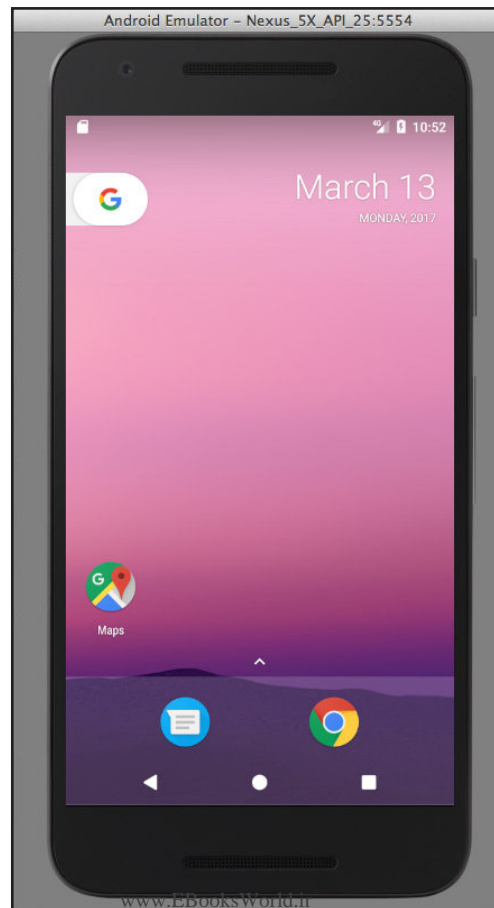
Here's an AVD running in the Android emulator. It looks just like a phone running on your computer.

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've set up the AVD in the first place. The AVD here is set up to mimic a Nexus 5X, so it will look and behave just like a Nexus 5X on your computer.

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

The Android emulator allows you to run your app on an Android virtual device (AVD), which behaves just like a physical Android device. You can set up numerous AVDs, each emulating a different type of device.



Once you've set up an AVD, you'll be able to see your app running on it. Android Studio launches the emulator for you.

Create an Android Virtual Device

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 Nexus 5X AVD running API level 25 so that you can see how your 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 Android on the Tools menu and choosing AVD Manager.

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.



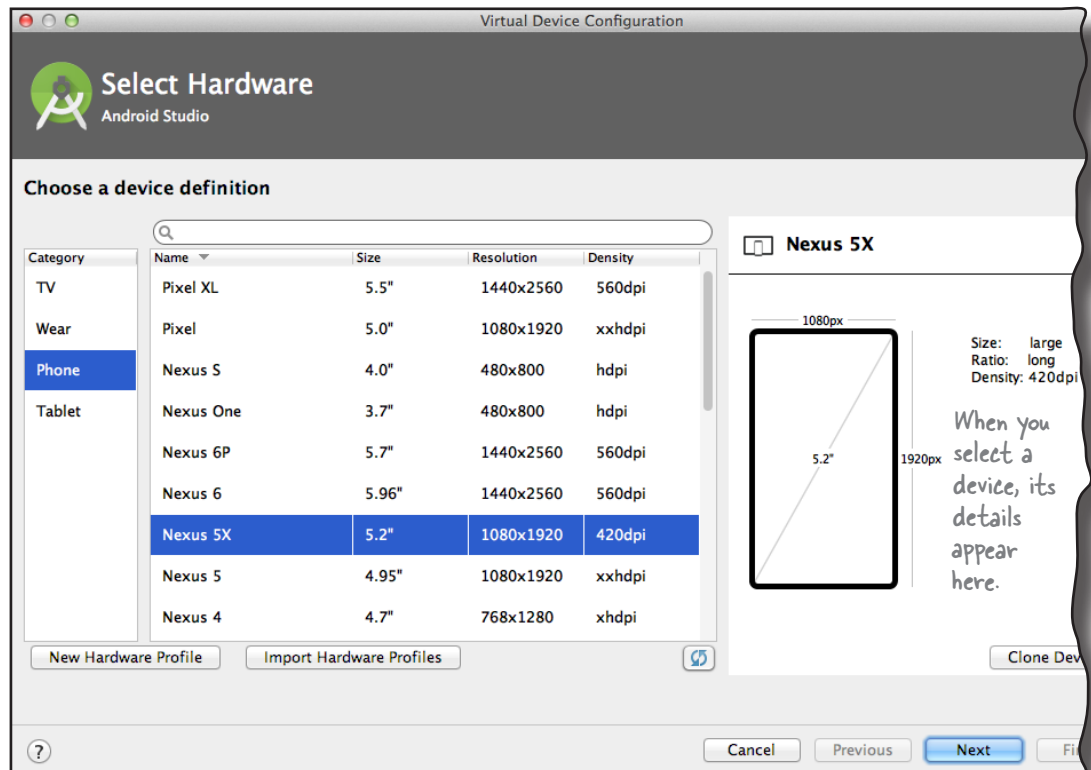
Set up environment
Build app
Run app
Change app



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 our app looks like running on a Nexus 5X phone. Choose Phone from the Category menu and Nexus 5X from the list. Then click the Next button.





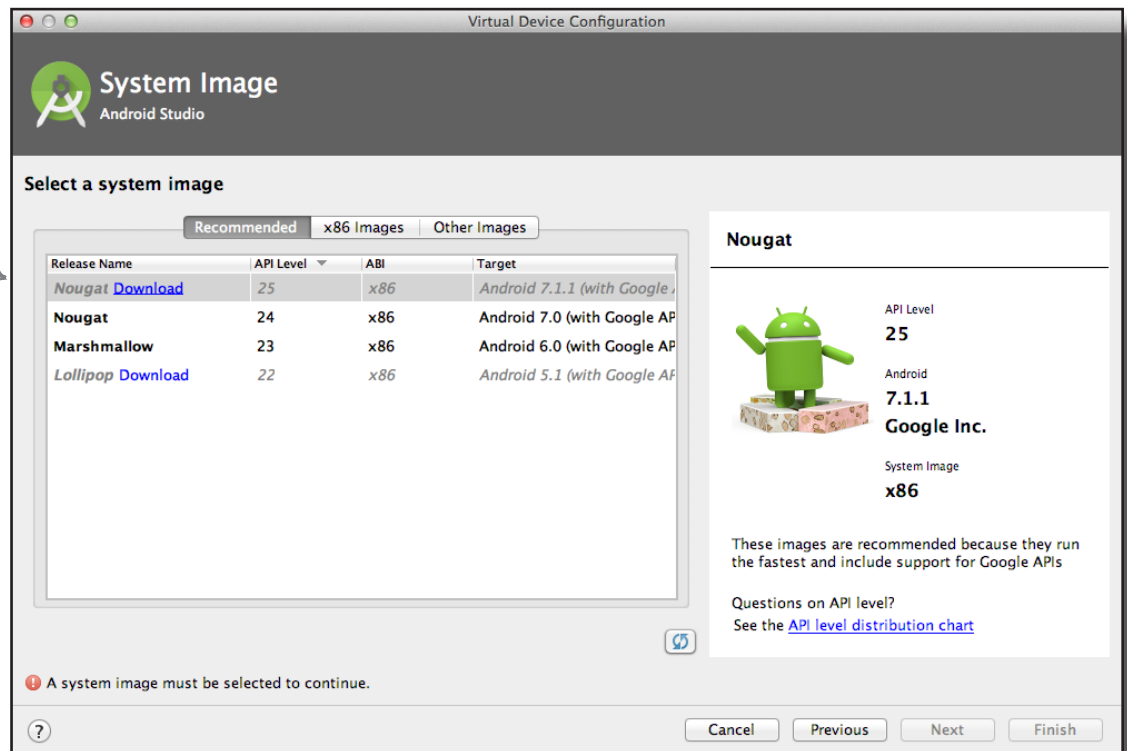
Creating an AVD (continued)

Select a system image

Next, you need to select a system image. The system image gives you an installed version of the Android operating system. You can choose the version of Android you want to be on your AVD.

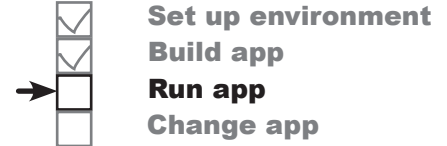
You need to choose a system image for an API level that's compatible with the app you're building. As an example, if you want your app to work on a minimum of API level 19, choose a system image for *at least* API level 19. We want our AVD to run API level 25, so choose the system image with a release name of Nougat and a target of Android 7.1.1 (API level 25). Then click on the Next button.

If you don't have this system image installed, you'll be given the option to download it.



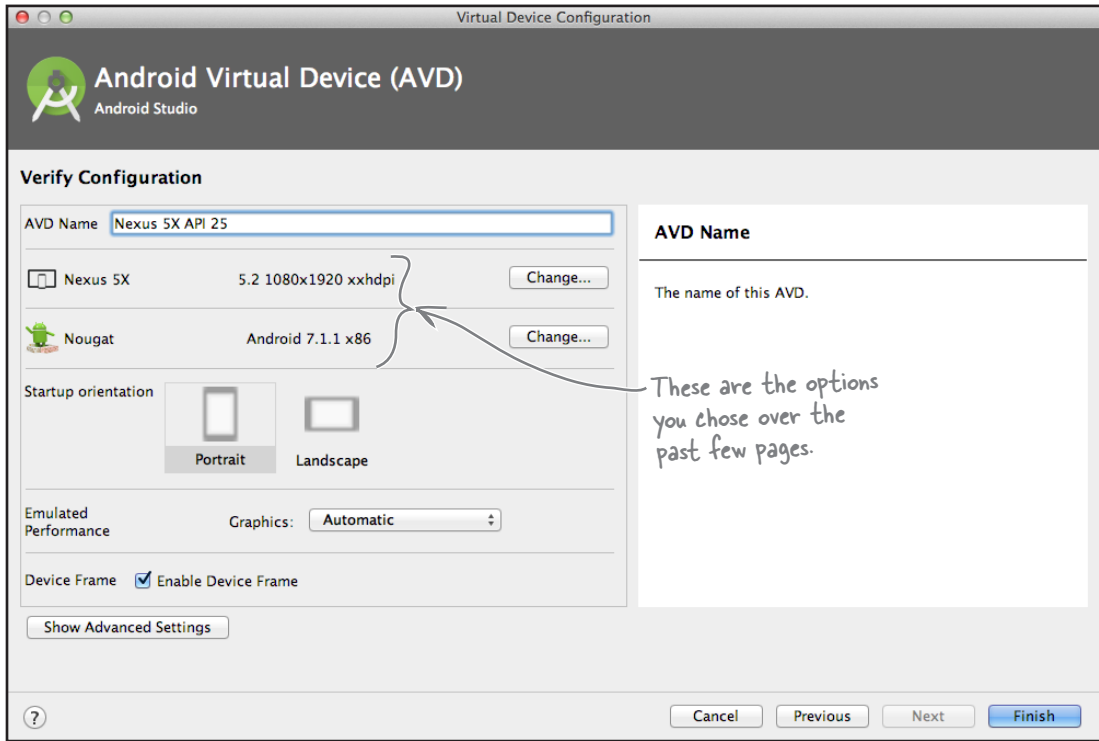
We'll continue setting up the AVD on the next page.

Creating an AVD (continued)

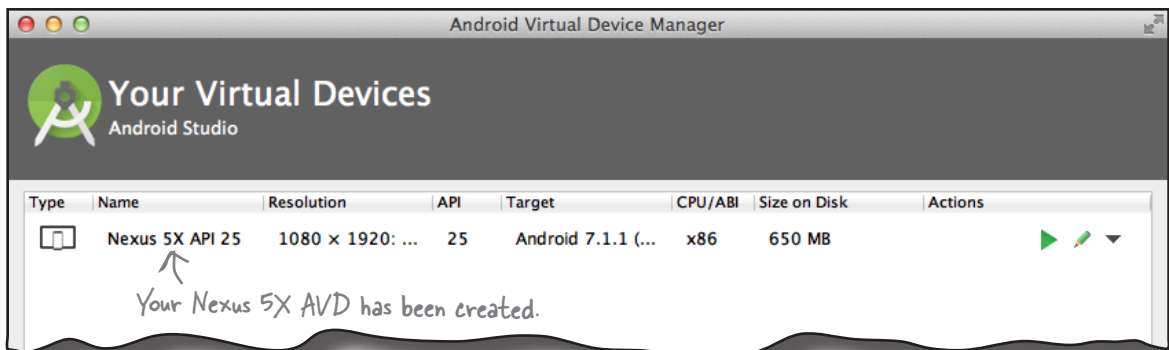


Verify the AVD configuration

On the next screen, you'll be asked to verify the AVD 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.



The AVD Manager will create the AVD for you, and when it's done, display it in the AVD Manager list of devices. You may now close the AVD Manager.





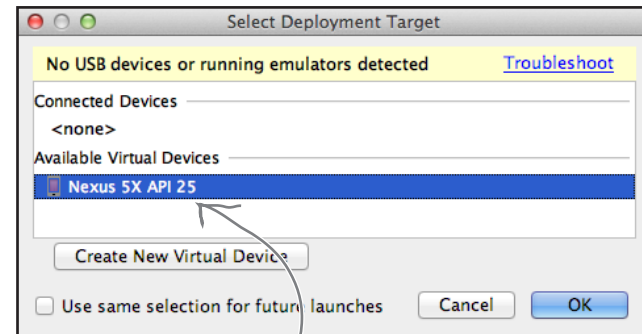
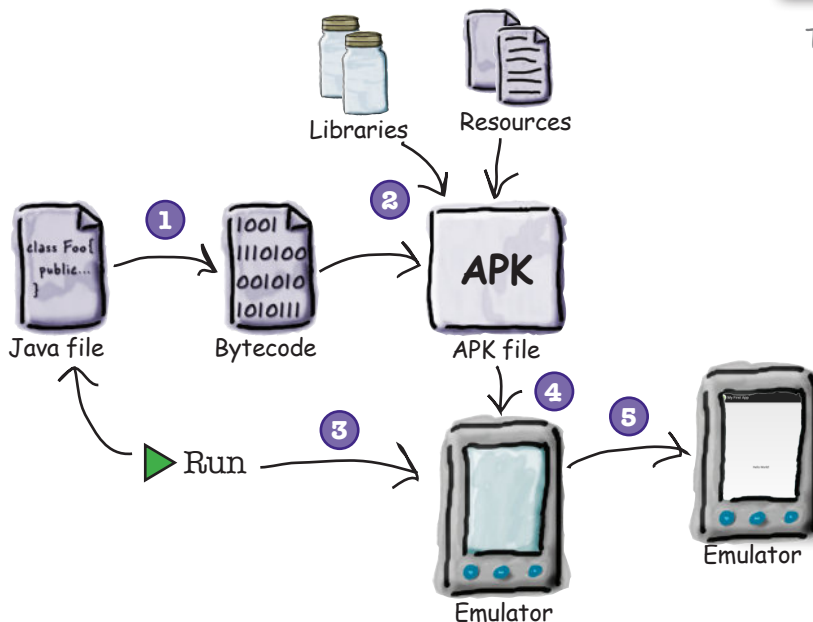
Run the app in the emulator

Now that you've set up your AVD, let's run the app on it. To do this, choose the "Run app" command from the Run menu. When you're asked to choose a device, select the Nexus 5X AVD you just created. Then click OK.

The AVD can take a few minutes to appear, so while we wait, let's take a look at what happens when you choose Run.

Compile, package, deploy, and 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:

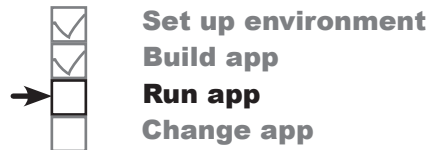


This is the AVD we just created.

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

- 1 The Java source files get compiled to bytecode.
- 2 An Android application package, or APK file, gets created.
The APK file includes the compiled Java files, along with any libraries and resources needed by your app.
- 3 Assuming there's not one already running, the emulator gets launched and then runs the AVD.
- 4 Once the emulator has been launched and the AVD is active, the APK file is uploaded to the AVD and installed.
- 5 The AVD starts the main activity associated with the app.
Your app gets displayed on the AVD screen, and it's all ready for you to test out.

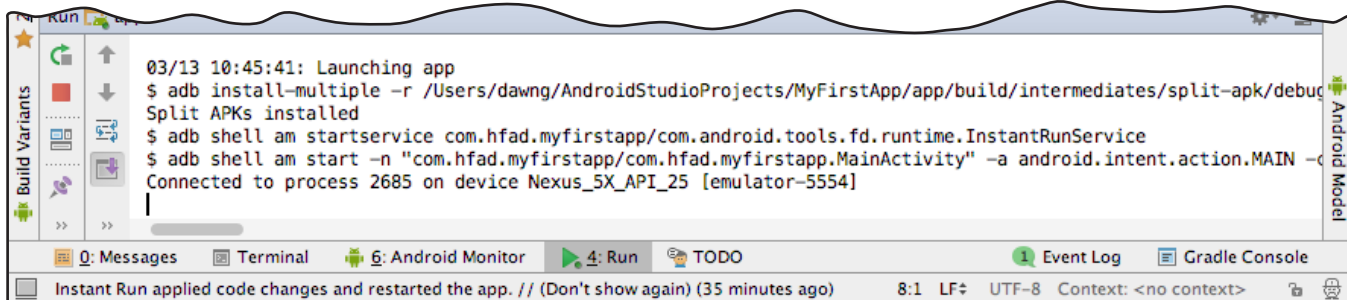
You can watch progress in the console



It can sometimes take quite a while for the emulator to launch with your AVD—often *several minutes*. If you like, you can watch what’s happening using the Android Studio console. The console gives you a blow-by-blow account of what the build system is doing, and if it encounters any errors, you’ll see them highlighted in the text.

We suggest finding something else to do while waiting for the emulator to start. Like quilting, or cooking a small meal.

You can find the console at the bottom of the Android Studio screen (click on the Run option at the bottom of the screen if it doesn’t appear automatically):



Here’s the output from our console window when we ran our app:

```
03/13 10:45:41: Launching app
$ adb install-multiple -r /Users/dawng/AndroidStudioProjects/MyFirstApp/app/build/intermediates/split-apk/debug/
split-apk/debug/dep/dependencies.apk /Users/dawng/AndroidStudioProjects/MyFirstApp/app/build/
intermediates/split-apk/debug/slices/slice_1.apk /Users/dawng/AndroidStudioProjects/MyFirstApp/
app/build/intermediates/split-apk/debug/slices/slice_2.apk /Users/dawng/AndroidStudioProjects/
MyFirstApp/app/build/intermediates/split-apk/debug/slices/slice_0.apk /Users/dawng/
AndroidStudioProjects/MyFirstApp/app/build/intermediates/split-apk/debug/slices/slice_3.apk /Users/
dawng/AndroidStudioProjects/MyFirstApp/app/build/intermediates/split-apk/debug/slices/slice_6.apk /
Users/dawng/AndroidStudioProjects/MyFirstApp/app/build/intermediates/split-apk/debug/slices/slice_4.
apk /Users/dawng/AndroidStudioProjects/MyFirstApp/app/build/intermediates/split-apk/debug/slices/
slice_5.apk /Users/dawng/AndroidStudioProjects/MyFirstApp/app/build/intermediates/split-apk/debug/
slices/slice_7.apk /Users/dawng/AndroidStudioProjects/MyFirstApp/app/build/intermediates/split-apk/
debug/slices/slice_8.apk /Users/dawng/AndroidStudioProjects/MyFirstApp/app/build/intermediates/
split-apk/debug/slices/slice_9.apk /Users/dawng/AndroidStudioProjects/MyFirstApp/app/build/outputs/
apk/app-debug.apk
Split APKs installed
$ adb shell am startservice com.hfad.myfirstapp/com.android.tools.fd.runtime.InstantRunService
$ adb shell am start -n "com.hfad.myfirstapp/com.hfad.myfirstapp.MainActivity" -a android.intent.
action.MAIN -c android.intent.category.LAUNCHER
Connected to process 2685 on device Nexus_5X_API_25 [emulator-5554]
```

Install the app.

Android Studio has finished launching the AVD we just set up.

The emulator launches our app by starting the main activity for it. This is the activity the wizard created for us.



Test drive

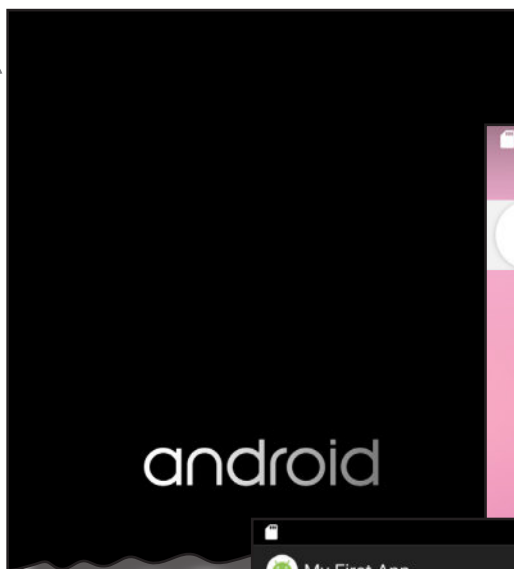
So let's look at what actually happens onscreen when you run your app.

First, the emulator fires up in a separate window. The emulator takes a while to load the AVD, but then you see what looks like an actual Android device.

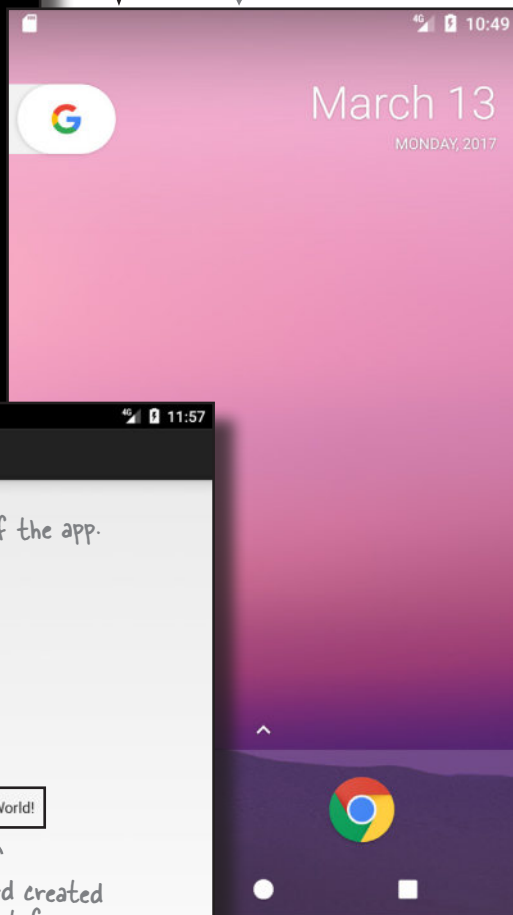


- getting started*
- Set up environment**
- Build app**
- Run app**
- Change app**

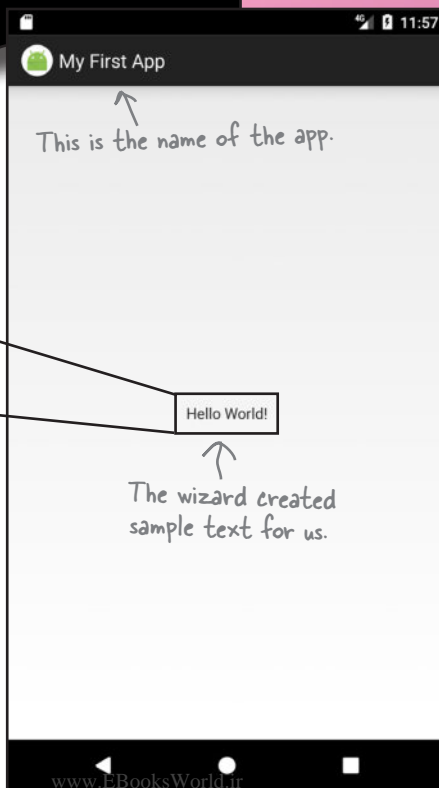
The emulator launches...



...and here's the AVD home screen. It looks and behaves just like a real Nexus 5X device.



Wait a bit longer, and you'll see the app you just created. The application name appears at the top of the screen, and the default sample text "Hello World!" is displayed in the middle of the screen.



Hello World!

Android Studio created the sample text "Hello World!" without us telling it to.

Hello World!

The wizard created sample text for us.

Here's the app running on the AVD.

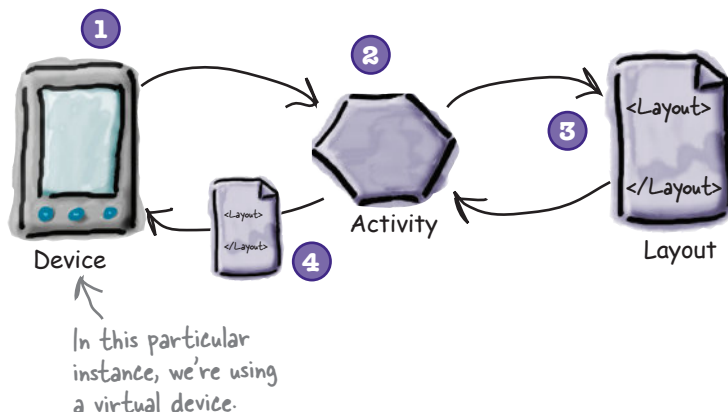
What just happened?

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



- Set up environment
- Build app
- Run app
- Change app

- 1 Android Studio launches the emulator, loads the AVD, and installs the app.
- 2 When the app gets launched, an activity object is created from MainActivity.java.
- 3 The activity specifies that it uses the layout activity_main.xml.
- 4 The activity tells Android to display the layout on the screen. The text "Hello World!" gets displayed.



there are no Dumb Questions

Q: You mentioned that when you create an APK file, the Java source code gets compiled into bytecode and added to the APK. Presumably you mean it gets compiled into Java bytecode, right?

A: It does, but that's not the end of the story. Things work a little differently on Android.

The big difference with Android is that your code doesn't actually run inside an ordinary Java VM. It runs on the Android runtime (ART) instead, and on older devices it runs in a predecessor to ART called Dalvik. This means that you write your Java source code and compile it into `.class` files using the Java compiler, and then the `.class` files get stitched into one or more files in DEX format, which is smaller, more efficient bytecode. ART then runs the DEX code. You can find more details about this in Appendix III.

Q: That sounds complicated. Why not just use the normal Java VM?

A: ART can convert the DEX bytecode into native code that can run directly on the CPU of the Android device. This makes the app run a lot faster, and use a lot less battery power.

Q: Is a Java virtual machine really that much overhead?

A: Yes. Because on Android, each app runs inside its own process. If it used ordinary JVMs, it would need a lot more memory.

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

A: No, once you've created the AVD you can use it for any of your apps. You may find it useful to create multiple AVDs in order to test your apps in different situations. As an example, in addition to a phone AVD you might want to create a tablet AVD so you can see how your app looks and behaves on larger devices.



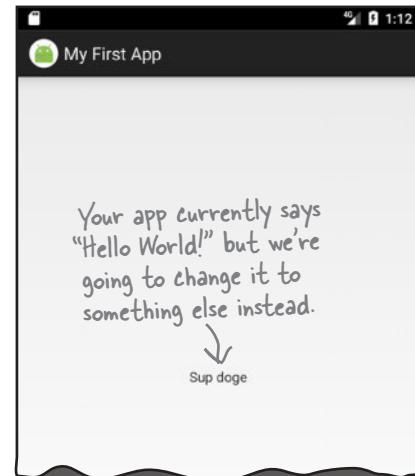
Refine the app

Over the past several pages, you've built a basic Android app and seen it running in the emulator. 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, let's take a step back and look at 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.

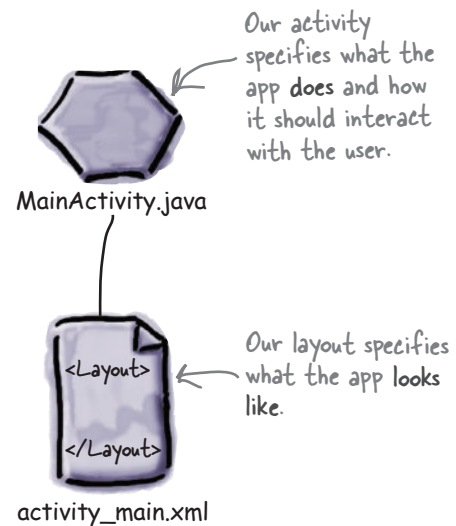


The activity controls what the app does

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

The layout controls the app's appearance

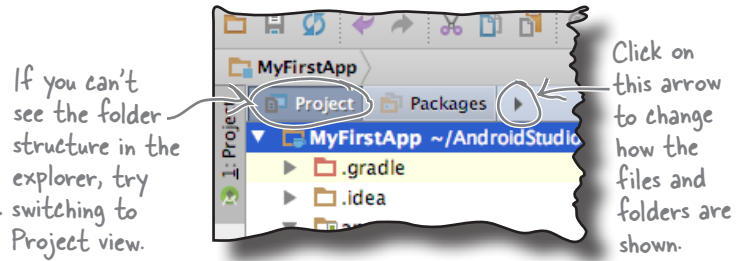
MainActivity.java specifies that it uses the layout Android Studio created for us called *activity_main.xml*. The layout specifies what the app **looks like**.



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

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*. If it isn’t already open in an editor, open it now by finding the file in the *app/src/main/res/layout* folder in the explorer and double-clicking on it.

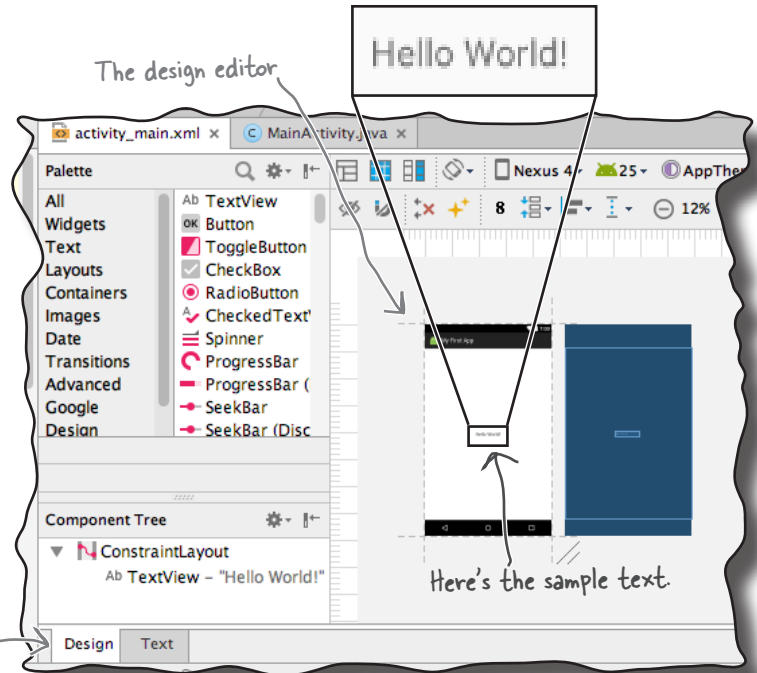


The design editor

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**.

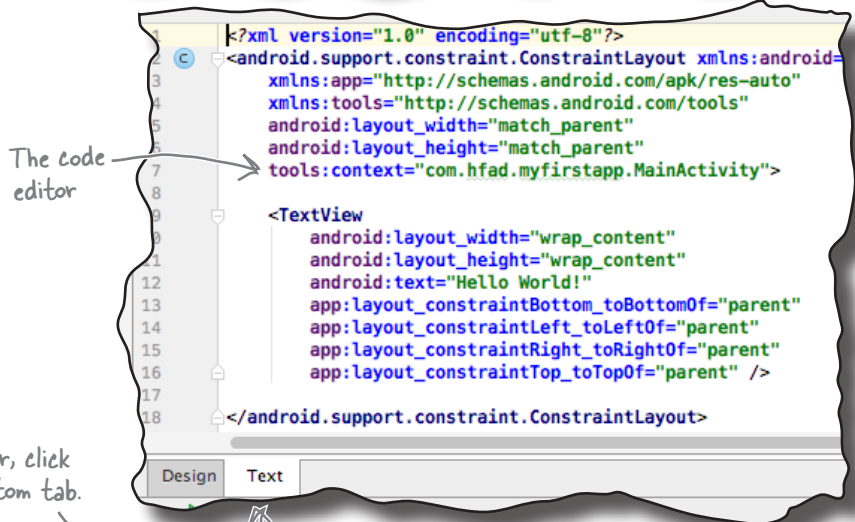
When you choose the design option, you can see that the sample text “Hello World!” appears in the layout as you might expect. But what’s in the underlying XML?

Let’s see by switching to the code editor.



The code editor

When you choose the code editor option, the content of *activity_main.xml* is displayed. Let’s take a closer look at it.



To see the code editor, click on “Text” in the bottom tab.

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 our code:

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

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

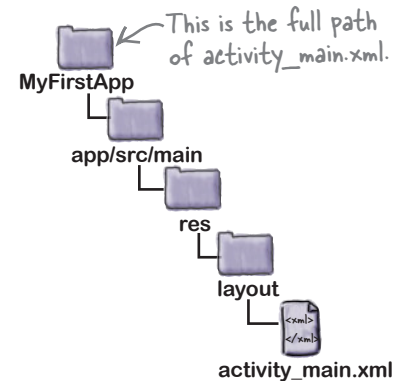
Android Studio gave us more XML here, but you don't need to think about that yet.

This is the `<TextView>` element.

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



Set up environment
Build app
Run app
Change app



As you can see, the code contains two elements.

The first is an `<android.support.constraint.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 element 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 our case the sample text "Hello World!"

The key part of the code within the `<TextView>` element is the line starting with `android:text`. This is a text property describing 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.

Let's change the text to something else.



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.

Update the text displayed in the layout

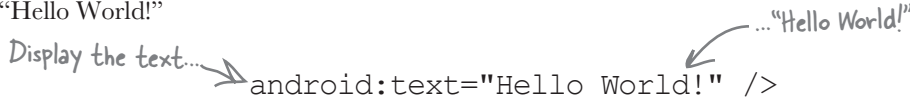


- Set up environment
- Build app
- Run app
- Change app

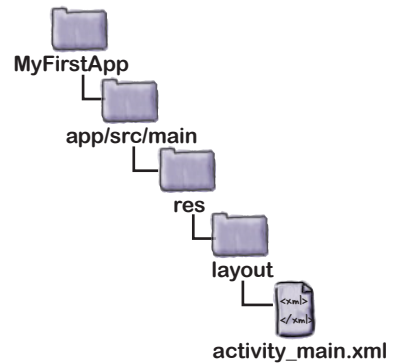
The key part of the <TextView> element is this line:

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

android:text means that this is the text property of the <TextView> element, so it specifies what text should be displayed in the layout. In this case, the text that's being displayed is "Hello World!"



To update the text that's displayed in the layout, simply change the value of the text property from "Hello World!" to "Sup doge". The new code for the <TextView> should look like this:



Once you've updated the file, go to the File menu and choose the Save All option to save your change.

there are no Dumb Questions

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

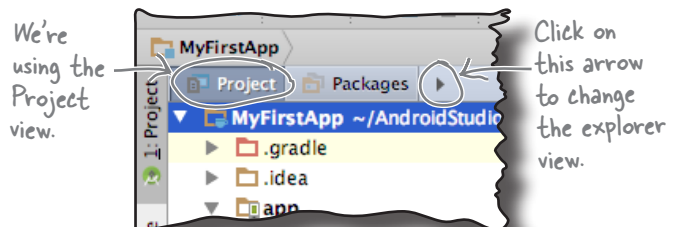
A: Yes, that's fine. Android Studio may generate slightly different code if you're using a different version than us, but that doesn't really matter. From now on you'll be learning how to create your own layout code, and you'll replace a lot of what Android Studio gives you.

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 your layouts, but you'll have to wait for the next chapter to learn what it is.

Q: The folders in my project explorer pane look different from yours. Why's 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 your explorer to the "Project" view by clicking on the arrow at the top of the explorer pane, and selecting the "Project" option.





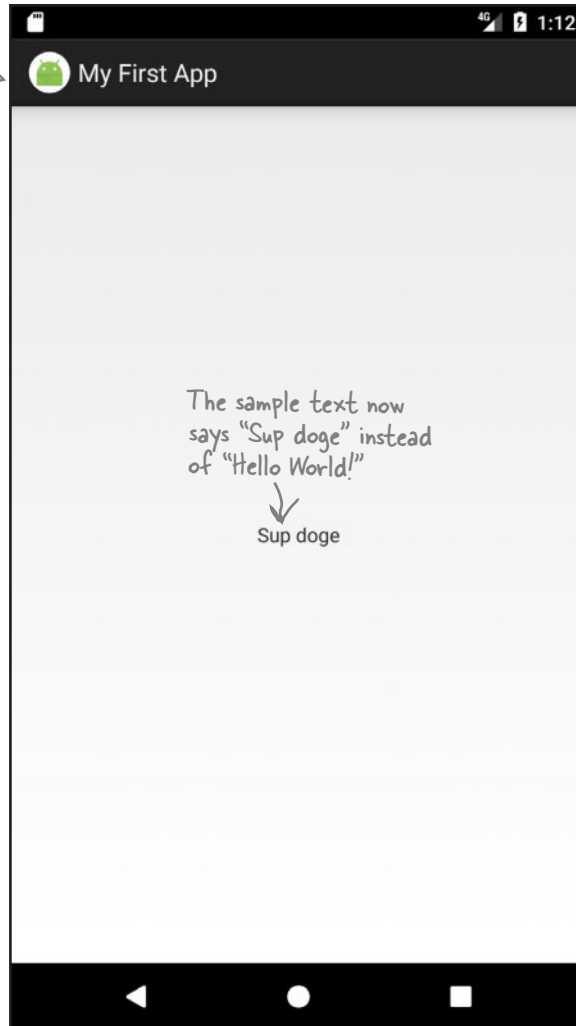
Take the app for a test drive

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



getting started
Set up environment
Build app
Run app
Change app

Here's the updated version of our app. →



You've now built and updated your first Android app.



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 <https://tinyurl.com/HeadFirstAndroid>.



BULLET POINTS

- Versions of Android have a version number, API level, and code name.
- Android Studio is a special version of IntelliJ IDEA that 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 your app looks like. They're held in the *app/src/main/res/layout* folder.
- Activities describe what your 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's like a JAR file for Android apps, and contains your 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.

2 building interactive apps

Apps That Do Something



I wonder what happens if I press the button marked "ejector seat"?

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

In this chapter, you'll see how you can make your apps **a bit more interactive**. You'll learn how to get your app to **do** something in response to the user, and **how to get your activity and layout talking to each other** like best buddies. Along the way, we'll take you a bit **deeper into how Android actually works** by introducing you to **R**, the hidden gem that glues everything together.

Let's build a Beer Adviser app

In Chapter 1, you saw how to create an app using the Android Studio New Project wizard, and how to change the text displayed in the layout. But when you create an Android app, you're usually going to want the app to *do* something.

In this chapter, we're going to show you how to create an app that the user can interact with: a Beer Adviser app. In the app, users can select the types of beer they enjoy, 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 specifies what the app will look like.

It includes three GUI components:

- A drop-down list of values called a spinner, which allows the user to choose which type of beer they want.
- A button that when pressed will return a selection of beer types.
- A text field that displays the types of beer.

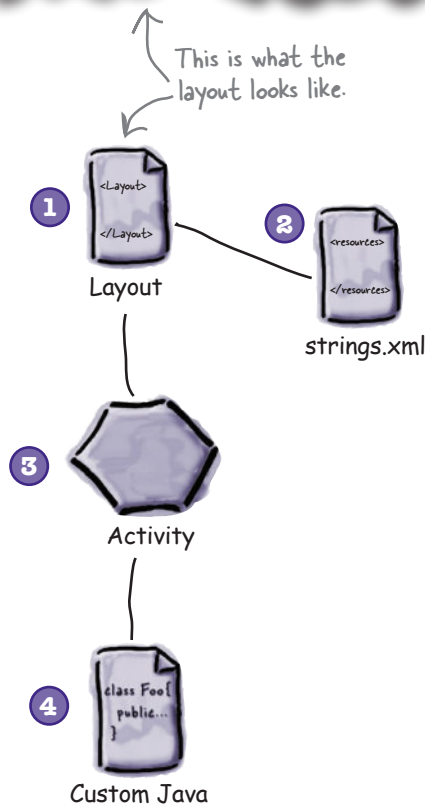
2 The file `strings.xml` includes any String resources needed by the layout—for example, the label of the button specified in the layout and the types of beer.

3 The activity specifies how the app should interact with the user.

It takes the type of beer the user chooses, and uses this to display a list of beers the user might be interested in. It achieves this with the help of a custom Java class.

4 The custom Java class contains the application logic for the app.

It includes a method that takes a type of beer as a parameter, and returns a list of beers of this type. The activity calls the method, passes it the type of beer, and uses the response.



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):

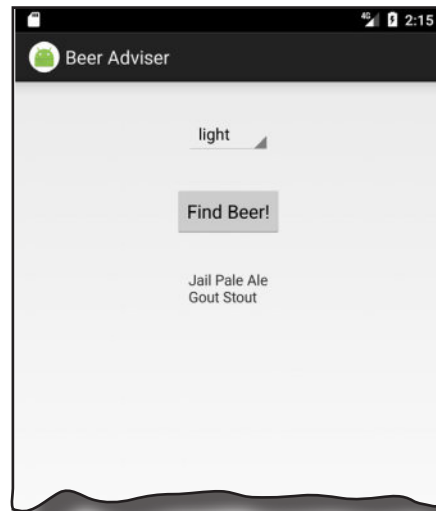
1 Create a project.

You're creating a brand-new app, so you'll need to create a new project. Just like before, you'll need to create an empty activity with a layout.

← We'll show you the details of how to do this on the next page.

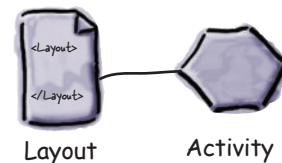
2 Update the layout.

Once you have the app set up, you need to amend the layout so that it includes all the GUI components your app needs.



3 Connect the layout to the activity.

The layout only creates the visuals. To add smarts to your app, you need to connect the layout to the Java code in your activity.



Layout

Activity

4 Write the application logic.

You'll add a Java custom class to the app, and use it to make sure users get the right beer based on their selection.



Layout

Create the project



- Create project**
- Update layout**
- Connect activity**
- Write logic**

Let's begin by creating the new app (the steps are similar to those we used in the previous chapter):

- 1** Open Android Studio and choose "Start a new Android Studio project" from the welcome screen. This starts the wizard you saw in Chapter 1.
- 2** When prompted, enter an application name of "Beer Adviser" and a company domain of "hfad.com", making your package name `com.hfad.beeradviser`. Make sure you uncheck the option to include C++ support.
- 3** We want the app to work on most phones and tablets, so choose a minimum SDK of API 19, and make sure the option for "Phone and Tablet" is selected. This means that any phone or tablet that runs the app must have API 19 installed on it as a minimum. Most Android devices meet this criterion.
- 4** Choose an empty activity for your default activity. Call the activity "FindBeerActivity" and the accompanying layout "activity_find_beer". Make sure the option to generate the layout is selected and you uncheck the Backwards Compatibility (AppCompat) option.

If your version of Android Studio has an option to include Kotlin support, uncheck this option too.

2 Application name: Beer Adviser
 Company domain: hfad.com
 Package name: com.hfad.beeradviser
 Include C++ support

The wizard will take you through these steps, just like before. Call your application "Beer Adviser," make sure it uses a minimum SDK of API 19, and then tell it to create an empty activity called "FindBeerActivity" and a layout called "activity_find_beer".

3 Phone and Tablet
 Minimum SDK: API 19: Android 4.4 (KitKat)
 Lower API levels target more devices, but have fewer features available. By targeting API 19 and later, your app will run on approximately 73.9% of the devices that are active on the Google Play Store.
[Help me choose](#)

Make sure you choose the Empty Activity option.

4 Activity Name: FindBeerActivity
 Generate Layout File
 Layout Name: activity_find_beer
 Backwards Compatibility (AppCompat)

Make sure you UNCHECK the Backwards Compatibility (AppCompat) option.

We've created a default activity and layout

When you click on the Finish button, Android Studio creates a new project containing an activity called *FindBeerActivity.java* and a layout called *activity_find_beer.xml*.

Let's start by changing the layout file. To do this, switch to the Project view of Android Studio's explorer, go to the *app/src/main/res/layout* folder, and open the file *activity_find_beer.xml*. Then switch to the text version of the code to open the code editor, and replace the code in *activity_find_beer.xml* with the following (we've bolded all the new code):

```
<?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="com.hfad.beeradviser.FindBeerActivity">
```

```
<TextView ← This is used to display text.
```

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

```
  android:layout_width="wrap_content"
```

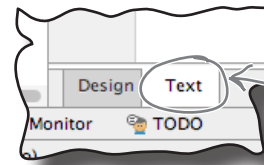
```
  android:layout_height="wrap_content"
```

```
  android:text="This is a text view" />
```

```
</LinearLayout>
```

We've just changed the code Android Studio gave us so that it uses a `<LinearLayout>`. This is used to display GUI components next to each other, either vertically or horizontally. If it's vertically, they're displayed in a single column, and if it's horizontally, they're displayed in a single row. You'll find out more about how this works as we go through the chapter.

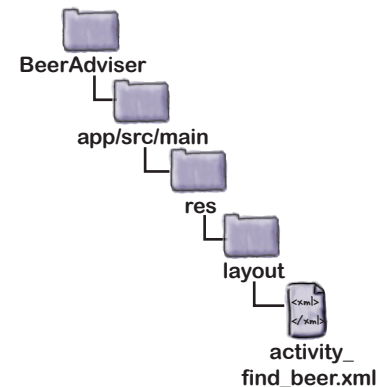
Any changes you make to a layout's XML are reflected in Android Studio's design editor, which you can see by clicking on the Design tab. We'll look at this in more detail on the next page.



Click on the Text tab to open the code editor.

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

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.

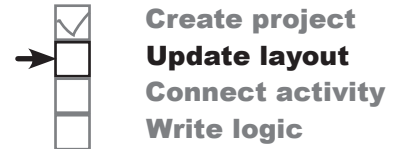


Click on the Design tab to open the design editor.



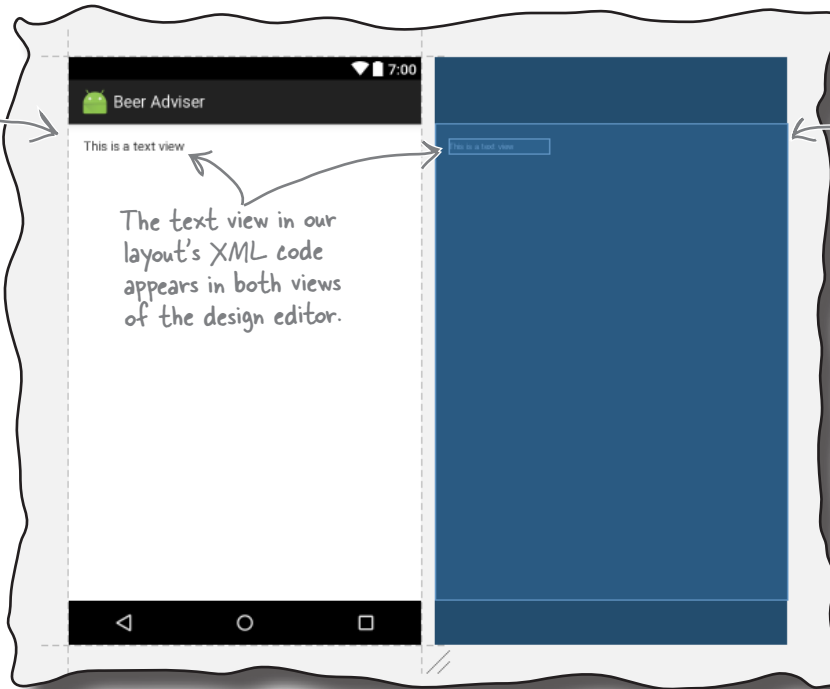
A closer look at the design editor

The design editor presents you with a more visual way of editing your 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 the "Show Design + Blueprint" icon in the design editor's toolbar.

This view of the design gives you an idea of how your layout will look on an actual device.



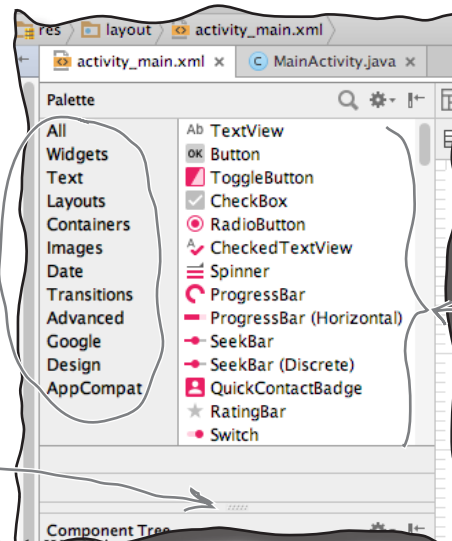
The text view in our layout's XML code appears in both views of the design editor.

This is the blueprint view, which focuses more on the layout's structure.

To the left of the design editor, there's a palette that contains components you can drag to your layout. We'll use this next.

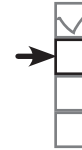
This list shows you the different categories of component you can add to your layout. You can click on them to filter the components displayed in the palette.

You can increase the size of the palette by clicking on this area and dragging it downward.



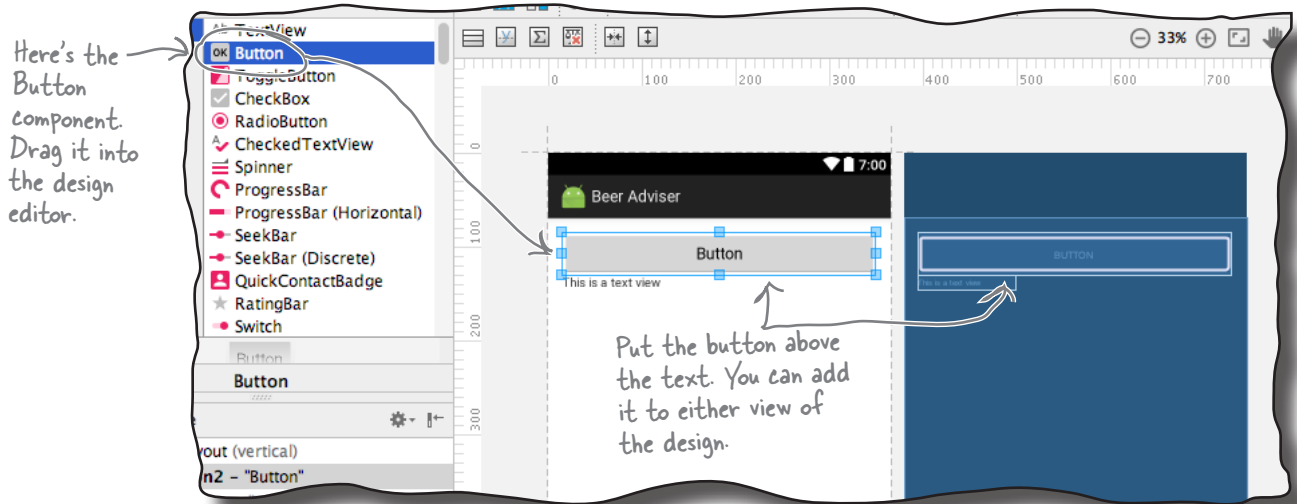
These are the components you'll find out more about them later in the book.

- ✓ Create project
- Update layout
- Connect activity
- Write logic



Add a button using the design editor

We're going to add a button to our layout using the design editor. Find the Button component in the palette, click on it, and then drag it into the design editor so that it's positioned above the text view. The button appears in the layout's design:



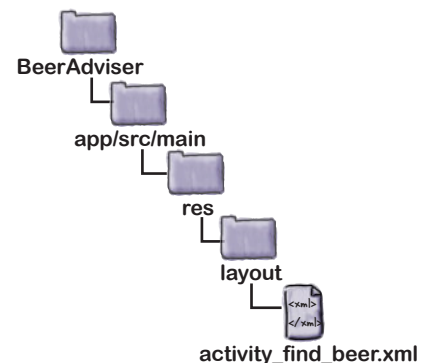
Changes in the design editor are reflected in the XML

Dragging GUI 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 lines of code to the file:

```

...
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/textView"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="This is a text view" />
...
    
```

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



activity_find_beer.xml has a new button

The editor added a new `<Button>` element to `activity_find_beer.xml`:

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

A button in Androidville is a pushbutton that the user can press to trigger an action. The `<Button>` element includes properties controlling its size and appearance. These properties aren't unique to buttons—other GUI components including 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 properties in common—they both inherit from the same Android **View** class. You'll find out more about this later in the book, but for now, here are some of the more common properties.

android:id

This gives the component an identifying name. The `id` property enables you to control what components do via activity code:

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

android:layout_width, android:layout_height

These properties 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

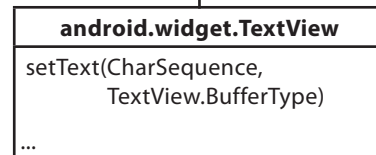
This tells Android what text the component should display. In the case of `<Button>`, it's the text that appears on the button:

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

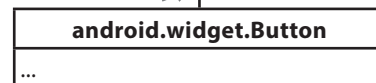
The View class includes lots of different methods. We'll look at this later in the book.

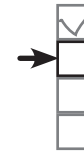


TextView is a type of View...



...and Button is a type of TextView, which means it's also a type of View.





- Create project
- Update layout
- Connect activity
- 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 with us):

The `<LinearLayout>` element

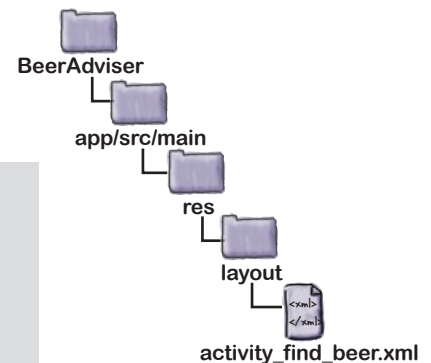
```
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:padding="16dp"
    android:orientation="vertical"
    tools:context="com.hfad.beeradviser.FindBeerActivity">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

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/textView"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="This is a text view" />
```



```
</LinearLayout>
```

← This closes the `<LinearLayout>` element.

The LinearLayout element

The first element in the layout code is `<LinearLayout>`. The `<LinearLayout>` element tells Android that the different GUI components in the layout should be displayed next to each other 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 GUI components are displayed in a single vertical column.

← There are other ways of laying out your GUI components too. You'll find out more about these later in the book.

A closer look at the layout code (continued)

The `<LinearLayout>` contains two 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" />
...
```

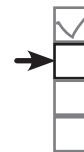
As this is the first element inside the `<LinearLayout>`, it appears first in the layout at the top of the screen. It has a `layout_width` of `"match_parent"`, which means that it should be as wide as its parent element, the `<LinearLayout>`. Its `layout_height` has been set to `"wrap_content"`, which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

The TextView element

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

```
...
<TextView
    android:id="@+id/textView"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="This is a text view" />
...
```

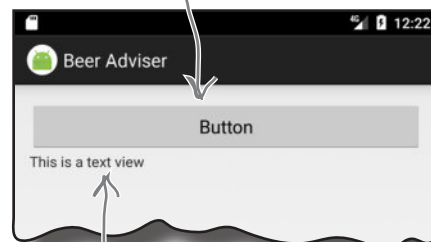
As this is the second element and we've set the linear layout's orientation to `"vertical"`, it's displayed underneath the button (the first element). Its `layout_width` and `layout_height` are set to `"wrap_content"` so that it takes up just enough space to contain its text.



Create project
Update layout
Connect activity
Write logic

Using a linear layout means that GUI components are displayed in a single row or column.

The button is displayed at the top as it's the first element in the XML.



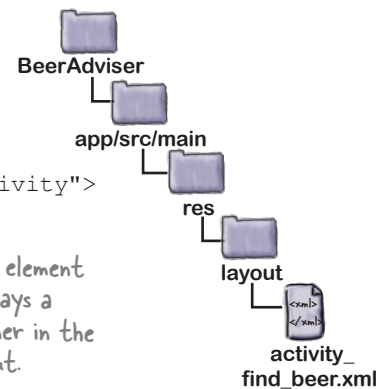
The text view is displayed underneath the button as it comes after it in the XML.

Changes to the XML...

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.

Try this now. Update your *activity_find_beer.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="com.hfad.beeradviser.FindBeerActivity">
```



A spinner is the Android name for a drop-down list of values. It allows you to choose a single value from a selection.

```
<Spinner
    android:id="@+id/color"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginTop="40dp"
    android:layout_gravity="center"
    android:layout_margin="16dp" />
```

This element displays a spinner in the layout.

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

Center the button horizontally and give it a margin.

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

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

```
<TextView
    android:id="@+id/textview_brands"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_gravity="center"
    android:layout_margin="16dp"
    android:text="This is a text view" />
</LinearLayout>
```

Center the text view and apply a margin.

Change the text view's ID to "brands".

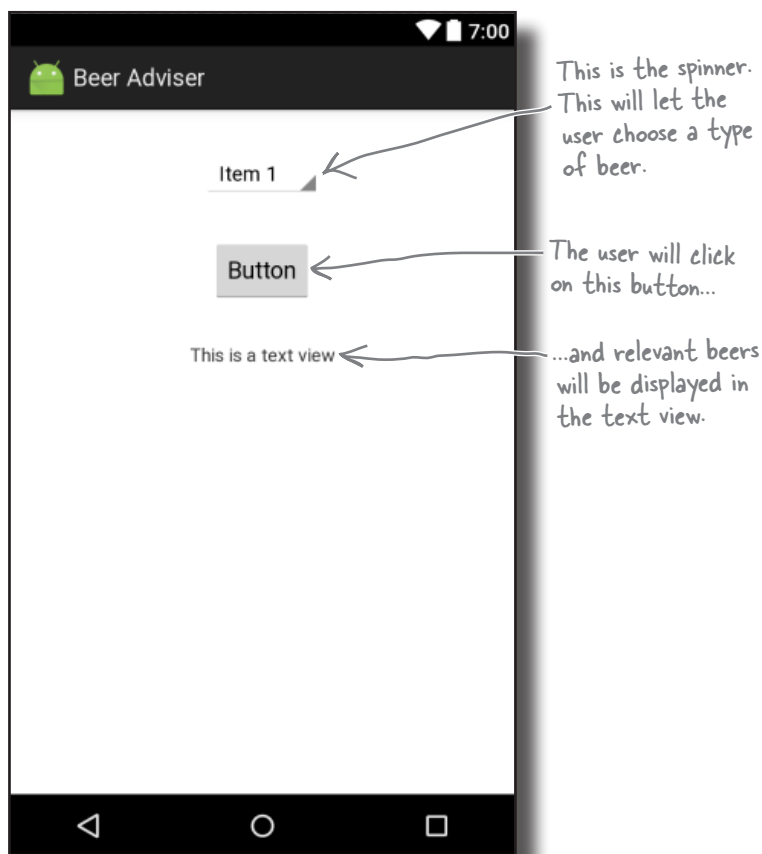
Do this!

Update the contents of *activity_find_beer.xml* with the changes shown here.

...are reflected in the design editor

Once you've changed the layout XML, switch to the design editor. Instead of a layout containing a button with a text view underneath it, you should now see a spinner, button, and text view centered in a single column.

A **spinner** is the Android term for a drop-down list of values. When you press it, it expands to show you the list so that you can pick a single value.



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

GUI components such as buttons, spinners, and text views have very similar attributes, as they are all types of View. Behind the scenes, they all inherit from the same Android View class.

We've shown you how to add GUI components to the layout with the aid of the design editor, and also by adding them through XML. In general, you're more likely to hack the XML for simple layouts to get the results you want without using the design editor. This is because editing the XML directly gives you more direct control over the layout.



Let's take the app for a test drive

We still have more work to do on the app, but let's see how it's looking so far. Save the changes you've made by choosing File→Save All, then choose the "Run 'app'" command from the Run menu. When prompted, select the option to launch the emulator.

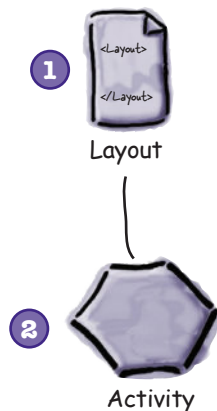
Wait patiently for the app to load, and eventually it should appear.

Try pressing the spinner. It's not immediately obvious, but when you press it, the spinner presents you with a drop-down list of values—it's just at this point we haven't added any values to it.

Here's what we've done so far

Here's a quick recap of what we've done so far:

- 1 **We've created a layout that specifies what the app looks like.**
It includes a spinner, a button, and a text view.
- 2 **The activity specifies how the app should interact with the user.**
Android Studio has created an activity for us, but we haven't done anything with it yet.

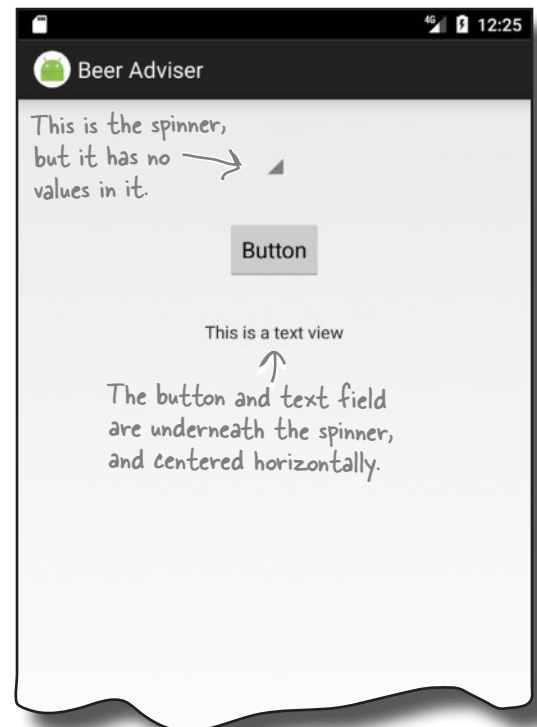


The next thing we'll do is look at replacing the hardcoded String values for the text view and button text.

building interactive apps



- Create project
- Update layout
- Connect activity
- Write logic



there are no Dumb Questions

Q: My layout looks slightly different in the AVD compared with how it looks in the design editor. Why's that?

A: The design editor does its best to show you how the layout will look on a device, but it's not always accurate depending on what version of Android Studio you're using. How the layout looks in the AVD reflects how the layout will look on a physical device.

Hardcoding text makes localization hard

So far, we've hardcoded the text we want to appear in our text views and buttons using the `android:text` property:

Display the text... → `android:text="Hello World!" />` ← ..."Hello World!"

While this is fine when you're just learning, hardcoding text 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 your app international will be difficult.

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

Put the text in a String resource file

A better approach is to put your text values into a String resource file called *strings.xml*.

Having a String resource file makes it much easier to internationalize your app. Rather than having to change hardcoded text values in a whole host of different activity and layout files, you can simply replace the *strings.xml* file with an internationalized version.

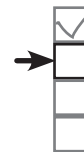
This approach also makes it much easier to make global changes to text across your whole application as you only need to edit one file. If you need to make changes to the text in your app, you only need to edit *strings.xml*.

How do you use String resources?

In order to use a String resource in your layout, there are two things you need to do:

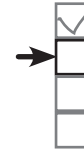
- 1 Create the String resource by adding it to *strings.xml*.
- 2 Use the String resource in your layout.

Let's see how this is done.



Create project
Update layout
Connect activity
Write logic

Put String values in *strings.xml* rather than hardcoding them. *strings.xml* is a resource file used to hold name/value pairs of Strings. Layouts and activities can look up String values using their names.



Create the String resource

We're going to create two String resources, one for the text that appears on the button, and another for the default text that appears in the text view.

To do this, use Android Studio's explorer to find the file *strings.xml* in the *app/src/main/res/values* folder. Then open it by double-clicking on it.

The file should look something like this:

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

strings.xml contains one string resource named "app_name", which has a value of Beer Adviser. Android Studio created this String resource for us automatically when we created the project.

This indicates that this is a String resource.

```
<string name="app_name">Beer Adviser</string>
```

This String resource has a name of "app_name", and a value of "Beer Adviser".

We're first going to add a new resource called "find_beer" that has a value of Find Beer! To do this, edit *strings.xml* so that you add it as a new line like this:

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

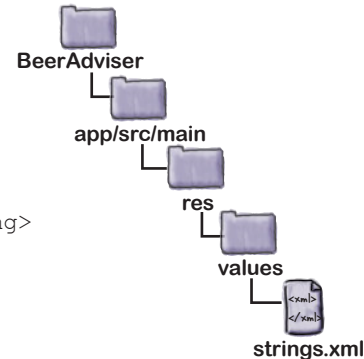
This adds a new String resource called "find_beer".

Then add a new resource named "brands" with a value of No beers selected:

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

This will be the default text in the text view.

Once you've updated the file, go to the File menu and choose the Save All option to save your changes. Next, we'll use the String resources in our layout.



Use the String resource in your layout

You use String resources in your layout using code like this:

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

You've seen the `android:text` part of the code before; it specifies what text should be displayed. But what does `"@string/find_beer"` 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. In our case, this is the file `strings.xml` that you just edited.

The second part, `find_beer`, tells Android to **look up the value of a resource with the name `find_beer`**. So `"@string/find_beer"` means "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`.

We want to change the button and text view elements in our layout XML so that they use the two String resources we've just added.

Go back to the layout file `activity_find_beer.xml` file, and make the following code changes:



Change the line:

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

to:

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



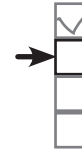
Change the line:

```
android:text="TextView"
```

to:

```
android:text="@string/brands"
```

You can see the code on the next page.



Create project
Update layout
Connect activity
Write logic



Watch it!

Android Studio sometimes displays the values of

references in the code editor in place of actual code.

As an example, it may display the text "Find Beer!" instead of the real code "@string/find_beer". Any such substitutions should be highlighted in the code editor. If you click on them, or hover over them with your mouse, the true code will be revealed.

```
<TextView
    android:text="Hello world!"
    android:text="@string/hello_world"
    android:layout_height= wrap_content />
```

The code for activity_find_beer.xml

Here's the updated code for `activity_find_beer.xml` (changes are in bold); update your version of the file to match ours.

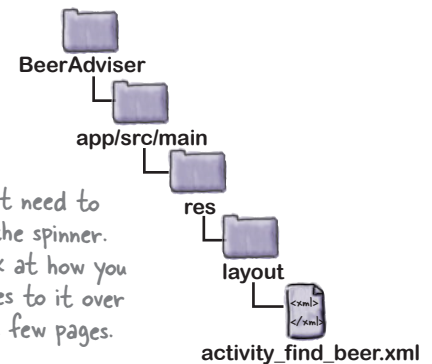
...

```
<Spinner
    android:id="@+id/color"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginTop="40dp"
    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="Button@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="This is a text view@string/brands" />
```

```
</LinearLayout>
```



We didn't need to change the spinner. We'll look at how you add values to it over the next few pages.

Delete the hardcoded text.

This will display the value of the `find_beer` String resource on the button.

Delete this hardcoded text too.

This will display the value of the `brands` String resource in the text view.

When you're done, save your changes.

We've put a summary of adding and using String resources on the next page.



String Resource Files Up Close

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

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>
```

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

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 simple values, such as Strings 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 Strings. 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 your String resource file *strings.xml*; if you want, you can call it something else, or split your Strings 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.



Time for a test drive

Let's see how the app's looking now. Save the changes you've made, then choose the "Run 'app'" command from the Run menu. When prompted, select the option to launch the emulator.

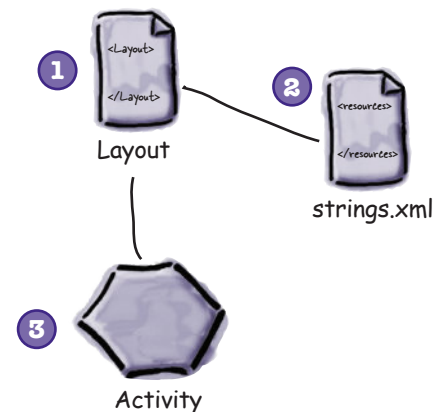
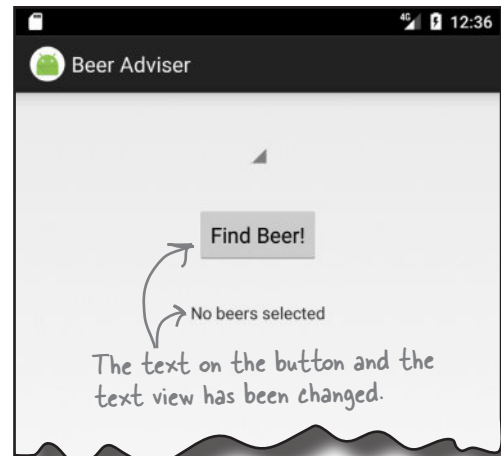
This time when we run the app, the text for the button and the text view has changed to the String values we added to *strings.xml*. The button says "Find Beer!" and the text view says "No beers selected."

Here's what we've done so far

Here's a quick recap of where we've got to:

- 1 **We've created a layout that specifies what the app looks like.**
It includes a spinner, a button, and a text view.
- 2 **The file *strings.xml* includes the String resources we need.**
We've added a label for the button, and default text for the list of suggested beer brands to try.
- 3 **The activity specifies how the app should interact with the user.**
Android Studio has created an activity for us, but we haven't done anything with it yet.

Next we'll look at how you add a list of beers to the spinner.



there are no Dumb Questions

Q: Do I absolutely have to put my text values in a String resource file such as *strings.xml*?

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 your application to be in English by default, but in French if the device language is set to French. Rather than hardcode different languages into your 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 your default English Strings resource file in the *app/src/main/res/values* folder as normal, and your French resource file in a new folder called *app/src/main/res/values-fr*. If the device is set to French, it will use the Strings in the *app/src/main/res/values-fr* folder. If the device is set to any other language, it will use the Strings in *app/src/main/res/values*.

Add values to the spinner

At the moment, the layout includes a spinner, but it doesn't have anything in it. Whenever you use a spinner, you need to get it to display a list of values so that the user can choose the value they want.

We can give the spinner a list of values in pretty much the same way that we set the text on the button and the text view: by using a **resource**. So far, we've used *strings.xml* to specify individual String values. For the spinner, all we need to do is specify an *array* of String values, and get the spinner to reference it.

Adding an array resource is similar to adding a String

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

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

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

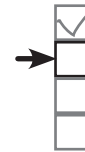
To add an array of Strings, you use the following syntax:

```
<string-array name="string_array_name"> ← This is the name of the array.
  <item>string_value1</item>
  <item>string_value2</item>
  <item>string_value3</item>
  ...
</string-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*, *string_value3* are the individual String values that make up the array.

Let's add a *string-array* resource to our app that can be used by the spinner.



Create project
Update layout
Connect activity
Write logic

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

Add the string-array to strings.xml

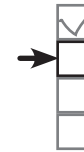
To add the string-array, open up *strings.xml*, and add the array like this:

```

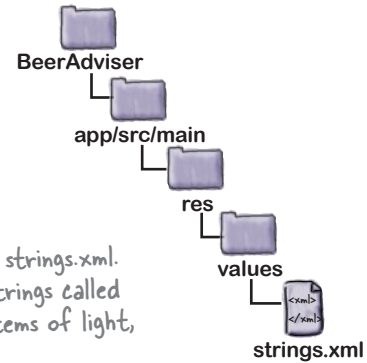
...
<string name="brands">No beer selected </string>
<string-array name="beer_colors">
  <item>light</item>
  <item>amber</item>
  <item>brown</item>
  <item>dark</item>
</string-array>
</resources>

```

Add this string-array to strings.xml. It defines an array of Strings called beer_colors with array items of light, amber, brown, and dark.



- Create project
- Update layout
- Connect activity
- Write logic



Get the spinner to reference a string-array

A layout can reference a string-array using similar syntax to how it would retrieve the value of a String. Rather than use:

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

you use the syntax:

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

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

where array_name is the name of the array.

Let's use this in the layout. Go to the layout file *activity_find_beer.xml* and add an entries attribute to the spinner like this:

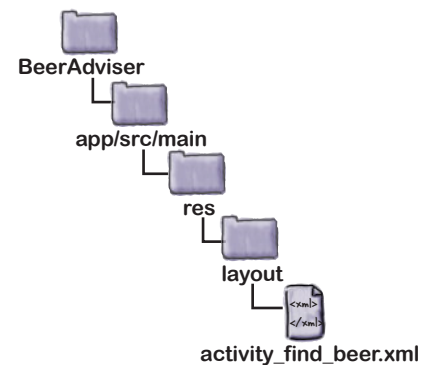
```

...
<Spinner
  android:id="@+id/color"
  android:layout_width="wrap_content"
  android:layout_height="wrap_content"
  android:layout_marginTop="40dp"
  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".

Those are all the changes you need in order to get the spinner to display a list of values. Let's see what it looks like.



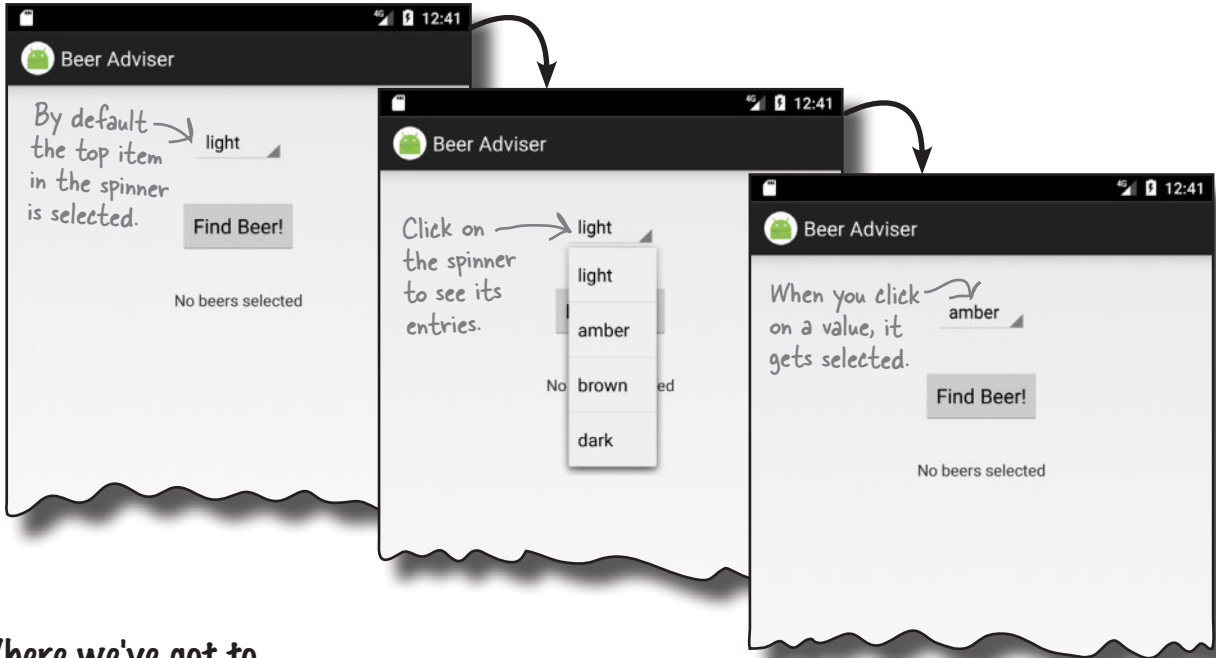


Test drive the spinner

So let's see what impact these changes have had on our app. Save your changes, then run the app. You should get something like this:



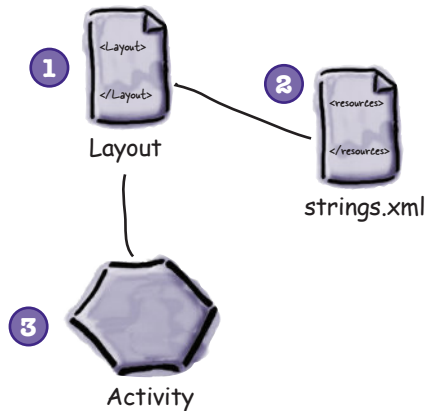
- Create project**
- Update layout**
- Connect activity**
- Write logic**



Where we've got to

Here's a reminder of what we've done so far:

- 1 We've created a layout that specifies what the app looks like.**
It includes a spinner, a button, and a text view.
- 2 The file strings.xml includes the String resources we need.**
We've added a label for the button, default text for the suggested beer brands, and an array of values for the spinner.
- 3 The activity specifies how the app should interact with the user.**
Android Studio has created an activity for us, but we haven't done anything with it yet.



So what's next?

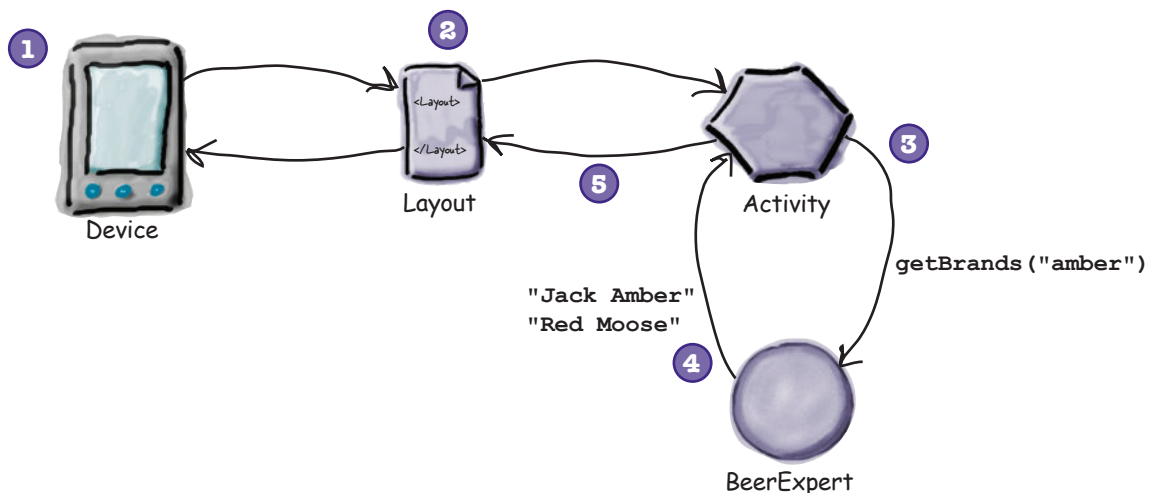
- ✓ Create project
- ✓ Update layout
- Connect activity
- Write logic

We need to make the button do something

What we need to do next is make the app react to the value we select in the spinner when the Find Beer button is clicked. We want our app to behave something like this:

- 1 The user chooses a type of beer from the spinner.
- 2 The user clicks the Find Beer button, and the layout specifies which method to call in the activity.
- 3 The method in the activity retrieves the value of the selected beer in the spinner and passes it to the `getBrands()` method in a Java custom class called `BeerExpert`.
- 4 `BeerExpert`'s `getBrands()` method finds matching brands for the type of beer and returns them to the activity as an `ArrayList` of `Strings`.
- 5 The activity gets a reference to the layout text view and sets its text value to the list of matching beers.

After all those steps are completed, the list is displayed on the device.



Let's start by getting the button to call a method.

Make the button call a method

Whenever you add a button to a layout, it's likely you'll want it to do something when the user clicks on it. To make this happen, you need to get the button to call a method in your activity.

To get our button to call a method in the activity when it's clicked, we need to make changes to two files:

- ★ **Change the layout file `activity_find_beer.xml`.**
We'll specify which method in the activity will get called when the button is clicked.
- ★ **Change the activity file `FindBeerActivity.java`.**
We need to write the method that gets called.

Let's start with the layout.

Use `onClick` to say which method the button calls

It only takes one line of XML to tell Android which method a button should call when it's clicked. All you need to do is add an `android:onClick` attribute to the `<button>` element, and tell it the name of the method you want to call:

`android:onClick="method_name"`

← This means "when the component is clicked, call the method in the activity called `method_name`".

Let's try this now. Go to the layout file `activity_find_beer.xml`, and add a new line of XML to the `<button>` element to say that the method `onClickFindBeer()` should be called when the button is clicked:

```
...
<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"
    android:onClick="onClickFindBeer" />
...

```

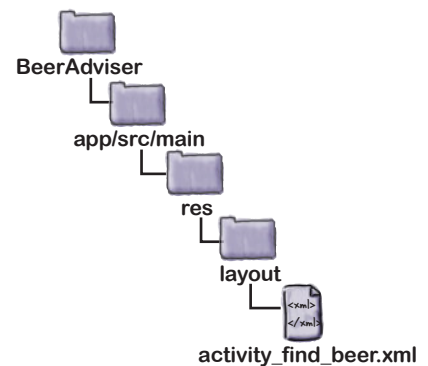
Once you've made these changes, save the file.

Now that the layout knows which method to call in the activity, we need to write the method. Let's take a look at the activity.

← When the button is clicked, call the method `onClickFindBeer()` in the activity. We'll create the method in the activity over the next few pages.



- Create project
- Update layout
- Connect activity
- Write logic



What activity code looks like

When we first created a project for our app, we asked the wizard to create an empty activity called `FindBeerActivity`. The code for this activity is held in a file called `FindBeerActivity.java`. Open this file by going to the `app/src/main/java` folder and double-clicking on it.

When you open the file, you'll see that Android Studio has generated some Java code for you. Rather than taking you through all the code that Android Studio may (or may not) have created, we want you to replace the code that's currently in `FindBeerActivity.java` with the code shown here:

```
package com.hfad.beeradviser;
```

```
import android.app.Activity;
import android.os.Bundle;
```

```
public class FindBeerActivity extends Activity {
```

```
    @Override
```

```
    protected void onCreate(Bundle savedInstanceState) {
```

```
        super.onCreate(savedInstanceState);
```

```
        setContentView(R.layout.activity_find_beer);
```

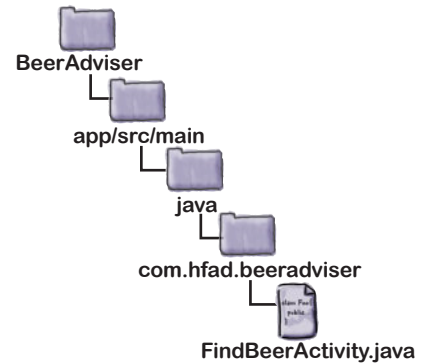
```
    }
```

```
}
```

Make sure class extends the Android Activity class.

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_find_beer`.



The above code is all you need to create a basic activity. As you can see, it's a class that extends the `android.app.Activity` class, and implements an `onCreate()` method.

All activities (not just this one) have to extend the `Activity` class or one of its subclasses. The `Activity` class contains a bunch of methods that transform your Java class from a plain old Java class into a full-fledged, card-carrying Android activity.

All activities also need to 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 the `setContentView()` method. In the example above, `setContentView(R.layout.activity_find_beer)` tells Android that this activity uses `activity_find_beer` as its layout.

On the previous page, we added an `onClick` attribute to the button in our layout and gave it a value of `onClickFindBeer`. We need to add this method to our activity so it will be called when the button gets clicked. This will enable the activity to respond when the user touches the button in the user interface.

Do this!

Replace the code in your version of `FindBeerActivity.java` with the code shown on this page.

Add an onClickFindBeer() method to the activity



- Create project
- Update layout
- Connect activity
- Write logic

The onClickFindBeer () method needs to have a particular signature, or otherwise it won't get called when the button specified in the layout gets clicked. The method needs to take the following form:

```
public void onClickFindBeer(View view) {
}
```

The method must be public.

The method must have a void return value.

The method must have a single parameter of type View.

If the method doesn't take this form, then it won't respond when the user presses the button. This is because behind the scenes, Android looks for a public method with a void return value, with a method name that matches the method specified in the layout XML.

The View parameter in the method may seem unusual at first glance, but there's a good reason for it being there. The parameter refers to the GUI component that triggers the method (in this case, the button). As we mentioned earlier, GUI components such as buttons and text views are all types of View.

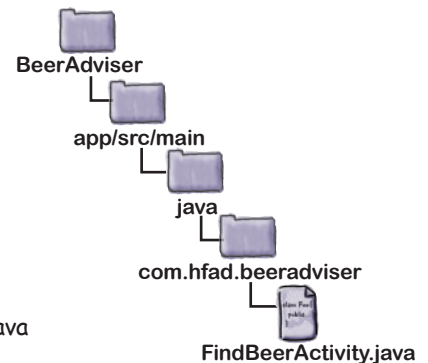
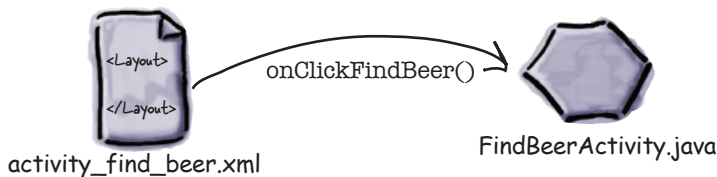
So let's update our activity code. Add the onClickFindBeer () method below to your activity code (FindBeerActivity.java):

If you want a method to respond to a button click, it must be public, have a void return type, and take a single View parameter.

We're using this class, so we need to import it. ... `import android.view.View;`

Add the onClickFindBeer() method to FindBeerActivity.java.

```
public class FindBeerActivity extends Activity {
...
//Called when the user clicks the button
public void onClickFindBeer(View view) {
}
}
```





onClickFindBeer() needs to do something

Now that we've created the `onClickFindBeer()` method in our activity, the next thing we need to do is get the method to do something when it runs. Specifically, we need to get our app to display a selection of different beers that match the beer type the user has selected.

In order to achieve this, we first need to get a reference to both the spinner and text view GUI components in the layout. This will allow us to retrieve the value of the chosen beer type from the spinner, and display text in the text view.

Use `findViewById()` to get a reference to a view

We can get references for our two GUI components using a method called `findViewById()`. This method takes the ID of the GUI component as a parameter, and returns a `View` object. You then cast the return value to the correct type of GUI component (for example, a `TextView` or a `Button`).

Here's how you'd use `findViewById()` to get a reference to the text view with an ID of `brands`:

```
TextView brands = (TextView) findViewById(R.id.brands);
```

brands is a `TextView`, so we have to cast it as one.

We want the view with an ID of `brands`.



Take a closer look at how we specified the ID of the text view. Rather than pass in the name of the text view, we passed in an ID of the form `R.id.brands`. So what does this mean? What's `R`?

`R.java` is a special Java file that gets generated by Android Studio whenever you create or build your app. It lives within the `app/build/generated/source/r/debug` folder in your project in a package with the same name as the package of your app. Android uses `R.java` to keep track of the resources used within the app, and among other things it enables you to get references to GUI components from within your activity code.

If you open up `R.java`, you'll see that it contains a series of inner classes, one for each type of resource. Each resource of that type is referenced within the inner class. As an example, `R.java` includes an inner class called `id`, and the inner class includes a `static final` `brands` value. Android added this code to `R.java` when we used the code `"@+id/brands"` in our layout. The line of code:

```
(TextView) findViewById(R.id.brands);
```

uses the value of `brands` to get a reference to the `brands` text view.

R is a special Java class that enables you to retrieve references to resources in your app.



`R.java` gets generated for you.

You never change any of the code within this file, but it's useful to know it's there.

Once you have a view, you can access its methods



Create project
Update layout
Connect activity
Write logic

The `findViewById()` method provides you with a Java version of your GUI component. This means that you can get and set properties in the GUI component using the methods exposed by the Java class. Let's take a closer look.

Setting the text in a text view

As you've seen, you can get a reference to a text view in Java using:

```
TextView brands = (TextView) findViewById(R.id.brands);
```

When this line of code gets called, it creates a `TextView` object called `brands`. You are then able to call methods on this `TextView` object.

Let's say you wanted to set the text displayed in the `brands` text view to "Gottle of geer". The `TextView` class includes a method called `setText()` that you can use to change the text property. You use it like this:

```
brands.setText("Gottle of geer");
```

← Set the text on the brands TextView to "Gottle of geer".

Retrieving the selected value in a spinner

You can get a reference to a spinner in a similar way to how you get a reference to a text view. You use the `findViewById()` method as before, but this time you cast the result as a spinner:

```
Spinner color = (Spinner) findViewById(R.id.color);
```

This gives you a `Spinner` object whose methods you can now access. As an example, here's how you retrieve the currently selected item in the spinner, and convert it to a `String`:

```
String.valueOf(color.getSelectedItem())
```

← This gets the selected item in the spinner and converts it to a String.

The code:

```
color.getSelectedItem()
```

actually returns a generic Java object. This is because spinner values can be something other than Strings, such as images. In our case, we know the values are all Strings, so we can use `String.valueOf()` to convert the selected item from an `Object` to a `String`.

Update the activity code

You now know enough to write some code in the `onClickFindBeer()` method. Rather than write all the code we need in one go, let's start by reading the selected value from the spinner, and displaying it in the text view.



Activity Magnets

Somebody used fridge magnets to write a new `onClickFindBeer()` method for us to slot into our activity. Unfortunately, a freak kitchen whirlwind has dislodged the magnets. Can you piece the code back together again?

The code needs to retrieve the type of beer selected in the spinner, and then display the type of beer in the text view.

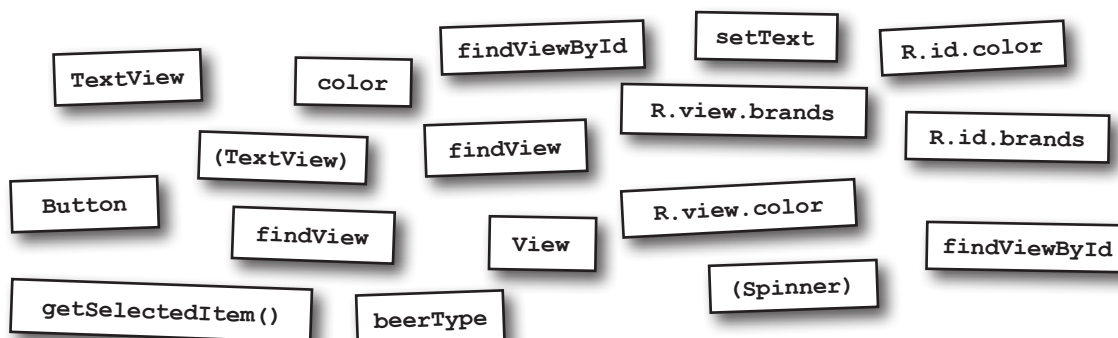
```
//Called when the button gets clicked
public void onClickFindBeer(.....view) {

    //Get a reference to the TextView
    ..... brands = ..... ( ..... );

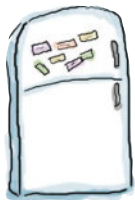
    //Get a reference to the Spinner
    Spinner ..... = ..... ( ..... );

    //Get the selected item in the Spinner
    String ..... = String.valueOf(color. ....);

    //Display the selected item
    brands.....(beerType);
}
```



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



Activity Magnets Solution

Somebody used fridge magnets to write a new `onClickFindBeer()` method for us to slot into our activity. Unfortunately, a freak kitchen whirlwind has dislodged the magnets. Can you piece the code back together again?

The code needs to retrieve the type of beer selected in the spinner, and then display the type of beer in the text view.

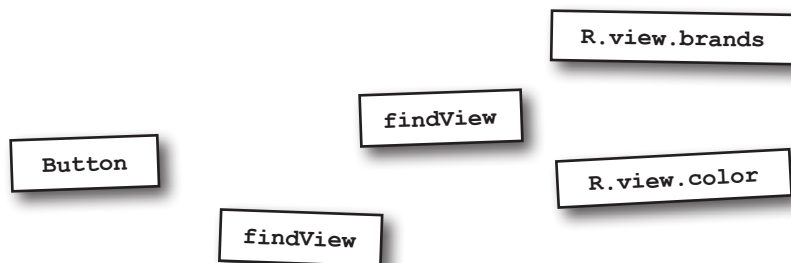
```
//Called when the button gets clicked
public void onClickFindBeer( ... View .....view) {

    //Get a reference to the TextView
    TextView brands = (TextView) ... findViewById ( R.id.brands );

    //Get a reference to the Spinner
    Spinner color ... = (Spinner) ... findViewById ( R.id.color );

    //Get the selected item in the Spinner
    String beerType ... = String.valueOf(color. ... getSelectedItem() ... );

    //Display the selected item
    brands. setText (beerType);
}
```



You didn't need to use these magnets.

Create project
Update layout
Connect activity
Write logic



The first version of the activity

Our cunning plan is to build the activity in stages and test it as we go along. In the end, the activity will take the selected value from the spinner, call a method in a custom Java class, and then display matching types of beer. For this first version, our goal is just to make sure that we correctly retrieve the selected item from the spinner.

Here is our activity code, including the method you pieced together on the previous page. Apply these changes to *FindBeerActivity.java*, then save them:

```
package com.hfad.beeradviser;

import android.app.Activity;
import android.os.Bundle;
import android.view.View;
import android.widget.Spinner;
import android.widget.TextView;

public class FindBeerActivity extends Activity {
```

```
@Override
```

```
protected void onCreate(Bundle savedInstanceState) { ← We've not changed this method.
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_find_beer);
}
```

```
//Called when the button gets clicked
```

```
public void onClickFindBeer(View view) {
```

```
    //Get a reference to the TextView
```

```
    TextView brands = (TextView) findViewById(R.id.brands);
```

```
    //Get a reference to the Spinner
```

```
    Spinner color = (Spinner) findViewById(R.id.color);
```

```
    //Get the selected item in the Spinner
```

```
    String beerType = String.valueOf(color.getSelectedItem());
```

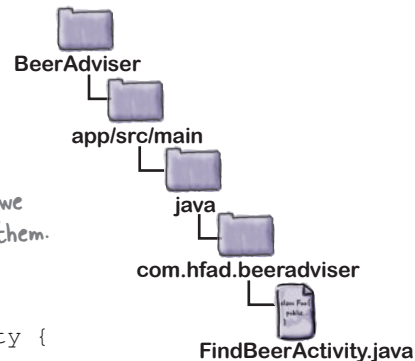
```
    //Display the selected item
```

```
    brands.setText(beerType);
```

```
}
```

```
}
```

We're using these extra classes so we need to import them.



findViewById returns a View. You need to cast it to the right type of View.

getSelectedItem returns an Object. You need to turn it into a String.

What the code does

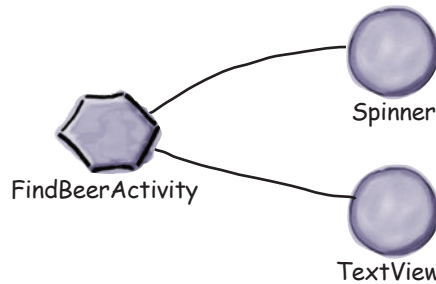


Before we take the app for a test drive, let's look at what the code actually does.

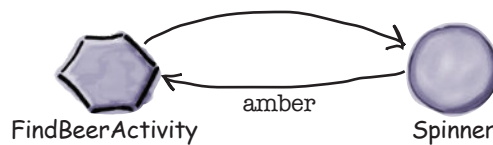
- 1 **The user chooses a type of beer from the spinner and clicks on the Find Beer button. This calls the public void `onClickFindBeer(View)` method in the activity.**
The layout specifies which method in the activity should be called when the button is clicked via the button's `android:onClick` property.



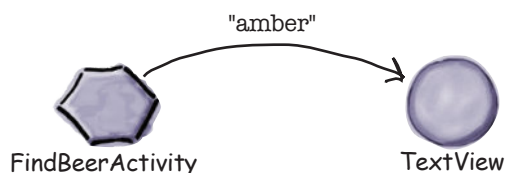
- 2 **The activity gets references to the Spinner and TextView GUI components using calls to the `findViewById()` method.**



- 3 **The activity retrieves the currently selected value of the spinner (in this case amber), and converts it to a `String`.**



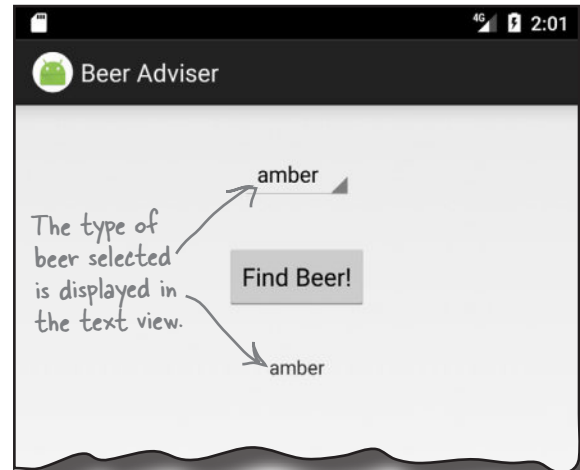
- 4 **The activity then sets the text property of the TextView to reflect the currently selected item in the spinner.**





Test drive the changes

Make the changes to the activity file, save it, and then run your app. This time when we click on the Find Beer button, it displays the value of the selected item in the spinner.



there are no Dumb Questions

Q: I added a `String` to my `strings.xml` file, but I can't see it in `R.java`. Why isn't it there?

A: Android Studio generates `R.java` when you save any changes you've made. If you've added a resource but can't see it in `R.java`, check that your changes have been saved.

`R.java` also gets updated when the app gets built. The app builds when you run the app, so running the app will also update `R.java`.

Q: The values in the spinner look like they're static as they're set to the values in the `string-array`. Can I change these values programmatically?

A: You can, but that approach is more complicated than just using static values. We'll show you later in the book how you can have complete control over the values displayed in components such as spinners.

Q: What type of object is returned by `getSelectedItem()`?

A: It's declared as type `Object`. Because we used a `string-array` for the values, the actual value returned in this case is a `String`.

Q: What do you mean "in this case"—isn't it always?

A: You can do more complicated things with spinners than just display text. As an example, the spinner might display an icon next to each value. As `getSelectedItem()` returns an object, it gives you a bit more flexibility than just returning a `String`.

Q: Does the name of `onClickFindBeer` matter?

A: All that matters is that the name of the method in the activity code matches the name used in the button's `onClick` attribute in the layout.

Q: Why did we have to replace the activity code that Android Studio created for us?

A: IDEs such as Android Studio include functions and utilities that can save you a lot of time. They generate a lot of code for you, and sometimes this can be useful. But when you're learning a new language or development area such as Android, we think it's best to learn about the fundamentals of the language rather than what the IDE generates for you. This way you'll develop a greater understanding of the language.

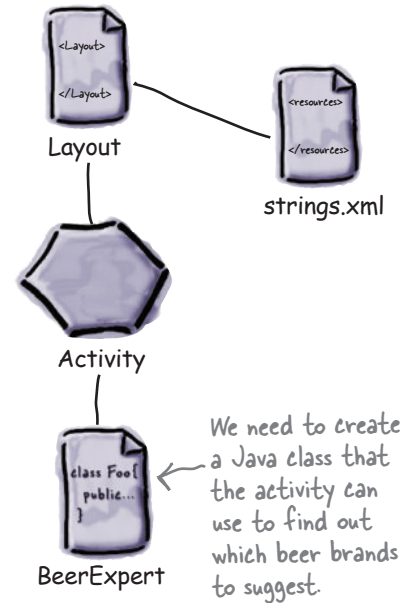
Build the custom Java class

As we said at the beginning of the chapter, the Beer Adviser app decides which beers to recommend with the help of a custom Java class. This Java class is written in plain old Java, with no knowledge of the fact it's being used by an Android app.

Custom Java class spec

The custom Java class should meet the following requirements:

- ★ The package name should be `com.hfad.beeradviser`.
- ★ The class should be called `BeerExpert`.
- ★ It should expose one method, `getBrands()`, that takes a preferred beer color (as a `String`), and return a `List<String>` of recommended beers.



Build and test the Java class

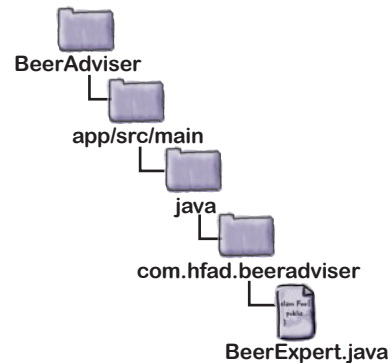
Java classes can be extremely complicated and involve calls to complex application logic. You can either build and test your own version of the class, or use our sophisticated version of the class shown here:

```
package com.hfad.beeradviser;

import java.util.ArrayList;
import java.util.List;

public class BeerExpert {
    List<String> getBrands(String color) {
        List<String> brands = new ArrayList<>();
        if (color.equals("amber")) {
            brands.add("Jack Amber");
            brands.add("Red Moose");
        } else {
            brands.add("Jail Pale Ale");
            brands.add("Gout Stout");
        }
        return brands;
    }
}
```

This is pure Java code; nothing Androidy about it.



Do this!

Add the BeerExpert class to your project. Select the `com.hfad.beeradviser` package in the `app/src/main/java` folder, go to `File→New...→Java Class`, name the file “BeerExpert”, and make sure the package name is “`com.hfad.beeradviser`”. This creates the `BeerExpert.java` file.



Enhance the activity to call the custom Java class so that we can get REAL advice

In version two of the activity we need to enhance the `onClickFindBeer()` method to call the `BeerExpert` class for beer recommendations. The code changes needed are plain old Java. You can try to write the code and run the app on your own, or you can follow along with us. But before we show you the code changes, try the exercise below; it'll help you create some of the activity code you'll need.



Sharpen your pencil

Enhance the activity so that it calls the `BeerExpert` `getBrands()` method and displays the results in the text view.

```

package com.hfad.beeradviser;

import android.app.Activity;
import android.os.Bundle;
import android.view.View;
import android.widget.Spinner;
import android.widget.TextView;
import java.util.List; ← We added this line for you.

public class FindBeerActivity extends Activity {
    private BeerExpert expert = new BeerExpert(); ← You'll need to use the BeerExpert
    ...                                     class to get the beer recommendations,
    //Called when the button gets clicked                                     so we added this line for you too.
    public void onClickFindBeer(View view) {
        //Get a reference to the TextView
        TextView brands = (TextView) findViewById(R.id.brands);
        //Get a reference to the Spinner
        Spinner color = (Spinner) findViewById(R.id.color);
        //Get the selected item in the Spinner
        String beerType = String.valueOf(color.getSelectedItem());
        //Get recommendations from the BeerExpert class

    }
}
  
```

↑
You need to update the `onClickFindBeer()` method.



Sharpen your pencil Solution

Enhance the activity so that it calls the `BeerExpert` `getBrands()` method and displays the results in the text view.

```
package com.hfad.beeradviser;

import android.app.Activity;
import android.os.Bundle;
import android.view.View;
import android.widget.Spinner;
import android.widget.TextView;
import java.util.List;

public class FindBeerActivity extends Activity {
    private BeerExpert expert = new BeerExpert();
    ...
    //Called when the button gets clicked
    public void onClickFindBeer(View view) {
        //Get a reference to the TextView
        TextView brands = (TextView) findViewById(R.id.brands);
        //Get a reference to the Spinner
        Spinner color = (Spinner) findViewById(R.id.color);
        //Get the selected item in the Spinner
        String beerType = String.valueOf(color.getSelectedItem());
        //Get recommendations from the BeerExpert class
```

```
List<String> brandsList = expert.getBrands(beerType); ← Get a List of brands.
```

```
StringBuilder brandsFormatted = new StringBuilder(); ← Build a String using
the values in the List.
```

```
for (String brand : brandsList) {
```

```
    brandsFormatted.append(brand).append('\n'); ← Display each brand
on a new line.
```

```
}
```

```
//Display the beers
```

```
brands.setText(brandsFormatted); ← Display the results in
the text view.
```

```
    }
}
```

↑
Using the `BeerExpert` requires pure Java code, so don't worry if your code looks a little different than ours.

- Create project
- Update layout
- Connect activity
- Write logic

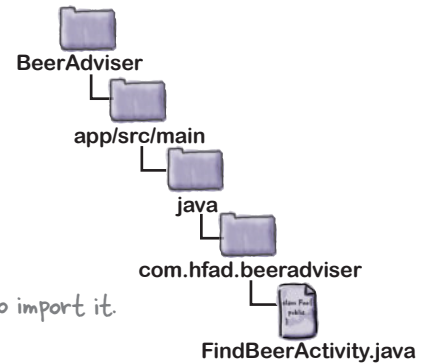


Activity code version 2

Here's our full version of the activity code. Apply the changes shown here to your version of *FindBeerActivity.java*, make sure you've added the *BeerExpert* class to your project, and then save your changes:

```
package com.hfad.beeradviser;

import android.app.Activity;
import android.os.Bundle;
import android.view.View;
import android.widget.Spinner;
import android.widget.TextView;
import java.util.List; ← We're using this extra class so we need to import it.
```



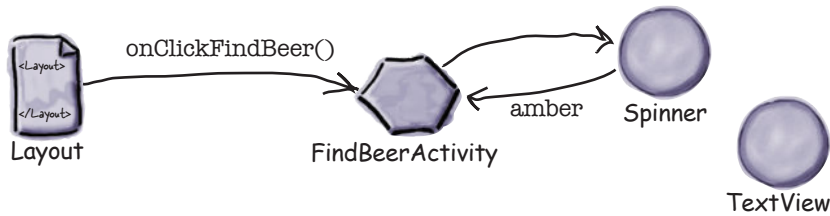
```
public class FindBeerActivity extends Activity {
    private BeerExpert expert = new BeerExpert();
    // ← Add an instance of BeerExpert as a private variable.

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

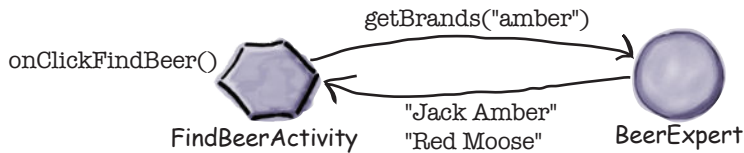
    //Called when the button gets clicked
    public void onClickFindBeer(View view) {
        //Get a reference to the TextView
        TextView brands = (TextView) findViewById(R.id.brands);
        //Get a reference to the Spinner
        Spinner color = (Spinner) findViewById(R.id.color);
        //Get the selected item in the Spinner
        String beerType = String.valueOf(color.getSelectedItem());
        //Get recommendations from the BeerExpert class
        List<String> brandsList = expert.getBrands(beerType); ← Use the BeerExpert class to get a List of brands.
        StringBuilder brandsFormatted = new StringBuilder();
        for (String brand : brandsList) {
            brandsFormatted.append(brand).append('\n'); ← Build a String, displaying each brand on a new line.
        }
        //Display the beers
        brands.setText(brandsFormatted); ← Display the String in the TextView.
        brands.setText(beerType); ← Delete this line.
    }
}
```

What happens when you run the code

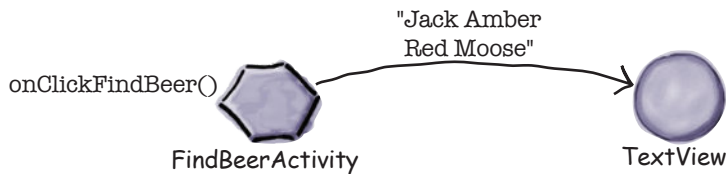
- 1** When the user clicks on the Find Beer button, the `onClickFindBeer()` method in the activity gets called. The method creates a reference to the spinner and text view, and gets the currently selected value from the spinner.



- 2** `onClickFindBeer()` calls the `getBrands()` method in the `BeerExpert` class, passing in the type of beer selected in the spinner. The `getBrands()` method returns a list of brands.



- 3** The `onClickFindBeer()` method formats the list of brands and uses it to set the text property in the text view.



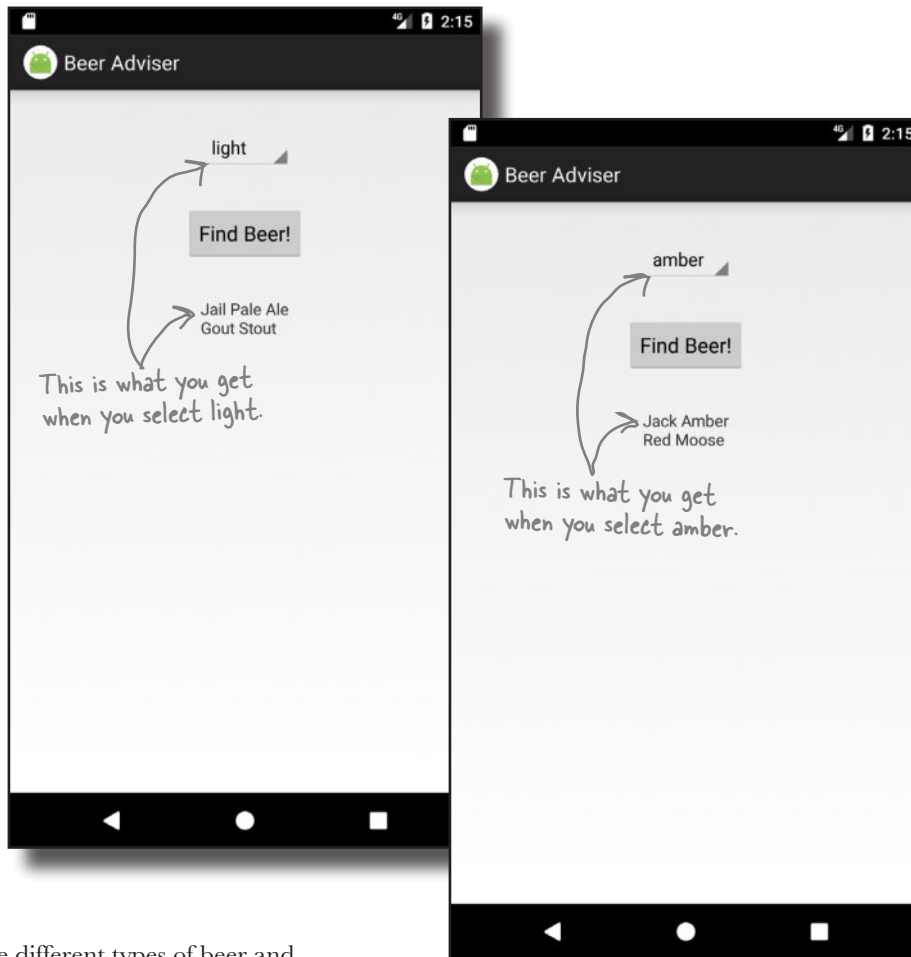


Test drive your app

Once you've made the changes to your app, go ahead and run it. Try selecting different types of beer and clicking on the Find Beer button.

building interactive apps

- Create project**
- Update layout**
- Connect activity**
- Write logic**



When you choose different types of beer and click on the Find Beer button, the app uses the BeerExpert class to provide you with a selection of suitable beers.



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 <https://tinyurl.com/HeadFirstAndroid>.

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 GUI components 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">Value</string>
```

- Reference a `String` in the layout using:

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

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

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

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

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

- Make a button call a method when clicked by adding the following to the layout:

```
android:onClick="clickMethod"
```

There needs to be a corresponding method in the activity:

```
public void clickMethod(View view) {
}
```

- `R.java` is generated for you. It enables you to get references for layouts, GUI components, `Strings`, and other resources in your Java code.
- Use `findViewById()` to get a reference to a view.
- Use `setText()` to set the text in a view.
- Use `getSelectedItem()` to get the selected item in a spinner.
- Add a custom class to an Android project by going to File menu → New... → Java Class.

3 multiple activities and intents

State Your Intent

I sent an intent asking who could handle my ACTION_CALL, and was offered all sorts of activities to choose from.



Most apps need more than one activity.

So far we've just looked at single-activity apps, which is fine for simple apps. But when things get more complicated, just having the one activity won't cut it. We're going to show you **how to build apps with multiple activities**, and how you can get your activities talking to each other using *intents*. We'll also look at how you can use intents to **go beyond the boundaries of your app and make activities in other apps on your device perform actions**. Things are about to get a whole lot more powerful...

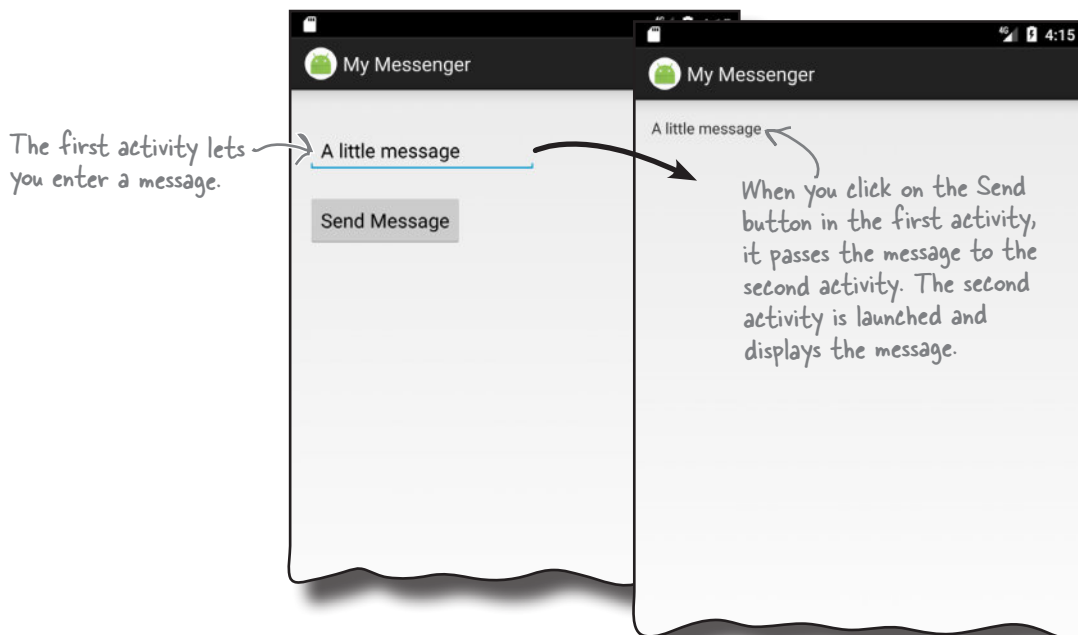
Apps can contain more than one activity

Earlier in the book, we said that an activity is a single, defined thing that your user can do, such as displaying a list of recipes. If your app is simple, this may be all that's needed.

But a lot of the time, you'll want users to do *more* than just one thing—for example, adding recipes as well as displaying a list of them. If this is the case, you'll need to use multiple activities: one for displaying the list of recipes and another for adding a single recipe.

The best way of understanding how this works is to see it in action. You're going to build an app containing two activities. The first activity will allow you to type a message. When you click on a button in the first activity, it will launch the second activity and pass it the message. The second activity will then display the message.

An activity is a single focused thing your user can do. If you chain multiple activities together to do something more complex, it's called a task.



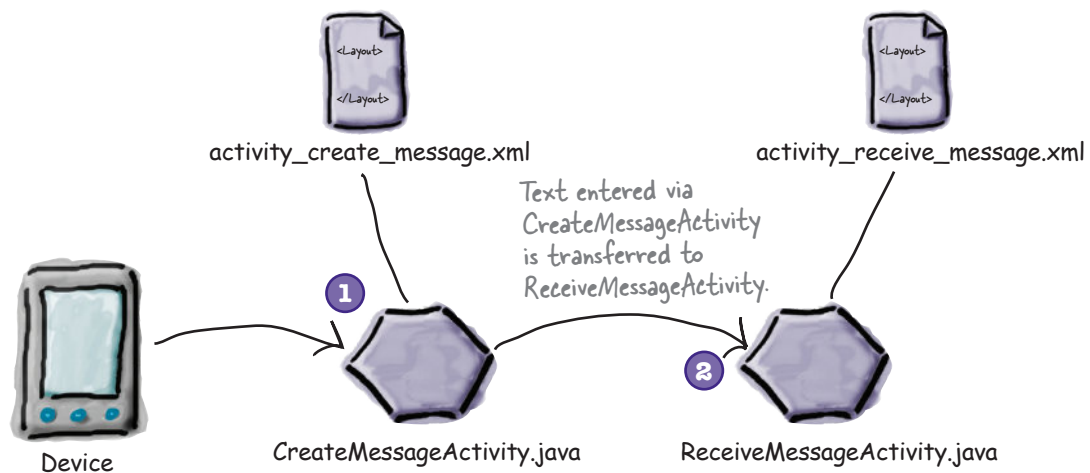
Here's what we're going to do in this chapter

- 1 Create an app with a single activity and layout.
- 2 Add a second activity and layout.
- 3 Get the first activity to call the second activity.
- 4 Get the first activity to pass data to the second activity.

Here's the app structure

The app contains two activities and two layouts.

- 1 **When the app gets launched, it starts activity `CreateMessageActivity`.**
This activity uses the layout `activity_create_message.xml`.
- 2 **When the user clicks a button in `CreateMessageActivity`, `ReceiveMessageActivity` is launched.**
This activity uses layout `activity_receive_message.xml`.



Get started: create the project

You create a project for the app in exactly the same way you did in previous chapters. Create a new Android Studio project for an application named “My Messenger” with a company domain of “hfad.com”, making the package name `com.hfad.mymessenger`. The minimum SDK should be API 19 so that it will work on most devices. You’ll need an empty activity named “`CreateMessageActivity`” with a layout named “`activity_create_message`” so that your code matches ours. **Make sure that you untick the Backwards Compatibility (AppCompat) option when you create the activity.**

On the next page, we’ll update the activity’s layout.



Create 1st activity
Create 2nd activity
Call 2nd activity
Pass data

Update the layout

Here's the XML for the `activity_create_message.xml` file. We're using a `<LinearLayout>` to display components in a single column, and we've added `<Button>` and `<EditText>` elements to it. The `<EditText>` element gives you an editable text field you can use to enter data.

Change your `activity_create_message.xml` file to match the XML 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="com.hfad.mymessenger.CreateMessageActivity">
    <EditText
        android:id="@+id/message"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginTop="20dp"
        android:hint="@string/hint"
        android:ems="10" />
    <Button
        android:id="@+id/send"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginTop="20dp"
        android:onClick="onSendMessage"
        android:text="@string/send" />
</LinearLayout>
```

We're using a linear layout with a vertical orientation.

This creates an editable text field.

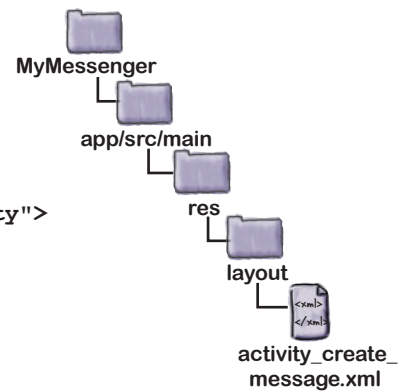
The hint attribute gives the user a hint of what text they should type into the text field. We need to add it as a String resource.

This describes how wide the `<EditText>` should be. It should be wide enough to accommodate 10 letter Ms.

Clicking on the button runs the `onSendMessage()` method in the activity.

This is a String resource we need to create.

This is the editable text field. If it's empty, it gives the user a hint about what text they should enter in it.



The `<EditText>` element defines an editable text field for entering text. It inherits from the same Android View class as the other GUI components we've seen so far.

Update strings.xml...

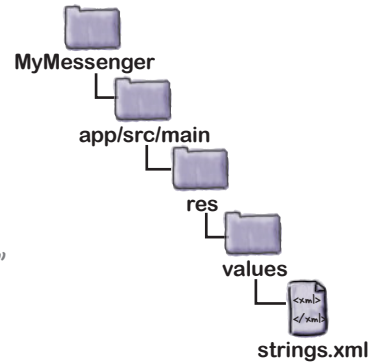
We used two String resources in our layout on the previous page. The button has a text value of @string/send that appears on the button, and the editable text field has a hint value of @string/hint that tells the user what to enter in the field. This means we need to add Strings called "send" and "hint" to strings.xml and give them values. Do this now:

```
<resources>
...
<string name="send">Send Message</string>
<string name="hint">Enter a message</string>
</resources>
```

The text "Send Message" will appear on the button.

The text "Enter a message" will appear as a hint in the text field if it's empty.

- multiple activities and intents
-
-
-
-
- Create 1st activity**
- Create 2nd activity**
- Call 2nd activity**
- Pass data**



...and add the method to the activity

This line in the <Button> element:

```
android:onClick="onSendMessage"
```

means that the onSendMessage () method in the activity will fire when the button is clicked. Let's add this method to the activity now.

Open up the CreateMessageActivity.java file and replace the code Android Studio created for you with the following:

```
package com.hfad.mymessenger;

import android.app.Activity;
import android.os.Bundle;
import android.view.View;

public class CreateMessageActivity extends Activity {

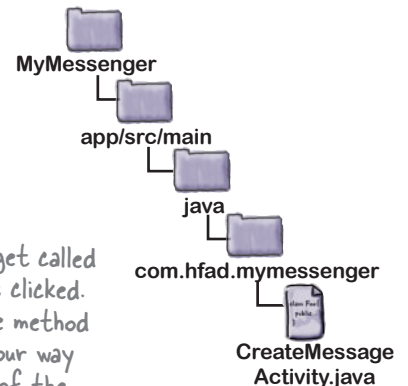
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_create_message);
    }

    //Call onSendMessage() when the button is clicked
    public void onSendMessage(View view) {
    }
}
```

Make sure your activity extends the Activity class.

The onCreate() method gets called when the activity is created.

This method will get called when the button's clicked. We'll complete the method body as we work our way through the rest of the chapter.



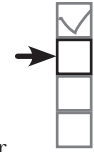
Now that you've created the first activity, let's move on to the second.

Create the second activity and layout

Android Studio has a wizard that lets you add extra activities and layouts to your apps. It's like a scaled-down version of the wizard you use to create an app, and you use it whenever you want to create a new activity.

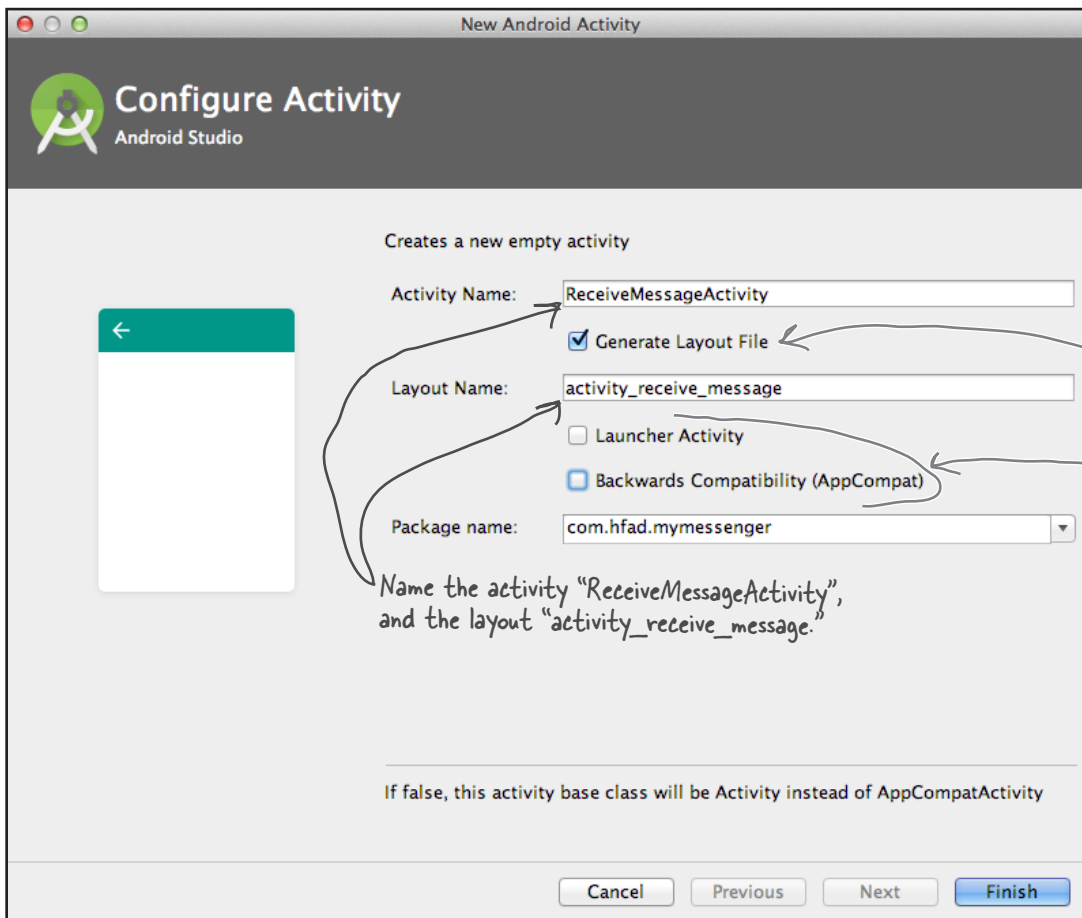
To create the new activity, switch to the Project view of Android Studio's explorer, click on the `com.hfad.mymessenger` package in the `app/src/main/java` folder, choose File → New → Activity, and choose the option for Empty Activity. You will be presented with a new screen where you can choose options for your new activity.

Every time you create a new activity and layout, you need to name them. Name the new activity "ReceiveMessageActivity" and the layout "activity_receive_message". Make sure that the option to generate a layout is checked, and the Launcher Activity and Backwards Compatibility (AppCompat) options are unchecked. Finally, confirm that the package name is `com.hfad.mymessenger`, and when you're done, click on the Finish button.



- Create 1st activity
- Create 2nd activity
- Call 2nd activity
- Pass data

Some versions of Android Studio may ask you what the source language of your activity should be. If prompted, select the option for Java.



Make sure the option to generate the layout is checked.

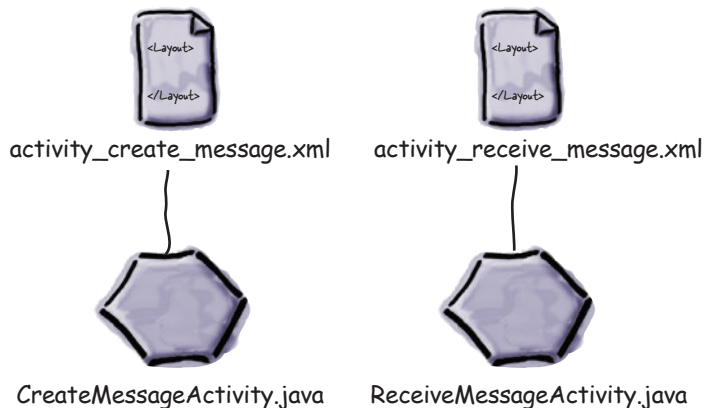
Uncheck the Launcher Activity and Backwards Compatibility (AppCompat) options.

Name the activity "ReceiveMessageActivity", and the layout "activity_receive_message."

What just happened?

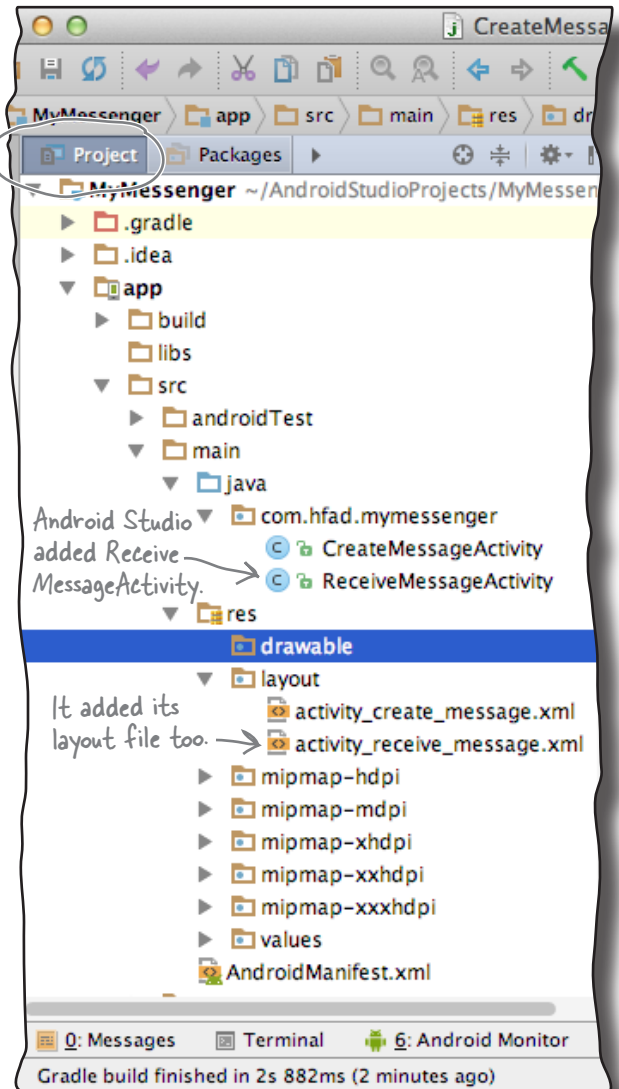
When you clicked on the Finish button, Android Studio created a shiny new activity file for you, along with a new layout. If you look in the explorer, you should see that a new file called *ReceiveMessageActivity.java* has appeared in the *app/src/main/java* folder, and a file called *activity_receive_message.xml* has appeared under *app/src/main/res/layout*.

Each activity uses a different layout. *CreateMessageActivity* uses the layout *activity_create_message.xml*, and *ReceiveMessageActivity* uses the layout *activity_receive_message.xml*:



Behind the scenes, Android Studio also made a configuration change to the app in a file called *AndroidManifest.xml*. Let's take a closer look.

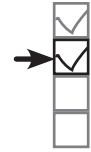
- multiple activities and intents
- Create 1st activity
- Create 2nd activity
- Call 2nd activity
- Pass data



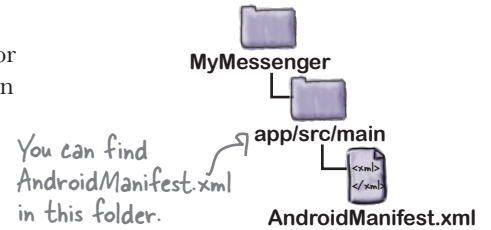
Welcome to the Android manifest file

Every Android app must include a file called *AndroidManifest.xml*. You can find it in the *app/src/main* folder of your project. The *AndroidManifest.xml* file contains essential information about your app, such as what activities it contains, required libraries, and other declarations. Android creates the file for you when you create the app. If you think back to the settings you chose when you created the project, some of the file contents should look familiar.

Here's what our copy of *AndroidManifest.xml* looks like:



- Create 1st activity
- Create 2nd activity
- Call 2nd activity
- Pass data



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

← This is the package name we specified.

← Android Studio gave our app default icons.

← The theme affects the appearance of the app. We'll look at this later.



Watch it!

If you develop Android apps without an IDE, you'll need to create this file manually.

```
    <activity android:name=".CreateMessageActivity">
        <intent-filter>
            <action android:name="android.intent.action.MAIN" />
            <category android:name="android.intent.category.LAUNCHER" />
        </intent-filter>
    </activity>

    <activity android:name=".ReceiveMessageActivity"></activity>

</application>
</manifest>
```

This is the first activity, Create Message Activity.

← This bit specifies that it's the main activity of the app.

↑ This says the activity can be used to launch the app.

↑ This is the second activity, ReceiveMessageActivity. Android Studio added this code when we added the second activity.

Every activity needs to be declared

All activities need to be declared in *AndroidManifest.xml*. If an activity isn't declared in the file, the system won't know it exists. And if the system doesn't know it exists, the activity will never run.

You declare an activity in the manifest by including an `<activity>` element inside the `<application>` element. In fact, *every* activity in your app needs a corresponding `<activity>` element. Here's the general format:

```
<application
  ...
  ...>
  <activity
    android:name=".MyActivityClassName"
    ...
    ...>
  ...
</activity>
...
</application>
```

← Each activity needs to be declared inside the `<application>` element.

← The activity may have other properties too.

← This line is mandatory; just replace *MyActivityClassName* with the name of your activity.



Activity

If I'm not included in *AndroidManifest.xml*, then as far as the system's concerned, I don't exist and will never run.

The following line is mandatory and is used to specify the class name of the activity, in this example `"MyActivityClassName"`:

```
android:name=".MyActivityClassName"
```

`MyActivityClassName` is the name of the class. It's prefixed with a `"."` because Android combines the class name with the name of the package to derive the *fully qualified* class name.

The activity declaration may include other properties too, such as security permissions, and whether it can be used by activities in other apps.



Watch it!

The second activity in our app was automatically declared

because we added it using the Android Studio wizard.

*If you add extra activities manually, you'll need to edit *AndroidManifest.xml* yourself. The same may be true if you use another IDE besides Android Studio.*

An intent is a type of message



So far we've created an app with two activities in it, and each activity has its own layout. When the app is launched, our first activity, `CreateMessageActivity`, will run. What we need to do next is get `CreateMessageActivity` to call `ReceiveMessageActivity` when the user clicks the Send Message button.

Whenever you want an activity to start a second activity, you use an **intent**. You can think of an intent as an "intent to do something." It's a type of message that allows you to bind separate objects (such as activities) together at runtime. If one activity wants to start a second activity, it does it by sending an intent to Android. Android will then start the second activity and pass it the intent.

You can create and send an intent using just a couple of lines of code. You start by creating the intent like this:

```
Intent intent = new Intent(this, Target.class);
```

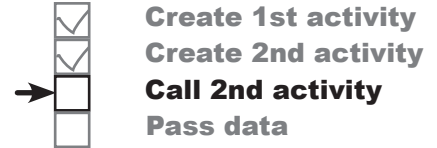
The first parameter tells Android which object the intent is from: you can use the word `this` to refer to the current activity. The second parameter is the class name of the activity that needs to receive the intent.

Once you've created the intent, you pass it to Android like this:

```
startActivity(intent);
```

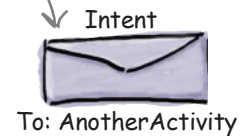
This tells Android to start the activity specified by the intent.

Once Android receives the intent, it checks that everything's OK and tells the activity to start. If it can't find the activity, it throws an **ActivityNotFoundException**.

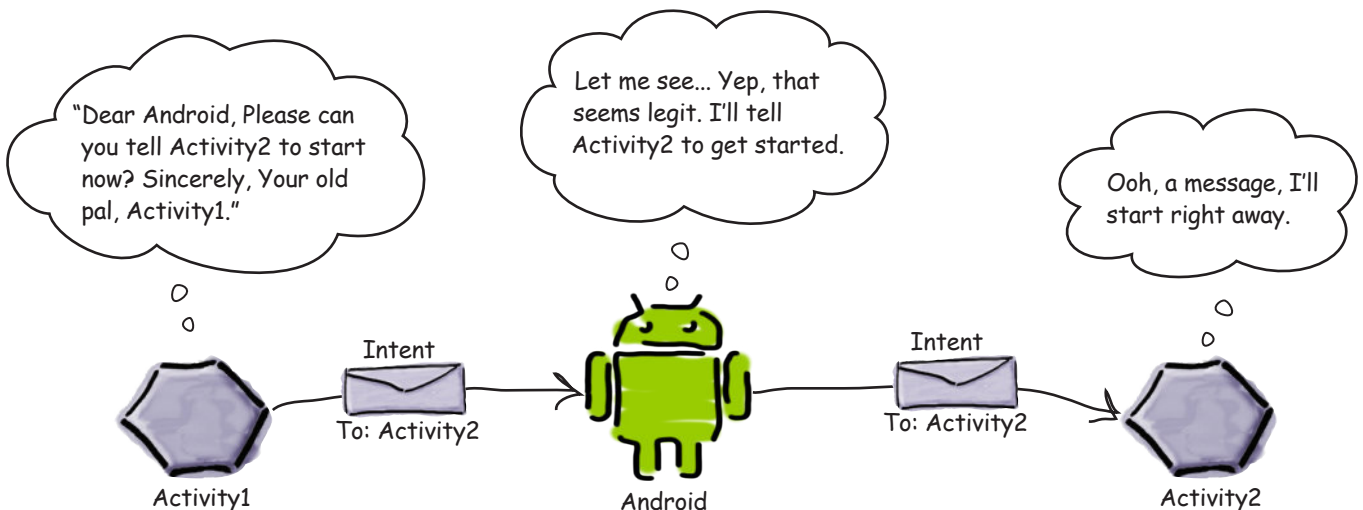


You start an activity by creating an intent and using it in the `startActivity()` method.

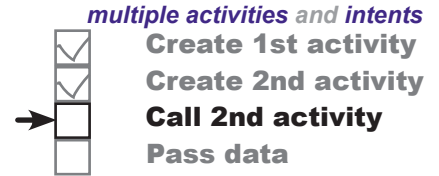
The intent specifies the activity you want to receive it. It's like putting an address on an envelope.



`startActivity()` starts the activity specified in the intent.



Use an intent to start the second activity



Let's put this into practice and use an intent to call `ReceiveMessageActivity`. We want to launch the activity when the user clicks on the Send Message button, so we'll add the two lines of code we discussed on the previous page to our `onSendMessage()` method.

Make the changes highlighted below:

```
package com.hfad.mymessenger;

import android.app.Activity;
import android.content.Intent;
import android.os.Bundle;
import android.view.View;

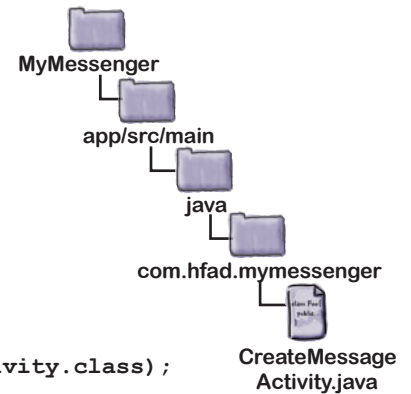
public class CreateMessageActivity extends Activity {

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

    //Call onSendMessage() when the button is clicked
    public void onSendMessage(View view) {
        Intent intent = new Intent(this, ReceiveMessageActivity.class);
        startActivity(intent);
    }
}
```

We need to import the `Intent` class `android.content.Intent` as we're using it in `onSendMessage()`.

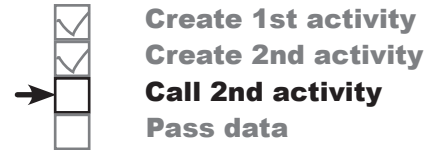
Start activity `ReceiveMessageActivity`.



So what happens now when we run the app?

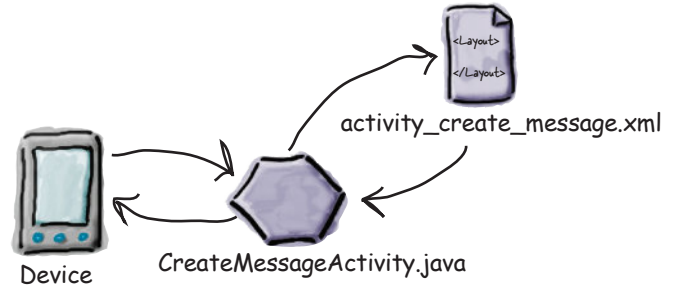
What happens when you run the app

Before we take the app out for a test drive, let's go over how the app we've developed so far will function:



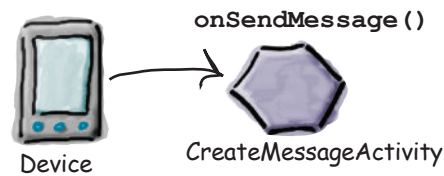
- 1 **When the app gets launched, the main activity, `CreateMessageActivity`, starts.**

When it starts, the activity specifies that it uses layout `activity_create_message.xml`. This layout gets displayed in a new window.



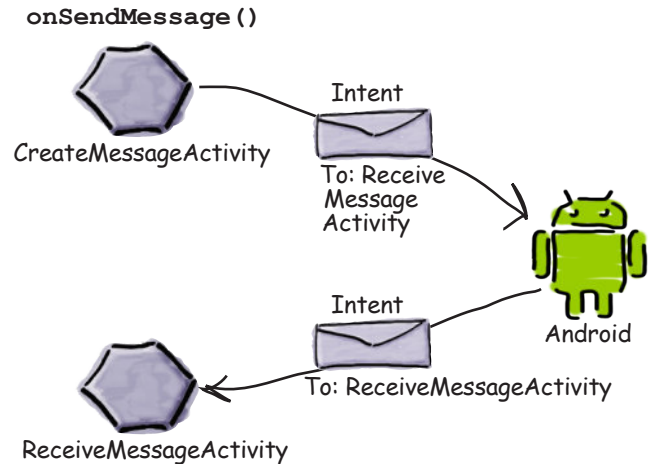
- 2 **The user types in a message and then clicks on the button.**

The `onSendMessage()` method in `CreateMessageActivity` responds to the click.



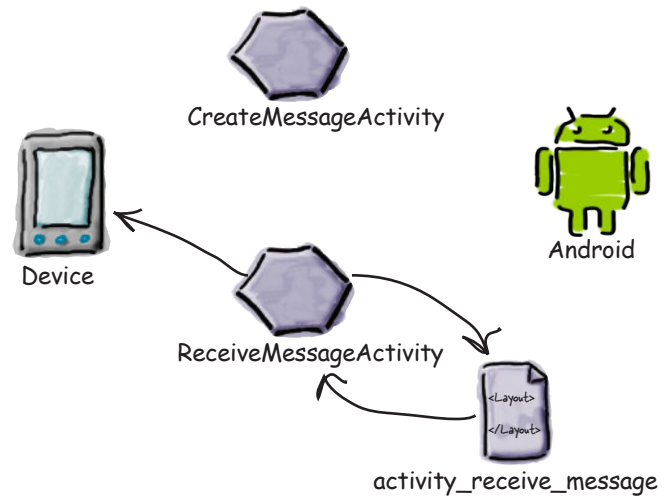
- 3 **The `onSendMessage()` method uses an intent to tell Android to start activity `ReceiveMessageActivity`.**

Android checks that the intent is valid, and then it tells `ReceiveMessageActivity` to start.



The story continues...

- 4 When `ReceiveMessageActivity` starts, it specifies that it uses layout `activity_receive_message.xml` and this layout gets displayed in a new window.

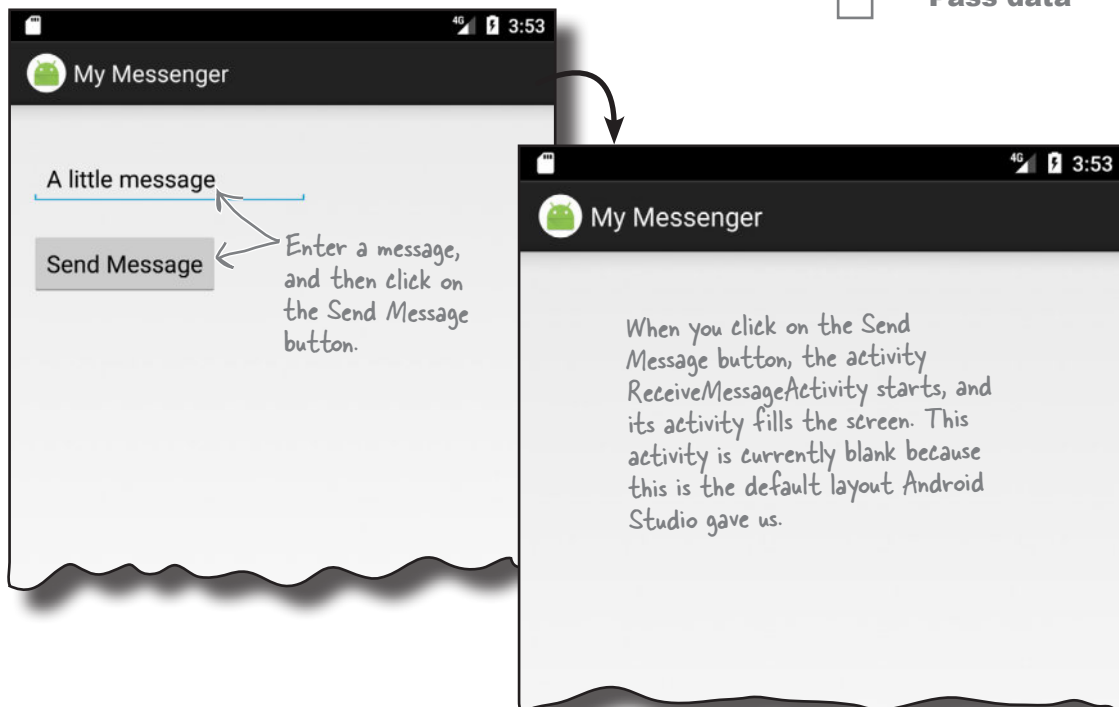


Test drive the app

Save your changes, and then run the app. `CreateMessageActivity` starts, and when you click on the Send Message button, it launches `ReceiveMessageActivity`.



Create 1st activity
Create 2nd activity
Call 2nd activity
Pass data



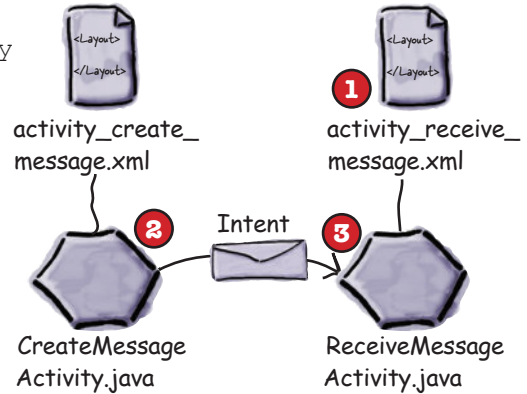
Pass text to a second activity

So far we've coded `CreateMessageActivity` to start `ReceiveMessageActivity` when the Send Message button is pressed. Next, we'll get `CreateMessageActivity` to pass text to `ReceiveMessageActivity` so that `ReceiveMessageActivity` can display it. In order to accomplish this, we'll do three things:

- 1 Tweak the layout `activity_receive_message.xml` so that it can display the text. At the moment it's simply the default layout the wizard gave us.
- 2 Update `CreateMessageActivity.java` so that it gets the text the user inputs, and then adds the text to the intent before it sends it.
- 3 Update `ReceiveMessageActivity.java` so that it displays the text sent in the intent.



Create 1st activity
Create 2nd activity
Call 2nd activity
Pass data



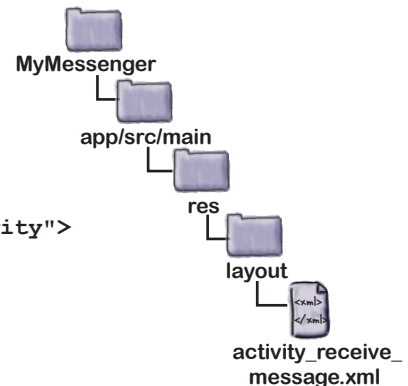
Let's start with the layout

We'll begin by changing the `activity_receive_message.xml` code Android Studio created for us so that it uses a `<LinearLayout>`. Update your version of the code so that it matches ours:

```
<?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="com.hfad.mymessenger.ReceiveMessageActivity">

</LinearLayout>
```

We're going to use a linear layout with a vertical orientation as we did in `activity_create_message.xml`.



Exercise

We need to change the layout so that it includes a text view. The text view needs to have an ID of "message" so that we can reference it in our activity code. How should we change the layout's code? Think about this before looking at the next page.

Update the text view properties

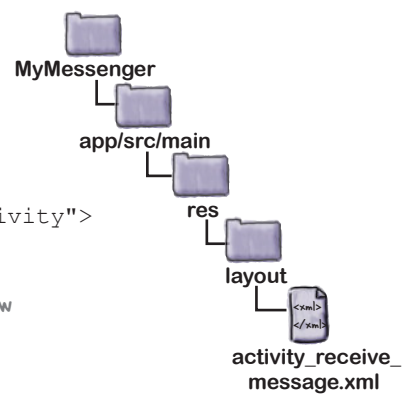
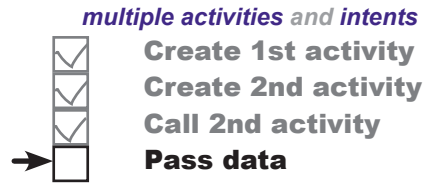
We need to add a `<TextView>` element to the layout, and give it an ID of “message.” This is because you have to add an ID to any GUI components you need to reference in your activity code, and we need to reference the text view so that we can update the text it displays.

We’ve updated our code so that it includes a new text view. Update your `activity_receive_message.xml` code so that it reflects ours (we’ve bolded our changes):

```
<?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="com.hfad.mymessenger.ReceiveMessageActivity">
    <TextView
        android:id="@+id/message"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" />
</LinearLayout>
```

← This adds the text view.

← This line gives the text view an ID of “message”.



We’ve not specified default text for the text view, as the only text we’ll ever want to display in the text view is the message passed to it by `CreateMessageActivity`.

Now that we’ve updated the layout, we can get to work on the activities. Let’s start by looking at how we can use an intent to pass a message to `ReceiveMessageActivity`.

there are no
Dumb Questions

Q: Do I have to use intents? Can't I just construct an instance of the second activity in the code for my first activity?

A: That's a good question, but no, that's not the "Android way" of doing things. One of the reasons is that passing intents to Android tells Android the sequence in which activities are started. This means that when you click on the Back button on your device, Android knows exactly where to take you back to.

putExtra() puts extra info in an intent



- Create 1st activity
- Create 2nd activity
- Call 2nd activity
- Pass data

You've seen how you can create a new intent using:

```
Intent intent = new Intent(this, Target.class);
```

You can add extra information to this intent that can be picked up by the activity you're targeting so it can react in some way. To do this, you use the putExtra () method like so:

```
intent.putExtra("message", value);
```

where message is a String name for the value you're passing in, and value is the value. The putExtra () method is overloaded so value has many possible types. As an example, it can be a primitive such as a boolean or int, an array of primitives, or a String. You can use putExtra () repeatedly to add numerous extra data to the intent. If you do this, make sure you give each one a unique name.

putExtra() lets you put extra information in the message you're sending.



To: ReceiveMessageActivity
message: "Hello!"

There are many different options for the type of value. You can see them all in the Google Android documentation. Android Studio will also give you a list as you type code in.

How to retrieve extra information from an intent

The story doesn't end there. When Android tells ReceiveMessageActivity to start, ReceiveMessageActivity needs some way of retrieving the extra information that CreateMessageActivity sent to Android in the intent.

There are a couple of useful methods that can help with this. The first of these is:

```
getIntent ();
```

getIntent () returns the intent that started the activity, and you can use this to retrieve any extra information that was sent along with it. How you do this depends on the type of information that was sent. As an example, if you know the intent includes a String value named "message", you would use the following:

```
Intent intent = getIntent (); ← Get the intent.  
String string = intent.getStringExtra("message"); ← Get the String passed along with the intent that has a name of "message".
```



To: ReceiveMessageActivity
message: "Hello!"

You're not just limited to retrieving String values. As an example, you can use:

```
int intNum = intent.getIntExtra("name", default_value);
```

to retrieve an int with a name of name. default_value specifies what int value you should use as a default.

Pool Puzzle



Your **job** is to take code snippets from the pool and place them into the blank lines in *CreateMessageActivity.java*. You may **not** use the same code snippet more than once, and you won't need to use all the code snippets. Your **goal** is to make the activity retrieve text from the message `<EditText>` and add it to the intent.

```
package com.hfad.mymessenger;

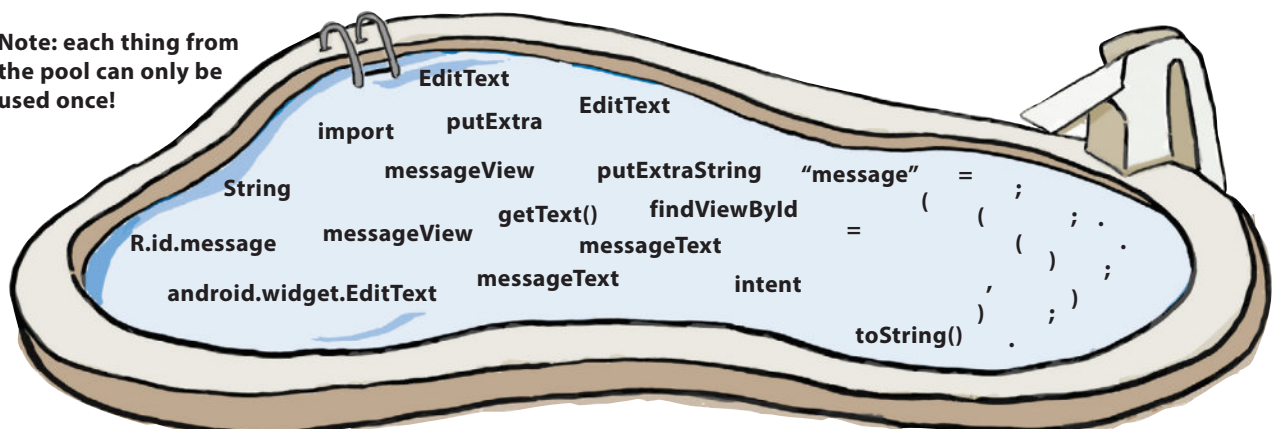
import android.app.Activity;
import android.os.Bundle;
import android.content.Intent;
import android.view.View;
.....

public class CreateMessageActivity extends Activity {

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

    //Call onSendMessage() when the button is clicked
    public void onSendMessage(View view) {
        .....
        .....
        Intent intent = new Intent(this, ReceiveMessageActivity.class);
        .....
        startActivity(intent);
    }
}
```

Note: each thing from the pool can only be used once!



```
package com.hfad.mymessenger;

import android.app.Activity;
import android.os.Bundle;
import android.content.Intent;
import android.view.View;
import android.widget.EditText;

public class CreateMessageActivity extends Activity {

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

    //Call onSendMessage() when the button is clicked
    public void onSendMessage(View view) {
        EditText messageView = (EditText) findViewById(R.id.message);
        String messageText = messageView.getText().toString();
        Intent intent = new Intent(this, ReceiveMessageActivity.class);
        intent.putExtra("message", messageText);
        startActivity(intent);
    }
}
```

You need to import the EditText class.

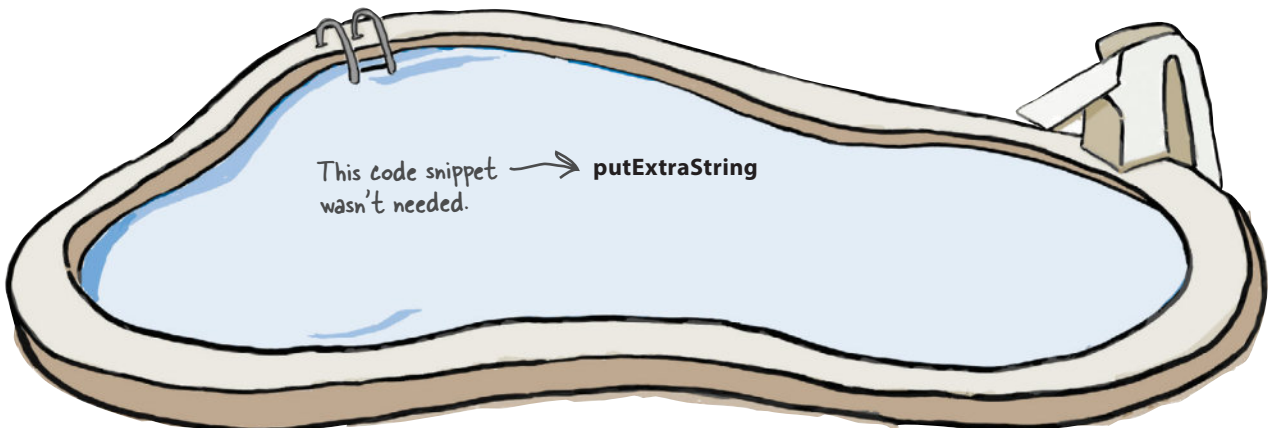


Pool Puzzle Solution

Your **job** is to take code snippets from the pool and place them into the blank lines in *CreateMessageActivity.java*. You may **not** use the same code snippet more than once, and you won't need to use all the code snippets. Your **goal** is to make the activity retrieve text from the message `<EditText>` and add it to the intent.

Get the text from the editable text field with an ID of "message."

Add the text to the intent, giving it a name of "message".



Update the CreateMessageActivity code

We updated our code for *CreateMessageActivity.java* so that it takes the text the user enters on the screen and adds it to the intent. Here's the full code (make sure you update your code to include these changes, shown in bold):

```

package com.hfad.mymessenger;

import android.app.Activity;
import android.os.Bundle;
import android.content.Intent;
import android.view.View;
import android.widget.EditText;

public class CreateMessageActivity extends Activity {

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

    //Call onSendMessage() when the button is clicked
    public void onSendMessage(View view) {
        EditText messageView = (EditText) findViewById(R.id.message);
        String messageText = messageView.getText().toString();
        Intent intent = new Intent(this, ReceiveMessageActivity.class);
        intent.putExtra(ReceiveMessageActivity.EXTRA_MESSAGE, messageText);
        startActivity(intent);
    }
}

```

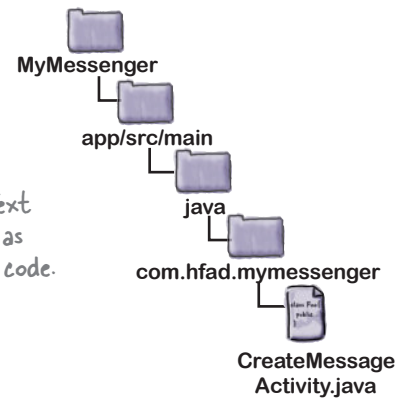
You need to import the `EditText` class `android.widget.EditText` as you're using it in your activity code.

Start `ReceiveMessageActivity` with the intent.

multiple activities and intents



- Create 1st activity
- Create 2nd activity
- Call 2nd activity
- Pass data



Get the text that's in the `EditText`.

Create an intent, then add the text to the intent. We're using a constant for the name of the extra information so that we know `CreateMessageActivity` and `ReceiveMessageActivity` are using the same String. We'll add this constant to `ReceiveMessageActivity` on the next page, so don't worry if Android Studio says it doesn't exist.

Now that `CreateMessageActivity` has added extra information to the intent, we need to retrieve that information and use it.

Get ReceiveMessageActivity to use the information in the intent



- Create 1st activity
- Create 2nd activity
- Call 2nd activity
- Pass data

Now that we've changed CreateMessageActivity to add text to the intent, we'll update ReceiveMessageActivity so that it uses that text.

We're going to get ReceiveMessageActivity to display the message in its text view when the activity gets created. Because the activity's onCreate() method gets called as soon as the activity is created, we'll add the code to this method.

To get the message from the intent, we'll first get the intent using the getIntent() method, then get the value of the message using getStringExtra().

Here's the full code for ReceiveMessageActivity.java (replace the code that Android Studio generated for you with this code, and then save all your changes):

```

package com.hfad.mymessenger;

import android.app.Activity;
import android.os.Bundle;
import android.content.Intent;
import android.widget.TextView;

public class ReceiveMessageActivity extends Activity {

    public static final String EXTRA_MESSAGE = "message";

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_receive_message);
        Intent intent = getIntent();
        String messageText = intent.getStringExtra(EXTRA_MESSAGE);
        TextView messageView = (TextView) findViewById(R.id.message);
        messageView.setText(messageText);
    }
}

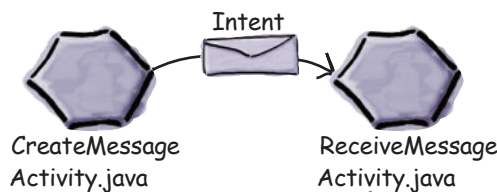
```

We need to import these classes.

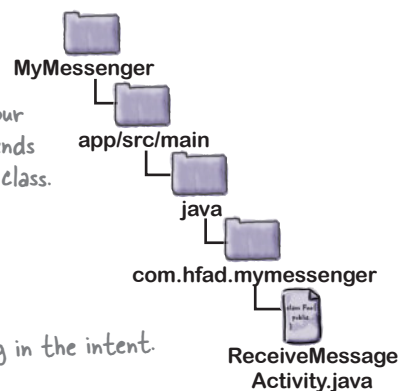
Make sure your activity extends the Activity class.

This is the name of the extra value we're passing in the intent.

Add the text to the message text view.



We need to make ReceiveMessageActivity deal with the intent it receives.



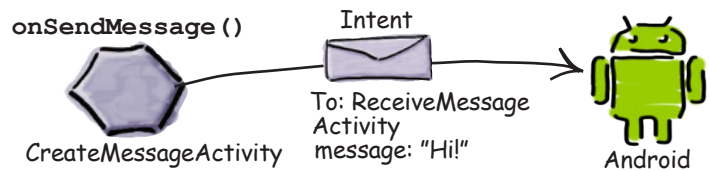
Get the intent, and get the message from it using getStringExtra().

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

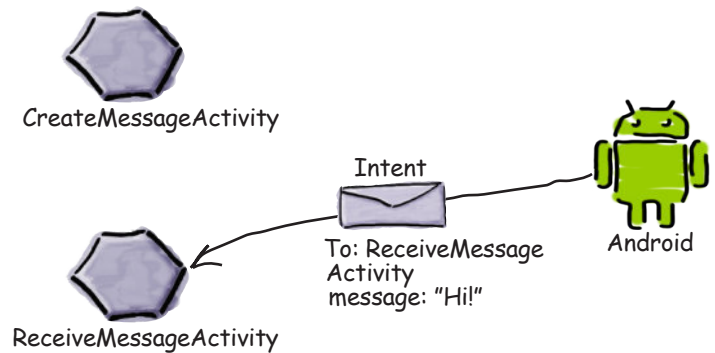
What happens when the user clicks the Send Message button

- 1** When the user clicks on the button, the `onSendMessage()` method is called.

Code within the `onSendMessage()` method creates an intent to start activity `ReceiveMessageActivity`, adds a message to the intent, and passes it to Android with an instruction to start the activity.

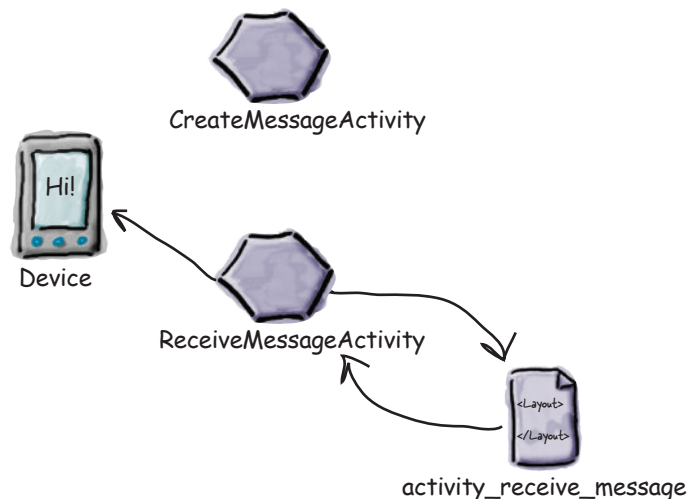


- 2** Android checks that the intent is OK, and then tells `ReceiveMessageActivity` to start.



- 3** When `ReceiveMessageActivity` starts, it specifies that it uses layout `activity_receive_message.xml`, and this gets displayed on the device.

The activity also updates the layout so that it displays the extra text included in the intent.



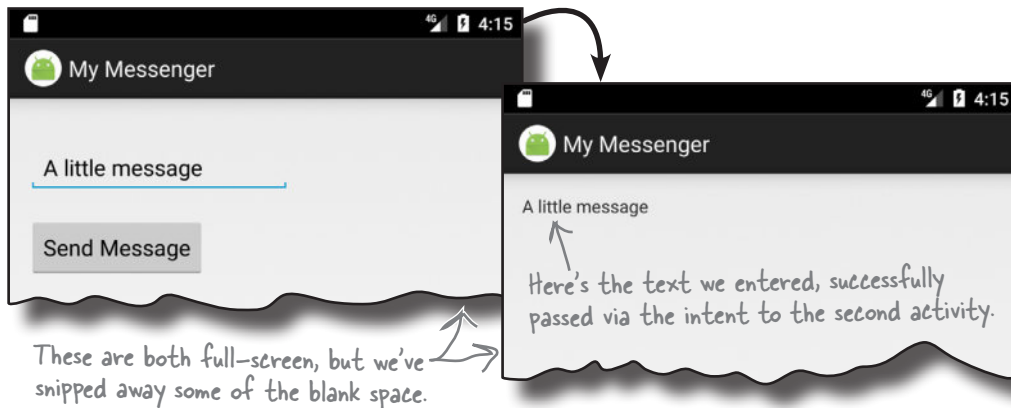


Test drive the app

Make sure you've updated the two activities, save your changes, and then run the app. CreateMessageActivity starts, and when you enter some text and then click on the Send Message button, it launches ReceiveMessageActivity. The text you entered is displayed in the text view.



Create 1st activity
Create 2nd activity
Call 2nd activity
Pass data



We can change the app to send messages to other people

Now that we have an app that sends a message to another activity, we can change it so that it can send messages to other people. We can do this by integrating with the message sending apps already on the device. Depending on what apps the user has, we can get our app to send messages via Gmail, Google+, Facebook, Twitter...

Hey, hold it right there! That sounds like a freaky amount of work to get our app working with all those other apps. And how the heck do I know what apps people have on their devices anyway?

It's not as hard as it sounds thanks to the way Android is designed to work.

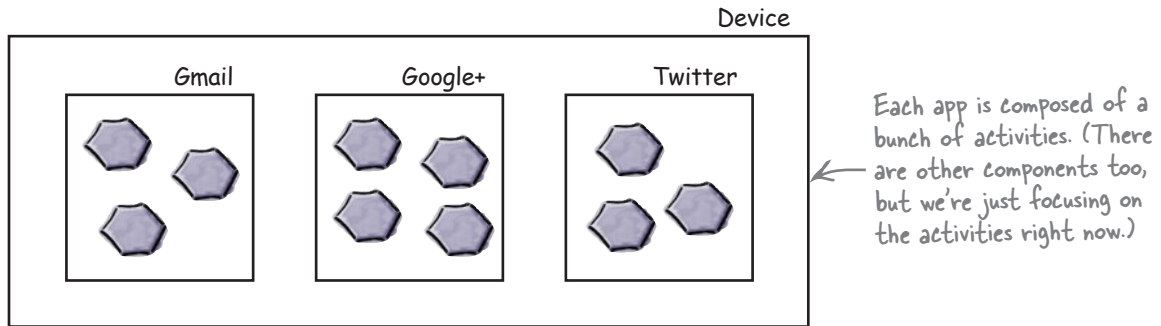
Remember right at the beginning of the chapter when we said that tasks are multiple activities chained together? Well, **you're not just limited to using the activities within your app.** You can go beyond the boundaries of your app to use activities within *other* apps as well.



How Android apps work

As you've seen, all Android apps are composed of one or more activities, along with other components such as layouts. Each activity is a single defined focused thing the user can do. As an example, apps such as Gmail, Google+, Facebook, and Twitter all have activities that enable you to send messages, even though they may achieve this in different ways.

- multiple activities and intents**
- ✓ Create 1st activity
 - ✓ Create 2nd activity
 - ✓ Call 2nd activity
 - ✓ Pass data

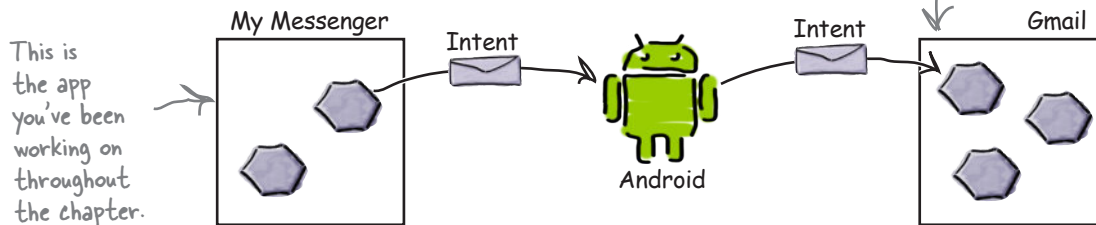


Intents can start activities in other apps

You've already seen how you can use an intent to start a second activity within the same app. The first activity passes an intent to Android, Android checks it, and then Android tells the second activity to start.

The same principle applies to activities in other apps. You get an activity in your app to pass an intent to Android, Android checks it, and then Android tells the second activity to start *even though it's in another app*. As an example, we can use an intent to start the activity in Gmail that sends messages, and pass it the text we want to send. That means that instead of writing our own activities to send emails, we can use the existing Gmail app.

You can create an intent to start another activity even if the activity is within another app.



This means that you can build apps that perform powerful tasks by chaining together activities across the device.

But we don't know what apps are on the user's device

There are three questions we need answers to before we can call activities in other apps:

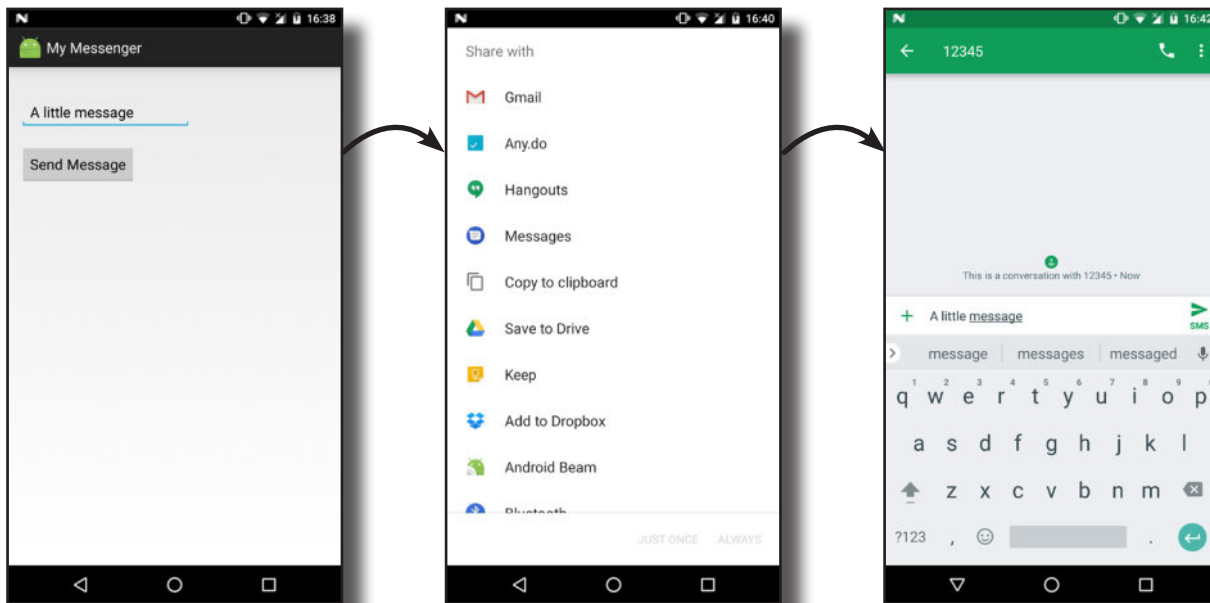
- ★ How do we know which activities are available on the user's device?
- ★ How do we know which of these activities are appropriate for what we want to do?
- ★ How do we know how to use these activities?

The great news is that we can solve all of these problems using **actions**. Actions are a way of telling Android what standard operations activities can perform. As an example, Android knows that all activities registered for a send action are capable of sending messages.

Let's explore how to create intents that use actions to return a set of activities that you can use in a standard way—for example, to send messages.

Here's what you're going to do

- 1 **Create an intent that specifies an action.**
The intent will tell Android you want to use an activity that can send a message. The intent will include the text of the message.
- 2 **Allow the user to choose which app to use.**
Chances are, there'll be more than one app on the user's device capable of sending messages, so the user will need to pick one. We want the user to be able to choose one every time they click on the Send Message button.



Create an intent that specifies an action

So far you've seen how to create an intent that launches a specific activity using:

```
Intent intent = new Intent(this, ReceiveMessageActivity.class);
```

This intent is an example of an **explicit intent**; you explicitly tell Android which class you want it to run.

If there's an action you want done but you don't care which activity does it, you create an **implicit intent**. You tell Android what sort of action you want it to perform, and you leave the details of which activity performs it to Android.

↑
 We've told the intent which class it's intended for, but what if we don't know?

How to create the intent

You create an intent that specifies an action using the following syntax:

```
Intent intent = new Intent(action);
```

where `action` is the type of activity action you want to perform. Android provides you with a number of standard actions you can use. As an example, you can use `Intent.ACTION_DIAL` to dial a number, `Intent.ACTION_WEB_SEARCH` to perform a web search, and `Intent.ACTION_SEND` to send a message. So, if you want to create an intent that specifies you want to send a message, you use:

```
Intent intent = new Intent(Intent.ACTION_SEND);
```

Adding extra information

Once you've specified the action you want to use, you can add extra information to it. We want to pass some text with the intent that will form the body of the message we're sending. To do this, you use the following lines of code:

```
intent.setType("text/plain");  

intent.putExtra(Intent.EXTRA_TEXT, messageText);
```

where `messageText` is the text you want to send. This tells Android that you want the activity to be able to handle data with a MIME data-type of `text/plain`, and also tells it what the text is.

You can make extra calls to the `putExtra()` method if there's additional information you want to add. As an example, if you want to specify the subject of the message, you can use:

```
intent.putExtra(Intent.EXTRA_SUBJECT, subject);
```

where `subject` is the subject of the message.

You can find out more about the sorts of activity actions you can use and the extra information they support in the Android developer reference material: <http://tinyurl.com/n57qb5>.

← These attributes relate to `Intent.ACTION_SEND`. They're not relevant for all actions.

← If subject isn't relevant to a particular app, it will just ignore this information. Any apps that know how to use it will do so.



Change the intent to use an action

We'll update *CreateMessageActivity.java* so that we create an implicit intent that uses a send action. Make the changes highlighted below, and save your work:

```
package com.hfad.mymessenger;

import android.app.Activity;
import android.os.Bundle;
import android.content.Intent;
import android.view.View;
import android.widget.EditText;

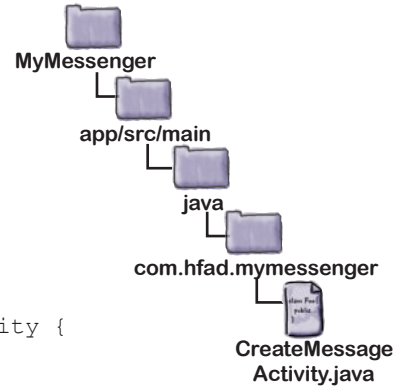
public class CreateMessageActivity extends Activity {

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

    //Call onSendMessage() when the button is clicked
    public void onSendMessage(View view) {
        EditText messageView = (EditText)findViewById(R.id.message);
        String messageText = messageView.getText().toString();
        Intent intent = new Intent(this, ReceiveMessageActivity.class);
        intent.putExtra(ReceiveMessageActivity.EXTRA_MESSAGE, messageText);
        Intent intent = new Intent(Intent.ACTION_SEND);
        intent.setType("text/plain");
        intent.putExtra(Intent.EXTRA_TEXT, messageText);
        startActivity(intent);
    }
}
```

Remove these
two lines.

Instead of creating an intent that's explicitly for *ReceiveMessageActivity*, we're creating an intent that uses a send action.

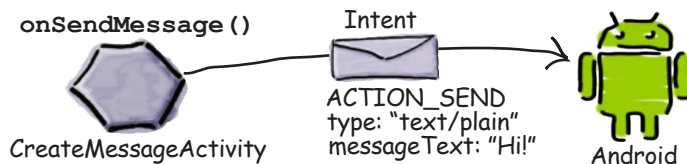




What happens when the code runs

- 1** When the `onSendMessage()` method is called, an intent gets created. The `startActivity()` method passes this intent to Android.

The intent specifies an action of `ACTION_SEND`, and a MIME type of `text/plain`.

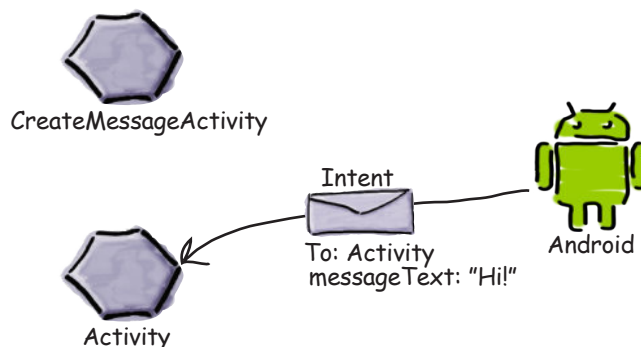


- 2** Android sees that the intent can only be passed to activities able to handle `ACTION_SEND` and `text/plain` data. Android checks all the activities on the user's device, looking for ones that are able to receive the intent.

If no actions are able to handle the intent, an `ActivityNotFoundException` is thrown.



- 3a** If just one activity is able to receive the intent, Android tells that activity to start and passes it the intent.





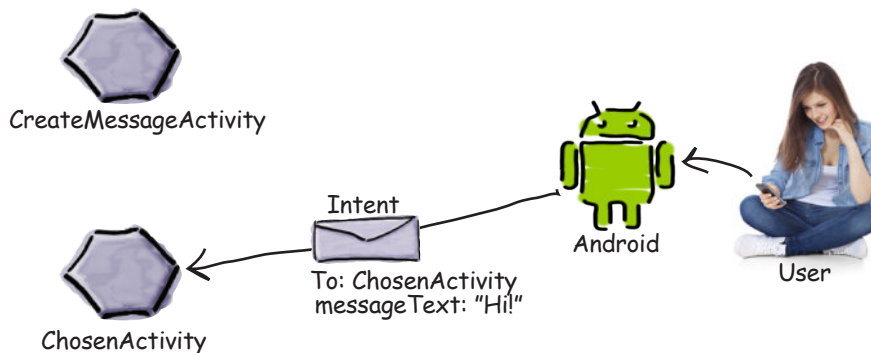
Specify action
Create chooser

The story continues...

- 3b If more than one activity is able to receive the intent, Android displays an activity chooser dialog and asks the user which one to use.



- 4 When the user chooses the activity she wants to use, Android tells the activity to start and passes it the intent. The activity displays the extra text contained in the intent in the body of a new message.



In order to pass the intent to an activity, Android must first know which activities are capable of receiving the intent. On the next couple of pages we'll look at how it does this.



The intent filter tells Android which activities can handle which actions

When Android is given an intent, it has to figure out which activity, or activities, can handle it. This process is known as **intent resolution**.

When you use an *explicit* intent, intent resolution is straightforward. The intent explicitly says which component the intent is directed at, so Android has clear instructions about what to do. As an example, the following code explicitly tells Android to start `ReceiveMessageActivity`:

```
Intent intent = new Intent(this, ReceiveMessageActivity.class);
startActivity(intent);
```

When you use an *implicit* intent, Android uses the information in the intent to figure out which components are able to receive it. It does this by checking the intent filters in every app's copy of `AndroidManifest.xml`.

An **intent filter** specifies what types of intent each component can receive. As an example, here's the entry for an activity that can handle an action of `ACTION_SEND`. The activity is able to accept data with MIME types of `text/plain` or `image/*`:

```
<activity android:name="ShareActivity">
  <intent-filter>
    <action android:name="android.intent.action.SEND"/>
    <category android:name="android.intent.category.DEFAULT"/>
    <data android:mimeType="text/plain"/>
    <data android:mimeType="image/*"/>
  </intent-filter>
</activity>
```

This is just an example; there's no activity called "ShareActivity" in our project.

This tells Android the activity can handle `ACTION_SEND`.

These are the types of data the activity can handle.

The intent filter must include a category of `DEFAULT` or it won't be able to receive implicit intents.

The intent filter also specifies a **category**. The category supplies extra information about the activity such as whether it can be started by a web browser, or whether it's the main entry point of the app. An intent filter **must** include a category of `android.intent.category.DEFAULT` if it's to receive implicit intents. If an activity has no intent filter, or it doesn't include a category name of `android.intent.category.DEFAULT`, it means that the activity can't be started with an implicit intent. It can only be started with an *explicit* intent using the full name (including the package) of the component.



How Android uses the intent filter

When you use an implicit intent, Android compares the information given in the intent with the information given in the intent filters specified in every app's *AndroidManifest.xml* file.

Android first considers intent filters that include a category of `android.intent.category.DEFAULT`:

```
<intent-filter>
    <category android:name="android.intent.category.DEFAULT"/>
    ...
</intent-filter>
```

Intent filters without this category will be omitted, as they can't receive implicit intents.

Android then matches intents to intent filters by comparing the action and MIME type contained in the intent with those of the intent filters. As an example, if an intent specifies an action of `Intent.ACTION_SEND` using:

```
Intent intent = new Intent(Intent.ACTION_SEND);
```

It will also look at the category of the intent filter if one is supplied by the intent. However, this feature isn't used very often, so we don't cover how to add categories to intents.

Android will only consider activities that specify an intent filter with an action of `android.intent.action.SEND` like this:

```
<intent-filter>
    <action android:name="android.intent.action.SEND"/>
    ...
</intent-filter>
```

Similarly, if the intent MIME type is set to `text/plain` using:

```
intent.setType("text/plain");
```

Android will only consider activities that can accommodate this type of data:

```
<intent-filter>
    <data android:mimeType="text/plain"/>
    ...
</intent-filter>
```

If the MIME type is left out of the intent, Android tries to infer the type based on the data the intent contains.

Once Android has finished comparing the intent to the component's intent filters, it sees how many matches it finds. If Android finds a single match, it starts the component (in our case, the activity) and passes it the intent. If it finds multiple matches, it asks the user to pick one.



BE the Intent

Your job is to play like you're the intent on the right and say which of the activities described below are compatible with your action and data. Say why, or why not, for each one.

Here's the intent.



```
Intent intent = new Intent(Intent.ACTION_SEND);
intent.setType("text/plain");
intent.putExtra(Intent.EXTRA_TEXT, "Hello");
```

```
<activity android:name="SendActivity">
  <intent-filter>
    <action android:name="android.intent.action.SEND"/>
    <category android:name="android.intent.category.DEFAULT"/>
    <data android:mimeType="*/"/>
  </intent-filter>
</activity>
```

```
<activity android:name="SendActivity">
  <intent-filter>
    <action android:name="android.intent.action.SEND"/>
    <category android:name="android.intent.category.MAIN"/>
    <data android:mimeType="text/plain"/>
  </intent-filter>
</activity>
```

```
<activity android:name="SendActivity">
  <intent-filter>
    <action android:name="android.intent.action.SENDTO"/>
    <category android:name="android.intent.category.MAIN"/>
    <category android:name="android.intent.category.DEFAULT"/>
    <data android:mimeType="text/plain"/>
  </intent-filter>
</activity>
```



BE the Intent Solution

Your job is to play like you're the intent on the right and say which of the activities described below are compatible with your action and data. Say why, or why not, for each one.

```
Intent intent = new Intent(Intent.ACTION_SEND);
intent.setType("text/plain");
intent.putExtra(Intent.EXTRA_TEXT, "Hello");
```

```
<activity android:name="SendActivity">
  <intent-filter>
    <action android:name="android.intent.action.SEND"/>
    <category android:name="android.intent.category.DEFAULT"/>
    <data android:mimeType="*/"/>
  </intent-filter>
</activity>
```



This activity accepts `ACTION_SEND` and can handle data of any MIME type, so it can respond to the intent.

```
<activity android:name="SendActivity">
  <intent-filter>
    <action android:name="android.intent.action.SEND"/>
    <category android:name="android.intent.category.MAIN"/>
    <data android:mimeType="text/plain"/>
  </intent-filter>
</activity>
```



This activity doesn't have a category of `DEFAULT` so can't receive the intent.

```
<activity android:name="SendActivity">
  <intent-filter>
    <action android:name="android.intent.action.SENDTO"/>
    <category android:name="android.intent.category.MAIN"/>
    <category android:name="android.intent.category.DEFAULT"/>
    <data android:mimeType="text/plain"/>
  </intent-filter>
</activity>
```



This activity can't accept `ACTION_SEND` intents, only `ACTION_SENDTO` (which allows you to send a message to someone specified in the intent's data).



You need to run your app on a REAL device

So far we've been running our apps using the emulator. The emulator only includes a small number of apps, and there may well be just one app that can handle `ACTION_SEND`. In order to test our app properly, we need to run it on a physical device where we know there'll be more than one app that can support our action—for example, an app that can send emails and an app that can send text messages.

Here's how you go about getting your app to run on a physical device.

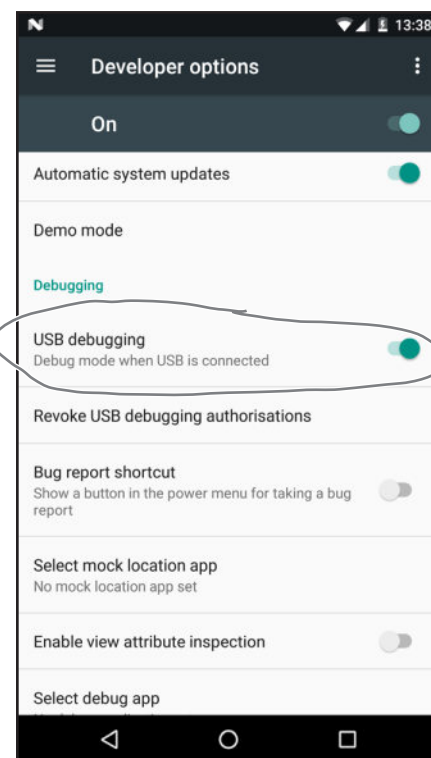
1. Enable USB debugging on your device

Yep,
seriously. →

On your device, open “Developer options” (in Android 4.0 onward, this is hidden by default). To enable “Developer options,” go to Settings → About Phone and tap the build number seven times. When you return to the previous screen, you should now be able to see “Developer options.”

Within “Developer options,” turn on USB debugging.

You need to enable USB debugging. →



2. Set up your system to detect your device

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

If you're using Windows, you need to install a USB driver. You can find the latest instructions here:

<http://developer.android.com/tools/extras/oem-usb.html>

If you're using Ubuntu Linux, you need to create a udev rules file. You can find the latest instructions on how to do this here:

<http://developer.android.com/tools/device.html#setting-up>

3. Plug your device into your computer with a USB cable

Your device may ask you if you want to accept an RSA key that allows USB debugging with your computer. If it does, you can check the “Always allow from this computer” option and choose OK to enable this.



Specify action
Create chooser

Running your app on a real device (continued)

4. Stop your app running on the emulator

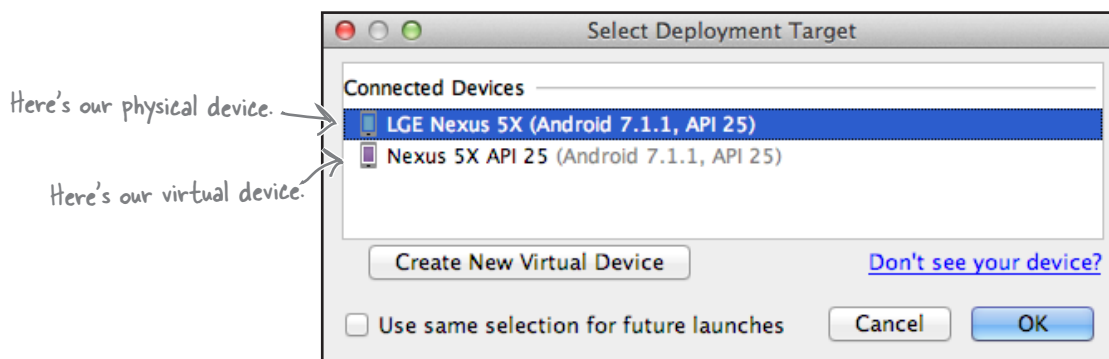
Before you can run your app on a different device, you need to stop it running on the current one (in this case the emulator). To do this, choose Run → “Stop ‘app’”, or click on the “Stop ‘app’” button in the toolbar.



Click on this button in Android Studio’s toolbar to stop the app running on the current device.

5. Run your app on the physical device

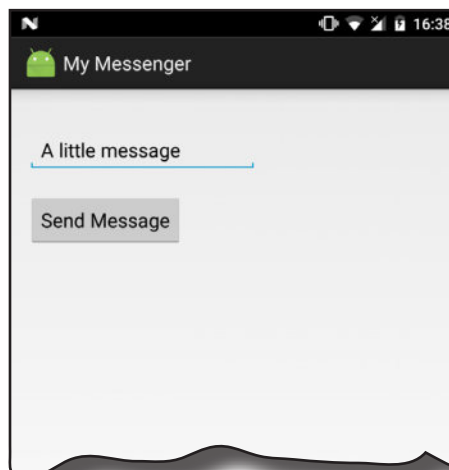
Run the app by choosing Run → “Run ‘app’”. Android Studio will ask you to choose which device you want to run your app on, so select your device from the list of available devices and click OK. Android Studio will install the app on your device and launch it.



And here’s the app running on the physical device

You should find that your app looks about the same as when you ran it through the emulator. You’ll probably find that your app installs and runs quicker too.

Now that you know how to run the apps you create on your own device, you’re all set to test the latest changes to your app.





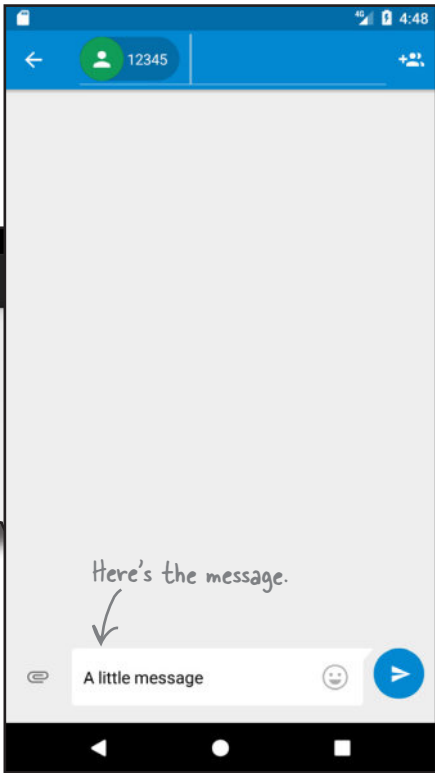
Test drive the app

Try running the app using the emulator, and then using your own device. The results you get will depend on how many activities you have on each that support using the Send action with text data.

If you have one activity

Clicking on the Send Message button will take you straight to that app.

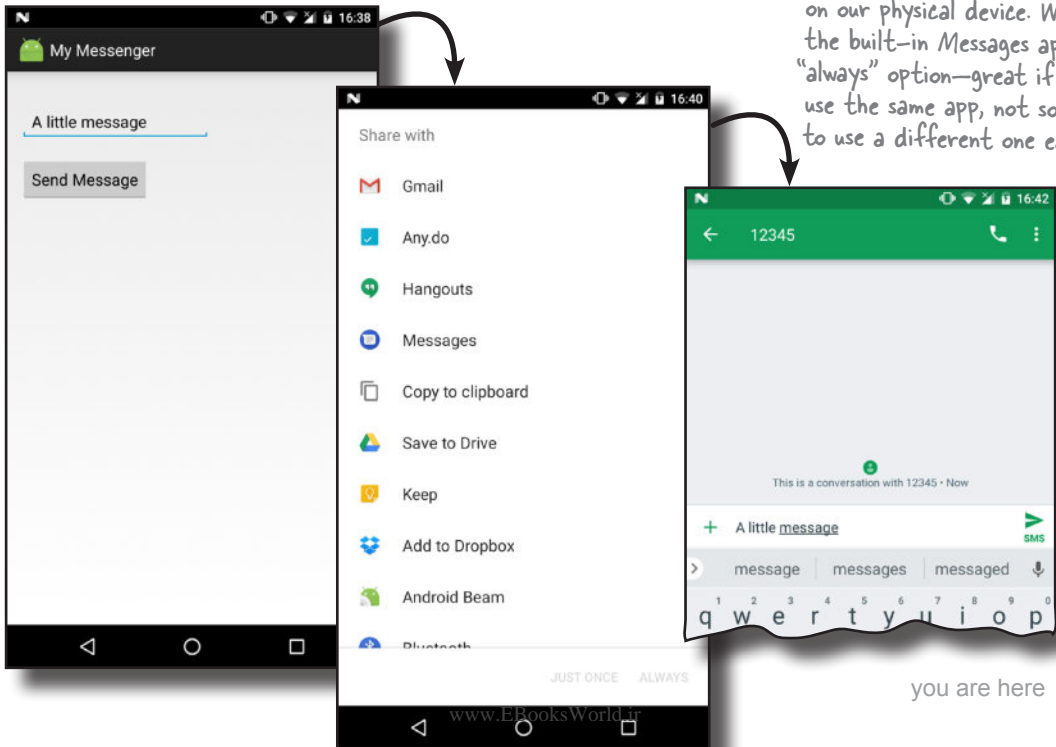
We only have one activity available on the emulator that can send messages with text data, so when we click on the Send Message button, Android starts that activity.



If you have more than one activity

Android displays a chooser and asks you to pick which one you want to use. It also asks you whether you want to use this activity just once or always. If you choose always, the next time you click on the Send Message button it uses the same activity by default.

We have lots of suitable activities available on our physical device. We decided to use the built-in Messages app. We selected the "always" option—great if we always want to use the same app, not so great if we want to use a different one each time.



What if you ALWAYS want your users to choose an activity?

You've just seen that if there's more than one activity on your device that's capable of receiving your intent, Android automatically asks you to choose which activity you want to use. It even asks you whether you want to use this activity all the time or just on this occasion.

There's just one problem with this default behavior: what if you want to *guarantee* that users can choose an activity every time they click on the Send Message button? If they've chosen the option to always use Gmail, for instance, they won't be asked if they want to use Twitter next time.

Fortunately, there's a way around this. You can create a chooser that asks users to pick an activity without giving them the option to always use that activity.

Intent.createChooser() displays a chooser dialog

You can achieve this using the `Intent.createChooser()` method, which takes the intent you've already created and wraps it in a chooser dialog. When you use this method, the user isn't given the option of choosing a default activity—they get asked to choose one every time.

You call the `createChooser()` method like this:

```
Intent chosenIntent = Intent.createChooser(intent, "Send message via...");
```

The method takes two parameters: an intent and an optional `String` title for the chooser dialog window. The `Intent` parameter needs to describe the types of activity you want the chooser to display. You can use the same intent we created earlier, as this specifies that we want to use `ACTION_SEND` with textual data.

The `createChooser()` method returns a brand-new `Intent`. This is a new explicit intent that's targeted at the activity chosen by the user. It includes any extra information supplied by the original intent, including any text.

To start the activity the user chose, you need to call:

```
startActivity(chosenIntent);
```

Over the next couple of pages we'll take a closer look at what happens when you call the `createChooser()` method.

createChooser() allows you to specify a title for the chooser dialog, and doesn't give the user the option of selecting an activity to use by default. It also lets the user know if there are no matching activities by displaying a message.

← This is the intent you created earlier.

↑ You can pass in a title for the chooser that gets displayed at the top of the screen.

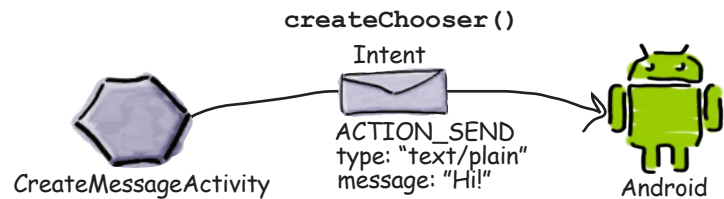


What happens when you call createChooser()

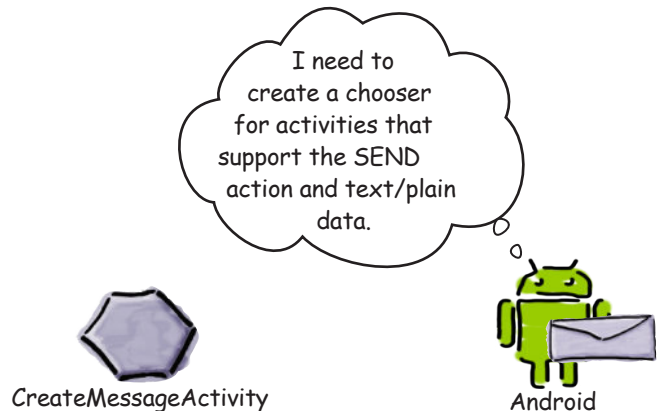
Here's what happens when you run the following two lines of code:

```
Intent chosenIntent = Intent.createChooser(intent, "Send message via...");
startActivity(chosenIntent);
```

- 1 The createChooser() method gets called.**
The method includes an intent that specifies the action and MIME type that's required.



- 2 Android checks which activities are able to receive the intent by looking at their intent filters.**
It matches on the actions, type of data, and categories they can support.



- 3 If more than one activity is able to receive the intent, Android displays an activity chooser dialog and asks the user which one to use.**
It doesn't give the user the option of always using a particular activity, and it displays "Send message via..." in the title.

If no activities are found, Android still displays the chooser but shows a message telling the user there are no apps that can perform the action.

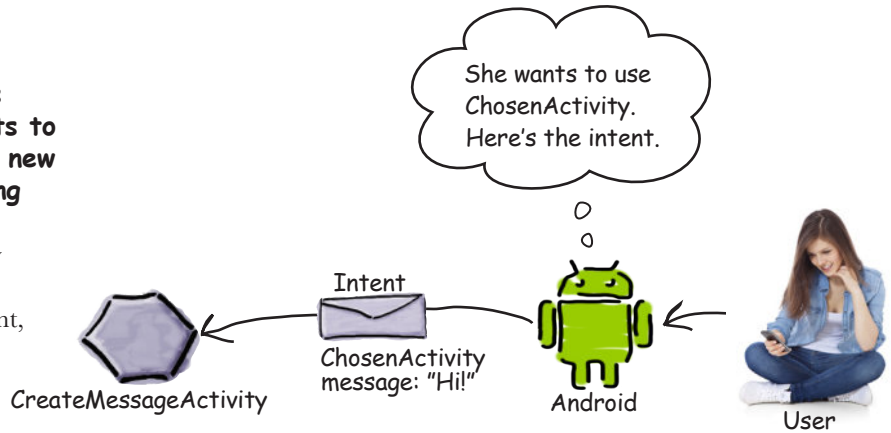


The story continues...

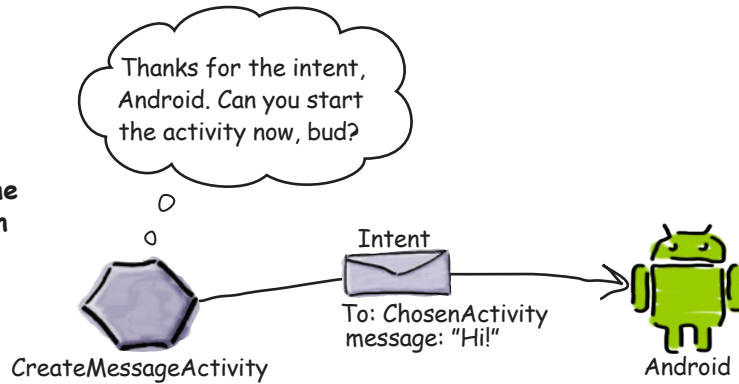
Specify action
Create chooser

- 4 When the user chooses which activity she wants to use, Android returns a new explicit intent describing the chosen activity.

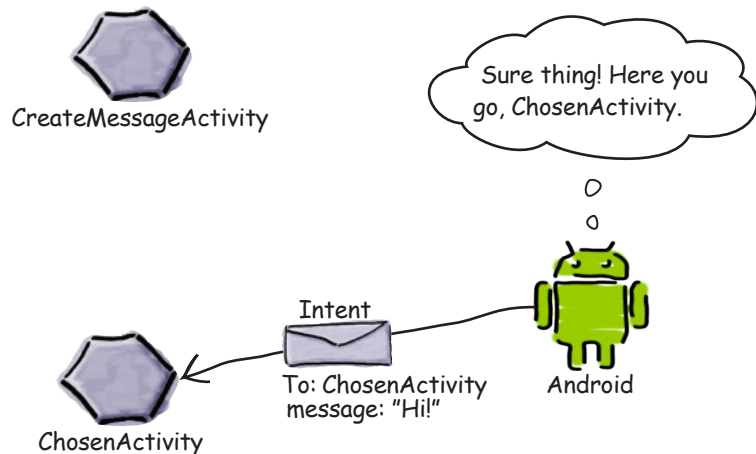
The new intent includes any extra information that was included in the original intent, such as any text.



- 5 The activity asks Android to start the activity specified in the intent.



- 6 Android starts the activity specified by the intent, and then passes it the intent.





Change the code to create a chooser

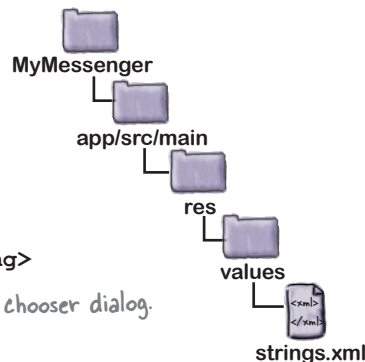
Let's change the code so that the user gets asked which activity they want to use to send a message every time they click on the Send Message button. We'll add a String resource to *strings.xml* for the chooser dialog title, and we'll update the *onSendMessage()* method in *CreateMessageActivity.java* so that it calls the *createChooser()* method.

Update strings.xml...

We want the chooser dialog to have a title of "Send message via...". Add a String called "chooser" to *strings.xml*, and give it the value Send message via... (make sure to save your changes):

```
...
<string name="chooser">Send message via...</string>
...
```

↑ This will be displayed in the chooser dialog.



...and update the onSendMessage() method

We need to change the *onSendMessage()* method so that it retrieves the value of the chooser String resource in *strings.xml*, calls the *createChooser()* method, and then starts the activity the user chooses. Update your code as follows:

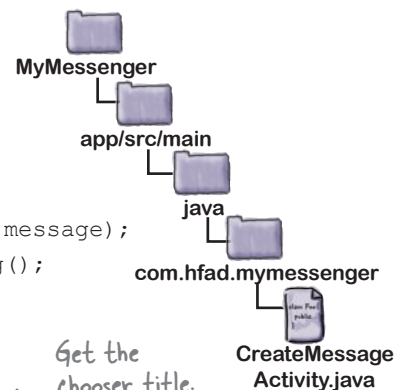
```
...
//Call onSendMessage() when the button is clicked
public void onSendMessage(View view) {
    EditText messageView = (EditText) findViewById(R.id.message);
    String messageText = messageView.getText().toString();
    Intent intent = new Intent(Intent.ACTION_SEND);
    intent.setType("text/plain");
    intent.putExtra(Intent.EXTRA_TEXT, messageText);
    String chooserTitle = getString(R.string.chooser);
    Intent chosenIntent = Intent.createChooser(intent, chooserTitle);
    startActivity(intent);
    startActivity(chosenIntent);
}
...
```

Delete this line. →

← Get the chooser title.

← Display the chooser dialog.

← Start the activity that the user selected.



The *getString()* method is used to get the value of a String resource. It takes one parameter, the ID of the resource (in our case, this is *R.string.chooser*):

```
getString(R.string.chooser);
```

← If you look in *R.java*, you'll find *chooser* in the inner class called *string*.

Now that we've updated the app, let's run the app to see our chooser in action.



Test drive the app



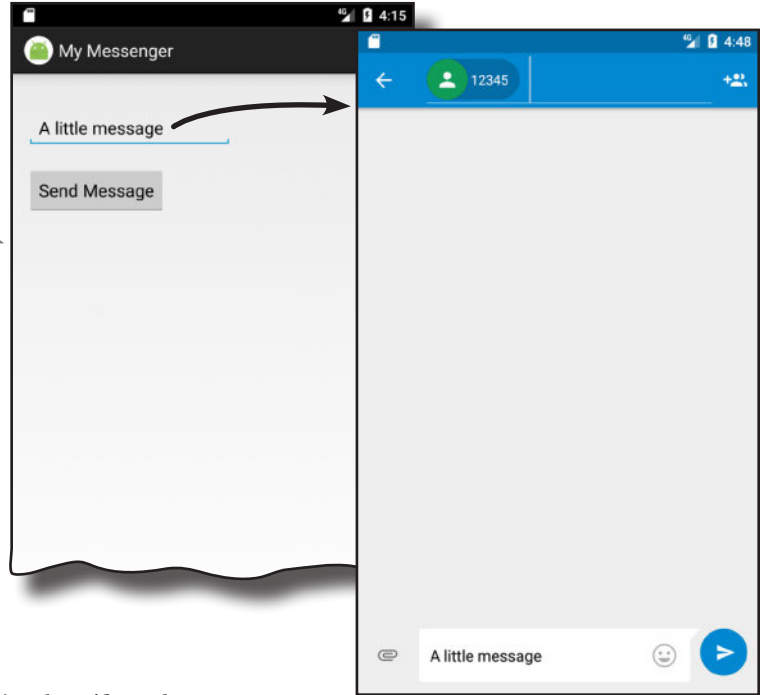
Specify action
Create chooser

Save your changes, then try running the app again on the device or the emulator.

If you have one activity

Clicking on the Send Message button will take you straight to that activity just like before.

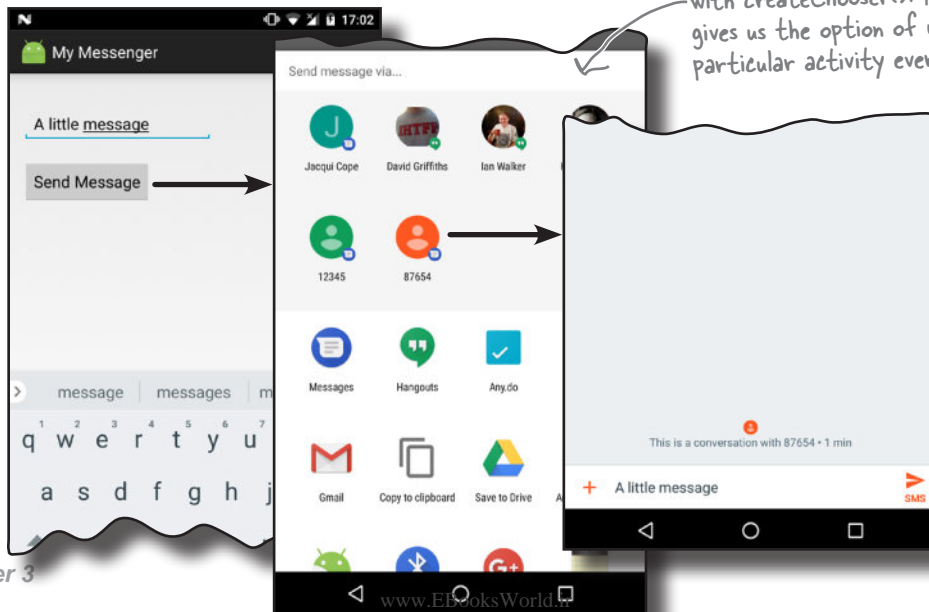
*There's no change here—
Android continues to take
you straight to the activity.*



If you have more than one activity

Android displays a chooser, but this time it doesn't ask us if we always want to use the same activity. It also displays the value of the chooser String resource in the title.

*Here's the chooser we created
with createChooser(). It no longer
gives us the option of using a
particular activity every time.*

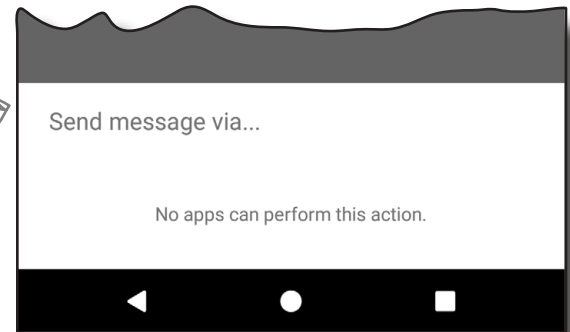




If you have NO matching activities

If you have no activities on your device that are capable of sending messages, the `createChooser()` method lets you know by displaying a message.

If you want to replicate this for yourself, try running the app in the emulator, and disable the built-in Messenger app that's on there.



there are no Dumb Questions

Q: So I can run my apps in the emulator or on a physical device. Which is best?

A: Each one has its pros and cons.

If you run apps on your physical device, they tend to load a lot quicker than using the emulator. This approach is also useful if you're writing code that interacts with the device hardware.

The emulator allows you to run apps against many different versions of Android, screen resolutions, and device specifications. It saves you from buying lots of different devices. The key thing is to make sure you test your apps thoroughly using a mixture of the emulator and physical devices before releasing them to a wider audience.

Q: Should I use implicit or explicit intents?

A: It comes down to whether you need Android to use a specific activity to perform your action, or whether you just want the action done. As an example, suppose you wanted to send an email. If you don't care which email app the user uses to send it, just as long as the email gets sent, you'd use an implicit intent. On the other hand, if you needed to pass an intent to a particular activity in your app, you'd need to use an explicit intent to explicitly say which activity needs to receive the intent.

Q: You mentioned that an activity's intent filter can specify a category as well as an action. What's the difference between the two?

A: An action specifies what an activity can do, and the category gives extra detail. We've not gone into details about adding categories because you don't often need to specify a category when you create an intent.

Q: You say that the `createChooser()` method displays a message in the chooser if there are no activities that can handle the intent. What if I'd instead used the default Android chooser and passed an implicit intent to `startActivity()`?

A: If the `startActivity()` method is given an intent where there are no matching activities, an `ActivityNotFoundException` is thrown. You can check whether any activities on the device are able to receive the intent by calling the intent's `resolveActivity()` method and checking its return value. If its return value is null, no activities on the device are able to receive the intent, so you shouldn't call `startActivity()`.



Your Android Toolbox

You've got Chapter 3 under your belt and now you've added multi-activity apps and intents to your toolbox.

You can download the full code for the chapter from <https://tinyurl.com/HeadFirstAndroid>.

BULLET POINTS

- A **task** is two or more activities chained together.
- The `<EditText>` element defines an editable text field for entering text. It inherits from the Android `View` class.
- You can add a new activity in Android Studio by choosing `File → New... → Activity`.
- Each activity you create must have an entry in `AndroidManifest.xml`.
- An **intent** is a type of message that Android components use to communicate with one another.
- An explicit intent specifies the component the intent is targeted at. You create an explicit intent using `Intent intent = new Intent(this, Target.class);`.
- To start an activity, call `startActivity(intent)`. If no activities are found, it throws an `ActivityNotFoundException`.
- Use the `putExtra()` method to add extra information to an intent.
- Use the `getIntent()` method to retrieve the intent that started the activity.
- Use the `get*Extra()` methods to retrieve extra information associated with the intent. `getStringExtra()` retrieves a `String`, `getIntExtra()` retrieves an `int`, and so on.
- An activity action describes a standard operational action an activity can perform. For example, to send a message, use `Intent.ACTION_SEND`.
- To create an implicit intent that specifies an action, use `Intent intent = new Intent(action);`.
- To describe the type of data in an intent, use the `setType()` method.
- Android resolves intents based on the named component, action, type of data, and categories specified in the intent. It compares the contents of the intent with the intent filters in each app's `AndroidManifest.xml`. An activity must have a category of `DEFAULT` if it is to receive an implicit intent.
- The `createChooser()` method allows you to override the default Android activity chooser dialog. It lets you specify a title for the dialog, and doesn't give the user the option of setting a default activity. If no activities can receive the intent it is passed, it displays a message. The `createChooser()` method returns an `Intent`.
- You retrieve the value of a `String` resource using `getString(R.string.stringname);`.

4 the activity lifecycle

Being an Activity

...so I told him that if he didn't `onStop()` soon, I'd `onDestroy()` him with a cattle prod.



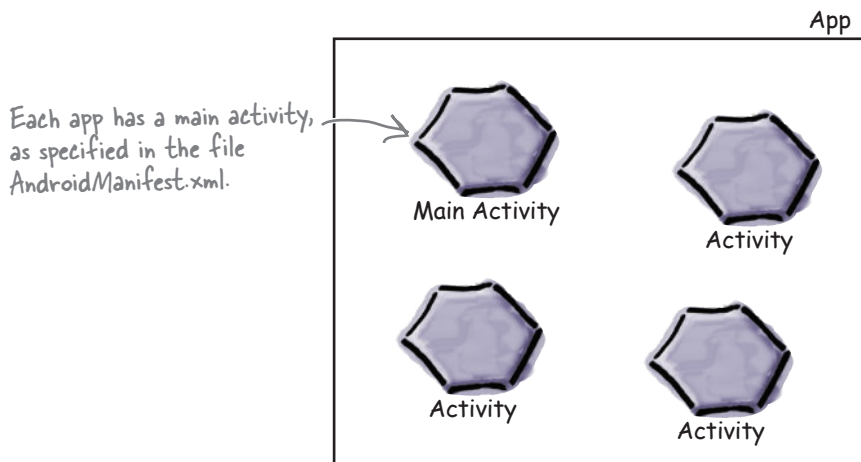
Activities form the foundation of every Android app.

So far you've seen how to create activities, and made one activity start another using an intent. But **what's really going on beneath the hood?** In this chapter, we're going to dig a little deeper into **the activity lifecycle**. What happens when an activity is **created** and **destroyed**? Which methods get called when an activity is **made visible and appears in the foreground**, and which get called when the activity **loses the focus and is hidden**? And **how do you save and restore your activity's state**? Read on to find out.

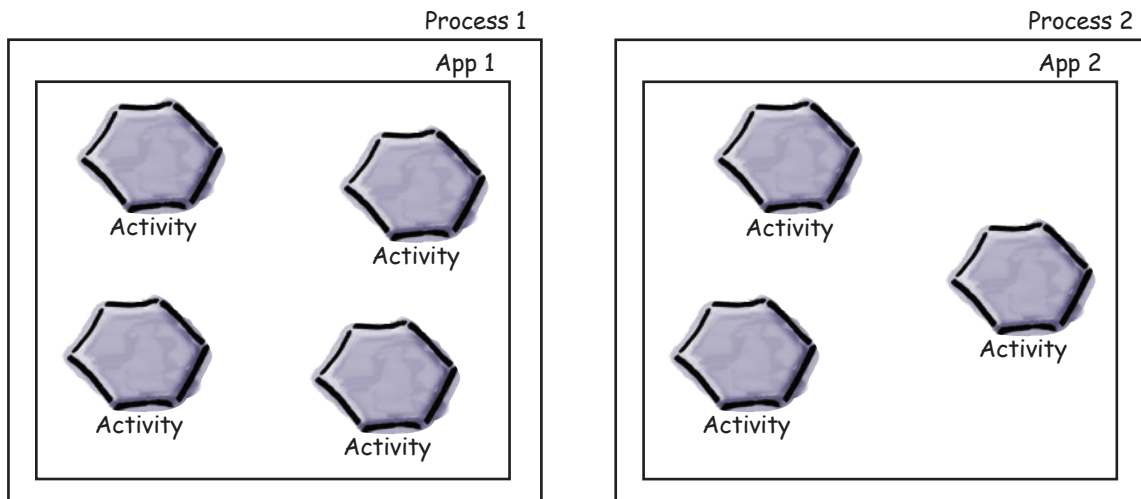
How do activities really work?

So far you've seen how to create apps that interact with the user, and apps that use multiple activities to perform tasks. Now that you have these core skills under your belt, it's time to take a deeper look at how activities *actually work*. Here's a recap of what you know so far, with a few extra details thrown in.

- ★ **An app is a collection of activities, layouts, and other resources.**
One of these activities is the main activity for the app.

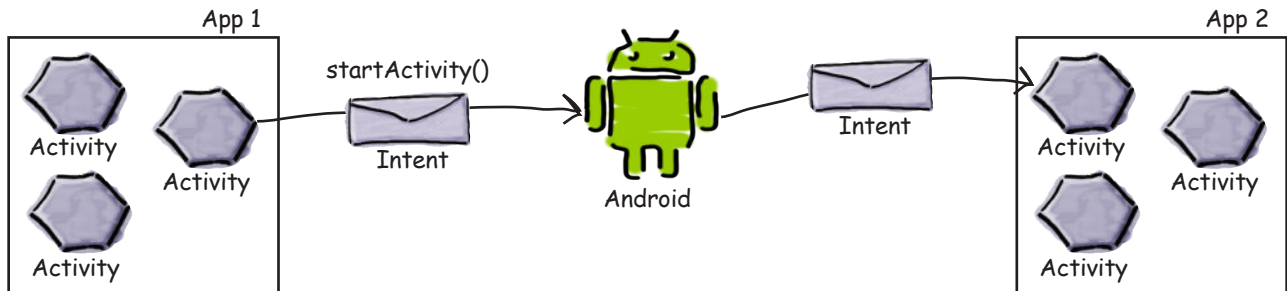


- ★ **By default, each app runs within its own process.**
This helps keep your apps safe and secure. You can read more about this in Appendix III (which covers the Android runtime, a.k.a. ART) at the back of this book.



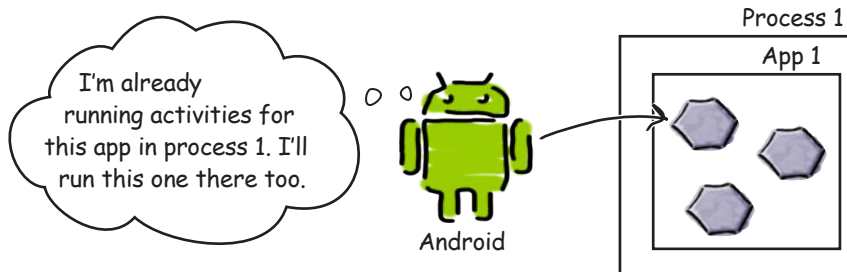
★ **Your app can start an activity in another application by passing an intent with `startActivity()`.**

The Android system knows about all the device's installed apps and their activities, and uses the intent to start the correct activity.



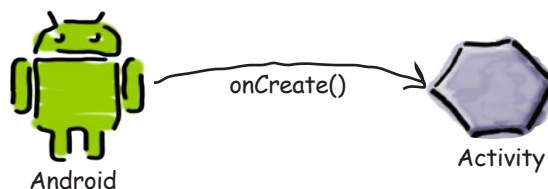
★ **When an activity needs to start, Android checks whether there's already a process for that app.**

If one exists, Android runs the activity in that process. If one doesn't exist, Android creates one.



★ **When Android starts an activity, it calls its `onCreate()` method.**

`onCreate()` is always run whenever an activity gets created.

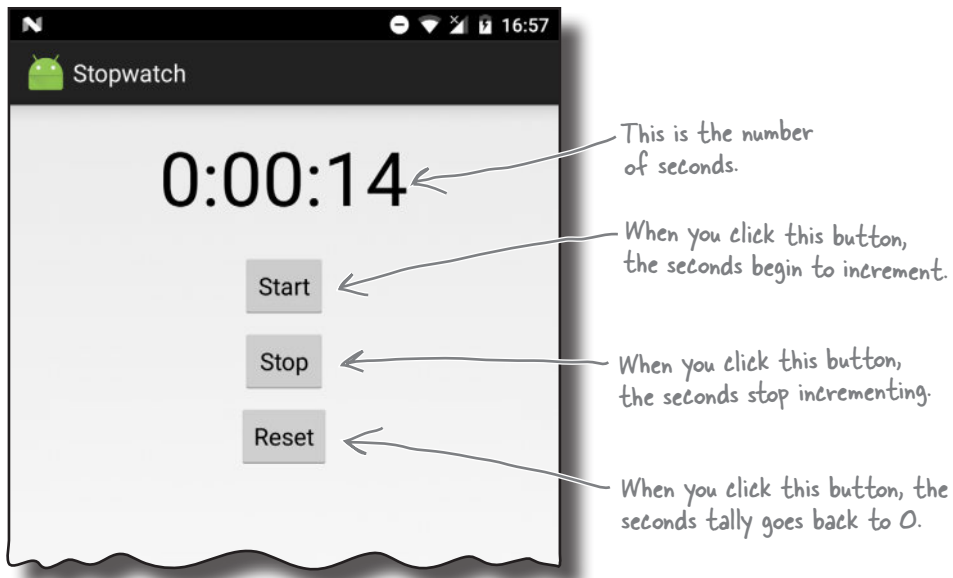


But there are still lots of things we don't yet know about how activities function. How long does an activity live for? What happens when your activity disappears from the screen? Is it still running? Is it still in memory? And what happens if your app gets interrupted by an incoming phone call? We want to be able to control the behavior of our activities in a *whole range of different circumstances*, but how?

The Stopwatch app

In this chapter, we're going to take a closer look at how activities work under the hood, common ways in which your apps can break, and how you can fix them using the activity lifecycle methods. We're going to explore the lifecycle methods using a simple Stopwatch app as an example.

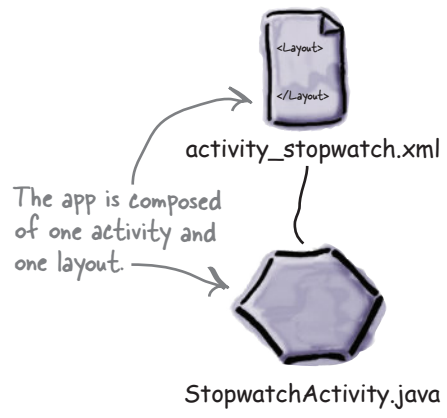
The Stopwatch app consists of a single activity and a single layout. The layout includes a text view showing you how much time has passed, a Start button that starts the stopwatch, a Stop button that stops it, and a Reset button that resets the timer value to 0.



Create a new project for the Stopwatch app

You have enough experience under your belt to build the app without much guidance from us. We're going to give you just enough code so you can build the app yourself, and then you can see what happens when you try to run it.

Start off by creating a new Android project for an application named "Stopwatch" with a company domain of "hfad.com", making the package name `com.hfad.stopwatch`. The minimum SDK should be API 19 so it can run on most devices. You'll need an empty activity called "StopwatchActivity" and a layout called "activity_stopwatch". **Make sure you uncheck the Backwards Compatibility (AppCompat) checkbox.**



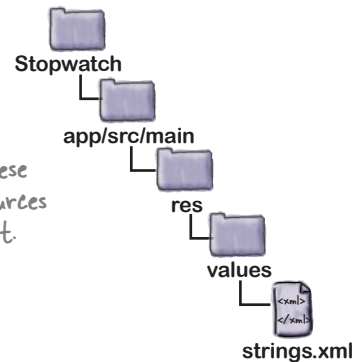
Add String resources

We're going to use three String values in our stopwatch layout, one for the text value of each button. These values are String resources, so they need to be added to *strings.xml*. Add the String values below to your version of *strings.xml*:

```
...
    <string name="start">Start</string>
    <string name="stop">Stop</string>
    <string name="reset">Reset</string>
...

```

We'll use these String resources in our layout.



Next, let's update the code for our layout.

Update the stopwatch layout code

Here's the XML for the layout. It describes a single text view that's used to display the timer, and three buttons to control the stopwatch. Replace the XML currently in *activity_stopwatch.xml* with the XML shown here:

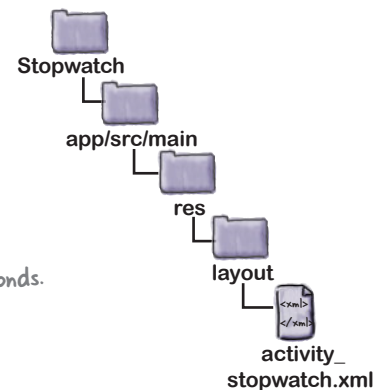
```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    android:padding="16dp"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

    <TextView
        android:id="@+id/time_view"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="center_horizontal"
        android:textAppearance="@android:style/TextAppearance.Large"
        android:textSize="56sp" />

```

We'll use this text view to display the number of seconds.

These attributes make the stopwatch timer nice and big.



The layout code continues over the page.

The layout code (continued)

<Button

```

    android:id="@+id/start_button"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_gravity="center_horizontal"
    android:layout_marginTop="20dp"
    android:onClick="onClickStart"
    android:text="@string/start" />
  
```

← This code is for the Start button.

← When it gets clicked, the Start button calls the `onClickStart()` method.

<Button

```

    android:id="@+id/stop_button"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_gravity="center_horizontal"
    android:layout_marginTop="8dp"
    android:onClick="onClickStop"
    android:text="@string/stop" />
  
```

← This is for the Stop button.

← When it gets clicked, the Stop button calls the `onClickStop()` method.

<Button

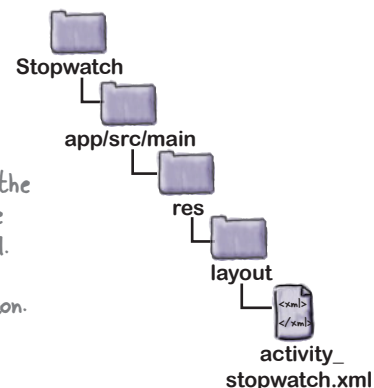
```

    android:id="@+id/reset_button"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_gravity="center_horizontal"
    android:layout_marginTop="8dp"
    android:onClick="onClickReset"
    android:text="@string/reset" />
  
```

← This is for the Reset button.

← When it gets clicked, the Reset button calls the `onClickReset()` method.

</LinearLayout>

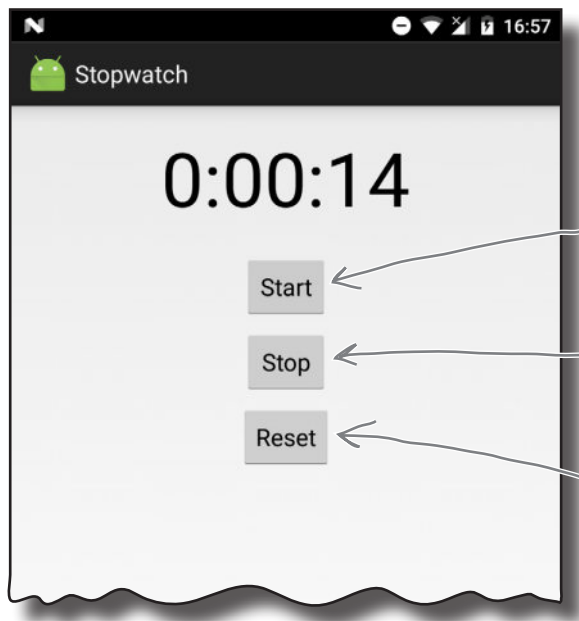


Make sure you update the layout and strings.xml in your app before continuing.

The layout is now done! Next, let's move on to the activity.

How the activity code will work

The layout defines three buttons that we'll use to control the stopwatch. Each button uses its `onClick` attribute to specify which method in the activity should run when the button is clicked. When the Start button is clicked, the `onClickStart()` method gets called, when the Stop button is clicked the `onClickStop()` method gets called, and when the Reset button is clicked the `onClickReset()` method gets called. We'll use these methods to start, stop, and reset the stopwatch.



When you click this button, the `onClickStart()` method is called.

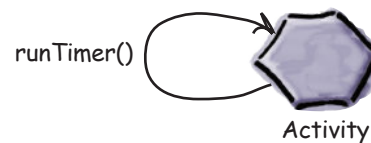
When you click this button, the `onClickStop()` method is called.

When you click this button, the `onClickReset()` method is called.

We'll update the stopwatch using a method we'll create called `runTimer()`. The `runTimer()` method will run code every second to check whether the stopwatch is running, and, if it is, increment the number of seconds and display the number of seconds in the text view.

To help us with this, we'll use two private variables to record the state of the stopwatch. We'll use an `int` called `seconds` to track how many seconds have passed since the stopwatch started running, and a `boolean` called `running` to record whether the stopwatch is currently running.

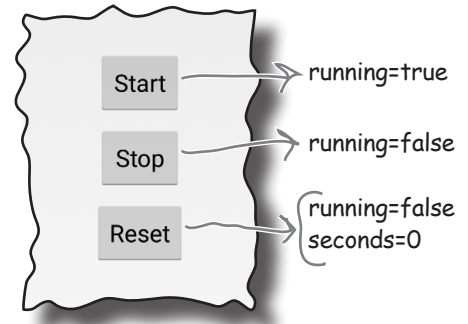
We'll start by writing the code for the buttons, and then we'll look at the `runTimer()` method.



Add code for the buttons

When the user clicks on the Start button, we'll set the running variable to `true` so that the stopwatch will start. When the user clicks on the Stop button, we'll set `running` to `false` so that the stopwatch stops running. If the user clicks on the Reset button, we'll set `running` to `false` and `seconds` to `0` so that the stopwatch is reset and stops running.

To do all that, replace the contents of `StopwatchActivity.java` with the code below:



```
package com.hfad.stopwatch;
```

```
import android.app.Activity;
import android.os.Bundle;
import android.view.View;
```

Make sure your activity extends the Activity class.

```
public class StopwatchActivity extends Activity {
```

```
    private int seconds = 0;
    private boolean running;
```

Use the seconds and running variables to record the number of seconds passed and whether the stopwatch is running.

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

```
//Start the stopwatch running when the Start button is clicked.
```

```
public void onClickStart(View view) {
    running = true;
}
```

This gets called when the Start button is clicked.

```
//Stop the stopwatch running when the Stop button is clicked.
```

```
public void onClickStop(View view) {
    running = false;
}
```

This gets called when the Stop button is clicked.

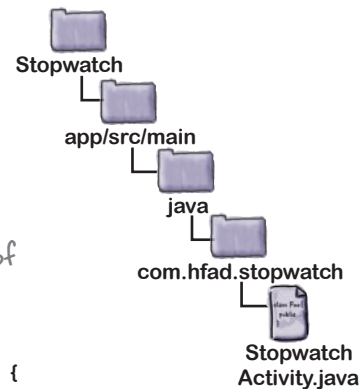
```
//Reset the stopwatch when the Reset button is clicked.
```

```
public void onClickReset(View view) {
    running = false;
    seconds = 0;
}
```

This gets called when the Reset button is clicked.

Stop the stopwatch and set the seconds to 0.

```
}
```



The runTimer() method

The next thing we need to do is create the `runTimer()` method. This method will get a reference to the text view in the layout; format the contents of the `seconds` variable into hours, minutes, and seconds; and then display the results in the text view. If the running variable is set to `true`, it will increment the `seconds` variable.

The code for the `runTimer()` method is below. We'll add it to `StopwatchActivity.java` in a few pages:

```
private void runTimer() {
    final TextView timeView = (TextView) findViewById(R.id.time_view);
    ...
    int hours = seconds/3600;
    int minutes = (seconds%3600)/60;
    int secs = seconds%60;
    String time = String.format(Locale.getDefault(),
        "%d:%02d:%02d", hours, minutes, secs);
    timeView.setText(time);
    if (running) {
        seconds++;
    }
    ...
}
```

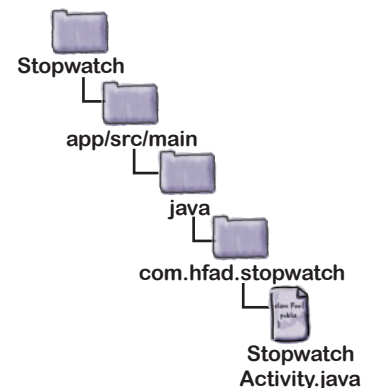
Get the text view.

We've left out a bit of code here. We'll look at it on the next page.

Format the seconds into hours, minutes, and seconds. This is plain Java code.

Set the text view text.

If running is true, increment the seconds variable.



We need this code to keep looping so that it increments the `seconds` variable and updates the text view every second. We need to do this in such a way that we don't block the main Android thread.

In non-Android Java programs, you can perform tasks like this using a background thread. In Androidville, that approach won't work—only the main Android thread can update the user interface, and if any other thread tries to do so, you get a `CalledFromWrongThreadException`.

The solution is to use a `Handler`. We'll look at this technique on the next page.

Handlers allow you to schedule code

A `Handler` is an Android class you can use to schedule code that should be run at some point in the future. You can also use it to post code that needs to run on a different thread than the main Android thread. In our case, we're going to use a `Handler` to schedule the stopwatch code to run every second.

To use the `Handler`, you wrap the code you wish to schedule in a `Runnable` object, and then use the `Handler.post()` and `postDelayed()` methods to specify when you want the code to run. Let's take a closer look at these methods.

The `post()` method

The `post()` method posts code that needs to be run as soon as possible (which is usually almost immediately). This method takes one parameter, an object of type `Runnable`. A `Runnable` object in Androidville is just like a `Runnable` in plain old Java: a job you want to run. You put the code you want to run in the `Runnable`'s `run()` method, and the `Handler` will make sure the code is run as soon as possible. Here's what the method looks like:

```
final Handler handler = new Handler();
handler.post(Runnable);
```

← You put the code you want to run in the `Runnable`'s `run()` method.

The `postDelayed()` method

The `postDelayed()` method works in a similar way to the `post()` method except that you use it to post code that should be run in the future. The `postDelayed()` method takes two parameters: a `Runnable` and a `long`. The `Runnable` contains the code you want to run in its `run()` method, and the `long` specifies the number of milliseconds you wish to delay the code by. The code will run as soon as possible after the delay. Here's what the method looks like:

```
final Handler handler = new Handler();
handler.postDelayed(Runnable, long);
```

← Use this method to delay running code by a specified number of milliseconds.

On the next page, we'll use these methods to update the stopwatch every second.

The full runTimer() code

To update the stopwatch, we're going to repeatedly schedule code using the `Handler` with a delay of 1 second each time. Each time the code runs, we'll increment the `seconds` variable and update the text view.

Here's the full code for the `runTimer()` method, which we'll add to `StopwatchActivity.java` in a couple of pages:

```
private void runTimer() {
    final TextView timeView = (TextView)findViewById(R.id.time_view);
    final Handler handler = new Handler(); ← Create a new Handler.
    handler.post(new Runnable() { ← Call the post() method, passing in a new Runnable. The post()
        @Override                               method processes code without a delay, so the code in the
        public void run() {                       Runnable will run almost immediately.
            int hours = seconds/3600;
            int minutes = (seconds%3600)/60;
            int secs = seconds%60;
            String time = String.format(Locale.getDefault(), ← The Runnable run()
                "%d:%02d:%02d", hours, minutes, secs);      method contains the code
            timeView.setText(time);                    you want to run—in our
            if (running) {                             case, the code to update
                seconds++;                               the text view.
            }
            handler.postDelayed(this, 1000); ← Post the code in the Runnable to be run again
        }                                             after a delay of 1,000 milliseconds. As this line
    });                                             of code is included in the Runnable run() method,
}                                                 it will keep getting called.
```

Using the `post()` and `postDelayed()` methods in this way means that the code will run as soon as possible after the required delay, which in practice means almost immediately. While this means the code will lag slightly over time, it's accurate enough for the purposes of exploring the lifecycle methods in this chapter.

We want the `runTimer()` method to start running when `StopwatchActivity` gets created, so we'll call it in the activity `onCreate()` method:

```
protected void onCreate(Bundle savedInstanceState) {
    ...
    runTimer();
}
```

We'll show you the full code for `StopwatchActivity` on the next page.

The full StopwatchActivity code

Here's the full code for *StopwatchActivity.java*. Update your code to match our changes below.

```
package com.hfad.stopwatch;
```

```
import android.app.Activity;
```

```
import android.os.Bundle;
```

```
import android.view.View;
```

```
import java.util.Locale;
```

```
import android.os.Handler;
```

```
import android.widget.TextView;
```

We're using these extra classes so we need to import them.

```
public class StopwatchActivity extends Activity {
```

```
    //Number of seconds displayed on the stopwatch.
```

```
    private int seconds = 0;
```

```
    //Is the stopwatch running?
```

```
    private boolean running;
```

Use the seconds and running variables to record the number of seconds passed and whether the stopwatch is running.

```
    @Override
```

```
    protected void onCreate(Bundle savedInstanceState) {
```

```
        super.onCreate(savedInstanceState);
```

```
        setContentView(R.layout.activity_stopwatch);
```

```
        runTimer();
```

We're using a separate method to update the stopwatch. We're starting it when the activity is created.

```
    }
```

```
    //Start the stopwatch running when the Start button is clicked.
```

```
    public void onClickStart(View view) {
```

```
        running = true;
```

```
    }
```

This gets called when the Start button is clicked.

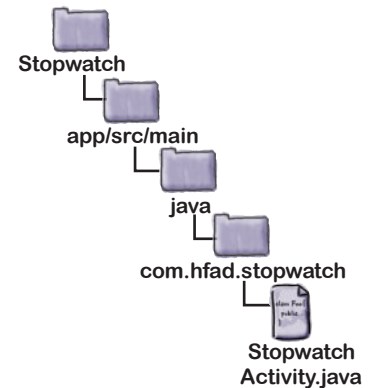
```
    //Stop the stopwatch running when the Stop button is clicked.
```

```
    public void onClickStop(View view) {
```

```
        running = false;
```

```
    }
```

This gets called when the Stop button is clicked.



The activity code (continued)

```

//Reset the stopwatch when the Reset button is clicked.
public void onClickReset(View view) {
    running = false;
    seconds = 0;
}

//Sets the number of seconds on the timer.
private void runTimer() {
    final TextView timeView = (TextView) findViewById(R.id.time_view);
    final Handler handler = new Handler();
    handler.post(new Runnable() {
        @Override
        public void run() {
            int hours = seconds/3600;
            int minutes = (seconds%3600)/60;
            int secs = seconds%60;
            String time = String.format(Locale.getDefault(),
                "%d:%02d:%02d", hours, minutes, secs);
            timeView.setText(time);
            if (running) {
                seconds++;
            }
            handler.postDelayed(this, 1000);
        }
    });
}

```

This gets called when the Reset button is clicked.

Stop the stopwatch and set the seconds to 0.

Get the text view.

Use a Handler to post code.

Format the seconds into hours, minutes, and seconds.

Set the text view text.

If running is true, increment the seconds variable.

Post the code again with a delay of 1 second.

Do this!

Make sure you update your activity code to reflect these changes.

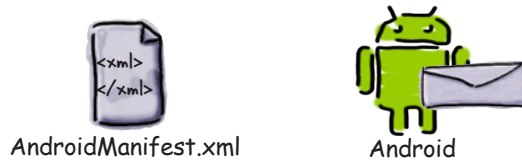
Let's look at what happens when the code runs.

What happens when you run the app

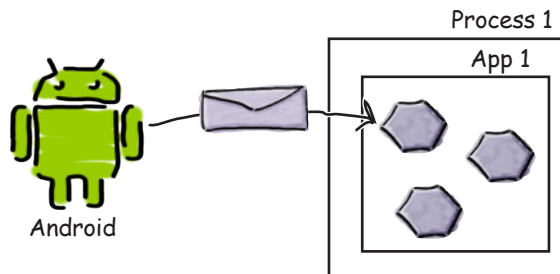
- 1 The user decides she wants to run the app.**
On her device, she clicks on the app's icon.



- 2 An intent is constructed to start this activity using `startActivity(intent)`.**
The *AndroidManifest.xml* file for the app specifies which activity to use as the launch activity.



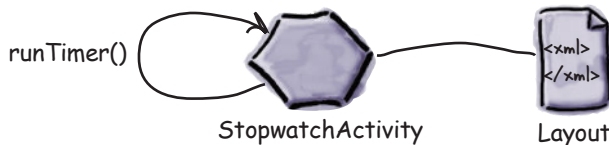
- 3 Android checks to see whether there's already a process running for the app, and if not, creates a new process.**
It then creates a new activity object—in this case, for `StopwatchActivity`.



The story continues

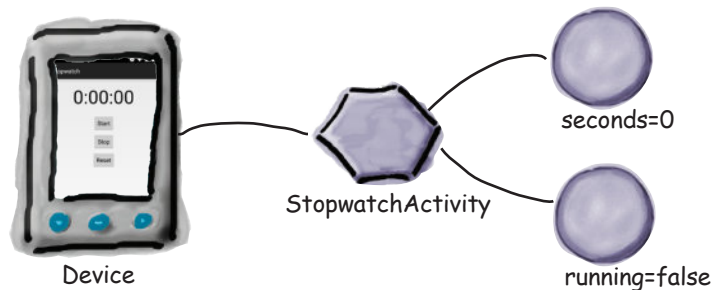
4 The onCreate() method in the activity gets called.

The method includes a call to `setContentView()`, specifying a layout, and then starts the stopwatch with `runTimer()`.



5 When the onCreate() method finishes, the layout gets displayed on the device.

The `runTimer()` method uses the `seconds` variable to determine what text to display in the text view, and uses the `running` variable to determine whether to increment the number of seconds. As `running` is initially `false`, the number of seconds isn't incremented.



there are no Dumb Questions

Q: Why does Android run each app inside a separate process?

A: For security and stability. This approach prevents one app from accessing the data of another. It also means if one app crashes, it won't take others down with it.

Q: Why have an `onCreate()` method in our activity? Why not just put that code inside a constructor?

A: Android needs to set up the environment for the activity after it's constructed. Once the environment is ready, Android calls `onCreate()`. That's why code to set up the screen goes inside `onCreate()` instead of a constructor.

Q: Couldn't I just write a loop in `onCreate()` to keep updating the timer?

A: No, `onCreate()` needs to finish before the screen will appear. An endless loop would prevent that from happening.

Q: `runTimer()` looks really complicated. Do I really need to do all this?

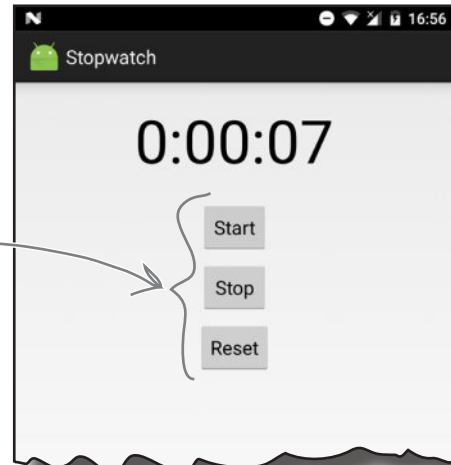
A: It's a little complex, but whenever you need to post code that runs in a loop, the code will look similar to `runTimer()`.



Test drive the app

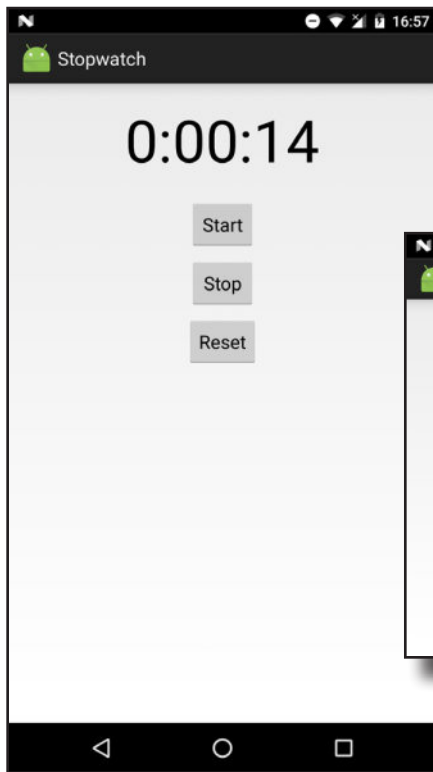
When we run the app in the emulator, the app works great. We can start, stop, and reset the stopwatch without any problems at all—the app works just as you’d expect.

These buttons work as you’d expect. The Start button starts the stopwatch, the Stop button stops it, and the Reset button sets the stopwatch back to 0.

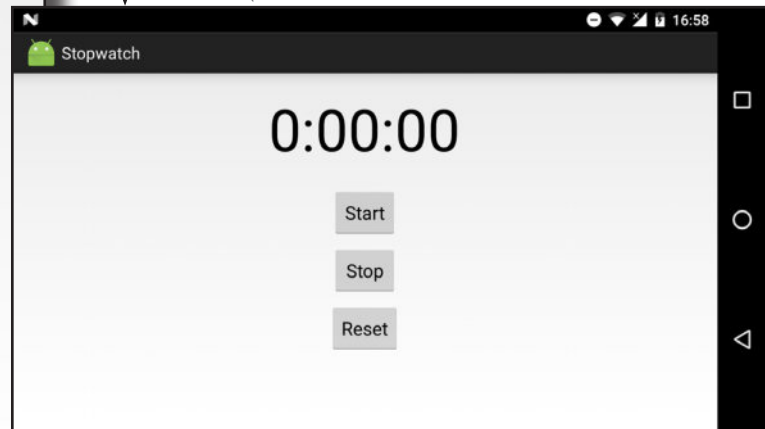


But there’s just one problem...

When we ran the app on a physical device, the app worked OK until someone rotated the device. When the device was rotated, the stopwatch set itself back to 0.



The stopwatch was running, but reset to 0 when the device was rotated.



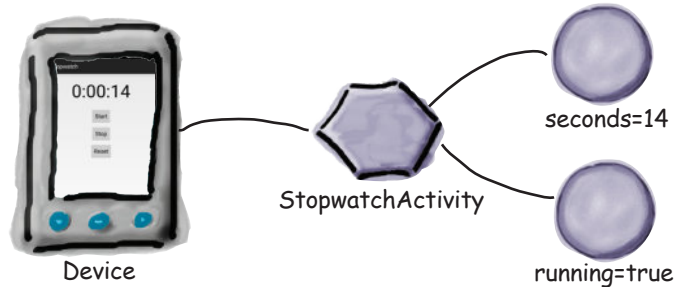
In Androidville, it’s surprisingly common for apps to break when you rotate the device. Before we fix the problem, let’s take a closer look at what caused it.

What just happened?

So why did the app break when the user rotated the screen?
Let's take a closer look at what really happened.

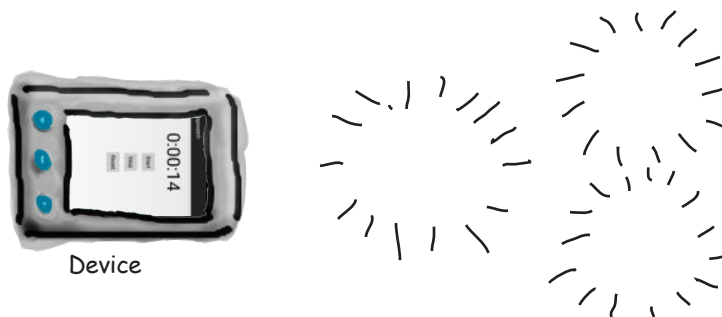
- 1 The user starts the app, and clicks on the Start button to set the stopwatch going.**

The `runTimer()` method starts incrementing the number of seconds displayed in the `time_view` text view using the `seconds` and `running` variables.



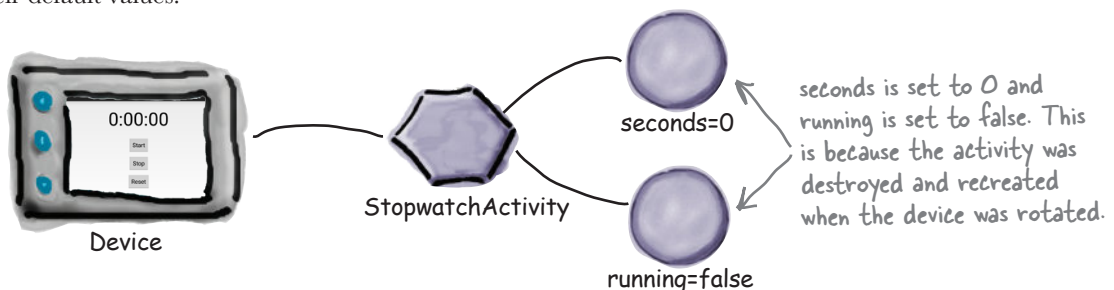
- 2 The user rotates the device.**

Android sees that the screen orientation and screen size has changed, and it destroys the activity, including any variables used by the `runTimer()` method.



- 3 StopwatchActivity is then recreated.**

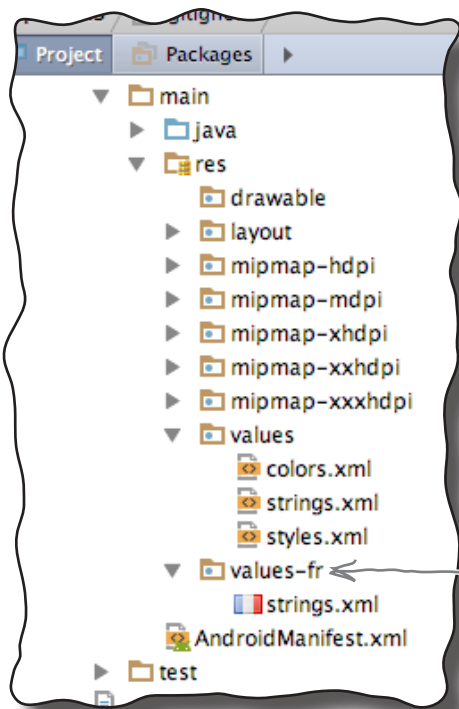
The `onCreate()` method runs again, and the `runTimer()` method gets called. As the activity has been recreated, the `seconds` and `running` variables are set to their default values.



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. A different layout might need to be used if the device screen is landscape rather than portrait, for instance, and a different set of String values might need to be used if the locale is France.



Android apps can contain multiple resource files in the `app/src/main/res` folder. For instance, if the device locale is set to France, Android will use the `strings.xml` file in the `values-fr` folder.

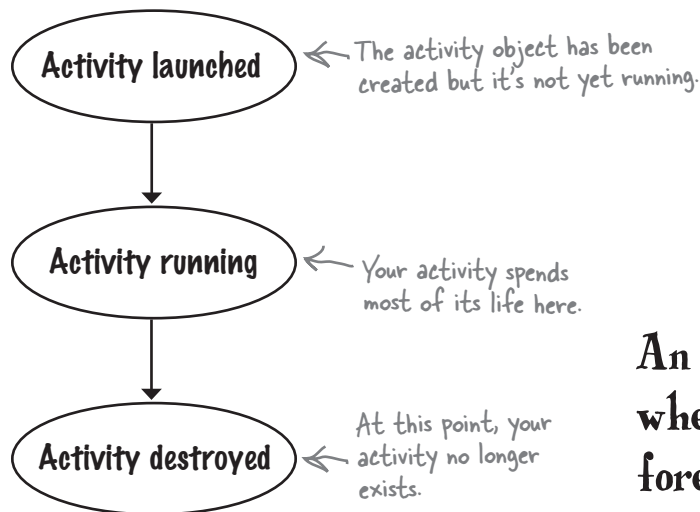
The device configuration includes options specified by the user (such as the locale), and options relating to the physical device (such as the orientation and screen size). A change to any of these options results in the activity being destroyed and then recreated.

When the device configuration changes, anything that displays a user interface needs to be updated to match the new configuration. If you rotate your device, Android spots that the screen orientation and screen size have 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 states of an activity

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 your activity inherits, and which you can override if necessary.

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.

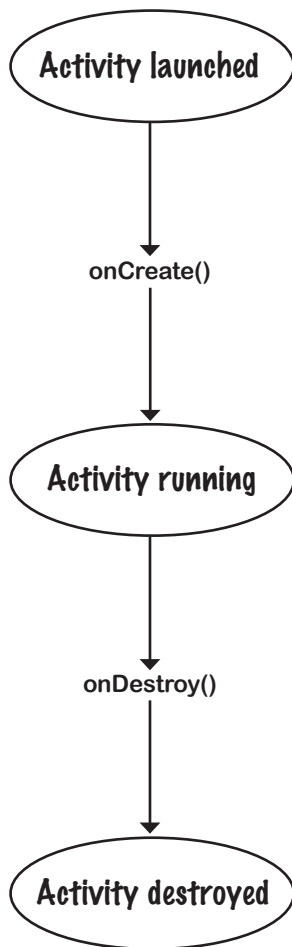
An activity is running when it's in the foreground of the screen.

`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, but at this point we're just focusing on the `onCreate()` and `onDestroy()` methods.



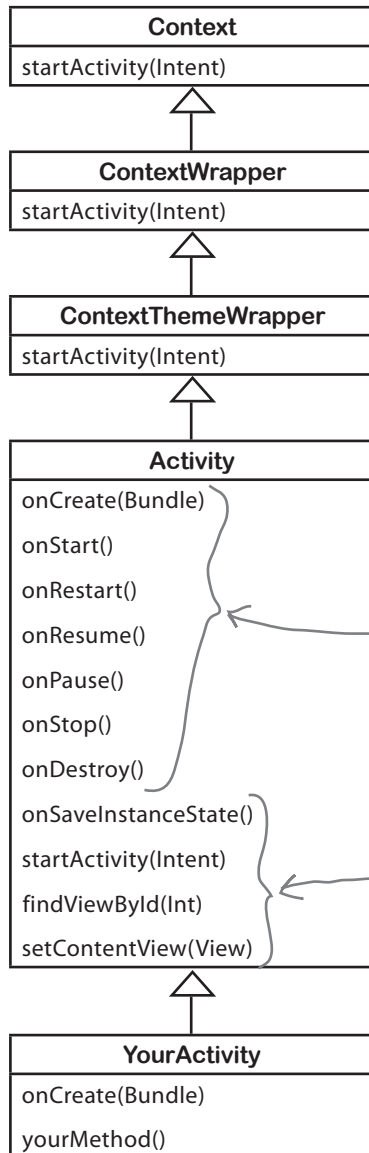
- 1 The activity gets launched.**
The activity object is created and its constructor is run.
- 2 The `onCreate()` method runs immediately after the activity is launched.**
The `onCreate()` method is where any initialization code should go, as this method always gets called after the activity has launched 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

As you saw earlier in the book, your activity extends the `android.app.Activity` class. It's this class that gives your activity access to the Android lifecycle methods. Here's a diagram showing the class hierarchy:



Context abstract class

(`android.content.Context`)

An interface to global information about the application environment. Allows access to application resources, classes, and operations.

ContextWrapper class

(`android.content.ContextWrapper`)

A proxy implementation for the `Context`.

ContextThemeWrapper class

(`android.view.ContextThemeWrapper`)

Allows you to modify the theme from what's in the `ContextWrapper`.

Activity class

(`android.app.Activity`)

The `Activity` class implements default versions of the lifecycle methods. It also defines methods such as `findViewById(Int)` and `setContentView(View)`.

These are the activity lifecycle methods. You'll find out 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.

YourActivity class

(`com.hfad.foo`)

Most of the behavior of your activity is handled by superclass methods your activity inherits. All you do is override the methods you need.

Now that you know more about the activity lifecycle methods, let's see how you deal with device configuration changes.

Save the current state...

As you saw, our app went wrong when the user rotated the screen. The activity was destroyed and recreated, which meant that local variables used by the activity were lost. So how do we get around this issue?

The best way of dealing with configuration changes is to save the current state of the activity, and then reinstate it in the `onCreate()` method of the activity.

To save the activity's current state, you need to implement the `onSaveInstanceState()` method. This method gets called before the activity gets destroyed, which means you get an opportunity to save any values you want to retain before they get lost.

The `onSaveInstanceState()` method takes one parameter, a `Bundle`. A `Bundle` allows you to gather together different types of data into a single object:

```
public void onSaveInstanceState(Bundle savedInstanceState) {
}
```

The `onCreate()` method gets passed the `Bundle` as a parameter. This means that if you add the values of the `running` and `seconds` variables to the `Bundle`, the `onCreate()` method will be able to pick them up when the activity gets recreated. To do this, you use `Bundle` methods to add name/value pairs to the `Bundle`. These methods take the form:

```
bundle.put*("name", value)
```

where `bundle` is the name of the `Bundle`, `*` is the type of value you want to save, and `name` and `value` are the name and value of the data. As an example, to add the `seconds` `int` value to the `Bundle`, you'd use:

```
bundle.putInt("seconds", seconds);
```

You can save multiple name/value pairs of data to the `Bundle`.

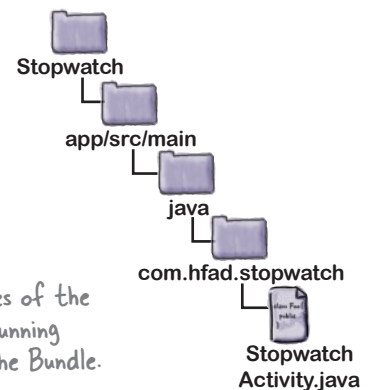
Here's our `onSaveInstanceState()` method in full (we'll add it to `StopwatchActivity.java` a couple of pages ahead):

```
@Override
public void onSaveInstanceState(Bundle savedInstanceState) {
    savedInstanceState.putInt("seconds", seconds);
    savedInstanceState.putBoolean("running", running);
}
```

Once you've saved variable values to the `Bundle`, you can use them in our `onCreate()` method.



The `onSaveInstanceState()` method gets called before `onDestroy()`. It gives you a chance to save your activity's state before the activity is destroyed.



Save the values of the `seconds` and `running` variables to the `Bundle`.

...then restore the state in onCreate()

As we said earlier, the `onCreate()` method takes one parameter, a `Bundle`. If the activity's being created from scratch, this parameter will be null. If, however, the activity's being recreated and there's been a prior call to `onSaveInstanceState()`, the `Bundle` object used by `onSaveInstanceState()` will get passed to the activity:

```
protected void onCreate(Bundle savedInstanceState) {
    ...
}
```

You can get values from `Bundle` by using methods of the form:

`bundle.get*("name");` *Instead of *, use Int, String, and so on, to specify the type of data you want to get.*

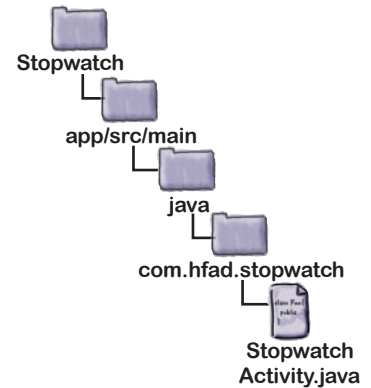
where `bundle` is the name of the `Bundle`, `*` is the type of value you want to get, and `name` is the name of the name/value pair you specified on the previous page. As an example, to get the `seconds` value from the `Bundle`, you'd use:

```
int seconds = bundle.getInt("seconds");
```

Putting all of this together, here's what our `onCreate()` method now looks like (we'll add this to `StopwatchActivity.java` on the next page):

```
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_stopwatch);
    if (savedInstanceState != null) {
        seconds = savedInstanceState.getInt("seconds");
        running = savedInstanceState.getBoolean("running");
    }
    runTimer();
}
```

Retrieve the values of the seconds and running variables from the Bundle.



We'll look at the full code to save and restore `StopwatchActivity`'s state on the next page.

The updated StopwatchActivity code

We've updated our StopwatchActivity code so that if the user rotates the device, its state gets saved via the onSaveInstanceState() method, and restored via the onCreate() method. Update your version of StopwatchActivity.java to include our changes (below in bold):

...

```
public class StopwatchActivity extends Activity {
    //Number of seconds displayed on the stopwatch.
    private int seconds = 0;
    //Is the stopwatch running?
    private boolean running;

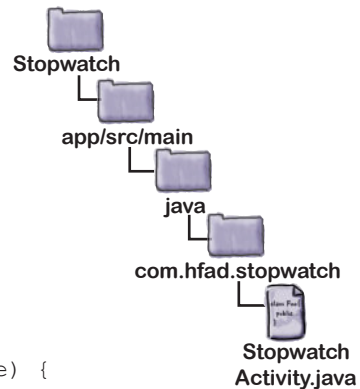
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_stopwatch);
        if (savedInstanceState != null) {
            seconds = savedInstanceState.getInt("seconds");
            running = savedInstanceState.getBoolean("running");
        }
        runTimer();
    }

    @Override
    public void onSaveInstanceState(Bundle savedInstanceState) {
        savedInstanceState.putInt("seconds", seconds);
        savedInstanceState.putBoolean("running", running);
    }
}
```

... ← We've left out some of the activity code, as we don't need to change it.

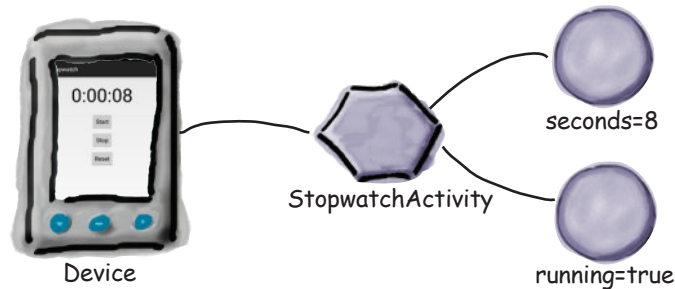
Save the state of the variables in the activity's onSaveInstanceState() method.

Restore the activity's state by getting values from the Bundle.

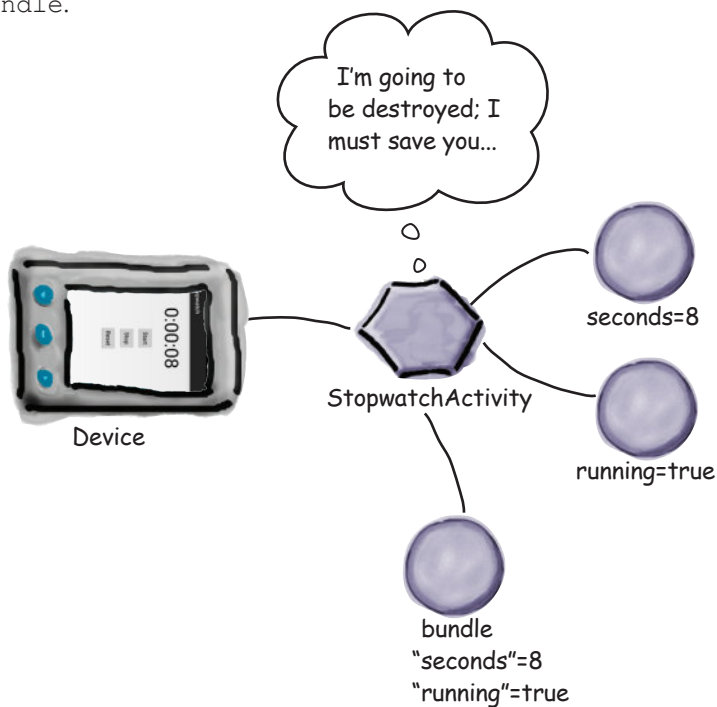


What happens when you run the app

- 1 The user starts the app, and clicks on the Start button to set the stopwatch going.**
 The `runTimer()` method starts incrementing the number of seconds displayed in the `time_view` text view.

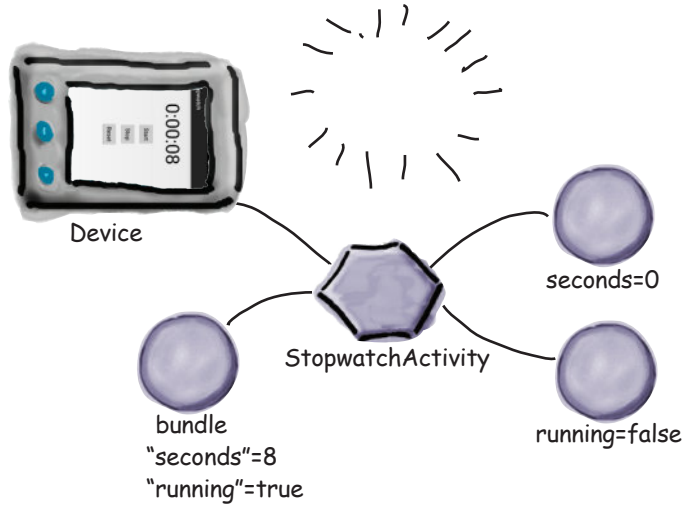


- 2 The user rotates the device.**
 Android views this as a configuration change, and gets ready to destroy the activity. Before the activity is destroyed, `onSaveInstanceState()` gets called. The `onSaveInstanceState()` method saves the `seconds` and `running` values to a `Bundle`.

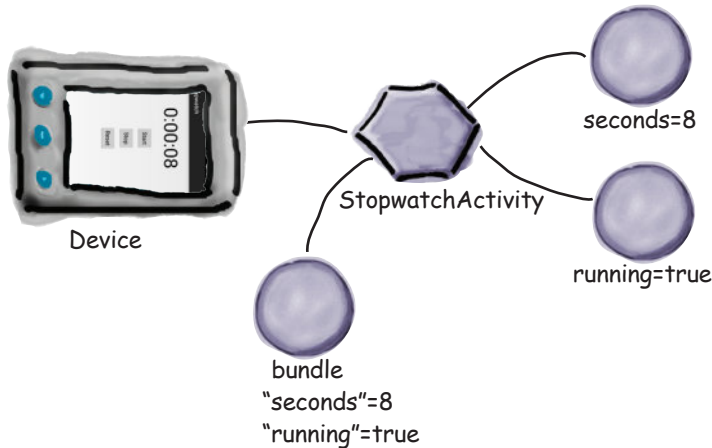


The story continues

- 3 Android destroys the activity, and then recreates it.**
The `onCreate ()` method gets called, and the `Bundle` gets passed to it.



- 4 The Bundle contains the values of the seconds and running variables as they were before the activity was destroyed.**
Code in the `onCreate ()` method sets the current variables to the values in the `Bundle`.



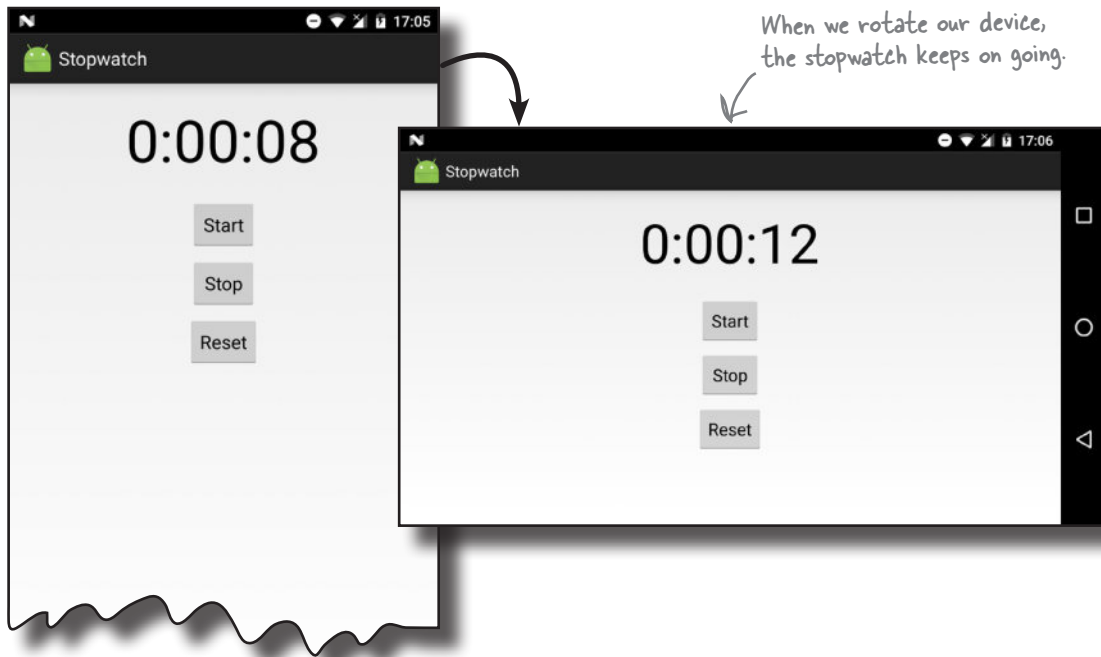
- 5 The `runTimer()` method gets called, and the timer picks up where it left off.**
The running stopwatch gets displayed on the device and continues to increment.





Test drive the app

Make the changes to your activity code, then run the app. When you click on the Start button, the timer starts, and it continues when you rotate the device.



there are no Dumb Questions

Q: Why does Android want to recreate an activity just because I rotated the screen?

A: The `onCreate()` method is normally used to set up the screen. If your code in `onCreate()` depended upon the screen configuration (for example, if you had different layouts for landscape and portrait), then you would want `onCreate()` to be called every time the configuration changed. Also, if the user changed their locale, you might want to recreate the UI in the local language.

Q: Why doesn't Android automatically store every instance variable automatically? Why do I have to write all of that code myself?

A: You might not want every instance variable 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: Is a `Bundle` some sort of Java map?

A: No, but it's designed to work like a `java.util.Map`. A `Bundle` has additional abilities compared to a `Map`. `Bundles` can be sent between processes, for example. That's really useful, because it allows the Android OS to stay in touch with the state of an activity.

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 in 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 stop 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 deal with 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.

`onStart()` gets called when your activity becomes visible to the user.

`onStop()` gets called when your activity has stopped being visible to the user. This might be because it's completely hidden by another activity that's appeared on top of it, or because the activity is going to be destroyed. If `onStop()` is called because the activity's going to be destroyed, `onSaveInstanceState()` gets called before `onStop()`.

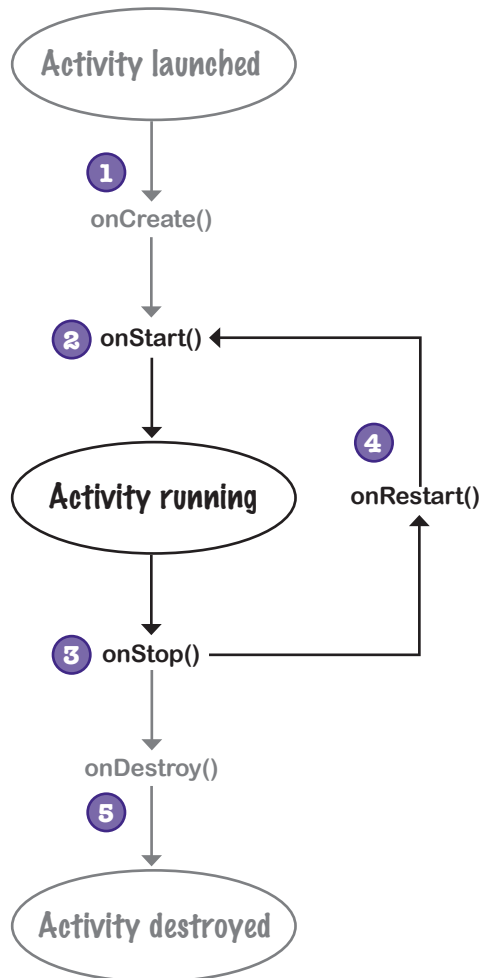
`onRestart()` gets called after your activity has been made invisible, before it gets made visible again.

We'll take a closer look at how these fit in with the `onCreate()` and `onDestroy()` methods on the next page.

An activity has a state of 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.

The activity lifecycle: the visible lifetime

Let's build on the lifecycle diagram you saw earlier in the chapter, this time including the `onStart()`, `onStop()`, and `onRestart()` methods (the bits you need to focus on are in bold):



- 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 becomes invisible and then 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 our Stopwatch app. First, we need to implement the activity's `onStop()` method so that the stopwatch stops running when the app isn't visible. Once we've done that, we need to implement the `onStart()` method so that the stopwatch starts again when the app is visible. Let's start with the `onStop()` method.

Implement `onStop()` to stop the timer

You override the `onStop()` method in the Android `Activity` class by adding the following method to your activity:

```
@Override
protected void onStop() {
    super.onStop(); ← This calls the onStop() method
    ...              in the activity's superclass,
}                  android.app.Activity.
```

The line of code:

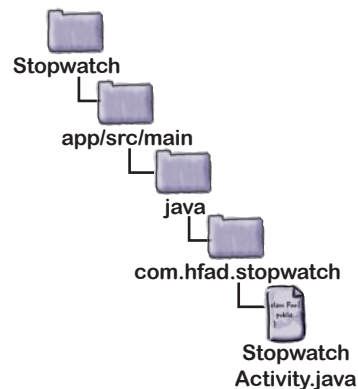
```
super.onStop();
```

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 that 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 Android will give you an exception.

We need to get the stopwatch to stop when the `onStop()` method is called. To do this, we need to set the value of the `running` boolean to `false`. Here's the complete method:

```
@Override
protected void onStop() {
    super.onStop();
    running = false;
}
```

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. If you don't, you'll get an exception.



Sharpen your pencil

Now it's your turn. Change the activity code so that if the stopwatch was running before `onStop()` was called, it starts running again when the activity regains the focus. Hint: you may need to add a new variable.

```
public class StopwatchActivity extends Activity {
    private int seconds = 0;
    private boolean running;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_stopwatch);
        if (savedInstanceState != null) {
            seconds = savedInstanceState.getInt("seconds");
            running = savedInstanceState.getBoolean("running");
        }
        runTimer();
    }

    @Override
    public void onSaveInstanceState(Bundle savedInstanceState) {
        savedInstanceState.putInt("seconds", seconds);
        savedInstanceState.putBoolean("running", running);
    }

    @Override
    protected void onStop() {
        super.onStop();
        running = false;
    }
}
```

Here's the first part of the activity code. You'll need to implement the `onStart()` method and change other methods slightly too.



Sharpen your pencil Solution

Now it's your turn. Change the activity code so that if the stopwatch was running before `onStop()` was called, it starts running again when the activity regains the focus. Hint: you may need to add a new variable.

```
public class StopwatchActivity extends Activity {
    private int seconds = 0;
    private boolean running;
    private boolean wasRunning;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_stopwatch);
        if (savedInstanceState != null) {
            seconds = savedInstanceState.getInt("seconds");
            running = savedInstanceState.getBoolean("running");
            wasRunning = savedInstanceState.getBoolean("wasRunning");
        }
        runTimer();
    }

    @Override
    public void onSaveInstanceState(Bundle savedInstanceState) {
        savedInstanceState.putInt("seconds", seconds);
        savedInstanceState.putBoolean("running", running);
        savedInstanceState.putBoolean("wasRunning", wasRunning);
    }

    @Override
    protected void onStop() {
        super.onStop();
        wasRunning = running;
        running = false;
    }

    @Override
    protected void onStart() {
        super.onStart();
        if (wasRunning) {
            running = true;
        }
    }
}
```

We added a new variable, `wasRunning`, to record whether the stopwatch was running before the `onStop()` method was called so that we know whether to set it running again when the activity becomes visible again.

We'll restore the state of the `wasRunning` variable if the activity is recreated.

Save the state of the `wasRunning` variable.

Record whether the stopwatch was running when the `onStop()` method was called.

Implement the `onStart()` method. If the stopwatch was running, set it running again.

The updated StopwatchActivity code

We've updated our activity code so that if the stopwatch was running before it lost the focus, it starts running again when it gets the focus back. Make the following changes (in bold) to your version of *StopwatchActivity.java*:

```
public class StopwatchActivity extends Activity {
    private int seconds = 0;
    private boolean running;
    private boolean wasRunning;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_stopwatch);
        if (savedInstanceState != null) {
            seconds = savedInstanceState.getInt("seconds");
            running = savedInstanceState.getBoolean("running");
            wasRunning = savedInstanceState.getBoolean("wasRunning");
        }
        runTimer();
    }

    @Override
    public void onSaveInstanceState(Bundle savedInstanceState) {
        savedInstanceState.putInt("seconds", seconds);
        savedInstanceState.putBoolean("running", running);
        savedInstanceState.putBoolean("wasRunning", wasRunning);
    }

    @Override
    protected void onStop() {
        super.onStop();
        wasRunning = running;
        running = false;
    }

    @Override
    protected void onStart() {
        super.onStart();
        if (wasRunning) {
            running = true;
        }
    }
}
```

A new variable, *wasRunning*, records whether the stopwatch was running before the *onStop()* method was called.

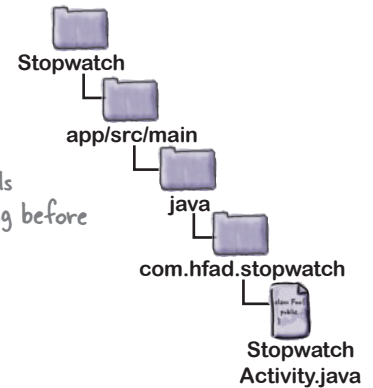
Restore the state of the *wasRunning* variable if the activity is recreated.

Save the state of the *wasRunning* variable.

Record whether the stopwatch was running when the *onStop()* method was called.

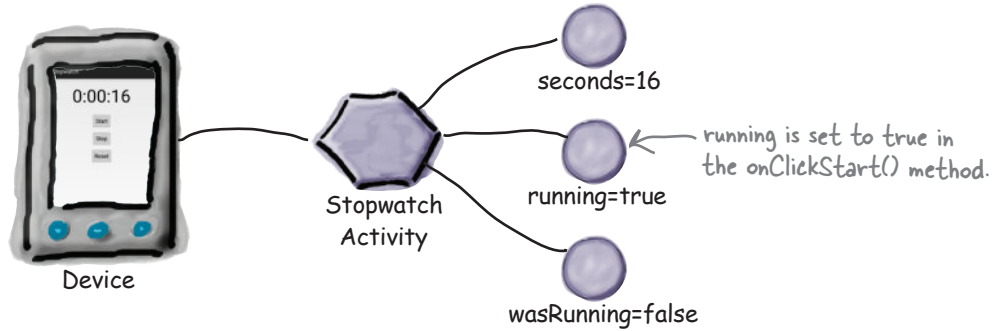
Implement the *onStart()* method. If the stopwatch was running, we'll set it running again.

... We've left out some of the activity code as we don't need to change it.

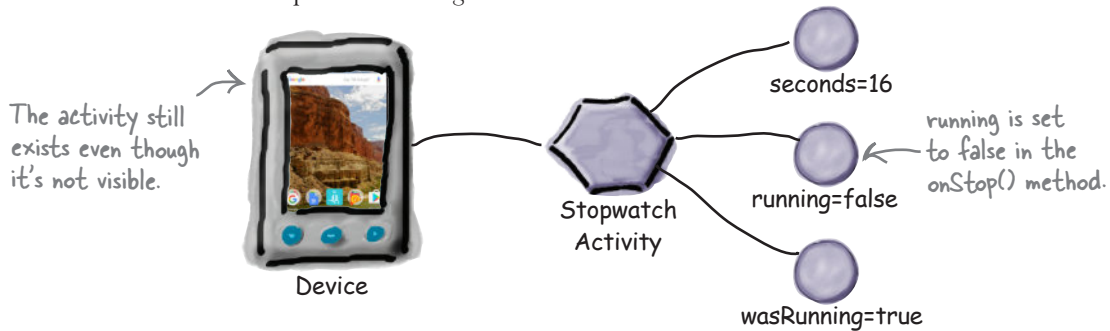


What happens when you run the app

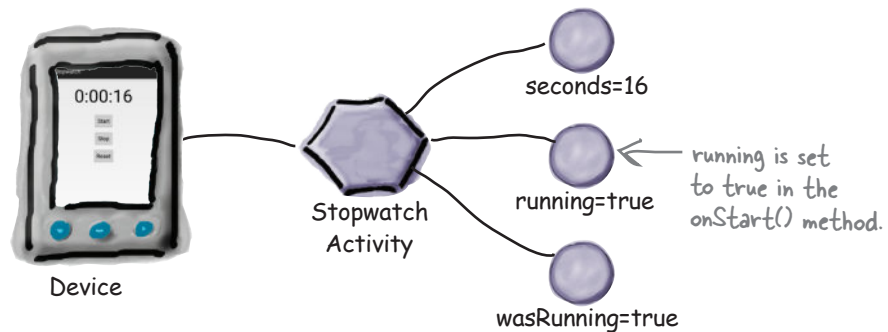
- 1 **The user starts the app, and clicks the Start button to set the stopwatch going.**
The `runTimer()` method starts incrementing the number of seconds displayed in the `time_view` text view.



- 2 **The user navigates to the device's home screen so the Stopwatch app is no longer visible.**
The `onStop()` method gets called, `wasRunning` is set to `true`, `running` is set to `false`, and the number of seconds stops incrementing.



- 3 **The user navigates back to the Stopwatch app.**
The `onStart()` method gets called, `running` is set to `true`, and the number of seconds starts incrementing again.





Test drive the app

Save the changes to your activity code, then run the app. When you click on the Start button the timer starts: it stops when the app is no longer visible, and it starts again when the app becomes visible again.



there are no Dumb Questions

Q: Could we have used the `onRestart()` method instead of `onStart()` to set the stopwatch running again?

A: `onRestart()` is used when you only want code to run when an app becomes visible after having previously been invisible. It doesn't run when the activity becomes visible for the first time. In our case, we wanted the app to still work when we rotated the device.

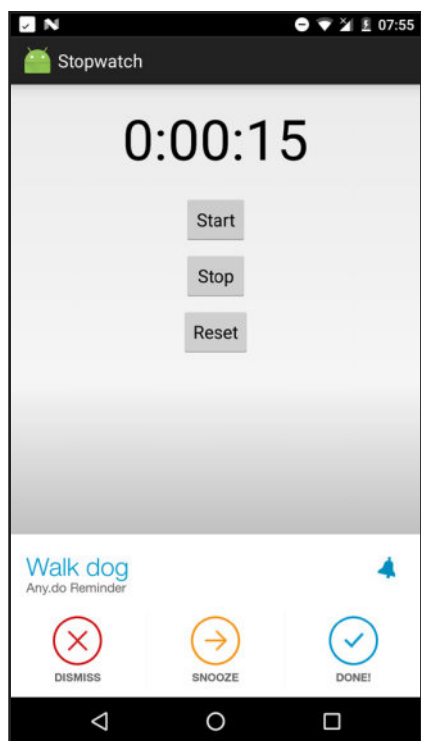
Q: Why should that make a difference?

A: When you rotate the device, the activity is destroyed and a new one is created in its place. If we'd put code to set the stopwatch running again in the `onRestart()` method instead of `onStart()`, it wouldn't have run when the activity was recreated. The `onStart()` method gets called in both situations.

What if an app 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 we 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.



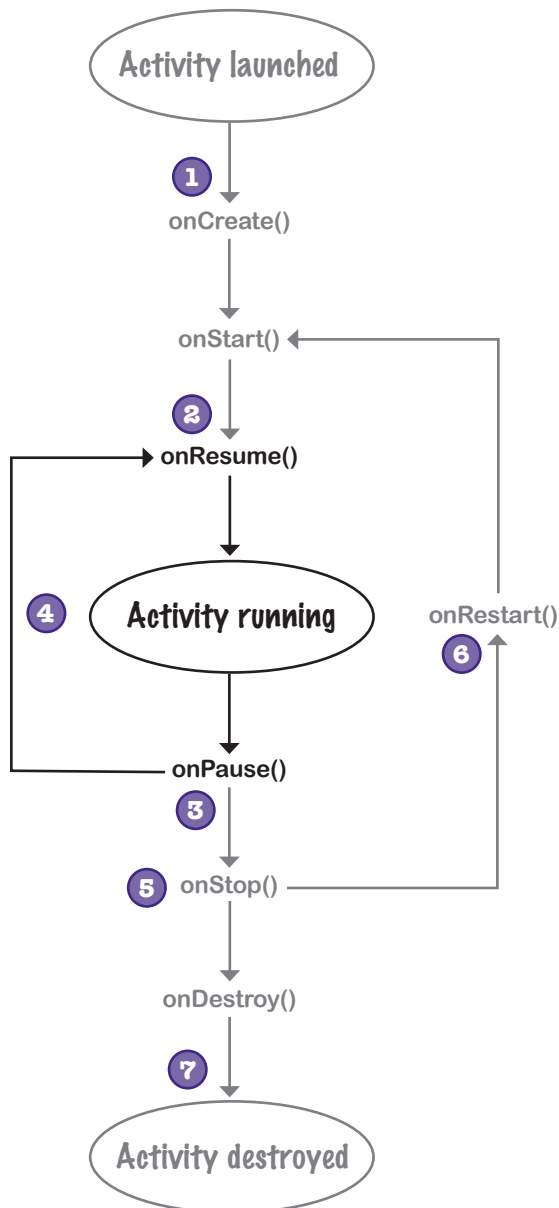
An activity has a state of paused if it's lost the focus but is still visible to the user. The activity is still alive and maintains all its state information.

There are two lifecycle methods that handle 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. If you need your app to react in some way when your activity is paused, you need to implement these methods.

You'll see on the next page how these methods fit in with the rest of the lifecycle methods you've seen so far.

The activity lifecycle: the foreground lifetime

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 bold):



- 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.

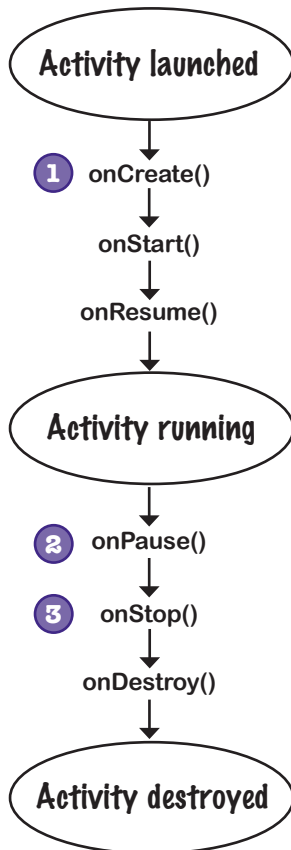


Earlier on you talked about how the activity is destroyed and a new one is created when the user rotates the device. What happens if the activity is paused when the device is rotated? Does the activity go through the same lifecycle methods?

That's a great question, so let's look at this in more detail before getting back to the Stopwatch app.

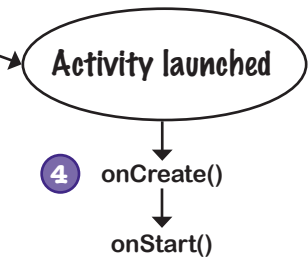
The original activity goes through all its lifecycle methods, from `onCreate()` to `onDestroy()`. A new activity is created when the original is destroyed. As this new activity isn't in the foreground, only the `onCreate()` and `onStart()` lifecycle methods get called. Here's what happens when the user rotates the device when the activity doesn't have the focus:

Original Activity



- 1 **The user launches the activity.**
The activity lifecycle methods `onCreate()`, `onStart()`, and `onResume()` get called.
- 2 **Another activity appears in front of it.**
The activity's `onPause()` method gets called.
- 3 **The user rotates the device.**
Android sees this as a configuration change. The `onStop()` and `onDestroy()` methods get called, and Android destroys the activity. A new activity is created in its place.
- 4 **The activity is visible but not in the foreground.**
The `onCreate()` and `onStart()` methods get called. As the activity is visible but doesn't have the focus, `onResume()` isn't called.

Replacement Activity





I see, the replacement activity doesn't reach a state of "running" because it's not in the foreground. But what if you navigate away from the activity completely so it's not even visible? If the activity's stopped, do onResume() and onPause() get called before onStop()?

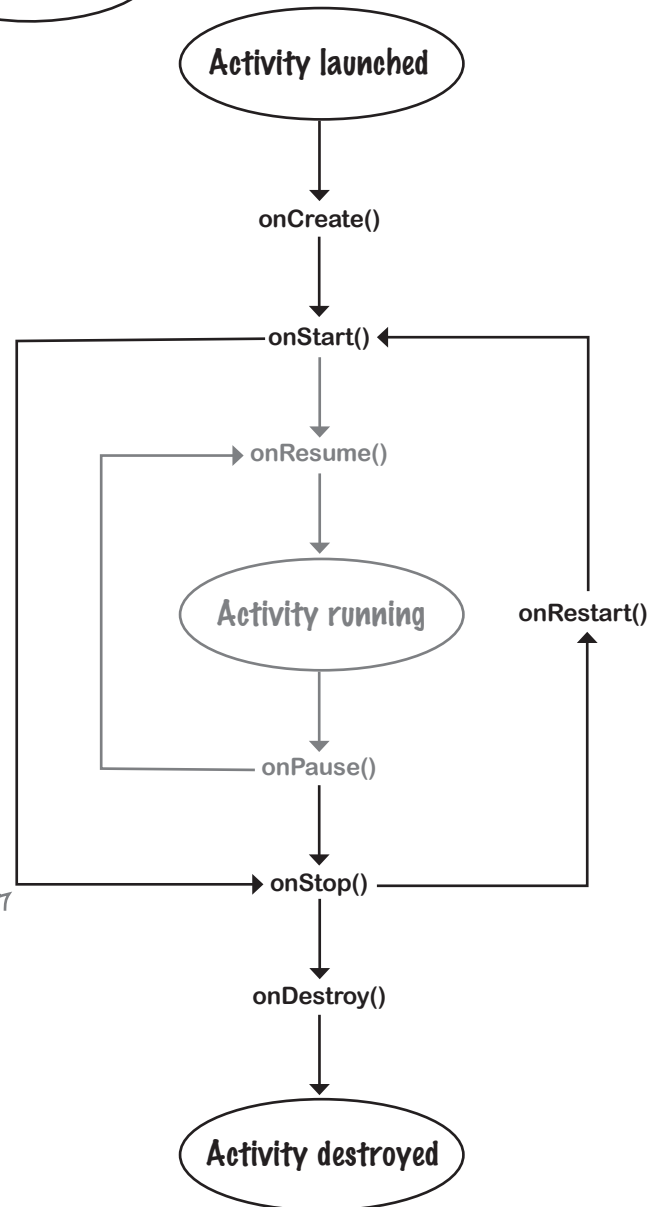
Activities can go straight from onStart() to onStop() and bypass onPause() and onResume().

If you have an activity that's visible, but never in the foreground and never has the focus, the onPause () and onResume () methods *never get called*.

The onResume () method gets called when the activity appears in the foreground and has the focus. If the activity is only visible behind other activities, the onResume () method doesn't get called.

Similarly, the onPause () method gets called only when the activity is no longer in the foreground. If the activity is never in the foreground, this method won't get called.

If an activity stops or gets destroyed before it appears in the foreground, the onStart() method is followed by the onStop() method. onResume() and onPause() are bypassed.



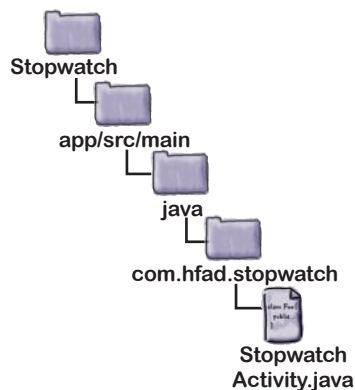
Stop the stopwatch if the activity's paused

Let's get back to the Stopwatch app.

So far we've made the stopwatch stop if the Stopwatch app isn't visible, and made it start again when the app becomes visible again. We did this by overriding the `onStop()` and `onStart()` methods like this:

```
@Override
protected void onStop() {
    super.onStop();
    wasRunning = running;
    running = false;
}

@Override
protected void onStart() {
    super.onStart();
    if (wasRunning) {
        running = true;
    }
}
```



Let's get the app to have the same behavior if the app is only partially visible. We'll get the stopwatch to stop if the activity is paused, and start again when the activity is resumed. So what changes do we need to make to the lifecycle methods?

We want the Stopwatch app to stop running when the activity is paused, and start it again (if it was running) when the activity is resumed. In other words, we want it to behave the same as when the activity is stopped or started. This means that instead of repeating the code we already have in multiple methods, we can use one method when the activity is paused or stopped, and another method when the activity is resumed or started.

Implement the onPause() and onResume() methods

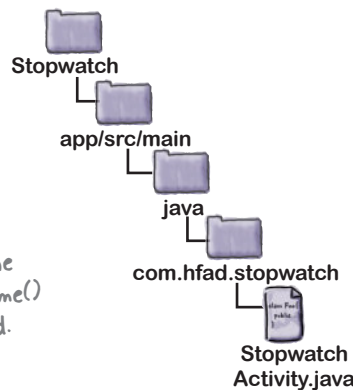
We'll start with when the activity is resumed or started.

When the activity is resumed, the activity's onResume() lifecycle method is called. If the activity is started, the activity's onResume() method is called after calling onStart(). The onResume() method is called irrespective of whether the activity is resumed or started, which means that if we move our onStart() code to the onResume() method, our app will behave the same irrespective of whether the activity is resumed or started. This means we can remove our onStart() method, and replace it with the onResume() method like this:

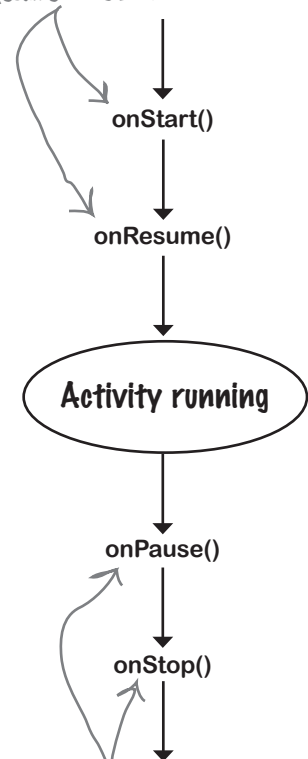
```

@Override
protected void onStart() {
    super.onStart();
    if (wasRunning) {
        running = true;
    }
}
    ← Delete the onStart() method.

@Override
protected void onResume() { ← Add the onResume() method.
    super.onResume();
    if (wasRunning) {
        running = true;
    }
}
    
```



The onResume() method is called when the activity is started or resumed. As we want the app to do the same thing irrespective of whether it's started or resumed, we only need to implement the onResume() method.



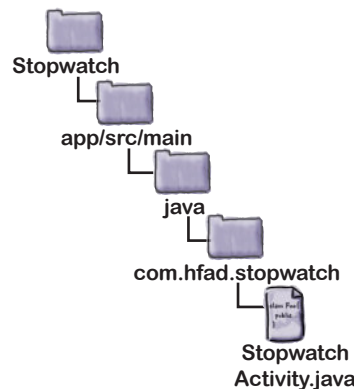
We can do something similar when the activity is paused or stopped.

When the activity is paused, the activity's onPause() lifecycle method is called. If the activity is stopped, the activity's onPause() method is called prior to calling onStop(). The onPause() method is called irrespective of whether the activity is paused or stopped, which means we can move our onStop() code to the onPause() method:

```

@Override
protected void onStop() {
    super.onStop();
    wasRunning = running;
    running = false;
}
    ← Delete the onStop() method.

@Override
protected void onPause() { ← Add the onPause() method.
    super.onPause();
    wasRunning = running;
    running = false;
}
    
```



The onPause() method is called when the activity is paused or stopped. This means we only need to implement the onPause() method.

The complete StopwatchActivity code

Here's the full *StopwatchActivity.java* code for the finished app (with changes in bold>):

```

package com.hfad.stopwatch;

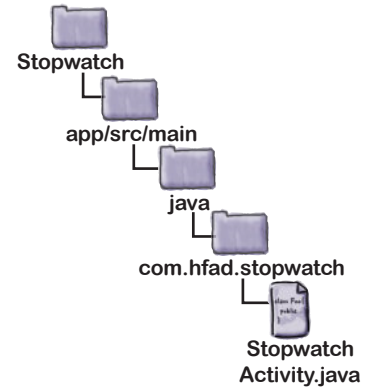
import android.app.Activity;
import android.os.Bundle;
import android.view.View;
import java.util.Locale;
import android.os.Handler;
import android.widget.TextView;

public class StopwatchActivity extends Activity {
    //Number of seconds displayed on the stopwatch.
    private int seconds = 0;
    //Is the stopwatch running?
    private boolean running;
    private boolean wasRunning;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_stopwatch);
        if (savedInstanceState != null) {
            seconds = savedInstanceState.getInt("seconds");
            running = savedInstanceState.getBoolean("running");
            wasRunning = savedInstanceState.getBoolean("wasRunning");
        }
        runTimer();
    }

    @Override
    public void onSaveInstanceState(Bundle savedInstanceState) {
        savedInstanceState.putInt("seconds", seconds);
        savedInstanceState.putBoolean("running", running);
        savedInstanceState.putBoolean("wasRunning", wasRunning);
    }
}

```



Use *seconds*, *running*, and *wasRunning* respectively to record the number of seconds passed, whether the stopwatch is running, and whether the stopwatch was running before the activity was paused.

Get the previous state of the stopwatch if the activity's been destroyed and recreated.

Save the state of the stopwatch if it's about to be destroyed.

The activity code continues over the page.

The activity code (continued)

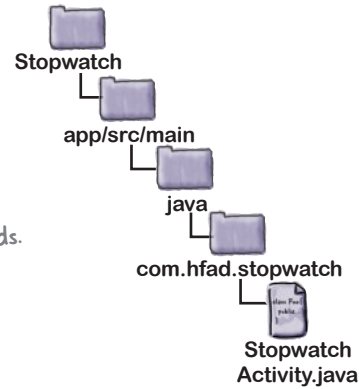
```

@Override
protected void onStop() {
    super.onStop();
    wasRunning = running;
    running = false;
}

@Override
protected void onStart() {
    super.onStart();
    if (wasRunning) {
        running = true;
}

```

Delete these two methods.



```

@Override
protected void onPause() {
    super.onPause();
    wasRunning = running;
    running = false;
}

```

← If the activity's paused, stop the stopwatch.

```

@Override
protected void onResume() {
    super.onResume();
    if (wasRunning) {
        running = true;
    }
}

```

← If the activity's resumed, start the stopwatch again if it was running previously.

```

//Start the stopwatch running when the Start button is clicked.
public void onClickStart(View view) {
    running = true;
}

```

← This gets called when the Start button is clicked.

The activity code continues over the page. →

The activity code (continued)

```
//Stop the stopwatch running when the Stop button is clicked.
public void onClickStop(View view) {
    running = false;
}

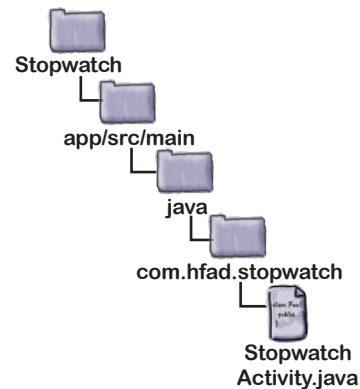
//Reset the stopwatch when the Reset button is clicked.
public void onClickReset(View view) {
    running = false;
    seconds = 0;
}

//Sets the number of seconds on the timer.
private void runTimer() {
    final TextView timeView = (TextView) findViewById(R.id.time_view);
    final Handler handler = new Handler();
    handler.post(new Runnable() {
        @Override
        public void run() {
            int hours = seconds/3600;
            int minutes = (seconds%3600)/60;
            int secs = seconds%60;
            String time = String.format(Locale.getDefault(),
                "%d:%02d:%02d", hours, minutes, secs);
            timeView.setText(time);
            if (running) {
                seconds++;
            }
            handler.postDelayed(this, 1000);
        }
    });
}
```

This gets called when the Stop button is clicked.

This gets called when the Reset button is clicked.

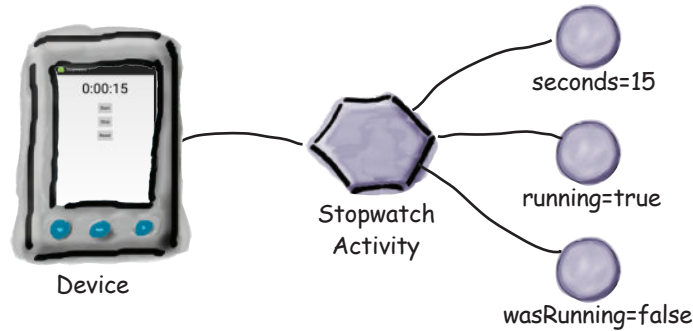
The runTimer() method uses a Handler to increment the seconds and update the text view.



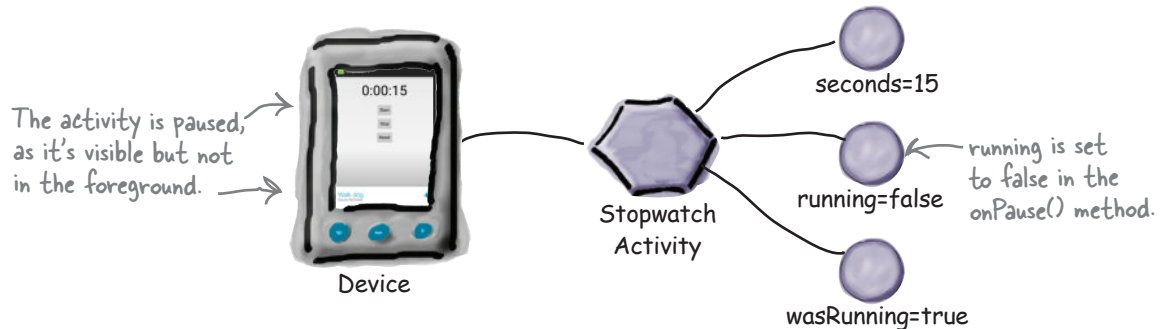
Let's go through what happens when the code runs.

What happens when you run the app

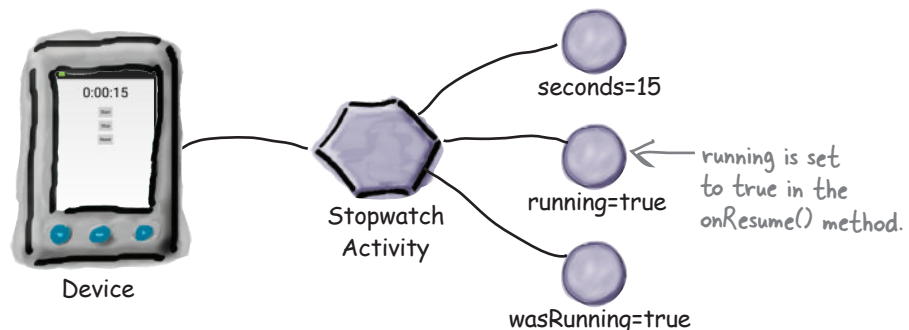
- 1** The user starts the app, and clicks on the Start button to set the stopwatch going. The `runTimer()` method starts incrementing the number of seconds displayed in the `time_view` text view.



- 2** Another activity appears in the foreground, leaving `StopwatchActivity` partially visible. The `onPause()` method gets called, `wasRunning` is set to `true`, `running` is set to `false`, and the number of seconds stops incrementing.



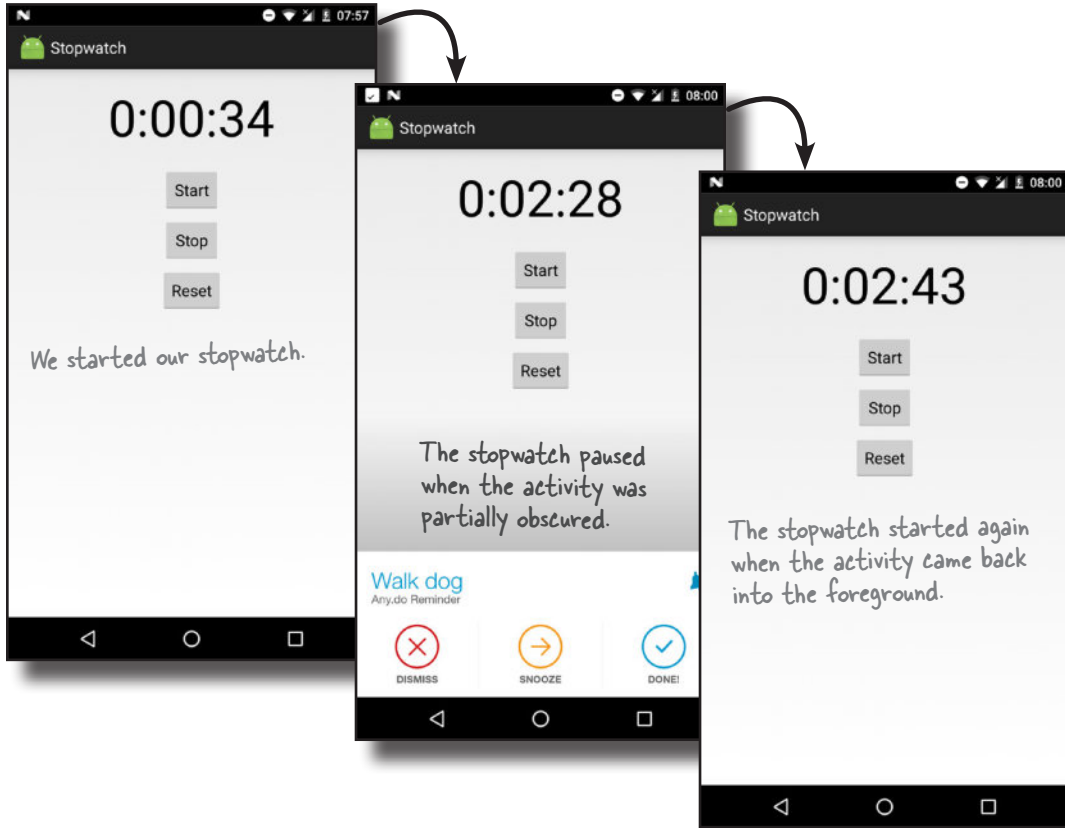
- 3** When `StopwatchActivity` returns to the foreground, the `onResume()` method gets called, `running` is set to `true`, and the number of seconds starts incrementing again.





Test drive the app

Save the changes to your activity code, then run the app. When you click on the Start button, the timer starts; it stops when the app is partially obscured by another activity; and it starts again when the app is back in the foreground.



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.

User starts the activity, starts using it, rotates the device, switches to another app, then goes back to the activity.

← This one's tough.

```

...
class MyActivity extends Activity{

    protected void onCreate(
        Bundle savedInstanceState) {
        A //Run code A
        ...
    }

    protected void onPause() {
        B //Run code B
        ...
    }

    protected void onRestart() {
        C //Run code C
        ...
    }

    protected void onResume() {
        D //Run code D
        ...
    }

    protected void onStop() {
        E //Run code E
        ...
    }

    protected void onRecreate() {
        F //Run code F
        ...
    }

    protected void onStart() {
        G //Run code G
        ...
    }

    protected void 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 MyActivity extends Activity{

protected void onCreate(
    Bundle savedInstanceState) {
    A //Run code A
    ...
}

protected void onPause() {
    B //Run code B
    ...
}

protected void onRestart() {
    C //Run code C
    ...
}

protected void onResume() {
    D //Run code D
    ...
}

protected void onStop() {
    E //Run code E
    ...
}

protected void onRecreate() {
    F //Run code F
    ...
}

protected void onStart() {
    G //Run code G
    ...
}

protected void onDestroy() {
    H //Run code H
    ...
}
}

```

There's no lifecycle method called `onRecreate()`.

Your handy guide to the lifecycle methods

Method	When it's called	Next method
onCreate ○	When the activity is first created. Use it for normal static setup, such as creating views. It also gives you a <code>Bundle</code> that contains the previously saved state of the activity.	<code>onStart ()</code>
onRestart ○	When your activity has been stopped but just before it gets started again.	<code>onStart ()</code>
onStart ○	When your activity is becoming visible. It's followed by <code>onResume ()</code> if the activity comes into the foreground, or <code>onStop ()</code> if the activity is made invisible.	<code>onResume ()</code> or <code>onStop ()</code>
onResume ○	When your activity is in the foreground.	<code>onPause ()</code>
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 <code>onResume ()</code> if the activity returns to the foreground, or <code>onStop ()</code> if it becomes invisible.	<code>onResume ()</code> or <code>onStop ()</code>
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 <code>onRestart ()</code> if the activity becomes visible again, or <code>onDestroy ()</code> if the activity is being destroyed.	<code>onRestart ()</code> or <code>onDestroy ()</code>
onDestroy ○	When your activity is about to be destroyed or because the activity is finishing.	None



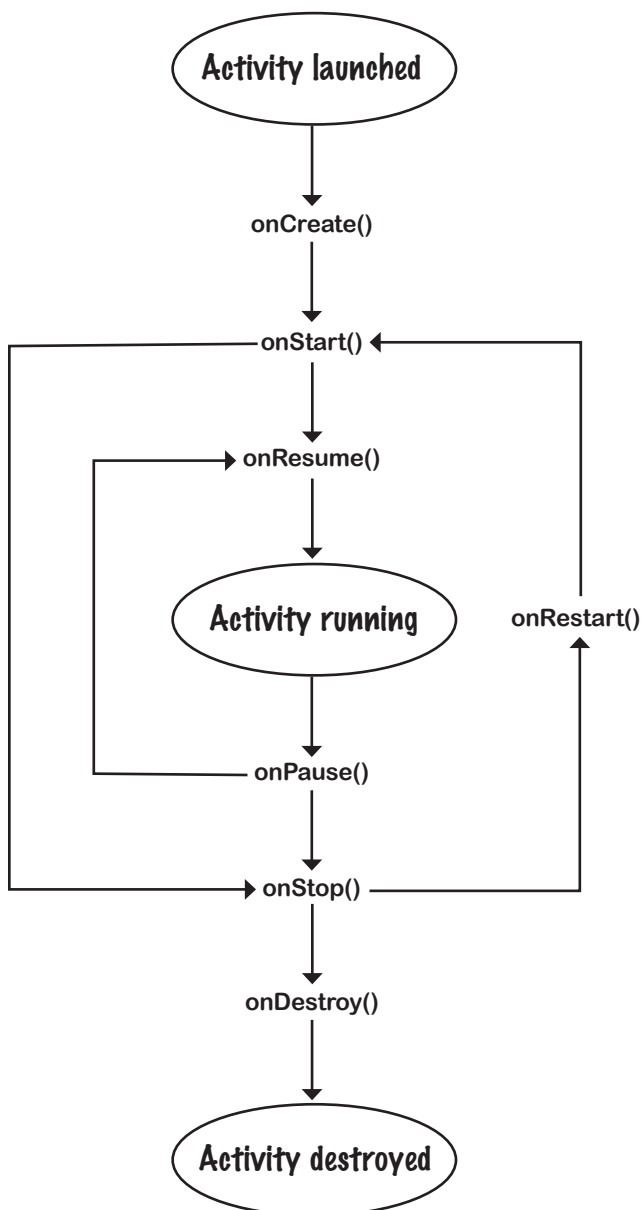
Your Android Toolbox

You've got Chapter 4 under your belt and now you've added the activity lifecycle to your toolbox.

You can download the full code for the chapter from <https://tinyurl.com/HeadFirstAndroid>.

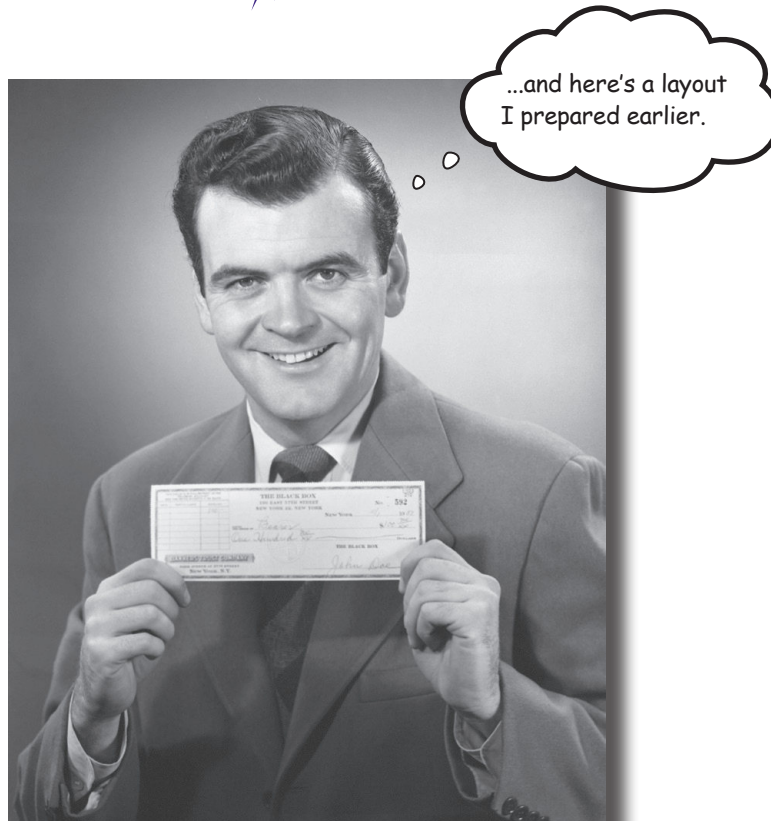
BULLET POINTS

- Each app runs in its own process by default.
- Only the main thread can update the user interface.
- Use a `Handler` to schedule code or post code to a different thread.
- 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` using `bundle.put*("name", value)`. You retrieve values from the bundle 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()` handle when the activity gains and loses the focus.



5 views and view groups

Enjoy the View



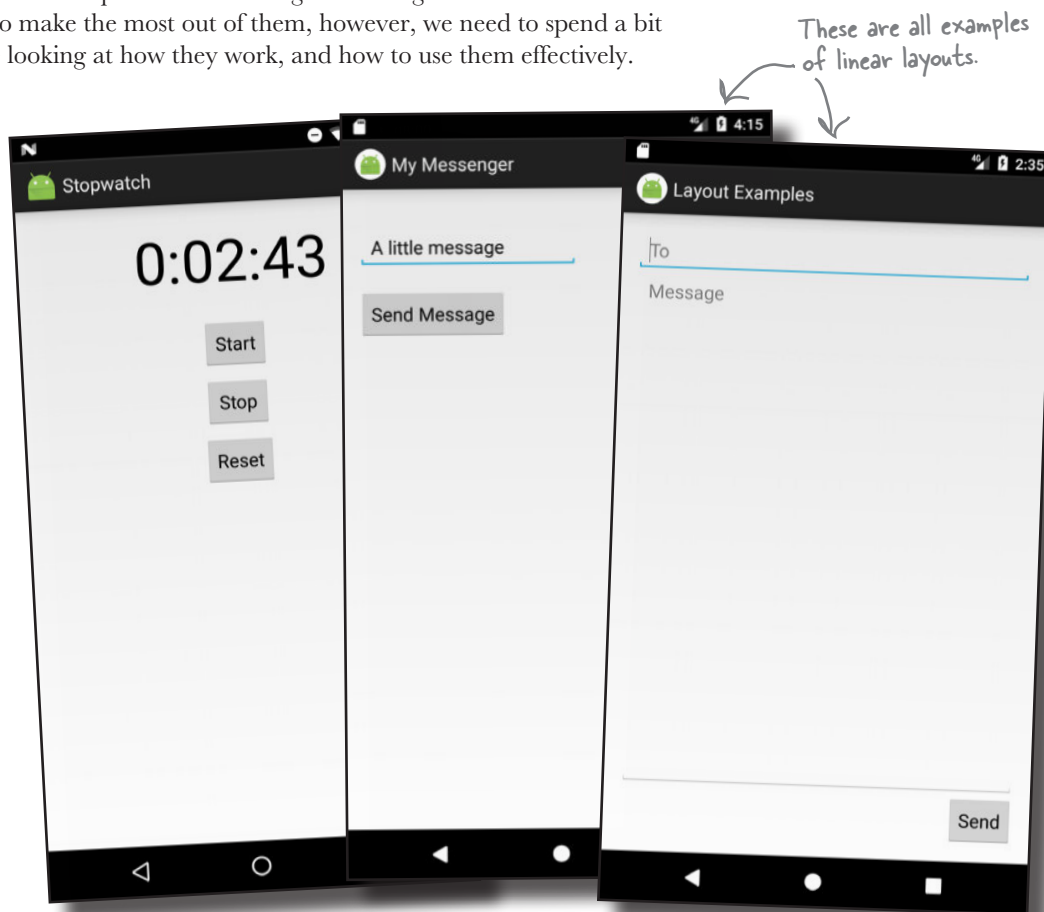
You've seen how to arrange GUI components using a linear layout, but so far we've only scratched the surface.

In this chapter we'll **look a little deeper** and show you how linear layouts *really work*. We'll introduce you to the **frame layout**, a simple layout used to stack views, and we'll also take a tour of the **main GUI components** and **how you use them**. By the end of the chapter, you'll see that even though they all look a little different, all layouts and GUI components have **more in common than you might think**.

Your user interface is made up of layouts and GUI components

As you already know, a layout defines what a screen looks like, and you define it using XML. Layouts usually contain GUI components such as buttons and text fields. Your user interacts with these to make your app do something.

All the apps you've seen in the book so far have used linear layouts, where GUI components are arranged in a single column or row. In order to make the most out of them, however, we need to spend a bit of time looking at how they work, and how to use them effectively.



In this chapter, we're going to take a closer look at linear layouts, introduce you to their close relative the frame layout, and show you other GUI components you can use to make your app more interactive.

Let's start with linear layouts.



LinearLayout displays views in a single row or column

As you already know, a linear layout displays its views next to each other, either vertically or horizontally. If it's vertically, the views are displayed in a single column. If it's horizontally, the views are displayed in a single row.

You define a linear layout using the `<LinearLayout>` element like this:

```

<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    ...>
</LinearLayout>

```

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 `xmlns:android` attribute is used to specify the Android namespace, and you must always set it to `"http://schemas.android.com/apk/res/android"`.

You MUST set the layout's width and height

The `android:layout_width` and `android:layout_height` attributes 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 in a couple of pages). You will usually set the layout width and height to `"match_parent"`.

You may sometimes see `android:layout_width` and `android:layout_height` set to `"fill_parent"`. `"fill_parent"` was used in older versions of Android, and it's now replaced by `"match_parent"`. `"fill_parent"` is deprecated.



Geek Bits

What are density-independent pixels?

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.



Orientation is vertical or horizontal

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

As you've seen in earlier chapters, you arrange views vertically using:

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

This displays the views in a single column.

You arrange views horizontally in a single row using:

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

When the orientation is horizontal, views are displayed from left to right by default. This is great for languages that are read from left to right, but what if the user has set the language on their device to one that's read from right to left?

For apps where the minimum SDK is *at least* API 17, you can get views to rearrange themselves depending on the language setting on the device. If the user's language is read from right to left, you can get the views to arrange themselves starting from the right.

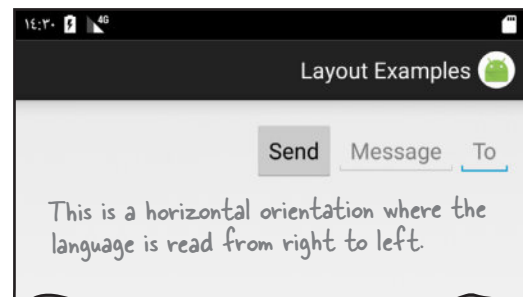
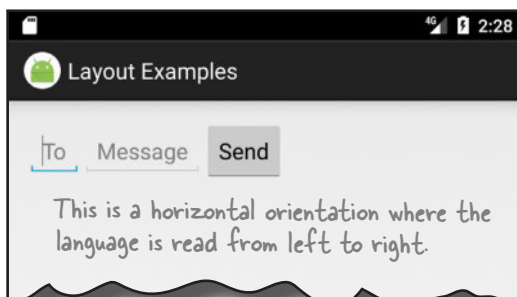
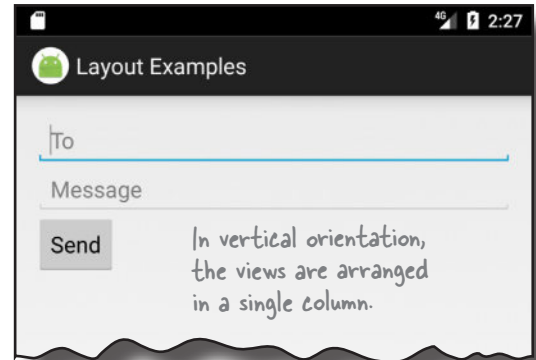
To do this, you declare that your app supports languages that are read from right to left in your *AndroidManifest.xml* file like this:

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

↑
 Android Studio may add this line of code for you. It must go inside the `<application>` tag.



`supportsRtl` means “supports right-to-left languages.”



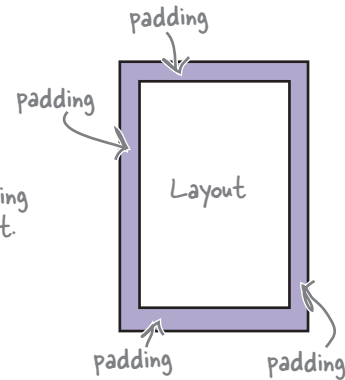


Padding adds space

If you want there to be a bit of space around the edge of the layout, you can set **padding** attributes. These attributes tell Android how much padding you want between each of the layout's sides and its parent. Here's how you would tell Android you want to add padding of 16dp to all edges of the layout:

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

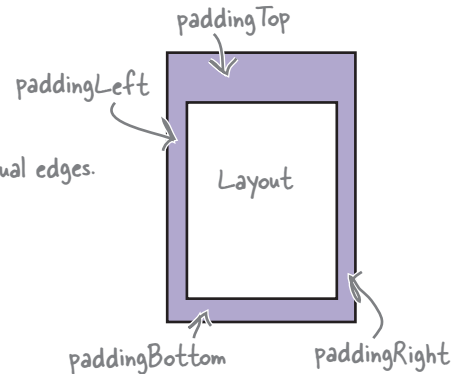
← This adds the same padding to all edges of the layout.



If you want to add different amounts of padding to different edges, you can specify the edges individually. Here's how you would add padding of 32dp to the top of the layout, and 16dp to the other edges:

```
<LinearLayout ...
    android:paddingBottom="16dp"
    android:paddingLeft="16dp"
    android:paddingRight="16dp"
    android:paddingTop="32dp" >
    ...
</LinearLayout>
```

Add padding to the individual edges.



If your app supports right-to-left languages, you can use:

```
android:paddingStart="16dp"
```

and:

```
android:paddingEnd="16dp"
```

to add padding to the start and end edges of the layout instead of their left and right edges.

`android:paddingStart` adds padding to the start edge of the layout. The start edge is on the left for languages that are read from left to right, and the right edge for languages that are read from right to left.

`android:paddingEnd` adds padding to the end edge of the layout. The end edge is on the right for languages that are read from left to right, and the left edge for languages that are read from right to left.



Watch it!

You can only use start and end properties with API 17 or above.

If you want your app to work on older versions of Android, you must use the left and right properties instead.



Add a dimension resource file for consistent padding across layouts

In the example on the previous page, we hardcoded the padding and set it to 16dp. An alternative approach is to specify the padding in a *dimension resource file* instead. Doing so makes it easier to maintain the padding dimensions for all the layouts in your app.

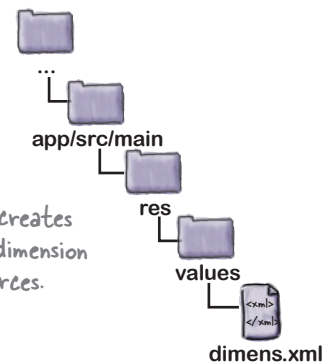
To use a dimension resource file, you first need to add one to your project. To do this, first select the `app/src/main/res/values` folder in Android Studio, then go to File menu and choose New→Values resource file. When prompted, enter a name of “dimens” and click on the OK button. This creates a new resource file called `dimens.xml`.

Android Studio may have already added this file for you, but it depends on which version of Android Studio you're using.

Once you've created the dimension resource file, you add dimensions to it using the `<dimen>` element. As an example, here's how you would add dimensions for the horizontal and vertical margins to `dimens.xml`:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <dimen name="activity_horizontal_margin">16dp</dimen>
  <dimen name="activity_vertical_margin">16dp</dimen>
</resources>
```

This creates two dimension resources.



You use the dimensions you create by setting the padding attributes in your layout file to the name of a dimension resource like this:

```
<LinearLayout ...
  android:paddingLeft="@dimen/activity_horizontal_margin"
  android:paddingRight="@dimen/activity_horizontal_margin"
  android:paddingTop="@dimen/activity_vertical_margin"
  android:paddingBottom="@dimen/activity_vertical_margin">
```

The `paddingLeft` and `paddingRight` attributes are set to `@dimen/activity_horizontal_margin`.

The `paddingTop` and `paddingBottom` attributes are set to `@dimen/activity_vertical_margin`.

With that kind of setup, at runtime, Android looks up the values of the attributes in the dimension resource file and applies the values it finds there.



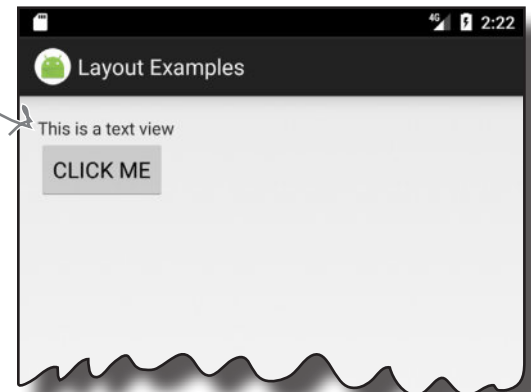
A linear layout displays views in the order they appear in the layout XML

When you define a linear layout, you add views to the layout in the order in which you want them to appear. So, if you want a text view to appear above a button in a linear layout, you *must* define the text view first:

```
<LinearLayout ... >
  <TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/text_view1" />

  <Button
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/click_me" />
</LinearLayout>
```

If you define the text view above the button in the XML, the text view will appear above the button when displayed.



You specify the width and height of any views using `android:layout_width` and `android:layout_height`. The code:

```
android:layout_width="wrap_content"
```

means that you want the view to be just wide enough for its content to fit inside it—for example, the text displayed on a button or in a text view. The code:

```
android:layout_width="match_parent"
```

means that you want the view to be as wide as the parent layout.

If you need to refer to a view elsewhere in your code, you need to give it an ID. As an example, you'd give the text view an ID of "text_view" using the code:

```
...
  <TextView
    android:id="@+id/text_view"
    ... />
...
```

`android:layout_width` and `android:layout_height` are mandatory attributes for all views, no matter which layout you use.

They can take the values `wrap_content`, `match_parent`, or a specific dimension value such as `16dp`.



Use margins to add distance between views

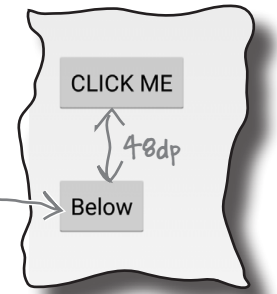
When you position a view using a linear layout, the layout doesn't leave much of a gap between views. You can increase the size of the gap by adding one or more **margins** to the view.

As an example, suppose you wanted to put one view below another, but add 48dp of extra space between the two. To do that, you'd add a margin of 48dp to the top of the bottom view:

```
LinearLayout ... >
  <Button
    android:id="@+id/button_click_me"
    ... />

  <Button
    android:id="@+id/button_below"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginTop="48dp"
    android:text="@string/button_below" />
</LinearLayout>
```

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



Here's a list of the margins 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_marginLeft,
layout_marginStart**

Adds extra space to the left (or start) of the view.

**layout_marginRight,
layout_marginEnd**

Adds extra space to the right (or end) of the view.



layout_margin

Adds equal space to each side of the view.



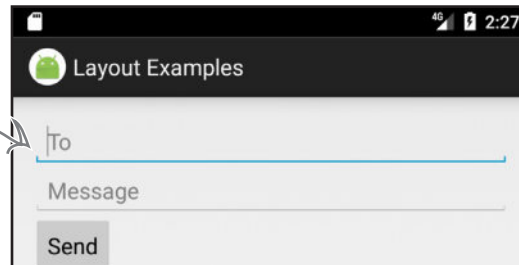


Let's change up a basic linear layout

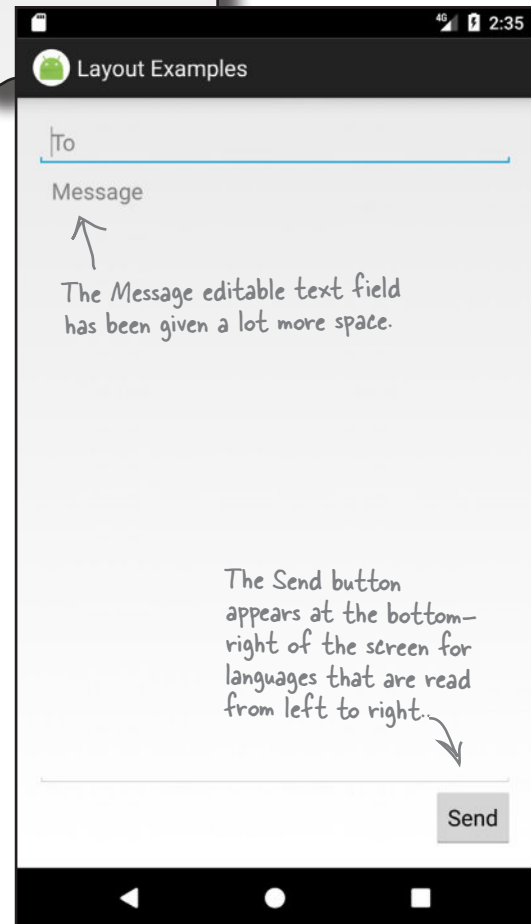
At first glance, a linear layout can seem basic and inflexible. After all, all it does is arrange views in a particular order. To give you more flexibility, you can tweak your layout's appearance using some more of its attributes. To show you how this works, we're going to transform a basic linear layout.

The layout is composed of two editable text fields and a button. To start with, these text fields are simply displayed vertically on the screen like this:

Each view takes up the least possible amount of vertical space.



We're going to change the layout so that the button is displayed in the end corner of the layout, and one of the editable text fields takes up any remaining space.



The Message editable text field has been given a lot more space.

The Send button appears at the bottom-right of the screen for languages that are read from left to right.



Here's the starting point for the linear layout

The linear layout contains two editable text fields and a button. The button is labeled “Send,” and the editable text fields contain hint text values of “To” and “Message.”

Hint text in an editable text field is text that's displayed when the field is empty. It's used to give users a hint as to what sort of text they should enter. You define hint text using the `android:hint` attribute:

```
<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="com.hfad.views.MainActivity" >
```

```
<EditText
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:hint="@string/to" />
```

```
<EditText
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:hint="@string/message" />
```

```
<Button
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/send" />
```

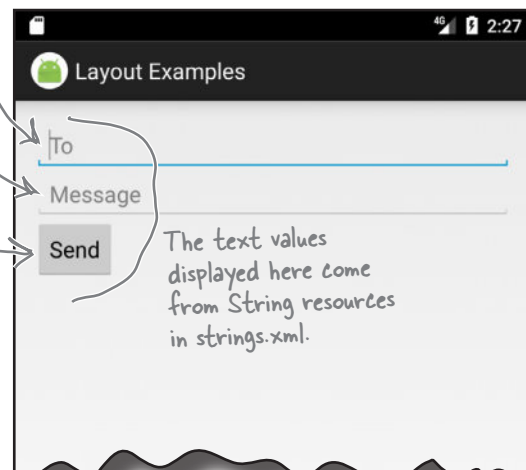
```
</LinearLayout>
```

The editable text fields are as wide as the parent layout.

`android:hint` displays a hint to the user as to what they should type in the editable text field.

The values of these Strings are defined in `strings.xml` as usual.

The text values displayed here come from String resources in `strings.xml`.



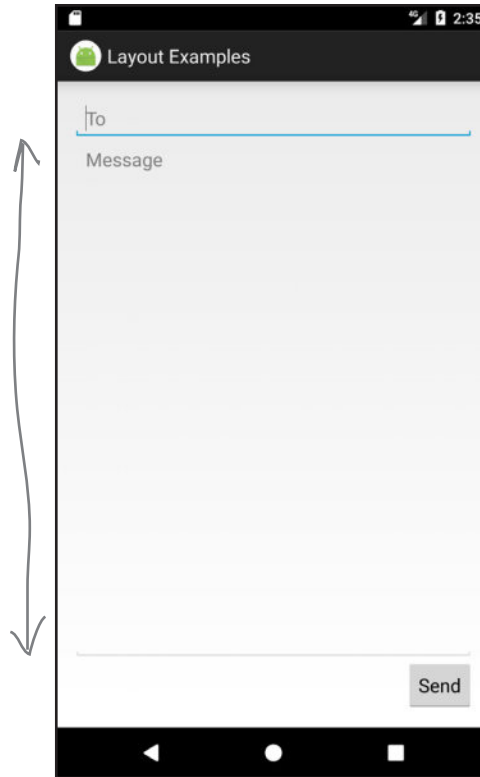
All of these views take up just as much vertical space in the layout as they need for their contents. So how do we make the Message text field taller?



Make a view stretch by adding weight

All of the views in our basic layout take up just as much vertical space as they need for their content. But what we actually want is to make the Message text field stretch to take up any vertical space in the layout that's not being used by the other views.

We want to make the Message text field stretch vertically so that it fills any spare space in the layout.



In order to do this, we need to allocate some **weight** to the Message text field. 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 editable text field 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.



Adding weight to one view

We need the Message editable text field to take up any extra space in the layout. To do this, we'll set its `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:

```
<LinearLayout ... >
```

```
  <EditText
```

```
    android:layout_width="match_parent"
```

```
    android:layout_height="wrap_content"
```

```
    android:hint="@string/to" />
```

```
  <EditText
```

```
    android:layout_width="match_parent"
```

```
    android:layout_height="0dp"
```

```
    android:layout_weight="1"
```

```
    android:hint="@string/message" />
```

```
  <Button
```

```
    android:layout_width="wrap_content"
```

```
    android:layout_height="wrap_content"
```

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

```
</LinearLayout>
```

← This `<EditText>` and the `<Button>` have no `layout_weight` attribute set. They'll take up as much room as their content needs, but no more.

This view is the only one 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`. Setting the `layout_height` to "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 editable text field 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.





Adding weight to multiple views

In this example, we only had one view with a weight attribute set. But what if we had *more* than one?

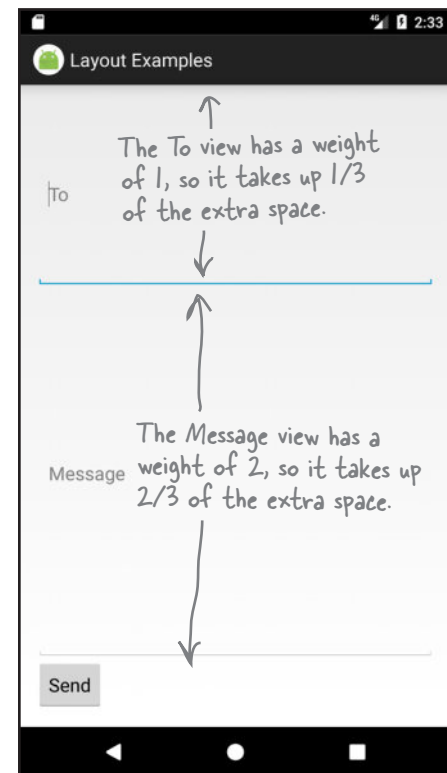
Suppose we gave the To text field a weight of 1, and the Message text field a weight of 2, like this:

```
<LinearLayout ... >
    ...
    <EditText
        android:layout_width="match_parent"
        android:layout_height="0dp"
        android:layout_weight="1"
        android:hint="@string/to" />

    <EditText
        android:layout_width="match_parent"
        android:layout_height="0dp"
        android:layout_weight="2"
        android:hint="@string/message" />
    ...
</LinearLayout>
```

To figure out how much extra space each view takes up, start by adding together the `layout_weight` attributes for each view. In our case, this is $1+2=3$. The amount of extra space taken up by each view will be 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.

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





Gravity controls the position of a view's contents

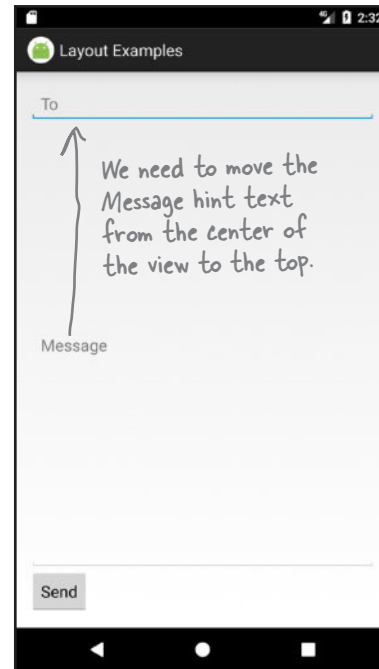
The next thing we need to do is move the hint text inside the Message text field. At the moment, it's centered vertically inside the view. We need to change it so that the text appears at the top of the text field. We can achieve this using the `android:gravity` attribute.

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 field. If you want the text inside a view to appear at the top, the following code will do the trick:

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

We'll add an `android:gravity` attribute to the Message text field so that the hint text moves to the top of the view:

```
<LinearLayout ... >
...
<EditText
    android:layout_width="match_parent"
    android:layout_height="0dp"
    android:layout_weight="1"
    android:gravity="top" ← Displays the text inside the text field
    android:hint="@string/message" />
...
</LinearLayout>
```



Test drive

Adding the `android:gravity` attribute to the Message text field moves the hint text to the top of the view, just like we want.

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





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"
```

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

Value	What it does
top	Puts the view's contents at the top of the view.
bottom	Puts the view's contents at the bottom of the view.
left	Puts the view's contents at the left of the view.
right	Puts the view's contents at the right 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.

start and end are only available if you're using API 17 or above.

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, use:

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

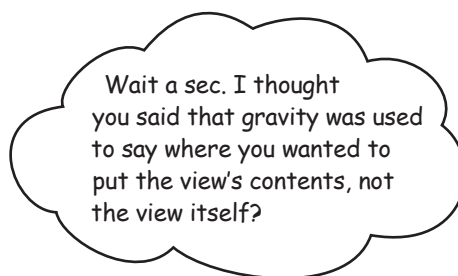
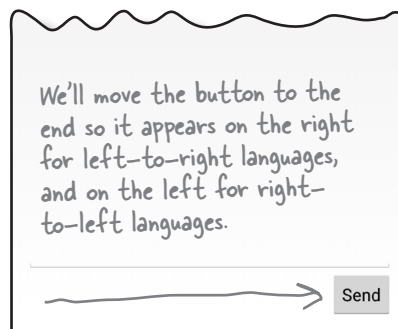


layout_gravity controls the position of a view within a layout

There's one final change we need to make to our layout. The Send button currently appears in the bottom-left corner. We need to move it over to the end instead (the bottom-right corner for left-to-right languages). To do this, we'll use the `android:layout_gravity` attribute.

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 our button to the end, we'd need to add the following to the button's code:

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



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

A couple of pages ago, we used the `android:gravity` attribute to position the Message text inside a text view. 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"
```

There's a list of some of the other values you can use with the `android:layout_gravity` attribute on the next page.





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"
```

You can apply multiple `layout_gravity` values by separating each one with a “|”. As an example, use `android:layout_gravity="bottom|end` to move a view to the bottom-end corner of its available space.

Value	What it does
top, bottom, left, right	Puts the view at the top, bottom, left, or right of its available space.
start, end	Puts the view at the 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.

`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 how the view’s contents are displayed.



The full linear layout code

Here's the full code for the linear layout:

```
<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="com.hfad.views.MainActivity" >
```

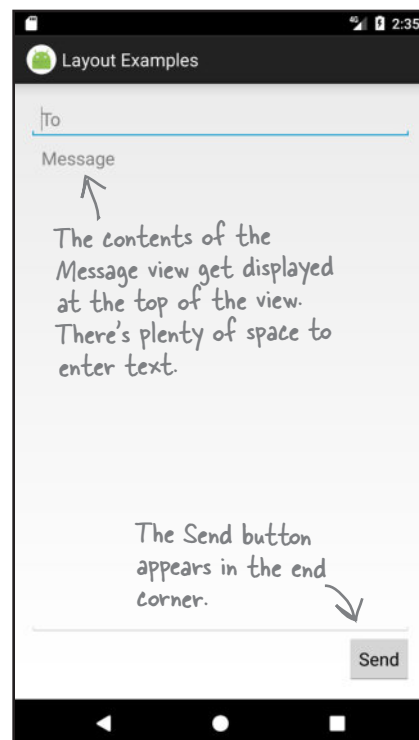
```
<EditText
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:hint="@string/to" />
```

```
<EditText
    android:layout_width="match_parent"
    android:layout_height="0dp"
    android:layout_weight="1"
    android:gravity="top"
    android:hint="@string/message" />
```

```
<Button
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_gravity="end"
    android:text="@string/send" />
```

```
</LinearLayout>
```

android:gravity is different from android:layout_gravity. android:gravity relates to the contents of the view, while android:layout_gravity relates to the view itself.





LinearLayout: a summary

Here's a summary of how you create linear layouts.

How you specify a linear layout

You specify a linear layout using `<LinearLayout>`. You must specify the layout's width, height, and orientation, but padding is optional:

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

Views get displayed in the order they're listed in the code

When you define a linear layout, you add views to the layout in the order in which you want them to appear.

Stretch views using weight

By default, all views take up just as much space as necessary for their content. If you want to make one or more of your views take up more space, you can use the `weight` attribute to make it stretch:

```
android:layout_weight="1"
```

Use gravity to specify where a view's contents appear within a view

The `android:gravity` attribute lets you specify how you want to position the contents of a view—for example, how you want to position text inside a text field.

Use `layout_gravity` to specify where a view appears in its available space

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

That's everything we've covered on linear layouts. Next we'll take a look at the **frame layout**.



Frame layouts stack their views

As you've already seen, linear layouts arrange their 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 on top of each other. This allows you to, for example, display text on top of an image.

How you define a frame layout

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

```

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

```

You use `<FrameLayout>` to define a frame layout.

These are the same attributes we used for our linear layout.

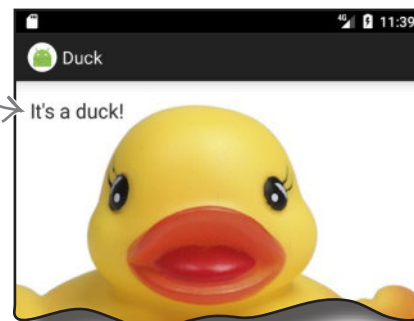
← This is where you add any views you wish to stack in the frame layout.

Just like a linear layout, the `android:layout_width` and `android:layout_height` attributes are mandatory and specify the layout's width and height.

Create a new project

To see how frame layouts work, we're going to use one to overlay text on an image. Create a new Android Studio project for an application named "Duck" with a company name of "hfad.com", making the package name `com.hfad.duck`. The minimum SDK should be API 19 so that it will work on most devices. You'll need an empty activity called "MainActivity" with a layout called "activity_main" so that your code matches ours. **Make sure you uncheck the Backwards Compatibility (AppCompat) option when you create the activity.**

Frame layouts let your views overlap one another. This is useful for displaying text on top of images, for example.





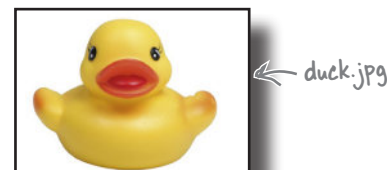
Add an image to your project

We're going to use an image called *duck.jpg* in our layout, so we need to add it to our project.

To do this, you first need to create a *drawable* resource folder (if Android Studio hasn't already created it 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.

Once you've created the *drawable* folder, download the file *duck.jpg* from <https://git.io/v9oet>, then add it to the *app/src/main/res/drawable* folder.

We're going to change *activity_main.xml* so that it uses a frame layout containing an image view (a view that displays an image) and a text view. To do this, replace the code in your version of *activity_main.xml* with ours below:



```
<?xml version="1.0" encoding="utf-8"?>
```

We're using a frame layout.

```
<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="com.hfad.duck.MainActivity">
```

This adds an image to the frame layout. You'll find out more about image views later in the chapter.

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

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

This line tells Android to use the image named "duck" located in the drawable folder.

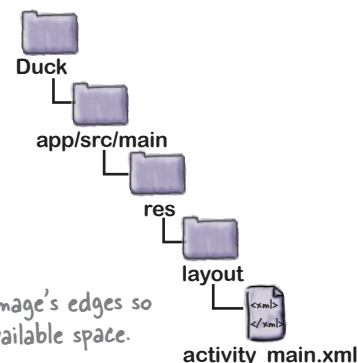
This adds a text view to the frame layout.

```
<TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:padding="16dp"
    android:textSize="20sp"
    android:text="It's a duck!" />
```

We've increased the size of the text.

```
</FrameLayout>
```

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



Then run your app, and we'll look at what the code does on the next page.

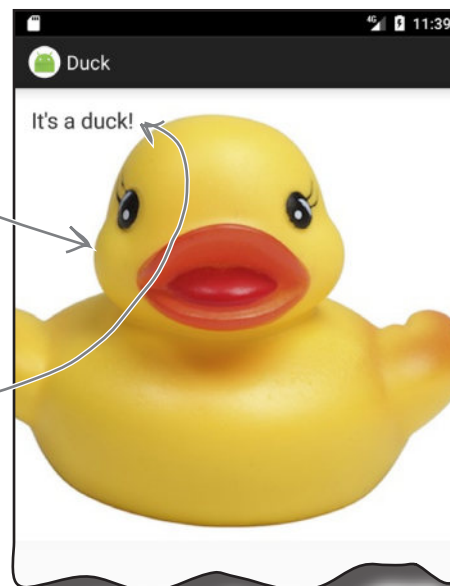


A frame layout stacks views in the order they appear in the layout XML

When you define a frame layout, you add views to the layout in the order in which you want them to be stacked. The first view is displayed first, the second is stacked on top of it, and so on. In our case, we've added an image view followed by a text view, so the text view appears on top of the image view:

```
<FrameLayout ...>
  <ImageView
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:scaleType="centerCrop"
    android:src="@drawable/duck"/>

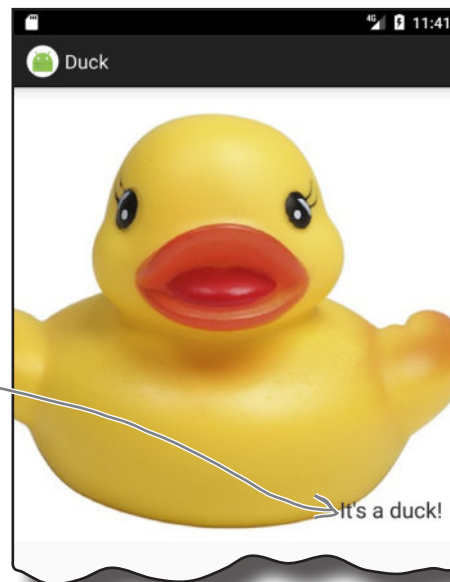
  <TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:padding="16dp"
    android:textSize="20sp"
    android:text="It's a duck!" />
</FrameLayout>
```



Position views in the layout using layout_gravity

By default, any views you add to a frame layout appear in the top-left corner. You can change the position of these views using the `android:layout_gravity` attribute, just as you could with a linear layout. As an example, here's how you would move the text view to the bottom-end corner of the image:

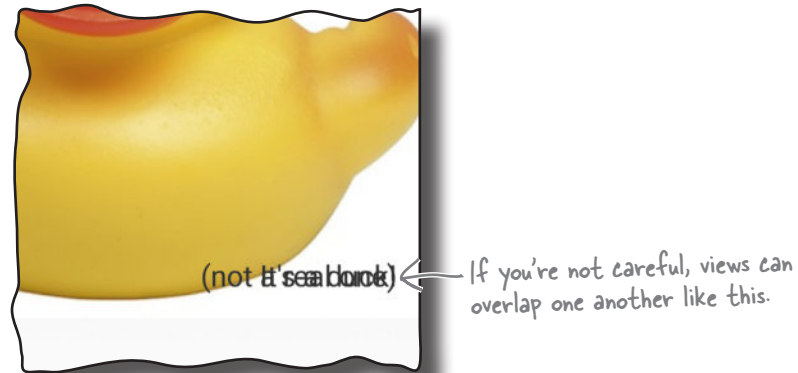
```
...
  <TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:padding="16dp"
    android:layout_gravity="bottom|end"
    android:textSize="20sp"
    android:text="It's a duck!" />
</FrameLayout>
```



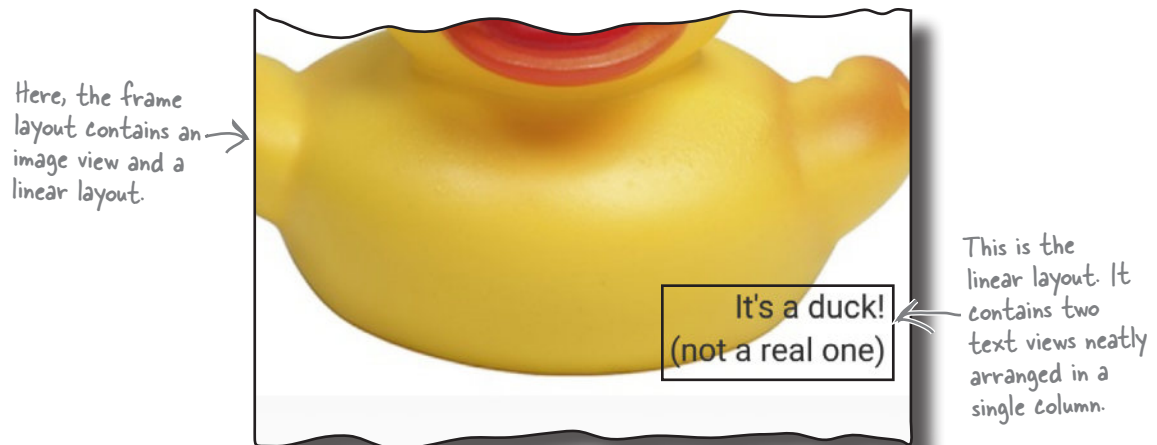


You can nest layouts

One of the disadvantages of using a frame layout is that it's easy for views to overlap one another when you don't want them to. As an example, you may want to display two text views in the bottom-end corner, one above the other:



It's possible to solve this problem by adding margins or padding to the text views. A neater solution, however, is to add them to a linear layout, which you then nest inside the frame layout. Doing this allows you to arrange the two text views linearly, then position them as a group inside the frame layout:



We'll show you the full code for how to do this on the next page.

The full code to nest a layout

Here's the full code to nest a linear layout in a frame layout. Update your version of `activity_main.xml` to include our changes, then run your app to see how it looks.

```
<?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="com.hfad.duck.MainActivity">

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

    <LinearLayout
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:orientation="vertical"
        android:layout_gravity="bottom|end"
        android:gravity="end"
        android:padding="16dp" >

        <TextView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:textSize="20sp"
            android:text="It's a duck!" />

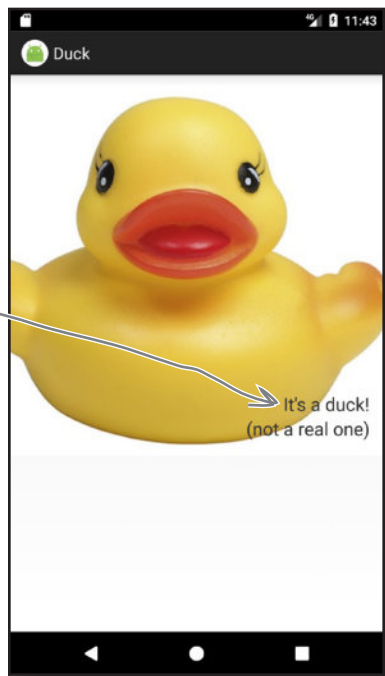
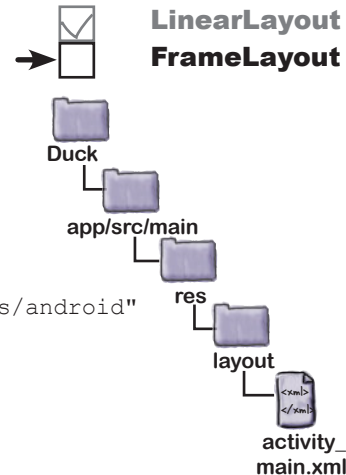
        <TextView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:textSize="20sp"
            android:text="(not a real one)" />

    </LinearLayout>
</FrameLayout>
```

Move each text view in the linear layout to the end of its available space.

We're adding a linear layout that's just big enough to contain its text views.

This line sinks the linear layout to the bottom-end corner of the frame layout.





FrameLayout: a summary

Here's a summary of how you create frame layouts.

How you specify a frame layout

You specify a frame layout using `<FrameLayout>`. You must specify the layout's width and height:

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

    ...

</FrameLayout>
```

Views are stacked in the order they appear

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

Use `layout_gravity` to specify where a view appears

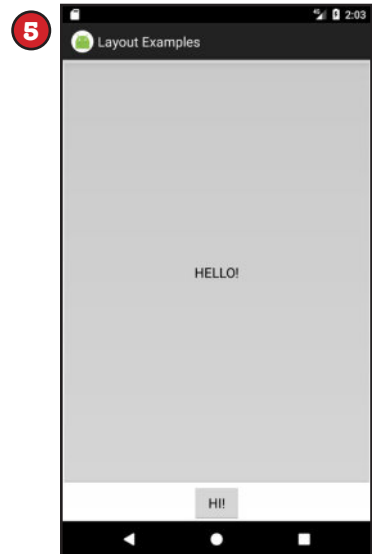
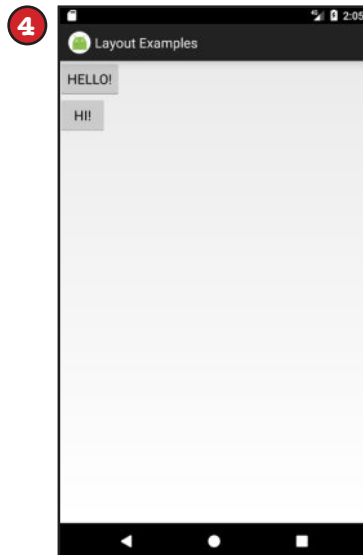
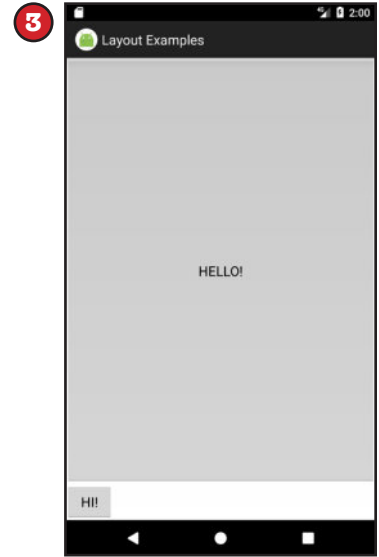
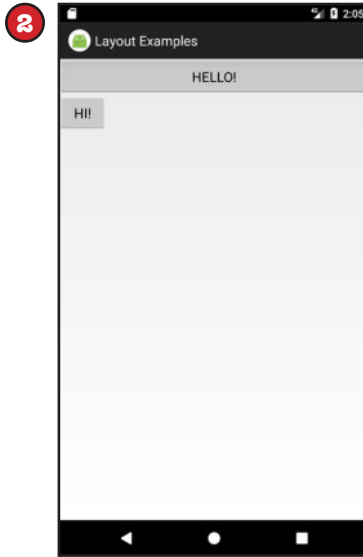
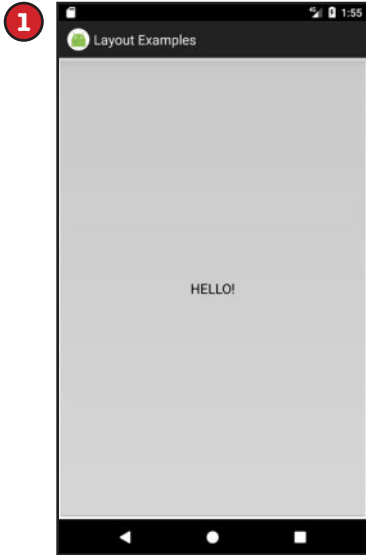
The `android:layout_gravity` attribute lets you specify where you want a view in a frame layout to appear. You can use it to push a view to the end, for instance, or sink it to the bottom-end corner.

Now that you've seen how to use two simple Android layouts, a linear layout and a frame layout, have a go at the following exercise.



BE the Layout

Three of the five screens below were made from layouts on the next page. Your job is to match each of the three layouts to the screen that the layout would produce.



A

```
<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="com.hfad.views.MainActivity" >
    <Button
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:text="HELLO!" />
</LinearLayout>
```

B

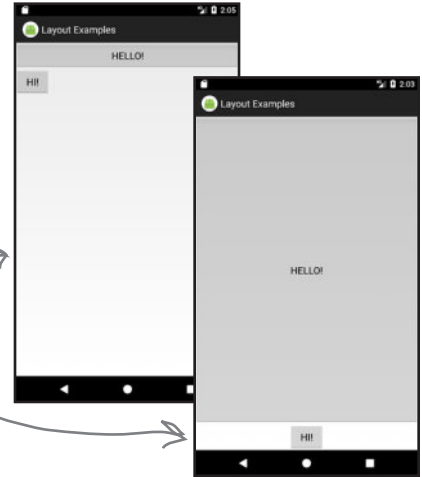
```
<LinearLayout xmlns:android=
    "http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    tools:context="com.hfad.views.MainActivity" >
    <Button
        android:layout_width="match_parent"
        android:layout_height="0dp"
        android:layout_weight="1"
        android:text="HELLO!" />
    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="HI!" />
</LinearLayout>
```

C

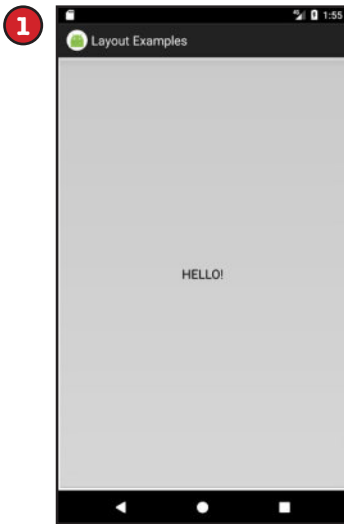
```
<LinearLayout xmlns:android=
    "http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    tools:context="com.hfad.views.MainActivity" >
    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="HELLO!" />
    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="HI!" />
</LinearLayout>
```

BE the Layout Solution

Three of the five screens below were made from layouts on the next page. Your job is to match each of the three layouts to the screen that the layout would produce.



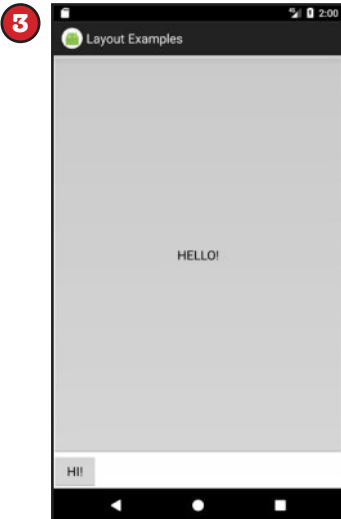
None of the layouts produce these screens.



A

```
<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="com.hfad.views.MainActivity" >
    <Button
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:text="HELLO!" />
</LinearLayout>
```

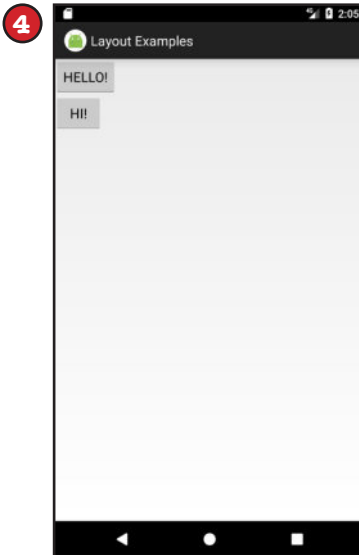
This has one button that fills the screen.



B

```
<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="com.hfad.views.MainActivity" >
    <Button
        android:layout_width="match_parent"
        android:layout_height="0dp"
        android:layout_weight="1"
        android:text="HELLO!" />
    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="HI!" />
</LinearLayout>
```

This button fills the screen, leaving space for another one underneath it.



```

C <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="com.hfad.views.MainActivity" >
    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="HELLO!" />
    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="HI!" />
</LinearLayout>

```

Both buttons have their `layout_width` and `layout_height` properties set to `"wrap_content"`, so they take up just enough space to display their contents.

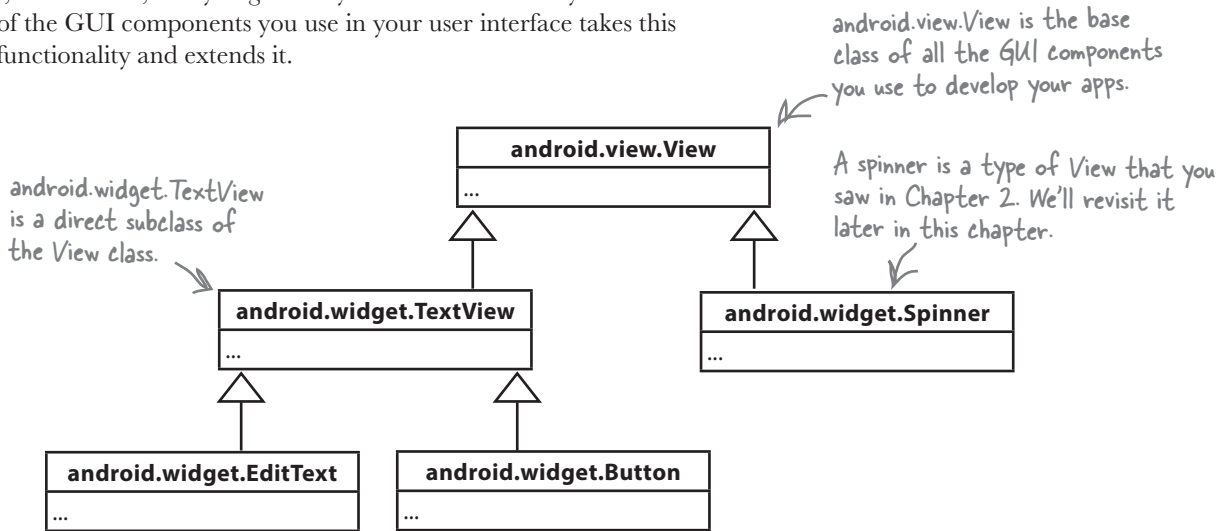
Layouts and GUI components have a lot in common

You may have noticed that all layout types have attributes in common. Whichever type of layout you use, you must specify the layout's width and height using the `android:layout_width` and `android:layout_height` attributes. And this requirement isn't just limited to layouts—the `android:layout_width` and `android:layout_height` are mandatory for all GUI components too.

This is because **all layouts and GUI components are subclasses of the Android View class**. Let's look at this in more detail.

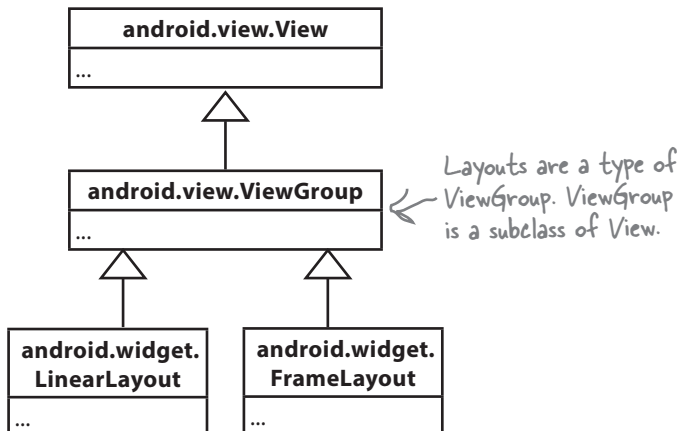
GUI components are a type of View

You've already seen that GUI components are all types of views—behind the scenes, they are all subclasses of the `android.view.View` class. This means that all of the GUI components in your user interface have attributes and behavior in common. They can all be displayed on the screen, for instance, and you get to say how tall or wide they should be. Each of the GUI components you use in your user interface takes this basic functionality and extends it.



Layouts are a type of View called a ViewGroup

It's not just the GUI components that are a type of view. Under the hood, a layout is a special type of view called a **view group**. All layouts are subclasses of the `android.view.ViewGroup` class. A view group is a type of view that can contain other views.



A GUI component is a type of view, an object that takes up space on the screen.

A layout is a type of view group, which is a special type of view that can contain other views.

What being a view buys you

A `View` object occupies rectangular space on the screen. It includes the functionality all views need in order to lead a happy helpful life in Androidville. Here are some of the qualities of views that we think are the most important:

Getting and setting properties

Each view is a Java object behind the scenes, and that means you can get and set its properties in your activity code. As an example, you can retrieve the value selected in a spinner or change the text in a text view. The exact properties and methods you can access depend on the type of view.

To help you get and set view properties, each view can have an ID associated with it so that you can refer to it in your code.

Size and position

You specify the width and height of views so that Android knows how big they need to be. You can also say whether any padding is needed around the view.

Once your view has been displayed, you can retrieve the position of the view, and its actual size on the screen.

Focus handling

Android handles how the focus moves depending on what the user does. This includes responding to any views that are hidden, removed, or made visible.

Event handling and listeners

Views can respond to events. You can also create listeners so that your app can react to things happening in the view. As an example, all views can react to getting or losing the focus, and a button (and all of its subclasses) can react to being clicked.

As a view group is also a type of view, this means that all layouts and GUI components share this common functionality.

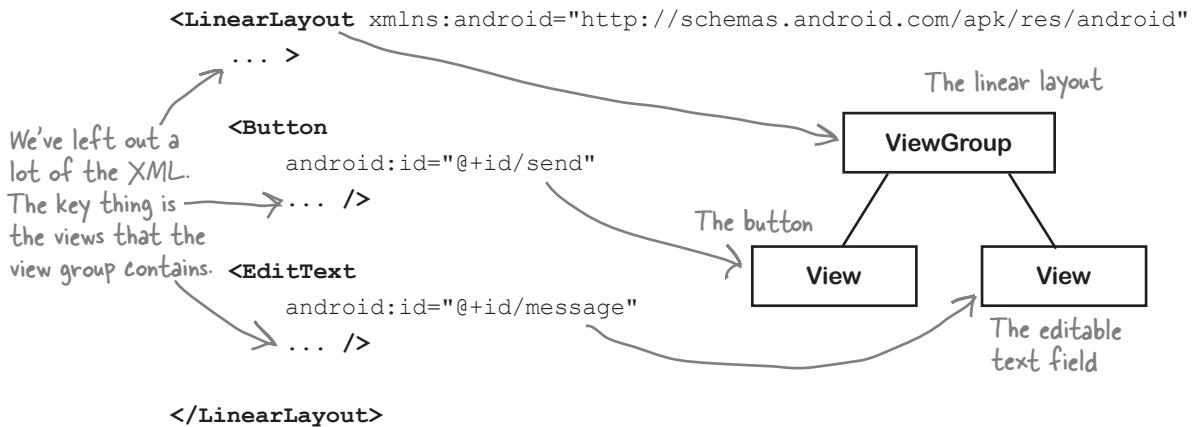
Here are some of the `View` methods you can use in your activity code. As these are in the base `View` class, they're common to all views and view groups.



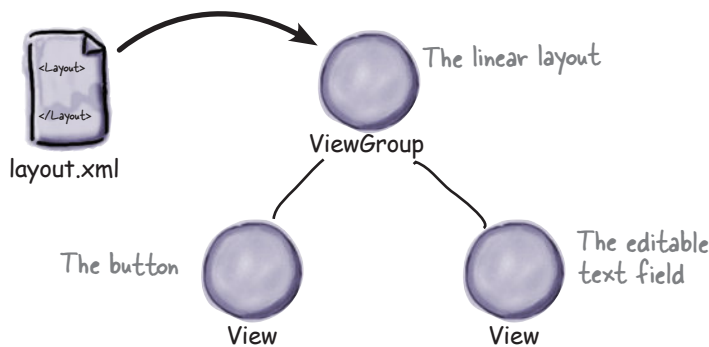
android.view.View
<code>getId()</code>
<code>getHeight()</code>
<code>getWidth()</code>
<code>setVisibility(int)</code>
<code>findViewById(int)</code>
<code>isClickable()</code>
<code>isFocused()</code>
<code>requestFocus()</code>
...

A layout is really a hierarchy of Views

The layout you define using XML gives you a *hierarchical tree of views and view groups*. As an example, here's a linear layout containing a button and an editable text field. The linear layout is a view group, and the button and text field are both views. The view group is the view's parent, and the views are the view group's children:



Behind the scenes, when you build your app, the layout XML is converted to a **ViewGroup** object containing a tree of Views. In the example above, the button gets translated to a **Button** object, and the text view gets translated to a **TextView** object. **Button** and **TextView** are both subclasses of **View**.



This is the reason why you can manipulate the views in your layout using Java code. Behind the scenes, all of the views are rendered to Java **View** objects.

Playing with views

Let's look at the most common GUI components. You've already seen some of these, but we'll review them anyway. We won't show you the whole API for each of these—just selected highlights to get you started.

Text view



This is a text view

A text view is used for displaying text.

Defining it in XML

You define a text view in your layout using the `<TextView>` element. You use `android:text` to say what text you want it to display, usually by using a String resource:

```
<TextView
    android:id="@+id/text_view"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/text" />
```

The `TextView` API includes many attributes to control the text view's appearance, such as the text size. To change the text size, you use the `android:textSize` attribute like this:

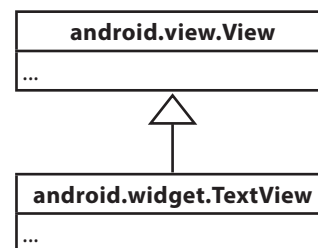
```
android:textSize="16sp"
```

You specify the text size using scale-independent pixels (sp). Scale-independent pixels take into account whether users want to use large fonts on their devices. A text size of 16sp will be physically larger on a device configured to use large fonts than on a device configured to use small fonts.

Using it in your activity code

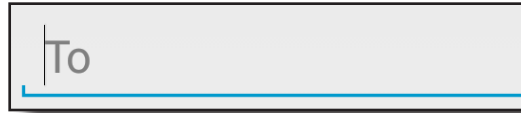
You can change the text displayed in your text view using code like this:

```
TextView textView = (TextView) findViewById(R.id.text_view);
textView.setText("Some other String");
```



Editable text view

This is like a text view, but editable.



Defining it in XML

You define an editable text view in XML using the `<EditText>` element. You use the `android:hint` attribute to give a hint to the user as to how to fill in the field.

```
<EditText
    android:id="@+id/edit_text"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:hint="@string/edit_text" />
```

You can use the `android:inputType` attribute to define what type of data you're expecting the user to enter so that Android can help them. As an example, if you're expecting the user to enter numbers, you can use:

```
android:inputType="number"
```

to provide them with a number keypad. Here are some more of our favorites:

Value	What it does
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.
textAutoCorrect	Automatically corrects the text being input.

You can find the entire list in the online Android developer documentation at https://developer.android.com/reference/android/widget/TextView.html#attr_android:inputType.

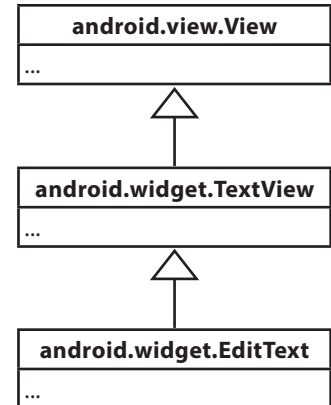
You can specify multiple input types using the `|` character. As an example, to capitalize the first word of a sentence and automatically correct any misspellings, you'd use:

```
android:inputType="textCapSentences|textAutoCorrect"
```

Using it in your activity code

You can retrieve the text entered in an editable text view like this:

```
EditText editText = (EditText) findViewById(R.id.edit_text);
String text = editText.getText().toString();
```



Button

Buttons are usually used to make your app do something when they're clicked.



Defining it in XML

You define a button in XML using the `<Button>` element. You use the `android:text` attribute to say what text you want the button to display:

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

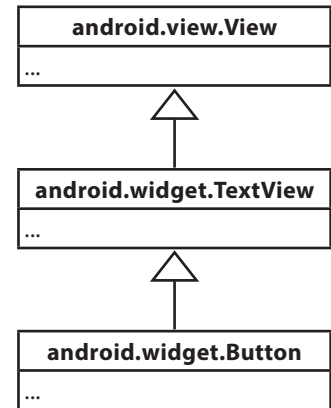
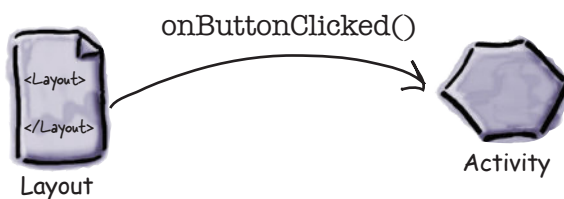
Using it in your activity code

You get the button to respond to the user clicking it by using the `android:onClick` attribute in the layout XML, and setting it to the name of the method you want to call in your activity code:

```
android:onClick="onButtonClicked"
```

You then define the method in your activity like this:

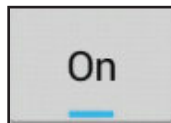
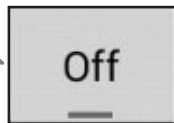
```
/** Called when the button is clicked */
public void onButtonClicked(View view) {
    // Do something in response to button click
}
```



Toggle button

A toggle button allows you to choose between two states by clicking a button.

This is what the toggle button looks like when it's off.



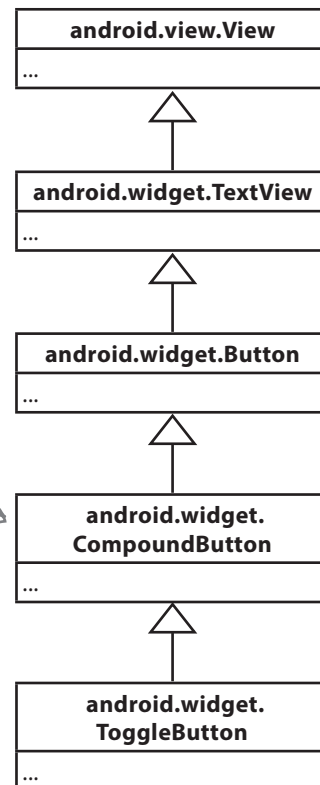
When you click the toggle button, it changes to being on.

Defining it in XML

You define a toggle button in XML using the `<ToggleButton>` element. You use the `android:textOn` and `android:textOff` attributes to say what text you want the button to display depending on the state of the button:

```
<ToggleButton
    android:id="@+id/toggle_button"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:textOn="@string/on"
    android:textOff="@string/off" />
```

A compound button is a button with two states, checked and unchecked. A toggle button is an implementation of a compound button.



Using it in your activity code

You get the toggle button to respond to the user clicking it by using the `android:onClick` attribute in the layout XML. You give it the name of the method you want to call in your activity code:

```
android:onClick="onToggleButtonClicked"
```

This is exactly the same as calling a method when a normal button gets clicked.

You then define the method in your activity like this:

```
/** Called when the toggle button is clicked */
public void onToggleButtonClicked(View view) {
    // Get the state of the toggle button.
    boolean on = ((ToggleButton) view).isChecked();
    if (on) {
        // On
    } else {
        // Off
    }
}
```

This returns true if the toggle button is on, and false if the toggle button is off.

Switch

A switch is a slider control that acts in the same way as a toggle button.



Defining it in XML

You define a switch in XML using the `<Switch>` element. You use the `android:textOn` and `android:textOff` attributes to say what text you want the switch to display depending on the state of the switch:

```
<Switch
    android:id="@+id/switch_view"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:textOn="@string/on"
    android:textOff="@string/off" />
```

Using it in your activity code

You get the switch to respond to the user clicking it by using the `android:onClick` attribute in the layout XML, and setting it to the name of the method you want to call in your activity code:

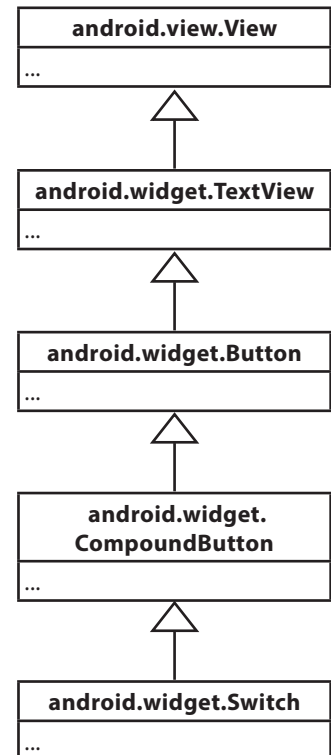
```
android:onClick="onSwitchClicked"
```

You then define the method in your activity like this:

```
/** Called when the switch is clicked */
public void onSwitchClicked(View view) {
    // Is the switch on?
    boolean on = ((Switch) view).isChecked();

    if (on) {
        // On
    } else {
        // Off
    }
}
```

↑
This code is very similar to that used with the toggle button.



Checkboxes

Checkboxes let you display multiple options to users. They can then select whichever options they want. Each of the checkboxes can be checked or unchecked independently of any others.



← Here we have two checkboxes. Users can choose milk, sugar, both milk and sugar, or neither.

Defining them in XML

You define each checkbox in XML using the `<CheckBox>` element. You use the `android:text` attribute to display text for each option:

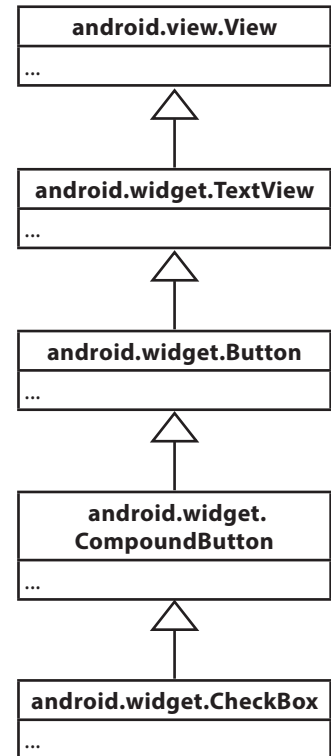
```
<CheckBox android:id="@+id/checkbox_milk"
  android:layout_width="wrap_content"
  android:layout_height="wrap_content"
  android:text="@string/milk" />
```

```
<CheckBox android:id="@+id/checkbox_sugar"
  android:layout_width="wrap_content"
  android:layout_height="wrap_content"
  android:text="@string/sugar" />
```

Using them in your activity code

You can find whether a particular checkbox is checked using the `isChecked()` method. It returns `true` if the checkbox is checked:

```
CheckBox checkbox = (CheckBox) findViewById(R.id.checkbox_milk);
boolean checked = checkbox.isChecked();
if (checked) {
    //do something
}
```



Checkboxes (continued)

Just like buttons, you can respond to the user clicking a checkbox by using the `android:onClick` attribute in the layout XML, and setting it to the name of the method you want to call in your activity code:

```
<CheckBox android:id="@+id/checkbox_milk"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/milk"
    android:onClick="onCheckboxClicked"/>

<CheckBox android:id="@+id/checkbox_sugar"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/sugar"
    android:onClick="onCheckboxClicked"/>
```

In this case, the `onCheckboxClicked()` method will get called no matter which checkbox gets clicked. We could have specified a different method for each checkbox if we'd wanted to.

You then define the method in your activity like this:

```
public void onCheckboxClicked(View view) {
    // Has the checkbox that was clicked been checked?
    boolean checked = ((CheckBox) view).isChecked();

    // Retrieve which checkbox was clicked
    switch(view.getId()) {
        case R.id.checkbox_milk:
            if (checked)
                // Milky coffee
            else
                // Black as the midnight sky on a moonless night
            break;
        case R.id.checkbox_sugar:
            if (checked)
                // Sweet
            else
                // Keep it bitter
            break;
    }
}
```

Radio buttons

These let you display multiple options to the user. The user can select a single option.

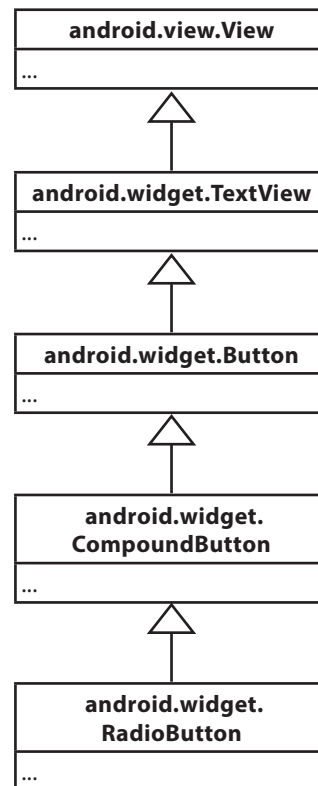


Defining them in XML

You start by defining a radio group, a special type of view group, using the `<RadioGroup>` tag. Within this, you then define individual radio buttons using the `<RadioButton>` tag:

```
<RadioGroup android:id="@+id/radio_group"
  android:layout_width="match_parent"
  android:layout_height="wrap_content"
  android:orientation="vertical">
  <RadioButton android:id="@+id/radio_cavemen"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/cavemen" />
  <RadioButton android:id="@+id/radio_astronauts"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/astronauts" />
</RadioGroup>
```

You can choose to display the radio buttons in a horizontal or vertical list.



Using them in your activity code

You can find which radio button is selected using the `getCheckedRadioButtonId()` method:

```
RadioGroup radioGroup = (RadioGroup) findViewById(R.id.radioGroup);
int id = radioGroup.getCheckedRadioButtonId();
if (id == -1){
    //no item selected
}
else{
    RadioButton radioButton = findViewById(id);
}
```

Radio buttons (continued)

You can respond to the user clicking a radio button by using the `android:onClick` attribute in the layout XML, and setting it to the name of the method you want to call in your activity code:

```
<RadioGroup android:id="@+id/radio_group"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:orientation="vertical">

    <RadioButton android:id="@+id/radio_cavemen"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/cavemen"
        android:onClick="onRadioButtonClicked" />

    <RadioButton android:id="@+id/radio_astronauts"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/astronauts"
        android:onClick="onRadioButtonClicked" />

</RadioGroup>
```

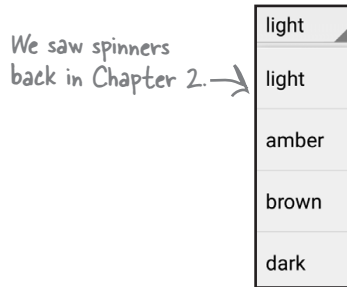
You then define the method in your activity like this:

```
public void onRadioButtonClicked(View view) {
    RadioGroup radioGroup = (RadioGroup) findViewById(R.id.radioGroup);
    int id = radioGroup.getCheckedRadioButtonId();
    switch(id) {
        case R.id.radio_cavemen:
            // Cavemen win
            break;
        case R.id.radio_astronauts:
            // Astronauts win
            break;
    }
}
```

The radio group containing the radio buttons is a subclass of `LinearLayout`. You can use the same attributes with a radio group as you can with a linear layout.

Spinner

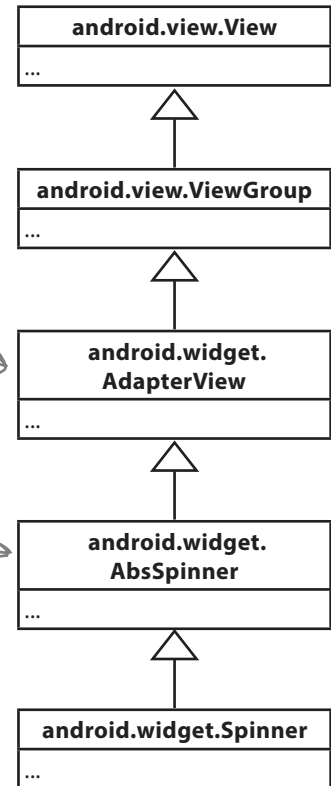
As you've already seen, a spinner gives you a drop-down list of values from which only one can be selected.



An AdapterView is a view that can use an adapter. You'll find out about adapters later in the book.

This is an abstract base class for Spinner widgets.

There are other ways of populating the spinner, which you'll see later in the book.



Defining it in XML

You define a spinner in XML using the `<Spinner>` element. You add a static array of entries to the spinner by using the `android:entries` attribute and setting it to an array of Strings.

```

<Spinner
    android:id="@+id/spinner"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:entries="@array/spinner_values" />
  
```

You can add an array of Strings to *strings.xml* like this:

```

<string-array name="spinner_values">
    <item>light</item>
    <item>amber</item>
    <item>brown</item>
    <item>dark</item>
</string-array>
  
```

Using it in your activity code

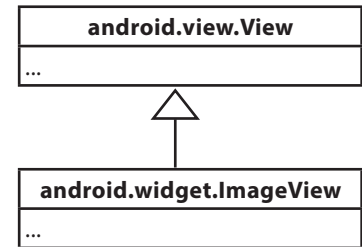
You can get the value of the currently selected item by using the `getSelectedItem()` method and converting it to a `String`:

```

Spinner spinner = (Spinner) findViewById(R.id.spinner);
String string = String.valueOf(spinner.getSelectedItem());
  
```

Image view

You use an image view to display an image:



The ImageView class is a direct subclass of View.

Adding an image to your project

You first need to create a *drawable* resource folder, the default folder for storing image resources in your app. To do this, select the *app/src/main/res* folder in your project, 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. You then need to add your image to the *app/src/main/res/drawable* folder.

If you want, you can use different image files depending on the screen density of the device. This means you can display higher-resolution images on higher-density screens, and lower-resolution images on lower-density screens. To do this, you create different *drawable* folders in *app/src/main/res* for the different screen densities. The name of the folder relates to the screen density of the device:

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.

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

You then put different resolution images in each of the *drawable** folders, making sure that each of the image files has the same name. Android decides which image to use at runtime, depending on the screen density of the device it's running on. As an example, if the device has an extra-high-density screen, 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.

Image view: the layout XML

You define an image view in XML using the `<ImageView>` element. You use the `android:src` attribute to specify what image you want to display, and the `android:contentDescription` attribute to add a String description of the image so that your app is more accessible:

```
<ImageView
    android:layout_width="200dp"
    android:layout_height="100dp"
    android:src="@drawable/starbuzz_logo"
    android:contentDescription="@string/starbuzz_logo" />
```

The `android:src` attribute takes a value of the form `"@drawable/image_name"`, where `image_name` is the name of the image (without its extension). Image resources are prefixed with `@drawable`, which tells Android that it's an image resource located in one or more of the `drawable*` folders.

Using image views in your activity code

You can set the image source and description in your activity code using the `setImageResource()` and `setContentDescription()` methods:

```
ImageView photo = (ImageView) findViewById(R.id.photo);
int image = R.drawable.starbuzz_logo;
String description = "This is the logo";
photo.setImageResource(image);
photo.setContentDescription(description);
```

This code looks for the image resource called `starbuzz_logo` in the `drawable*` folders, and sets it as the source of an image view with an ID of `photo`. When you need to refer to an image resource in your activity code, you use `R.drawable.image_name` where `image_name` is the name of the image (without its extension).

Adding images to buttons

In addition to displaying images in image views, you can also display images on buttons.

Displaying text and an image on a button

To display text on a button with an image to the right of it, use the `android:drawableRight` attribute and specify the image to be used:

```
<Button
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:drawableRight="@drawable/android"
    android:text="@string/click_me" />
```

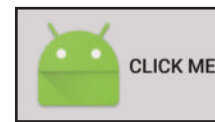
Display the android image resource on the right side of the button.



If you want to display the image on the left, use the `android:drawableLeft` attribute:

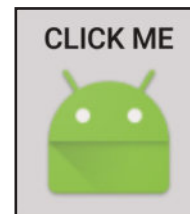
```
<Button
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:drawableLeft="@drawable/android"
    android:text="@string/click_me" />
```

You can also use `drawableStart` and `drawableEnd` to support right-to-left languages.



Use the `android:drawableBottom` attribute to display the image underneath the text:

```
<Button
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:drawableBottom="@drawable/android"
    android:text="@string/click_me" />
```



The `android:drawableTop` attribute displays the image above the text:

```
<Button
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:drawableTop="@drawable/android"
    android:text="@string/click_me" />
```



Image button

An image button is just like a button, except it contains an image and no text.



Defining it in XML

You define an image button in XML using the `<ImageButton>` element. You use the `android:src` attribute to say what image you want the image button to display:

```
<ImageButton
    android:id="@+id/button"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:src="@drawable/button_icon" />
```

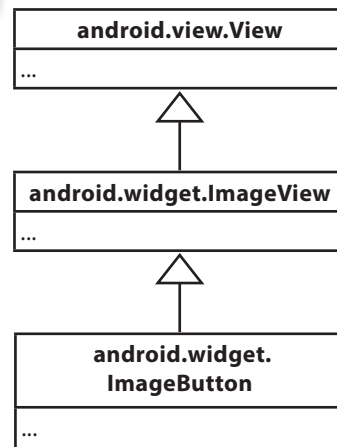
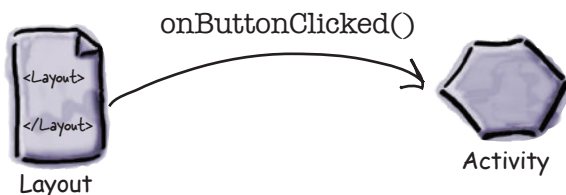
Using it in your activity code

You get the image button to respond to the user clicking it by using the `android:onClick` attribute in the layout XML, and setting it to the name of the method you want to call in your activity code:

```
android:onClick="onButtonClicked"
```

You then define the method in your activity like this:

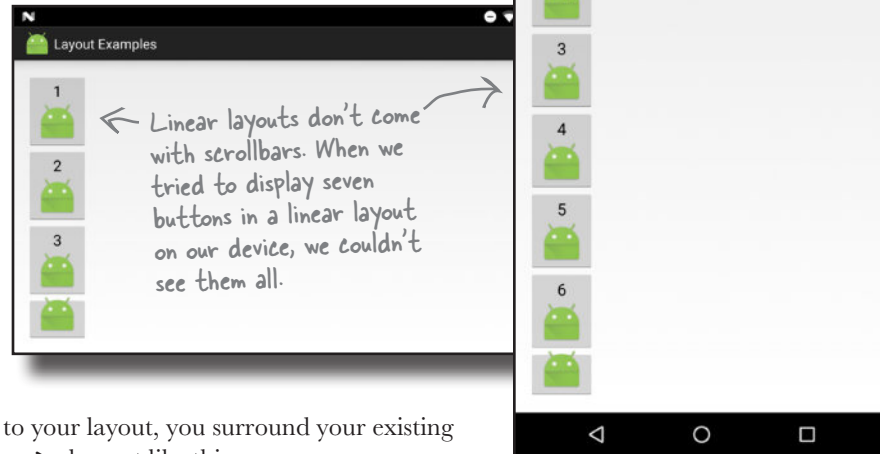
```
/** Called when the image button is clicked */
public void onButtonClicked(View view) {
    // Do something in response to button click
}
```



The `ImageButton` class extends the `ImageView` class, not the `Button` class.

Scroll view

If you add lots of views to your layouts, you may have problems on devices with smaller screens—most layouts don't come with scrollbars to allow you to scroll down the page. As an example, when we added seven large buttons to a linear layout, we couldn't see all of them.



To add a vertical scrollbar to your layout, you surround your existing layout with a **<ScrollView>** element like this:

```
<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"
    tools:context="com.hfad.views.MainActivity" >
```

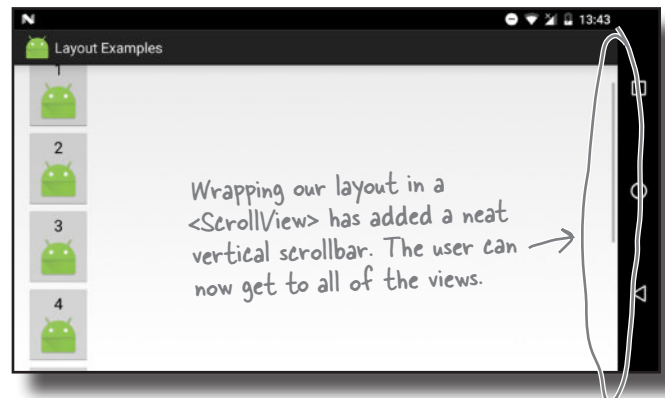
Move these attributes from the original layout to the **<ScrollView>**, as the **<ScrollView>** is now the root element.

```
<LinearLayout
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingBottom="16dp"
    android:paddingLeft="16dp"
    android:paddingRight="16dp"
    android:paddingTop="16dp"
    android:orientation="vertical" >
```

...

```
</LinearLayout>
```

```
</ScrollView>
```



To add a horizontal scrollbar to your layout, wrap your existing layout inside a **<HorizontalScrollView>** element instead.

Toasts

There's one final widget we want to show you in this chapter: a toast. A toast is a simple pop-up message you can display on the screen.

Toasts are purely informative, as the user can't interact with them. While a toast is displayed, the activity stays visible and interactive. The toast automatically disappears when it times out.

Using it in your activity code

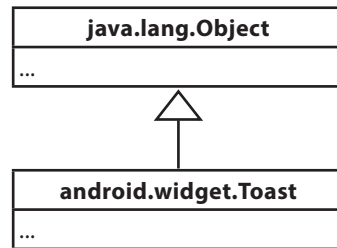
You create a toast using activity code only. You can't define one in your layout.

To create a toast, you call the `Toast.makeText()` method, and pass it three parameters: a `Context` (usually `this` for the current activity), a `CharSequence` that's the message you want to display, and an `int` duration. Once you've created the toast, you call its `show()` method to display it.

Here's the code you would use to create a toast that appears on screen for a short duration:

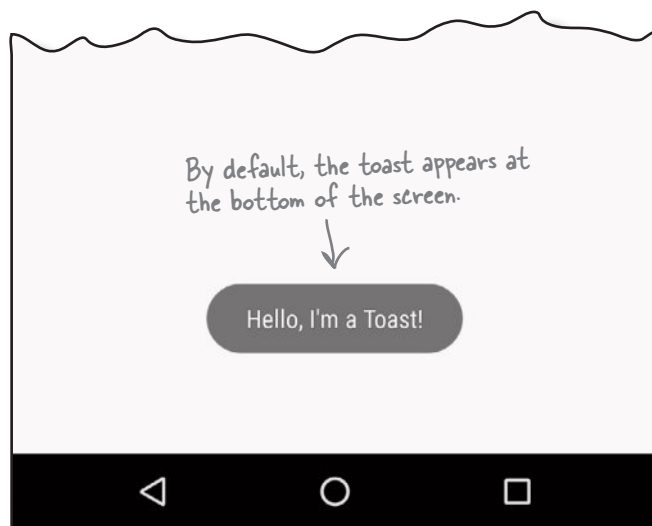
```
CharSequence text = "Hello, I'm a Toast!";
int duration = Toast.LENGTH_SHORT;

Toast toast = Toast.makeText(this, text, duration);
toast.show();
```



A toast isn't actually a type of view. It's a useful way of displaying a short message to the user, though, so we're sneaking it into this chapter.

A toast is a message that pops up like toast in a toaster.

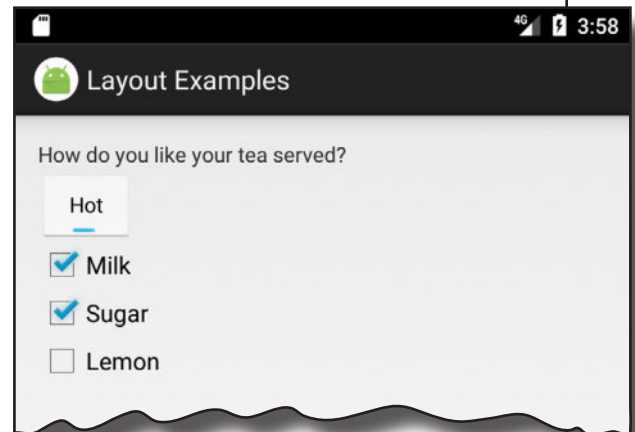




Exercise

It's time for you to try out some of the views we've introduced you to this chapter. Create a layout that will create this screen:

You probably won't want to write the code here, but why not experiment in the IDE?





Exercise Solution

Here's one of the many ways in which you can create the layout. Don't worry if your code looks different, as there are many different solutions.

```
<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="com.hfad.layoutexamples.MainActivity" >
```

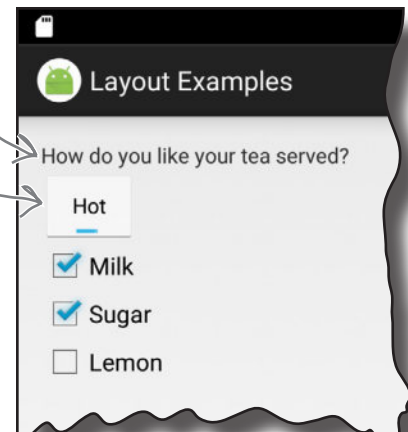
```
<TextView
```

```
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="How do you like your tea served?" />
```

```
<ToggleButton
```

```
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:textOn="Hot"
    android:textOff="Cold" />
```

We used a toggle button to display whether the drink should be served hot or cold.

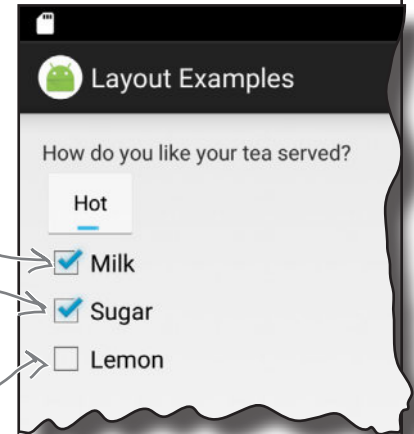


We used a checkbox for each of the values (Milk, Sugar, and Lemon). We put each one on a separate row.

```

<CheckBox android:id="@+id/checkbox_milk"
  android:layout_width="wrap_content"
  android:layout_height="wrap_content"
  android:text="Milk" />
<CheckBox android:id="@+id/checkbox_sugar"
  android:layout_width="wrap_content"
  android:layout_height="wrap_content"
  android:text="Sugar" />
<CheckBox android:id="@+id/checkbox_lemon"
  android:layout_width="wrap_content"
  android:layout_height="wrap_content"
  android:text="Lemon" />

```



```
</LinearLayout>
```

Remember, your code may look different from ours. This is just one way of building the layout.



Your Android Toolbox

You've got Chapter 5 under your belt and now you've added views and view groups to your toolbox.

You can download the full code for the chapter from <https://tinyurl.com/HeadFirstAndroid>.



BULLET POINTS

- GUI components are all types of view. They are all subclasses of the `android.view.View` class.
- All layouts are subclasses of the `android.view.ViewGroup` class. A view group is a type of view that can contain multiple views.
- The layout XML file gets converted to a `ViewGroup` containing a hierarchical tree of views.
- A linear layout lists views either horizontally or vertically. You specify the direction using the `android:orientation` attribute.
- A frame layout stacks views.
- Use `android:padding*` attributes to specify how much padding you want around a view.
- In a linear layout, use `android:layout_weight` if you want a view to use up extra space in the layout.
- `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.
- `<ToggleButton>` defines a toggle button that allows you to choose between two states by clicking a button.
- `<Switch>` defines a switch control that behaves in the same way as a toggle button. It requires API level 14 or above.
- `<CheckBox>` defines a checkbox.
- 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>`.
- Use `<ImageView>` to display an image.
- `<ImageButton>` defines a button with no text, just an image.
- Add scrollbars using `<ScrollView>` or `<HorizontalScrollView>`.
- A `Toast` is a pop-up message.

6 constraint layouts

Put Things in Their Place



Of course I'm sure, I have the blueprint right in front of me. It says the toe bone's connected to the foot bone, and the foot bone's connected to the ankle bone.

Let's face it, you need to know how to create great layouts.

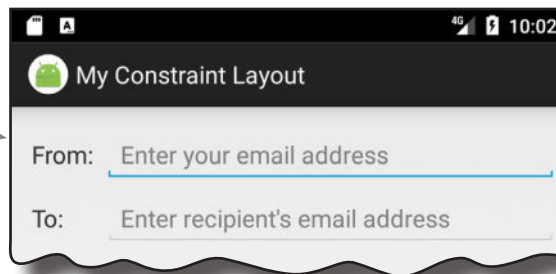
If you're building apps you want people to **use**, you need to make sure they **look exactly the way you want**. So far you've seen how to use linear and frame layouts, but *what if your design is more complex?* To deal with this, we'll introduce you to Android's new **constraint layout**, a type of layout you **build visually using a blueprint**. We'll show you how **constraints** let you position and size your views, *irrespective of screen size and orientation*. Finally, you'll find out how to save time by making Android Studio **infer and add constraints** on your behalf.

Nested layouts can be inefficient

You've seen how to build a simple user interface using linear layouts and frame layouts, but what if you need to create something more complex? While complex UIs are possible if you nest your linear and frame layouts deep enough, they can slow down your app and make your code hard to read and maintain.

As an example, suppose you wanted to create a layout containing two rows, each containing two items. One possibility would be to create this layout using three linear layouts: one linear layout at the root, and one nested linear layout for each row:

Linear layouts only allow you to display views in a single column or row, so you can't use a single linear layout to construct layouts like this. You can, however, nest linear layouts to produce the results you need.



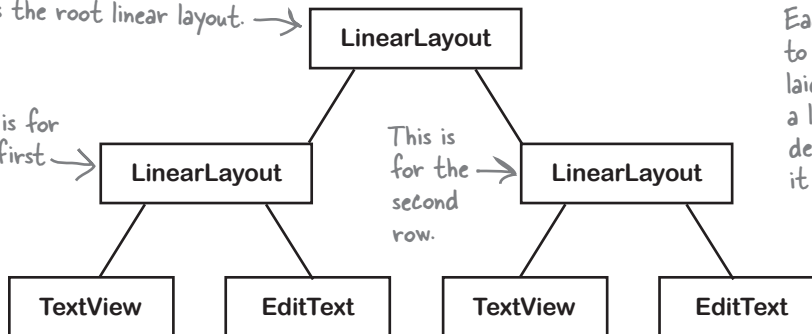
For this layout, you could use one linear layout at the root, and one nested linear layout for each row.

When Android displays a layout on the device screen, it creates a hierarchy of views based on the layout components which helps it figure out where each view should be placed. If the layout contains nested layouts, the hierarchy is more complex, and Android may need to make more than one pass through the hierarchy:

This is the root linear layout.

This is for the first row.

This is for the second row.



Each view and layout needs to be initialized, measured, laid out, and drawn. This is a lot of work if you have deeply nested layouts, and it can slow down your app.

While the above hierarchy is still relatively simple, imagine if you needed more views, more nested layouts, and a deeper hierarchy. This could lead to bottlenecks in your app's performance, and leave you with a mass of code that's difficult to read and maintain.

If you have a more complex UI that requires you to nest multiple layouts, it's usually better to **use a different type of layout**.

Introducing the constraint layout

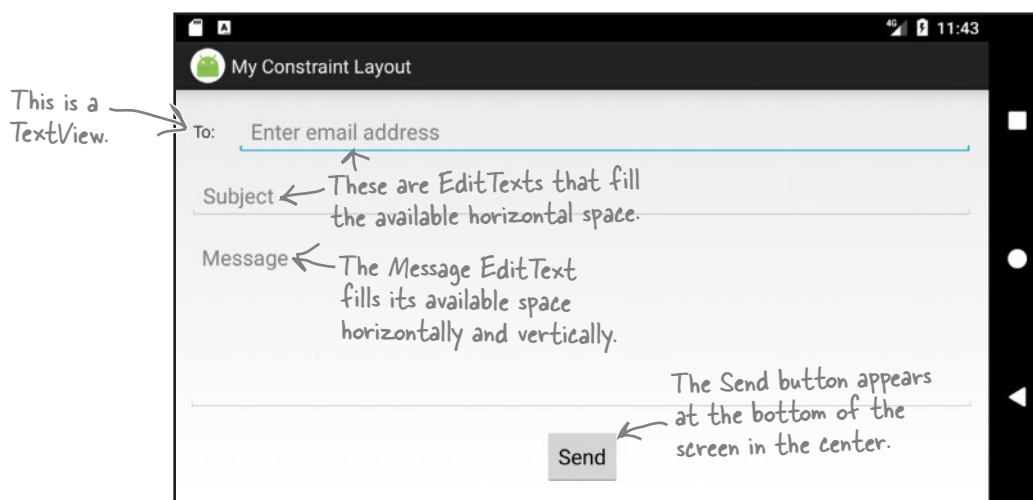
In this chapter we're going to focus on using a new type of layout called a **constraint layout**. This type of layout is more complex than a linear or frame layout, but it's a lot more flexible. It's also much more efficient for complex UIs as it gives you a flatter view hierarchy, which means that Android has less processing to do at runtime.

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 direct in XML, you build constraint layouts *visually*. You drag and drop GUI components onto the design editor's blueprint tool, and give it instructions for how each view should be displayed.

To see this in action, we're going to take you on a tour of using a constraint layout, then build the following UI:

To build constraint layouts using the visual tools, you need Android Studio 2.3 or above. If you're using an older version, check for updates.



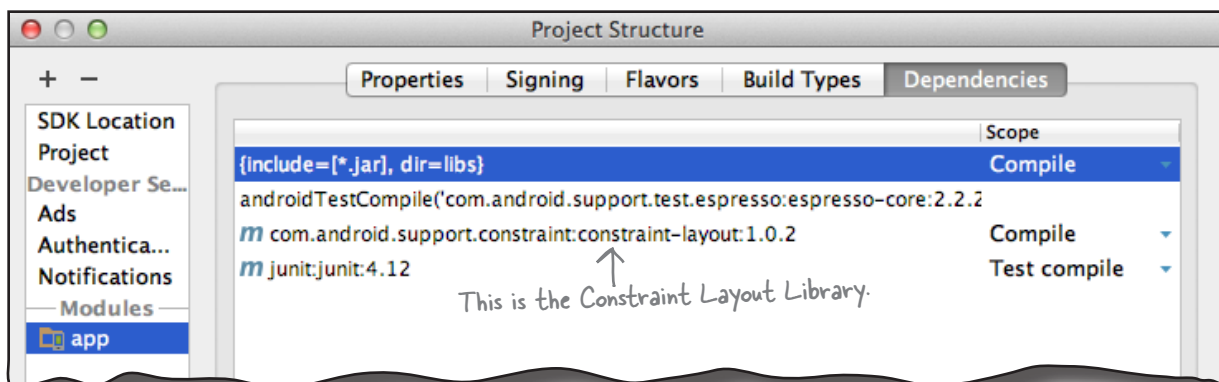
Create a new project

We'll start by creating a new Android Studio project for an application named "My Constraint Layout" with a company domain of "hfad.com", making the package name `com.hfad.myconstraintlayout`. The minimum SDK should be API 19 so that it will work on most devices. You'll need an empty activity called "MainActivity" with a layout called "activity_main" so that your code matches ours. **Make sure you uncheck the Backwards Compatibility (AppCompat) option when you create the activity.**

Make sure your project includes the Constraint Layout Library

Unlike the other layouts you've seen so far, constraint layouts come in their own library, which you need to add to your project as a dependency before you can use it. Adding a library as a dependency means that the library gets included in your app, and downloaded to the user's device.

It's likely that Android Studio added the Constraint Layout Library to your project automatically, but let's check. In Android Studio, choose File→Project Structure. Then click on the app module and choose Dependencies. You'll be presented with the following screen:



If Android Studio has already added the Constraint Layout Library for you, you will see it listed as “com.android.support.constraint:constraint-layout,” as shown above.

If the library hasn't been added for you, you will need to add it yourself. To do this, click on the “+” button at the bottom or right side of the Project Structure screen. Choose the Library Dependency option, and select the Constraint Layout Library option from the list. If you don't see it listed, type the following text into the search box:

```
com.android.support.constraint:constraint-layout:1.0.2
```

You only need to type this in if the Constraint Layout Library hasn't already been added to your project as a dependency.

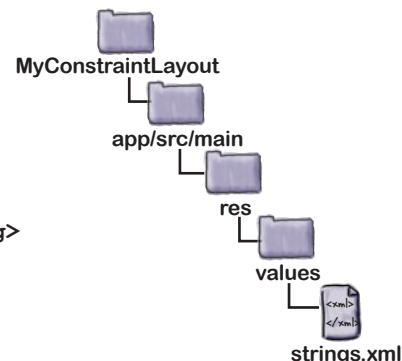
When you click on the OK button, the Constraint Layout Library should be added to the list of dependencies. Click on OK again to save your changes and close the Project Structure window.

Now that we know that our project contains the Constraint Layout Library, let's add the String resources we'll need for our layout.

Add the String resources to strings.xml

Each of the views in our layout will display text values or hints, so we'll add these as String resources. Add the String values below to *strings.xml*:

```
...
<string name="to_label">To:</string>
<string name="email_hint">Enter email address</string>
<string name="subject_hint">Subject</string>
<string name="message_hint">Message</string>
<string name="send_button">Send</string>
...
```



Now that we've added our String resources, we'll update the layout.

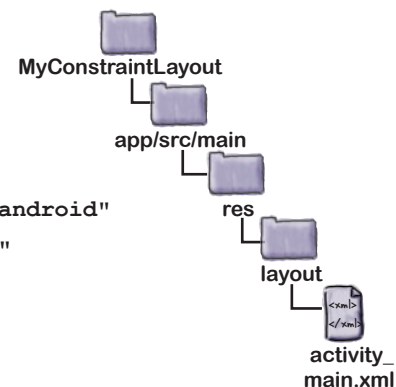
Change activity_main.xml to use a constraint layout

We're going to use a constraint layout. To do this (and to make sure that your layout matches ours), update your code in *activity_main.xml* to match the code below (our changes are in bold):

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

This is how you define a constraint layout.

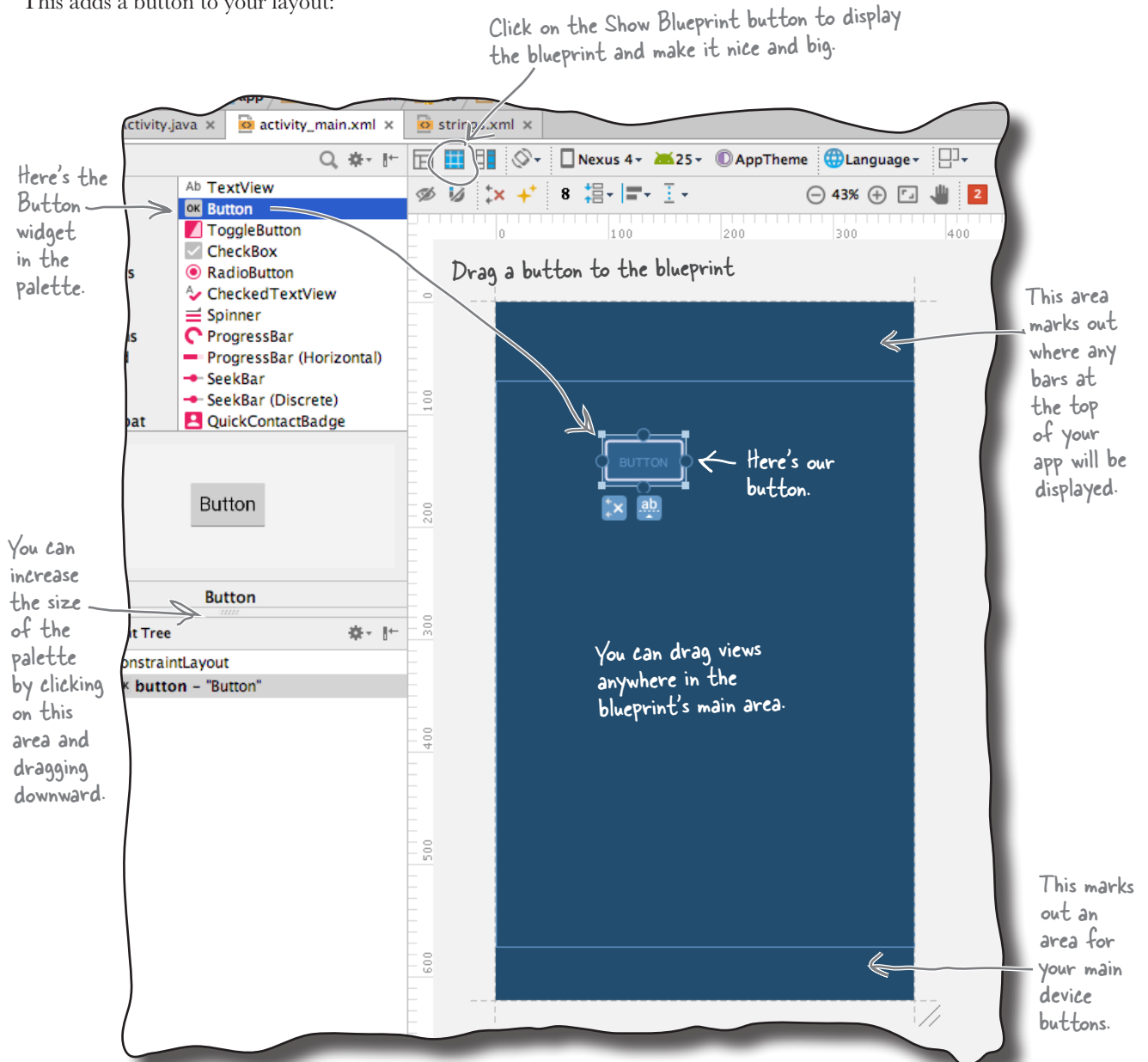
← If Android Studio added any extra views for you, delete them.



This defines a constraint layout to which we can add views. We'll do this using the design editor's blueprint tool.

Use the blueprint tool

To use the blueprint tool, first switch to the design view of your layout code by clicking on the Design tab. Then click on the Show Blueprint button in the design editor's toolbar to display the blueprint. Finally, drag a Button widget from the design editor's palette to the blueprint. This adds a button to your layout:



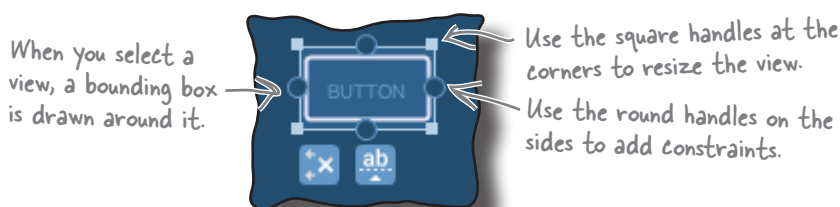
Position views using constraints

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. For example, you can use a constraint to attach a view to the start edge of the layout, or underneath another view.

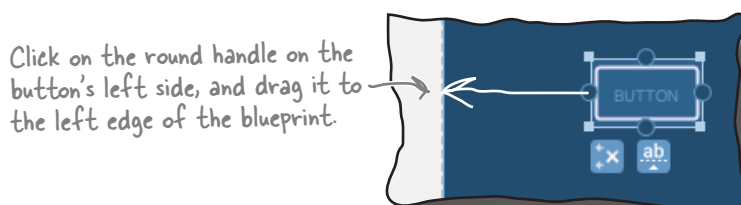
We'll add a horizontal constraint to the button

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

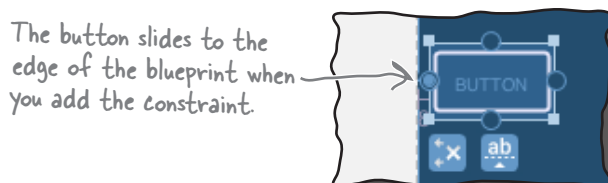
First, make sure the button's selected by clicking it. When you select a view, a bounding box is drawn around it, and handles are added to its corners and sides. The square handles in the corners let you resize the view, and the round handles on the sides let you add constraints:



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 our 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. We'll add a vertical constraint to the button next.

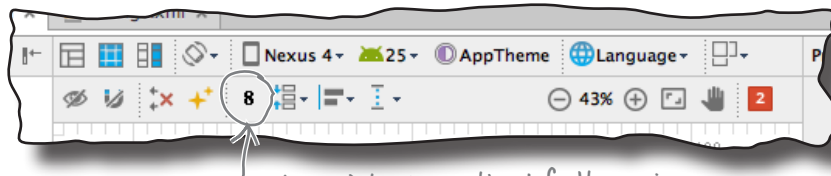
Add a vertical constraint

Let's add a second constraint to the button to attach it 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's main area. This adds the second constraint, and the button slides to the top of the main area.

Each view in a constraint layout must have at least two constraints—a horizontal constraint and a vertical one—in order to specify where it should be positioned. If you omit the horizontal constraint, the view is displayed next to the start edge of the layout at runtime. If you omit the the vertical constraint, the view is displayed at the top of the layout. **This is irrespective of where the view is positioned in the blueprint.**

Changing the view's margins

When you add a constraint to a view, the design editor automatically adds a margin on the same edge as the constraint. You can set the size of the default margin in the design editor's toolbar by changing the number in the Default Margin box:



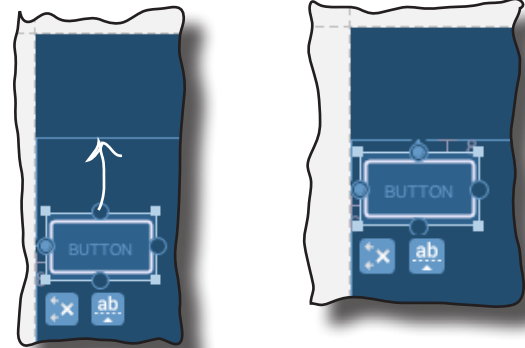
Change the number here (in dps) to change the default margin.

Changing the size of the default margin specifies the size of any *new* margins that get added. The size of any existing margins remain unchanged, but you can change these using the property window.

The property window is displayed in a separate panel at the side of the design editor. When you select a view, it displays a diagram featuring the view's constraints and the size of its margins. To change the size of a margin, you change the number next to the relevant side of the view.

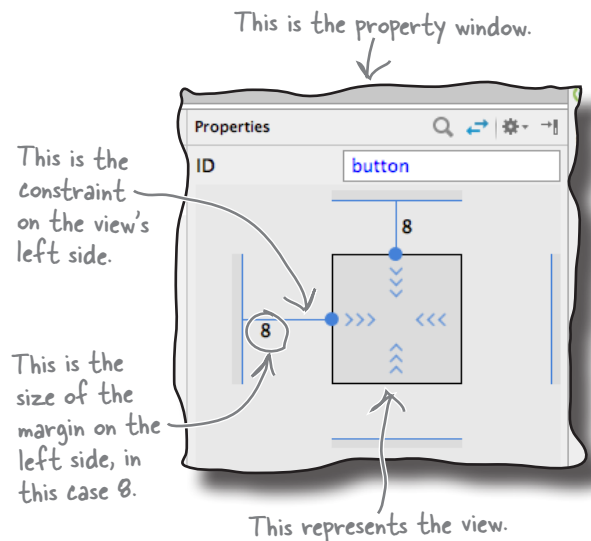
You can also change the size of a view's margins by clicking and dragging the view in the blueprint. This technique has the same effect as changing the size of the margin in the property window, but it's harder to be precise.

Try changing the size of both margins using each method before looking at the next page.



Click on the round 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's main area.



Changes to the blueprint are reflected in the XML

When you add views to the blueprint and specify constraints, these changes are reflected in the layout's underlying XML. To see this, switch to the text view of your code. Your code should look something like this (but don't worry if it's slightly different):

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.constraint.ConstraintLayout
...>
```

The design editor's added a button.

```
<Button
```

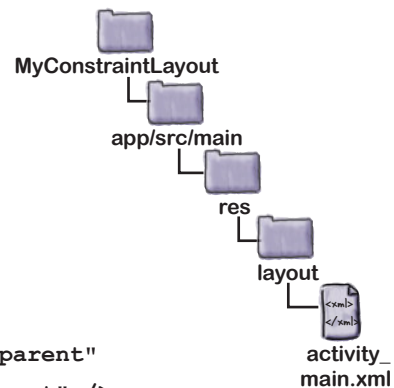
These are all attributes you've seen before.

```
    android:id="@+id/button"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginLeft="8dp"
    android:layout_marginTop="32dp"
    android:text="Button"
```

These lines only apply to constraint layouts.

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

```
</android.support.constraint.ConstraintLayout>
```



As you can see, our XML now includes a button. Most of the code for the button should look familiar to you, as it's material we covered in Chapter 5. The button's width, height, and margins are specified in exactly the same way as before. The only unfamiliar code is the two lines that specify the view's constraints on its left and top edges:

```
<Button>
```

```
...
```

```
    app:layout_constraintLeft_toLeftOf="parent"
```

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

These lines describe the constraints on the button's left and top edges.

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

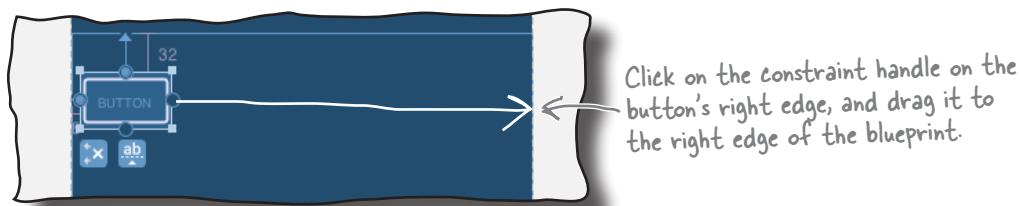
Next, switch your code back to design view, and we'll look at other techniques for positioning your views in a constraint layout.

How to center views

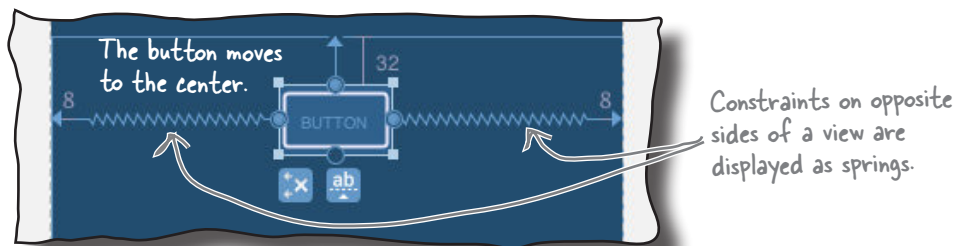
So far you've seen how you can use constraints to attach a view to the edge of its layout. This works well if you want to position a view in the top-left corner, for example, but what if you want to position it in the center?

To position views in the center of its layout, you add constraints to opposite sides of the view. Let's see how this works by centering our button horizontally.

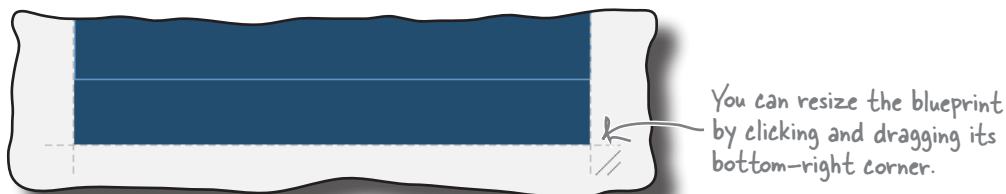
Make sure the button is selected, then click on the constraint handle on its right edge, and drag it to the right edge of the layout:



This adds a constraint to the view's right edge. As the button now has two horizontal constraints, one on each side, the button is pulled to the center, and the two opposing constraints are displayed in the blueprint as springs:



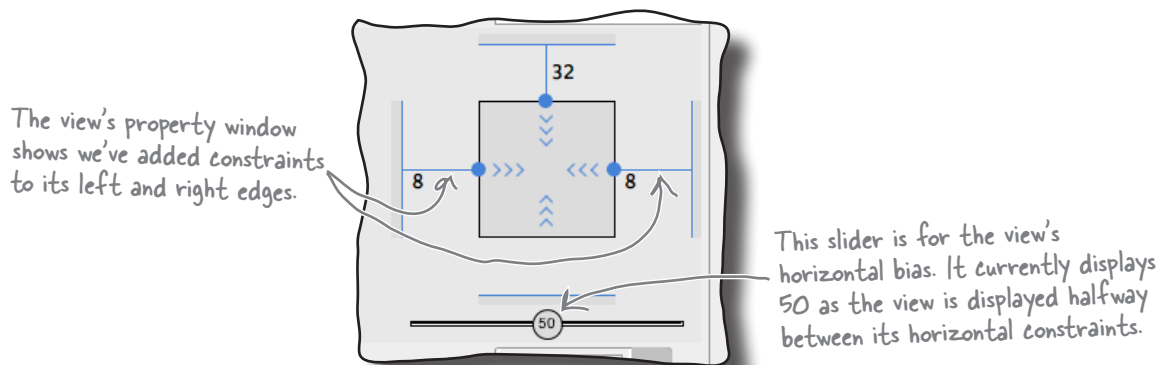
As the button is now attached to both sides of the layout, it's displayed in the center irrespective of screen size or orientation. You can experiment with this by running the app, or changing the size of the blueprint by dragging the blueprint's bottom-right corner:



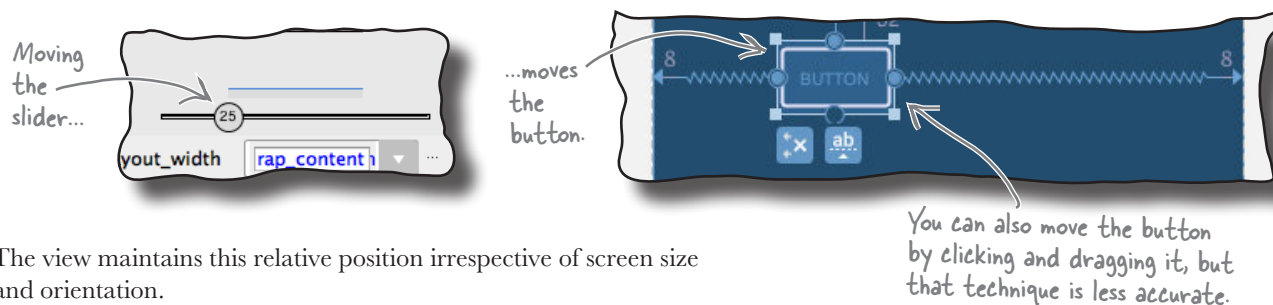
Adjust a view's position by updating its bias

Once you've added constraints to opposite sides of your view, you can control where it should be positioned relative to each side by changing its **bias**. This tells Android what the proportionate length of the constraint should be on either side of the view.

To see this in action, let's change our button's horizontal bias so that it's positioned off-center. First, make sure the button's selected, then look in the view's property window. Underneath the diagram of the view, you should see a slider with a number in it. This represents the view's horizontal bias as a percentage:



To change the value of the bias, simply move the slider. If you move the slider to the left, for example, it moves the button in the blueprint to the left too:



The view maintains this relative position irrespective of screen size and orientation.

When you add a bias to a view in the design editor, this is reflected in the underlying XML. If you change the horizontal bias of your view to 25%, for example, the following code gets added to the view's XML:

```
app:layout_constraintHorizontal_bias="0.25"
```

Now that you know how bias works, let's look at how you specify a view's size.

How to change a view's size

With a constraint layout, you have several different options for specifying a view's size:

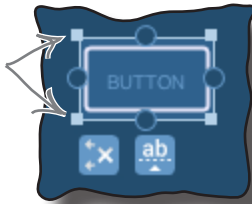
- ★ Make it a fixed size by specifying a specific width and height.
- ★ Use `wrap_content` to make the view just big enough to display its contents.
- ★ Tell it to match the size of its constraints (if you've added constraints to opposite sides of the view).
- ★ Specify a ratio for the width and height so that, for example, the view's width is twice the size of its height.

We'll go through these options one-by-one.

1. Make the view a fixed size

There are a couple of ways of using the design editor to make the view a fixed size. One way is to simply resize the view in the blueprint by clicking and dragging the square resizing handles on its corners. The other way is to type values into the `layout_width` and `layout_height` fields in the properties window:

You can resize a view using the square resizing handles on its corners.



You can also hardcode the width and height in the view's property window.

In general, **making your view a fixed size is a bad idea**, as it means the view can't grow or shrink to fit the size of its contents or the size of the screen.

2. Make it just big enough

To make the view just large enough to display its contents, change the view's `layout_width` and `layout_height` properties to `wrap_content`. You do this in the view's property window as shown here:

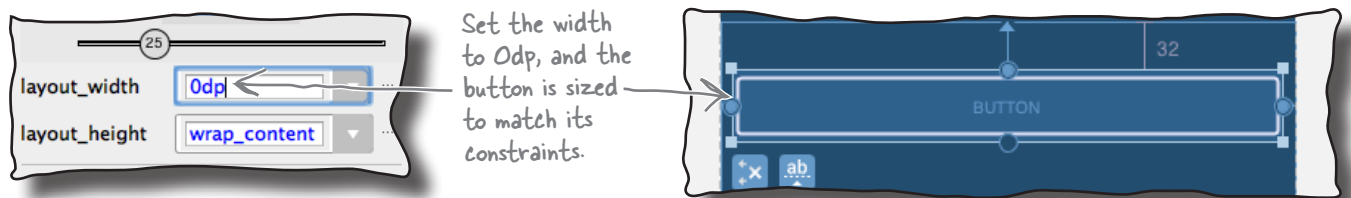


Setting the width and height to "wrap_content" makes it just large enough to display its contents, just as it does in other layouts.

3. Match the view's constraints

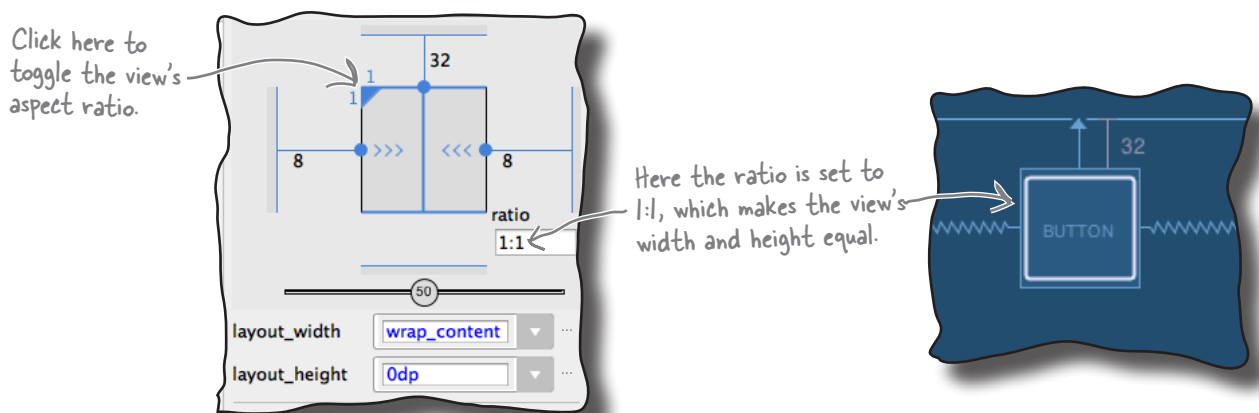
If you've added constraints to opposite sides of your view, you can make the view as wide as its constraints. You do this by setting its width and/or height to 0dp: set its width to 0dp to get the view to match the size of its horizontal constraints, and set its height to 0dp to get it to match the size of its vertical constraints.

In our case, we've added constraints to the left and right sides of our button, so we can get the button to match the size of these constraints. To do this, go to the view's property window, and change the `layout_width` property to 0dp. In the blueprint, the button should expand to fill the available horizontal space (allowing for any margins):



4. Specify the width:height ratio

Finally, you can specify an aspect ratio for the view's width and height. To do this, change the view's `layout_width` or `layout_height` to 0dp as you did above, then click in the top-left corner of the view diagram that's displayed in the property window. This should display a ratio field, which you can then update:



Now that you've seen how to resize a view, try experimenting with the different techniques before having 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.

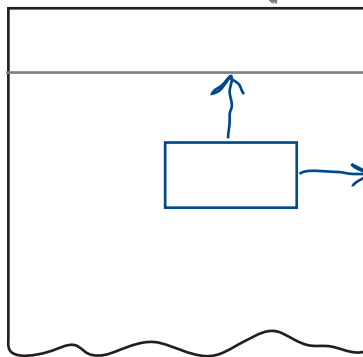
You also need to specify the `layout_width`, `layout_height`, and `bias` (when needed) for each view. We've completed the first one for you.



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

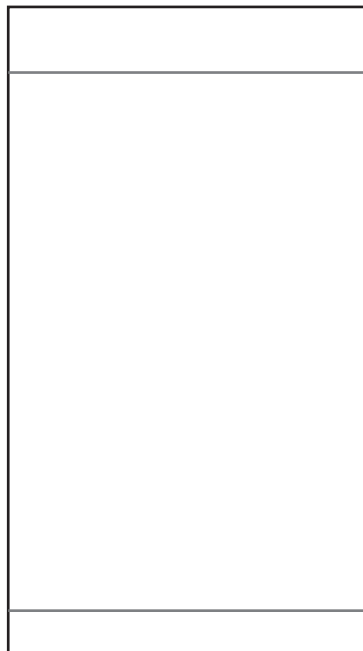
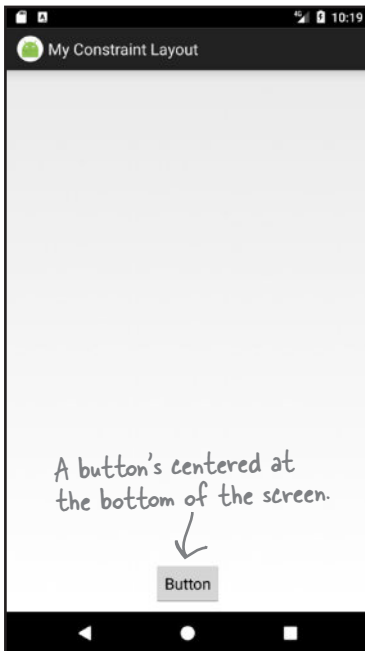
A

This is how we want the screen to look.



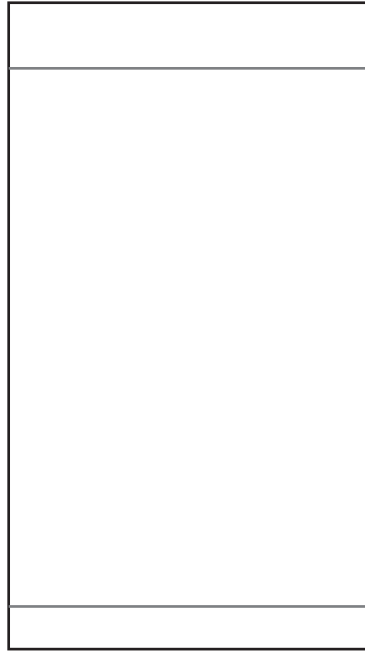
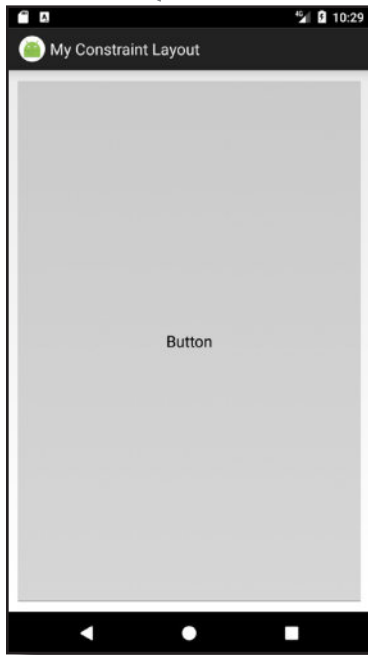
`layout_width: wrap_content`
`layout_height: wrap_content`

B

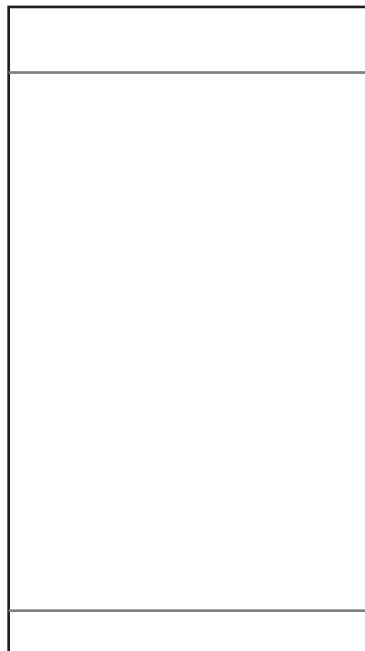


The button fills the available space.

C



D



BE the Constraint Solution

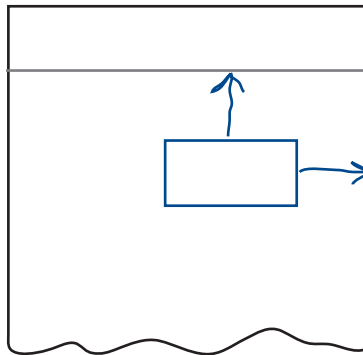
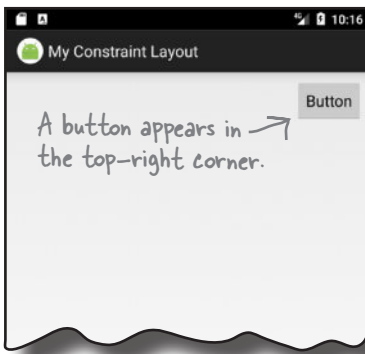
Your job is to play like you're the constraint layout and draw the constraints that are needed to produce each layout.

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.



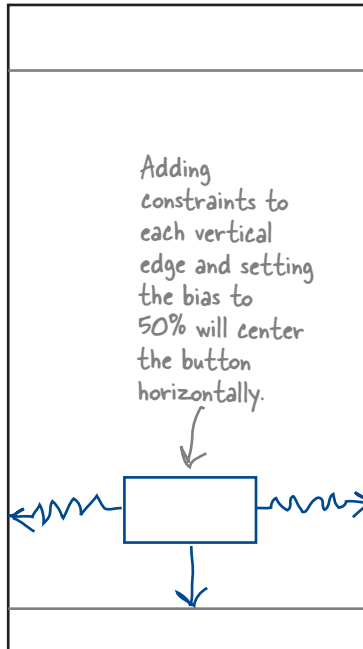
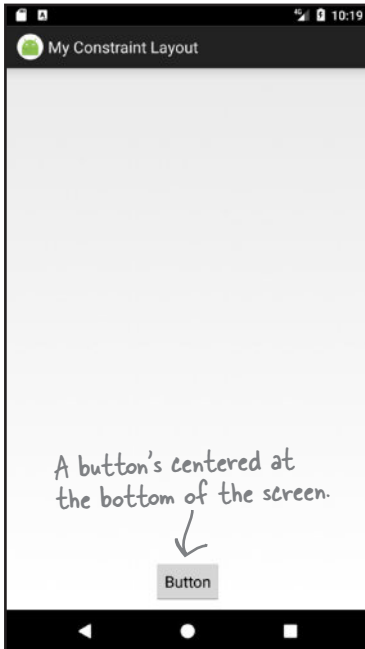
A

This is how we want the screen to look.



`layout_width: wrap_content`
`layout_height: wrap_content`

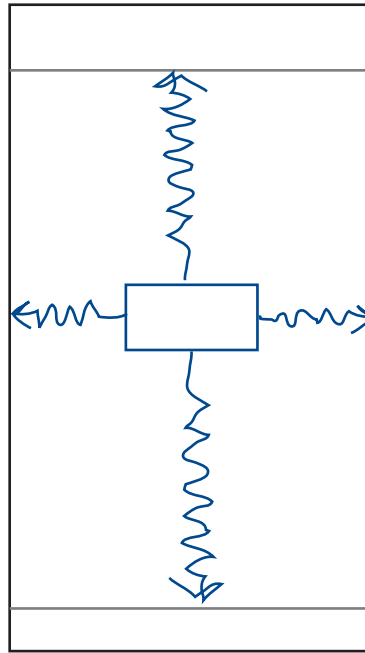
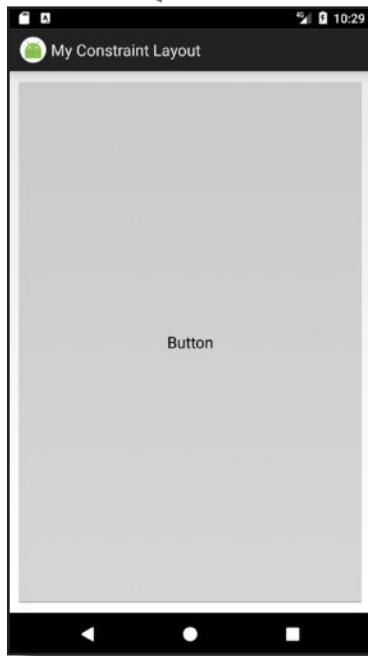
B



`layout_width: wrap_content`
`layout_height: wrap_content`
`bias: 50%`

The button fills the available space.

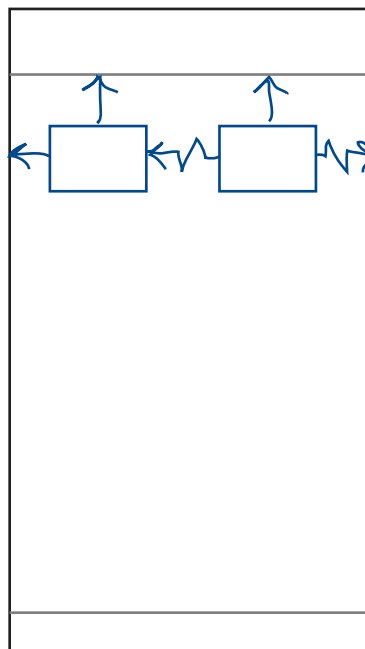
G



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 `Odp`.

`layout_width: Odp`
`layout_height: Odp`

D



Button 1:
`layout_width: wrap_content`
`layout_height: wrap_content`

Button 2:
`layout_width: Odp`
`layout_height: wrap_content`

To make Button 2 fill the remaining horizontal space, we add constraints to each vertical edge and set its width to `Odp`. Its left edge is attached to Button 1's right edge.

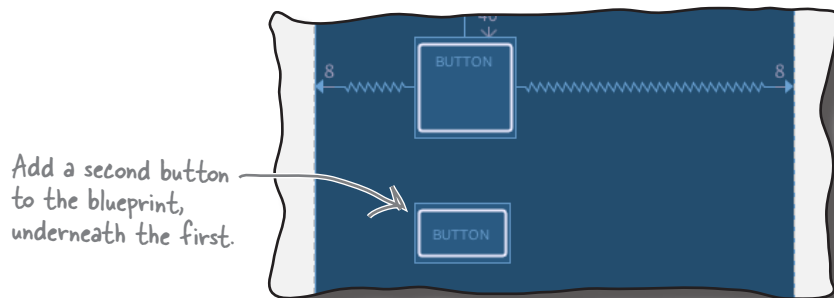
How to align views

So far you've seen how to position and size a single view. Next, let's examine how you align it with another view.

First, click on the Show Constraints button in the design edit toolbar to display all the constraints in the blueprint (not just the ones for the selected view). Then drag a second button from the palette to the blueprint, and place it underneath the first:

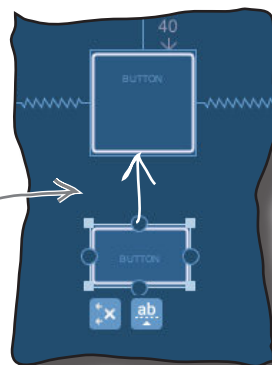


This is the Show Constraints button. Clicking on it shows (or hides) all the constraints in the layout.



To display the second button underneath the first when the app runs, we need to add a constraint to the second button's top edge, and attach it to the first button's bottom edge. To do this, select the second button, and draw a constraint from its top edge to the bottom edge of the other button:

This adds a constraint attaching the top of one button to the bottom edge of the other.

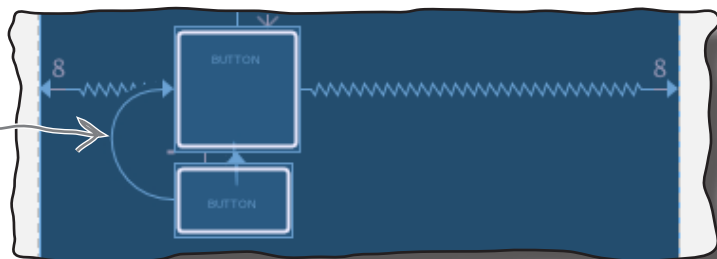


To align the left edges of both buttons, select both buttons by holding down the Shift key when you select each one, then click on the Align Left Edges button in the design editor toolbar:



Clicking on this button gives you different options for aligning views.

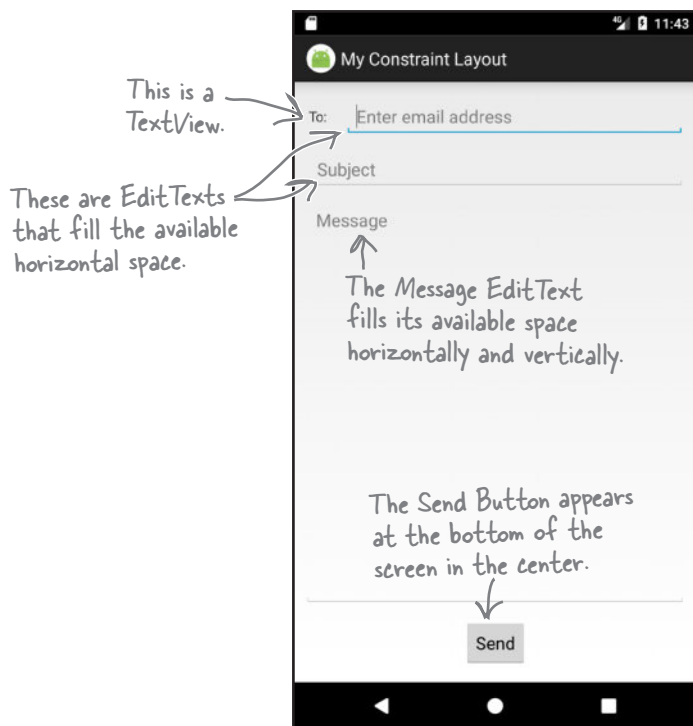
Aligning the view's left edges adds another constraint.



This adds a constraint from the left edge of the second button to the left edge of the first, and this constraint aligns the view's edges.

Let's build a real layout

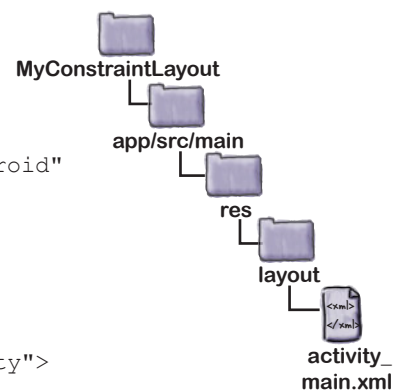
You now know enough about constraint layouts to start building a real one. Here's the layout we're going to create:



We'll build it from scratch in `activity_main.xml`, so before we get started, delete any views that are already in the layout so that the blueprint's empty, and make sure your `activity_main.xml` code looks like this:

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

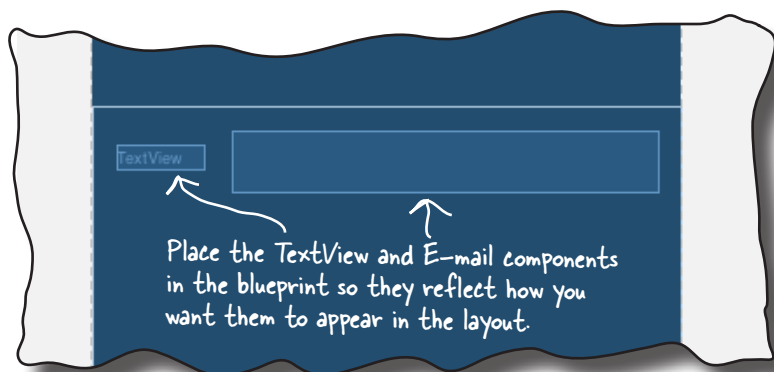
</android.support.constraint.ConstraintLayout>
```



First, add the top line of views

We're going to start by adding the views we want to appear at the top of the layout: a text view and an editable text field.

To do this, switch to the design editor if you haven't already done so, then drag a TextView from the palette to the top-left corner of the blueprint. Next, drag an E-mail component to the blueprint so it's positioned to the right of the text view. This is an editable text field that uses Android's email keyboard for data entry. Manually resize the E-mail component so that it lines up with the text view and fills any remaining horizontal space:



Notice that we haven't added any constraints to either view yet, and we've positioned them where we want them to appear when the layout's displayed on a device. There's a good reason for this: to save us some work, **we're going to get the design editor to figure out the constraints.**

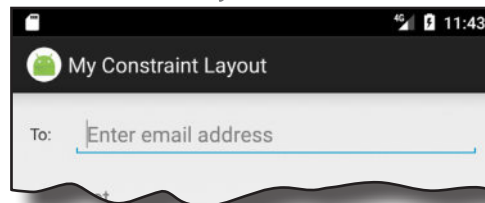
Get the design editor to infer the constraints

As you already know, a constraint layout uses constraints to determine where its views should be positioned. The great news is that the design editor has an Infer Constraints button that's designed to work out what it thinks the constraints should be, and add them. To use this feature, simply click on the Infer Constraints button in the design editor's toolbar:



This is the Infer Constraints button. Click it now.

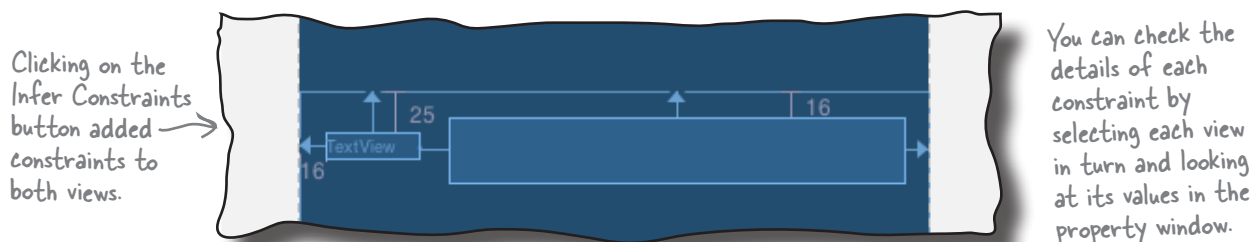
The top line of the layout features a TextView label and an EditText for the email address.



The Infer Constraints feature guesses which constraints to add

When you click on the Infer Constraints button, the design editor tries to figure out what the constraints should be and adds them for you. It's not completely foolproof, as it can't read your mind (as far as we know) to determine how you want the layout to behave on a real device. It simply guesses based on each view's position in the blueprint.

Here are the changes the design editor made for us when we clicked on the Infer Constraints button (yours may look different if you positioned your views differently):



If you don't like what the Infer Constraints feature has done, you can undo the changes it's made by choosing Undo Infer Constraints from the Edit menu, or adjust individual constraints.

We're going to tweak our views before adding more items to the blueprint. First, select the text view in the blueprint, then edit its properties in the properties panel to give it an ID of `to_label` and a text value of `"@string/to_label"`. This does the same thing as adding the following lines of code to the `<TextView>` element in the XML:

```
android:id="@+id/to_label"
android:text="@string/to_label"
```

Android Studio adds these lines of code when you change the view's ID and text value.

Next, select the E-mail component `EditText`, and change its ID to `email_address`, its `layout_height` to `"wrap_content"`, and its hint to `"@string/email_hint"`. This does the same thing as adding these lines to the `<EditText>` element in the XML:

```
android:id="@+id/email_address"
android:layout_height="wrap_content"
android:hint="@string/email_hint"
```

Android Studio adds these lines of code when you change the view's `layout_height` and hint value.

Now that we've added the first line of views to the blueprint, let's add some more.

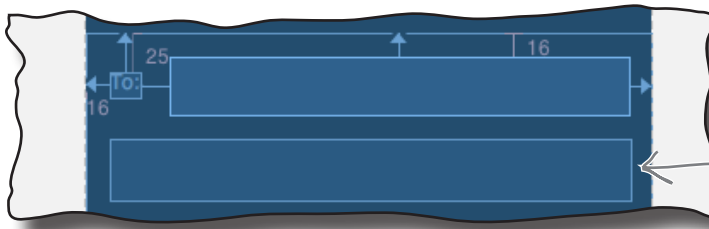
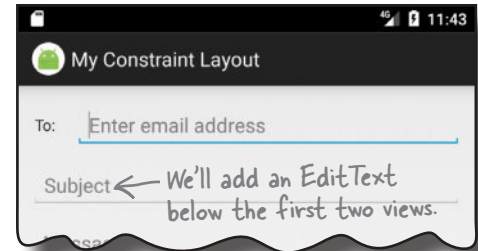
When you update the ID, don't worry if Android Studio displays a message telling you it's going to make changes to the code. We want it to do this because we're changing the view's ID.

Update this property to change the TextView's text value.

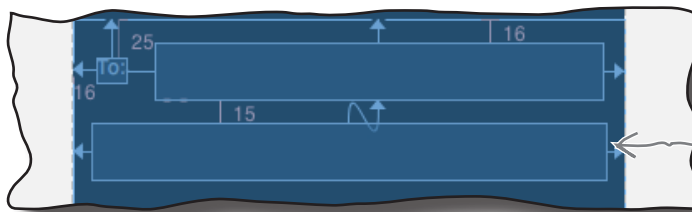


Add the next line to the blueprint...

The next row of the layout contains an editable text field for the message subject. Drag a Plain Text component from the palette to the blueprint, and position it underneath the two items we've already added. This adds an `EditText` to the blueprint. Then change the component's size and position so that it lines up with the other views and fills the horizontal space:



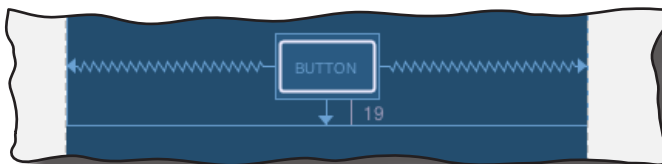
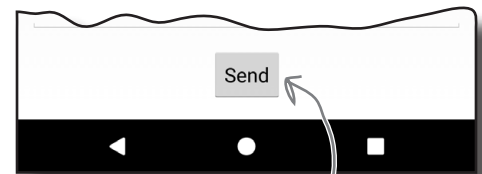
Then click on the Infer Constraints button again. The design editor adds more constraints, this time positioning the new component:



Select the new view in the blueprint, and then change its ID to `subject`, its `layout_height` to `"wrap_content"`, and its hint to `"@string/subject_hint"`, and delete any text in the text property that the design editor may have added.

...and then add the button

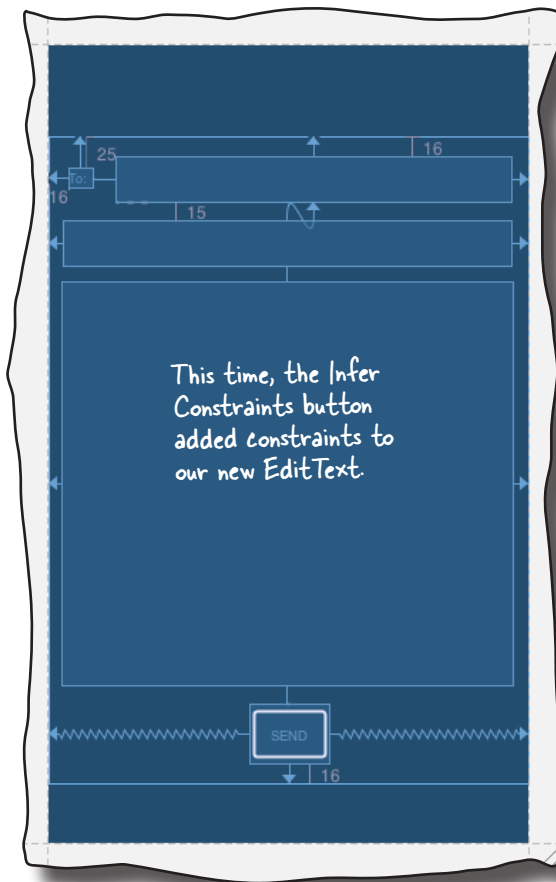
Next, we'll add a button to the bottom of the layout. Drag a Button component to the bottom of the blueprint and center it horizontally. When you click on the Infer Constraints button this time, the design editor adds these constraints to it:



Change the button's ID to `send_button` and its text to `"@string/send_button"`.

Finally, add a view for the message

We have one more view to add to our layout, an editable text field that we want to be able to grow to fill any remaining space. Drag a Plain Text component from the palette to the middle of the blueprint, change its size so that it fills the entire area, and then click on the Infer Constraints button:

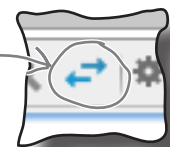


Note that we could have added all these views in one go, and clicked the Infer Constraints button when we reached the end. We've found, however, that building it up step-by-step gives the best results. Why not experiment and try this out for yourself?

Select the new component in the blueprint, then change its ID to `message`, its hint to `"@string/message_hint"`, and its gravity to `top`, and delete any text in the text property that the design editor may have added.

Let's take the app for a test drive and see what the layout looks like.

You may need to click on the "View all properties" button to see the gravity property.

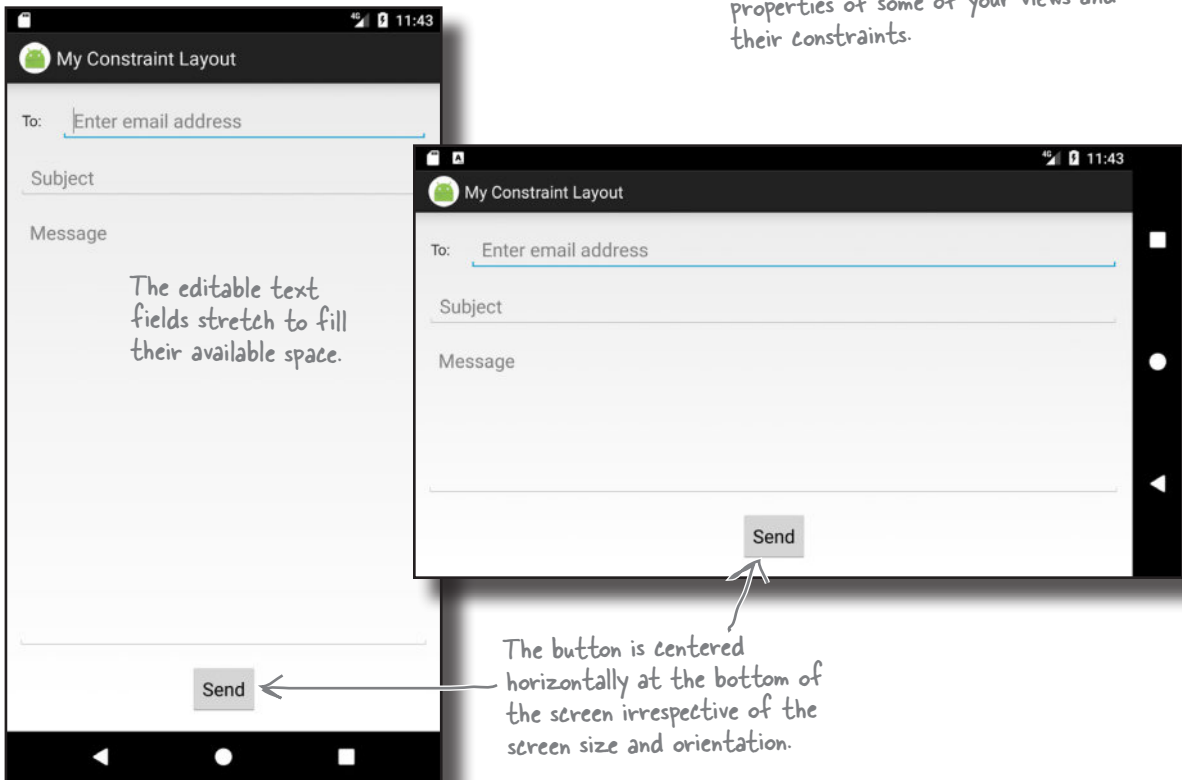




Test drive the app

When we run the app, MainActivity's layout looks almost exactly how we want it to. When we rotate the device, the button stays centered, the email and subject editable text fields expand to fill the horizontal space, and the message view fills the remaining area:

Test your constraint layout on a variety of device sizes and orientations to make sure it behaves the way you want. If it doesn't, you may need to change the properties of some of your views and their constraints.



Remember that your layout may look and behave differently than ours depending on what constraints the design editor added when you clicked on the Infer Constraints button. The feature isn't perfect, but it usually takes you most of the way there, and you can undo or update any changes it makes.



Your Android Toolbox

You've got Chapter 6 under your belt and now you've added constraint layouts to your toolbox.

You can download the full code for the chapter from <https://tinyurl.com/HeadFirstAndroid>.



BULLET POINTS

- Constraint layouts are designed to work with Android Studio's design editor. They have their own library and can be used in apps where the minimum SDK is API level 9 or above.
- 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 specify a width:height aspect ratio for the view's size.
- Clicking on the Infer Constraints button adds constraints to views based on their position in the blueprint.

there are no Dumb Questions

Q: Is a constraint layout my only option if I want to create complex layouts?

A: There are other types of layout as well, such as relative and grid layouts, but the constraint layout does everything that these do. Also it's designed to work with Android Studio's design editor, which makes building constraint layouts much easier.

If you're interested in finding out more about relative and grid layouts, they're covered in Appendix I at the back of the book.

Q: Why do constraint layouts have a separate library?

A: Constraint layouts are a fairly recent addition to Android compared to other types of layout. They're in a separate library so that they can be added to apps that support older versions of Android. You'll find out more about backward compatibility in later chapters.

Q: Can I still edit constraint layouts using XML?

A: Yes, but as they're designed to be edited visually, we've concentrated on building them using the design editor.

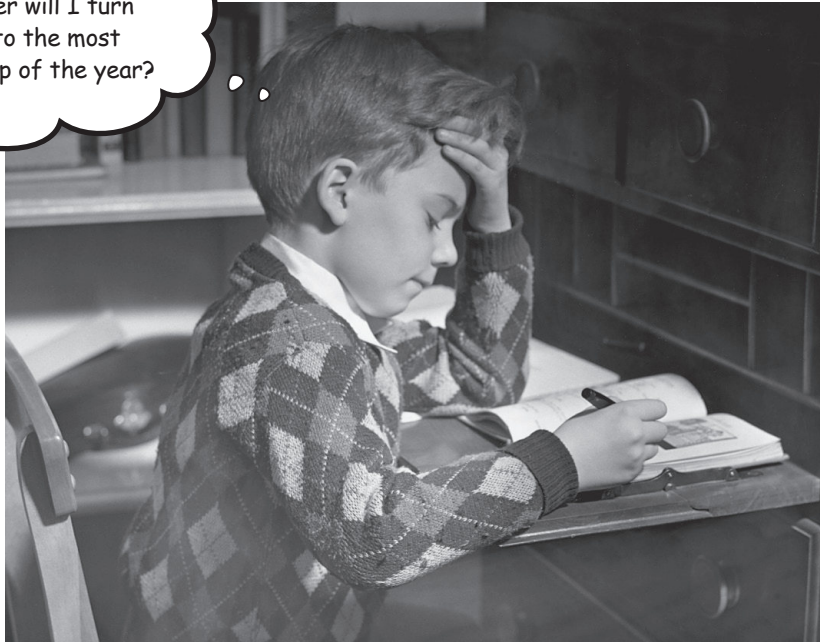
Q: I tried using the Infer Constraints feature but it didn't give me the results I wanted. Why not?

A: The Infer Constraints feature can only make guesses based on where you position views in the blueprint, so it may not always give you the results you want. You can, however, edit the changes the Infer Constraints feature makes to your app.

7 list views and adapters

✦ **Getting Organized** ✦

Sheesh! So many ideas... However will I turn all of these into the most downloaded app of the year?



Want to know how best to structure your Android app?

You've learned about some of the basic building blocks that are used to create apps, and now **it's time to get organized**. In this chapter, we'll show you how you can take a bunch of ideas and **structure them into an awesome app**. You'll learn how **lists of data** can form the core part of your app design, and how **linking them together** can create a **powerful and easy-to-use app**. Along the way, you'll get your first glimpse of using **event listeners** and **adapters** to make your app more dynamic.

Every app starts with ideas

When you first come up with an idea for an app, you'll have lots of thoughts about what the app should contain.

As an example, the guys at Starbuzz Coffee want a new app to entice more customers to their stores. These are some of the ideas they came up with for what the app should include:



These are all ideas that users of the app will find useful. But how do you take all of these ideas and organize them into an intuitive, well-structured app?

Organize your ideas: top-level, category, and detail/edit activities

A useful way to bring order to these ideas is to organize them into three different types of activity: **top-level** activities, **category** activities, and **detail/edit** activities.

Top-level activities

A top-level activity contains the things that are most important to the user, and gives them an easy way of navigating to those things. In most apps, the first activity the user sees will be a top-level activity.

Display a start screen with a list of options.

Category activities

Category activities show the data that belongs to a particular category, often in a list. These type of activities are often used to help the user navigate to detail/edit activities. An example of a category activity is a list of all the drinks available at Starbuzz.

Display a menu showing all the food you can buy.

Show a list of all our stores.

Display a list of the drinks we sell.

Detail/edit activities

Detail/edit activities display details for a particular record, let the user edit the record, or allow the user to enter new records. An example of a detail/edit activity would be an activity that shows the user the details of a particular drink.

Show details of each drink.

Show details of an item of food.

Display the address and opening times of each store.

Once you've organized your activities, you can use them to construct a hierarchy showing how the user will navigate between activities.



Think of an app you'd like to create. What activities should it include? Organize these activities into top-level activities, category activities, and detail/edit activities.

Navigating through the activities

When you organize the ideas you have into top-level, category, and detail/edit activities, you can use this organization scheme to figure out how to navigate through your app. In general, you want your users to navigate from top-level activities to detail/edit activities via category activities.

Top-level activities go at the top

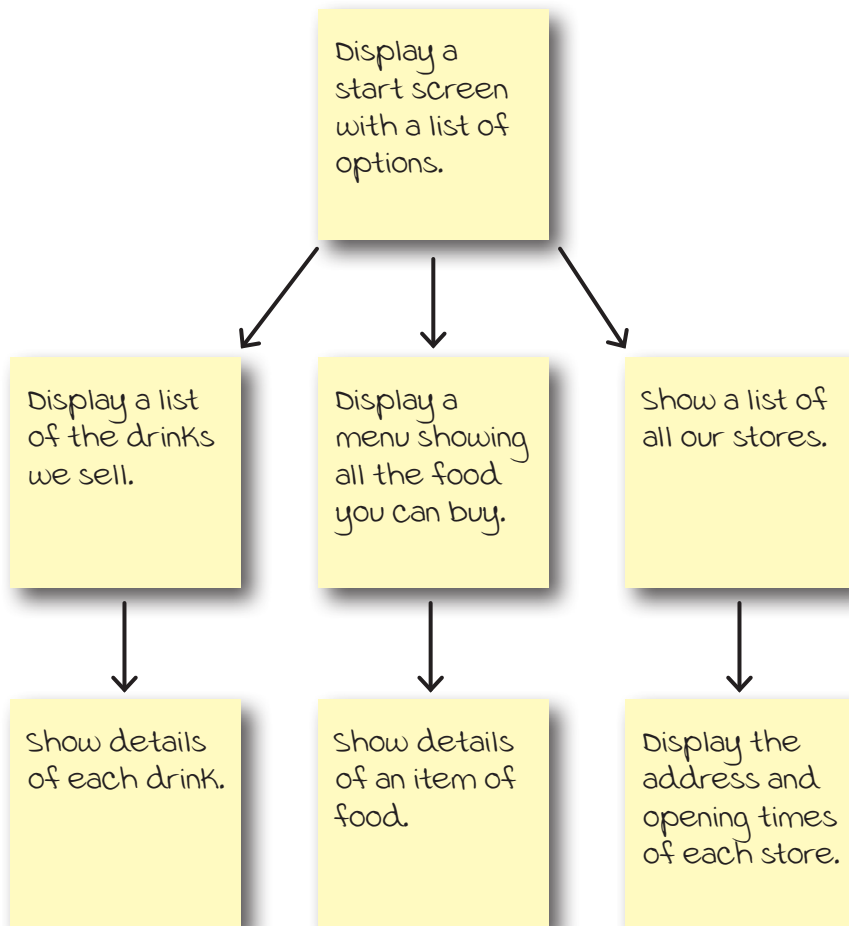
These are the activities your user will encounter first, so they go at the top.

Category activities go between top-level and detail/edit activities

Your users will navigate from the top-level activity to the category activities. In complex apps, you might have several layers of categories and subcategories.

Detail/edit activities

These form the bottom layer of the activity hierarchy. Users will navigate to these from the category activities.

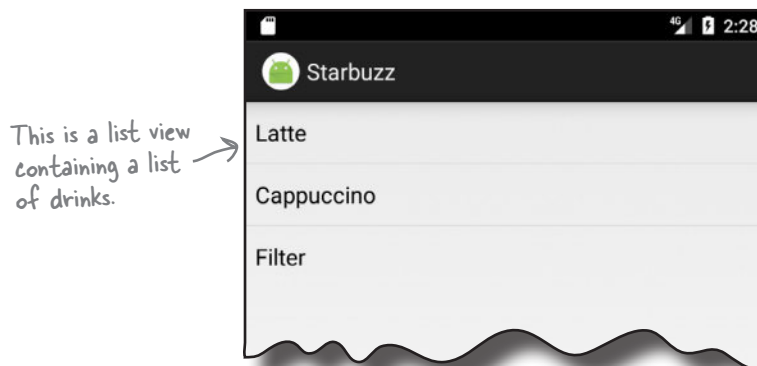


As an example, suppose a user wanted to look at details of one of the drinks that Starbuzz serves. To do this, she would launch the app, and be presented with the top-level activity start screen showing her a list of options. The user would click on the option to display a list of drinks. To see details of a particular drink, she would then click on her drink of choice in the list.

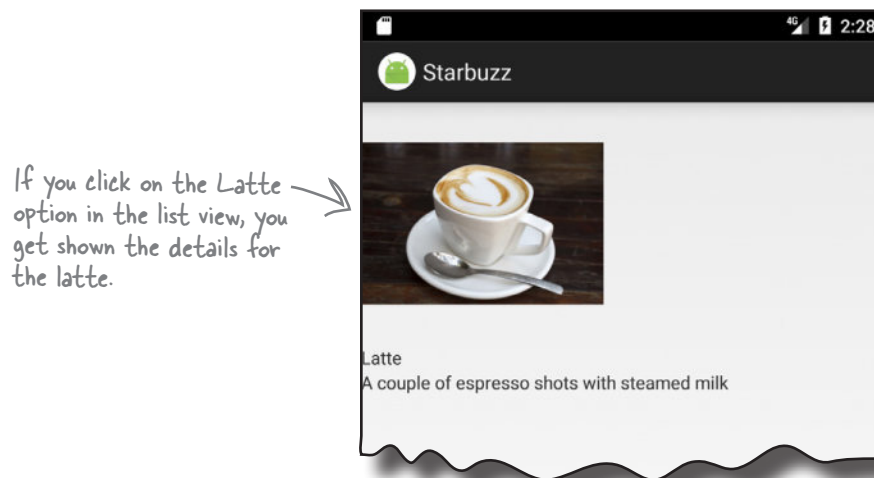
Use list views to navigate to data

When you structure your app in this way, you need a way of navigating between your activities. A common approach is to use **list views**. A list view allows you to display a list of data that you can then use to navigate through the app.

As an example, on the previous page, we said we'd have a category activity that displays a list of the drinks sold by Starbuzz. Here's what the activity might look like:



The activity uses a list view to display all the drinks that are sold by Starbuzz. To navigate to a particular drink, the user clicks on one of the drinks, and the details of that drink are displayed.



We're going to spend the rest of this chapter showing you how to use list views to implement this approach, using the Starbuzz app as an example.

part of We're going to build the Starbuzz app

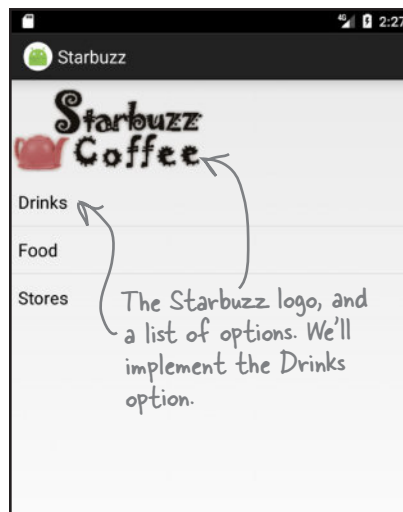
Rather than build all the category and detail/edit activities required for the entire Starbuzz app, **we'll focus on just the drinks.**

We're going to build a top-level activity that the user will see when they launch the app, a category activity that will display a list of drinks, and a detail/edit activity that will display details of a single drink.

The top-level activity

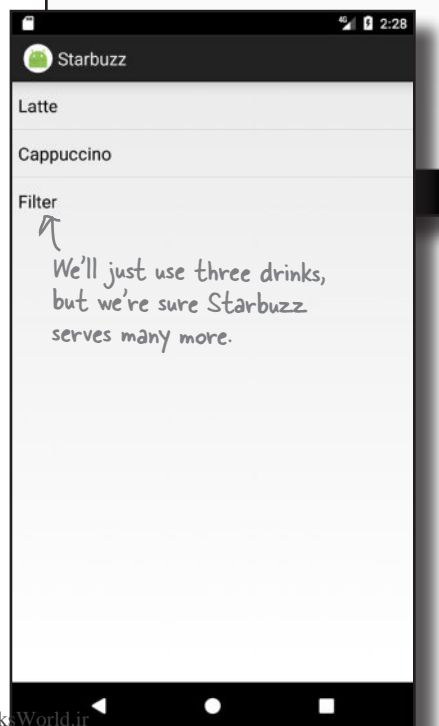
When the user launches the app, she will be presented with the top-level activity, the main entry point of the app. This activity includes an image of the Starbuzz logo, and a navigational list containing entries for Drinks, Food, and Stores.

When the user clicks on an item in the list, the app uses her selection to navigate to a separate activity. As an example, if the user clicks on Drinks, the app starts a category activity relating to drinks.



The drink category activity

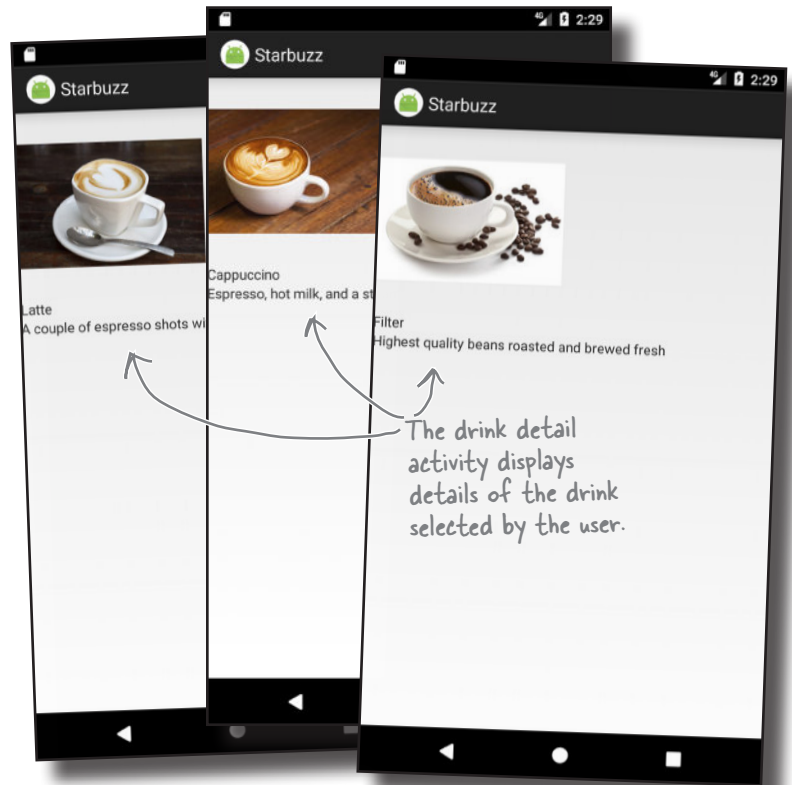
This activity is launched when the user chooses Drinks from the navigational list in the top-level activity. The activity displays a list of all the drinks that are available at Starbuzz. The user can click on one of these drinks to see more details of it.



The drink detail activity

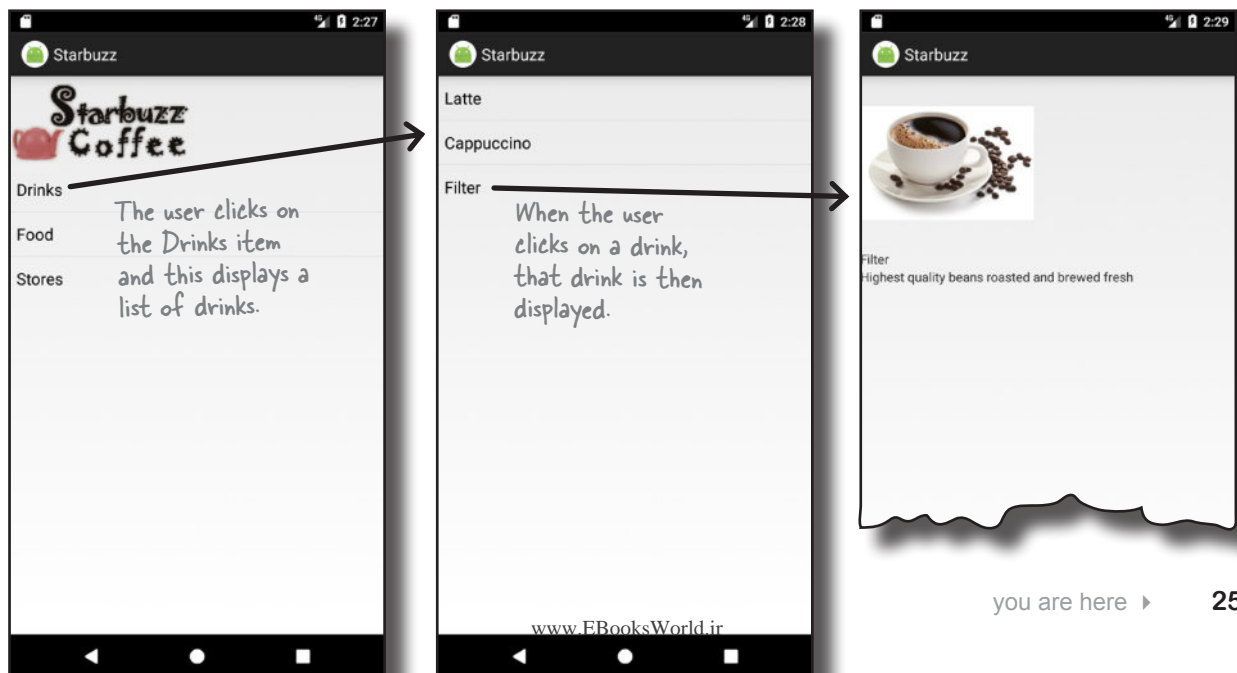
The drink detail activity is launched when the user clicks on one of the drinks listed by the drink category activity.

This activity displays details of the drink the user has selected: its name, an image of it, and a description.



How the user navigates through the app

The user navigates from the top-level activity to the drink category activity by clicking on the Drinks item in the top-level activity. She then navigates to the drink detail activity by clicking on a drink.



The Starbuzz app structure

The app contains three activities. `TopLevelActivity` is the app's top-level activity and allows the user to navigate through the app. `DrinkCategoryActivity` is a category activity; it contains a list of all the drinks. The third activity, `DrinkActivity`, displays details of a given drink.

For now, we're going to hold the drink data in a Java class. In a later chapter, we're going to move it into a database, but for now we want to focus on building the rest of the app without teaching you about databases too.

Here's how the app will work:

1 When the app gets launched, it starts activity `TopLevelActivity`.

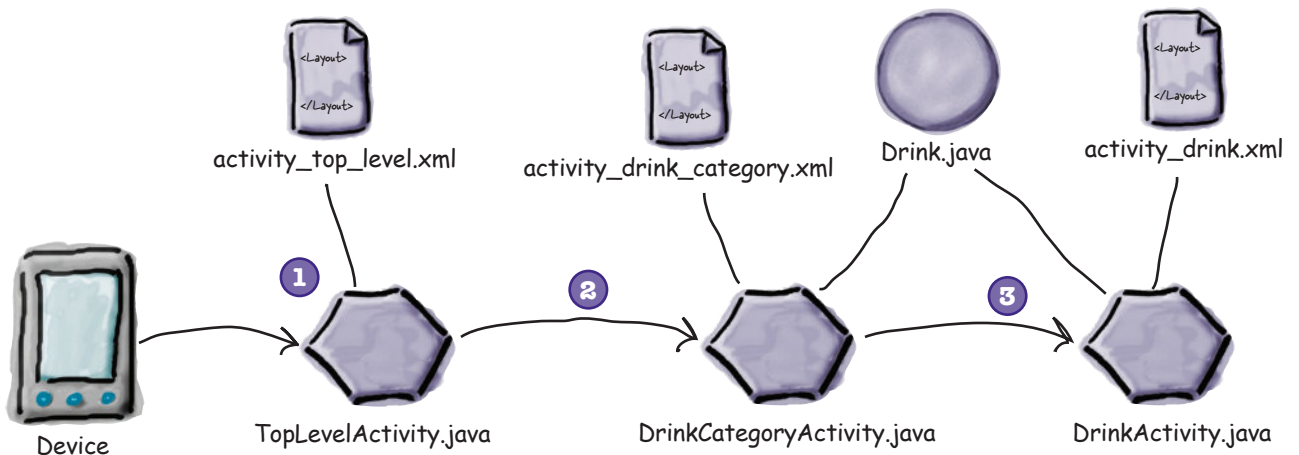
This activity uses layout `activity_top_level.xml`. The activity displays a list of options for Drinks, Food, and Stores.

2 The user clicks on Drinks in `TopLevelActivity`, which launches activity `DrinkCategoryActivity`.

This activity uses layout `activity_drink_category.xml` and displays a list of drinks. It gets information about the drinks from the `Drink.java` class file.

3 The user clicks on a drink in `DrinkCategoryActivity`, which launches activity `DrinkActivity`.

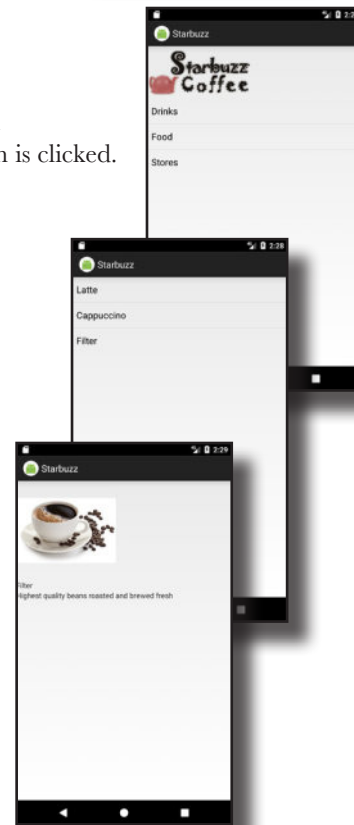
The activity uses layout `activity_drink.xml`. This activity also gets details about the drinks from the `Drink.java` class file.



Here's what we're going to do

There are a number of steps we'll go through to build the app:

- 1 **Add the Drink class and image resources.**
This class contains details of the available drinks, and we'll use images of the drinks and Starbuzz logo in the app.
- 2 **Create TopLevelActivity and its layout.**
This activity is the entry point for the app. It needs to display the Starbuzz logo and include a navigational list of options. TopLevelActivity needs to launch DrinkCategoryActivity when the Drinks option is clicked.
- 3 **Create DrinkCategoryActivity and its layout.**
This activity contains a list of all the drinks that are available. When a drink is clicked, it needs to launch DrinkActivity.
- 4 **Create DrinkActivity and its layout.**
This activity displays details of the drink the user clicked on in DrinkCategoryActivity.



Add resources
TopLevelActivity
DrinkCategoryActivity
DrinkActivity

Create the project

You create the project for the app in exactly the same way you did in the previous chapters.

Create a new Android project for an application named “Starbuzz” with a company domain of “hfad.com”, making the package name `com.hfad.starbuzz`. The minimum SDK should be API 19. You'll need an empty activity called “TopLevelActivity” and a layout called “activity_top_level”. **Make sure you uncheck the Backwards Compatibility (AppCompat) checkbox.**

The Drink class

We'll start by adding the `Drink` class to the app. `Drink.java` is a pure Java class file that activities will get their drink data from. The class defines an array of three drinks, where each drink is composed of a name, description, and image resource ID. Switch to the Project view of Android Studio's explorer, select the `com.hfad.starbuzz` package in the `app/src/main/java` folder, then go to File→New...→Java Class. When prompted, name the class "Drink", and make sure the package name is `com.hfad.starbuzz`. Then replace the code in `Drink.java` with the following, and save your changes.

```
package com.hfad.starbuzz;

public class Drink {
    private String name;
    private String description;
    private int imageResourceId;

    //drinks is an array of Drinks
    public static final Drink[] drinks = {
        new Drink("Latte", "A couple of espresso shots with steamed milk",
            R.drawable.latte),
        new Drink("Cappuccino", "Espresso, hot milk, and a steamed milk foam",
            R.drawable.cappuccino),
        new Drink("Filter", "Highest quality beans roasted and brewed fresh",
            R.drawable.filter)
    };

    //Each Drink has a name, description, and an image resource
    private Drink(String name, String description, int imageResourceId) {
        this.name = name;
        this.description = description;
        this.imageResourceId = imageResourceId;
    }

    public String getDescription() {
        return description;
    }

    public String getName() {
        return name;
    }

    public int getImageResourceId() {
        return imageResourceId;
    }

    public String toString() {
        return this.name;
    }
}
```

Each Drink has a name, description, and image resource ID. The image resource ID refers to drink images we'll add to the project on the next page.

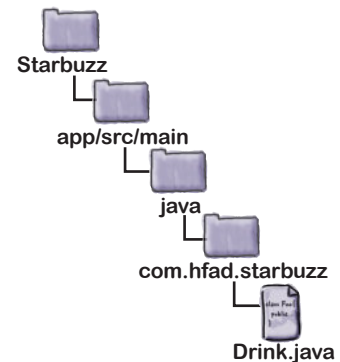
drinks is an array of three Drinks.

These are images of the drinks. We'll add these next.

The Drink constructor

These are getters for the private variables.

The String representation of a Drink is its name.



The image files

The `Drink` code includes three image resources for its drinks with IDs of `R.drawable.latte`, `R.drawable.cappuccino`, and `R.drawable.filter`. These are so we can show the user images of the drinks. `R.drawable.latte` refers to an image file called *latte*, `R.drawable.cappuccino` refers to an image file called *cappuccino*, and `R.drawable.filter` refers to a file called *filter*.

We need to add these image files to the project, along with an image of the Starbucks logo so that we can use it in our top-level activity. First, create the `app/src/main/res/drawable` folder in your Starbucks project (if it doesn't already exist). To do this, make sure you've switched 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.

Once your project includes the *drawable* folder, download the files *starbuzz-logo.png*, *cappuccino.png*, *filter.png*, and *latte.png* from <https://git.io/v9oet>. Finally, add each file to the `app/src/main/res/drawable` folder.

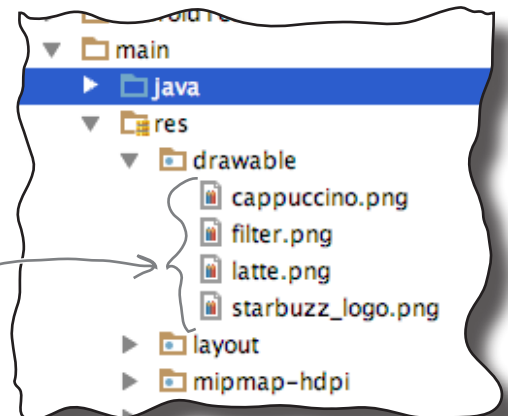
When you add images to your apps, you need to decide whether to display different images for different density screens. In our case, we're going to use the same resolution image irrespective of screen density, so we've put a single copy of the images in one folder. If you decide to cater to different screen densities in your own apps, put images for the different screen densities in the appropriate *drawable** folders as described in Chapter 5.

Here are the four image files. You need to create the *drawable* folder, then add the image files to it.

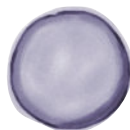


Add resources
TopLevelActivity
DrinkCategoryActivity
DrinkActivity

← Android Studio may have already added this folder for you. If so, you don't need to recreate it.



When you save images to your project, Android assigns each of them an ID in the form `R.drawable.image_name` (where *image_name* is the name of the image). As an example, the file *latte.png* is given an ID of `R.drawable.latte`, which matches the value of the *latte*'s image resource ID in the `Drink` class.



Drink

name: "Latte"
 description: "A couple of espresso shots with steamed milk"
 imageResourceId: `R.drawable.latte`

The image *latte.png* is given an ID of `R.drawable.latte`.



`R.drawable.latte`

Now that we've added the `Drink` class and image resources to the project, let's work on the activities. We'll start with the top-level activity.

The top-level layout contains an image and a list

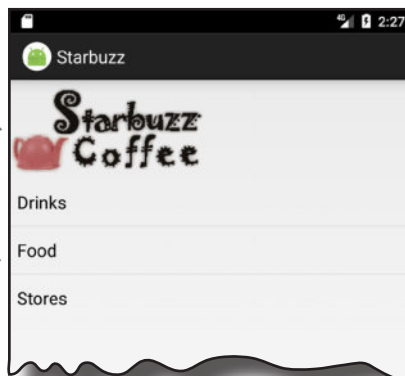


Add resources
TopLevelActivity
DrinkCategoryActivity
DrinkActivity

When we created our project, we called our default activity *TopLevelActivity.java*, and its layout *activity_top_level.xml*. We need to change the layout so it displays an image and a list.

This is the Starbuzz logo. We added this image to the project on the previous page.

We'll add these items as a static list of options; then we'll make each list item clickable.



You saw how to display images in Chapter 5 using an image view. In this case, we need an image view that displays the Starbuzz logo, so we'll create one that uses *starbuzz_logo.png* as its source.

Here's the code to define the image view in the layout:

```

<ImageView
    android:layout_width="200dp"
    android:layout_height="100dp"
    android:src="@drawable/starbuzz_logo"
    android:contentDescription="@string/starbuzz_logo" />
    
```

We're going to add this to activity_top_level.xml. We'll show you the full code soon.

These are the dimensions we want the image to have.

The source of the image is the starbuzz_logo.png file we added to the app.

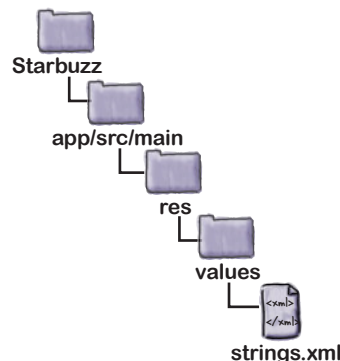
Adding a content description makes your app more accessible.

When you use an image view in your app, you use the `android:contentDescription` attribute to add a description of the image; this makes your app more accessible. In our case, we're using a String value of `"@string/starbuzz_logo"`, so add this to *strings.xml*:

```

<resources>
    ...
    <string name="starbuzz_logo">Starbuzz logo</string>
</resources>
    
```

That's everything we need to add the image to the layout, so let's move on to the list.



Use a list view to display the list of options

As we said earlier, a list view allows you to display a vertical list of data that people can use to navigate through the app. We're going to add a list view to the layout that displays the list of options, and later on we'll use it to navigate to a different activity.

How to define a list view in XML

You add a list view to your layout using the `<ListView>` element. You then add an array of entries to the list view by using the `android:entries` attribute and setting it to an array of Strings. The array of Strings then gets displayed in the list view as a list of text views.

Here's how you add a list view to your layout that gets its values from an array of Strings named options:

We're going to add this to `activity_top_level.xml` on the next page.

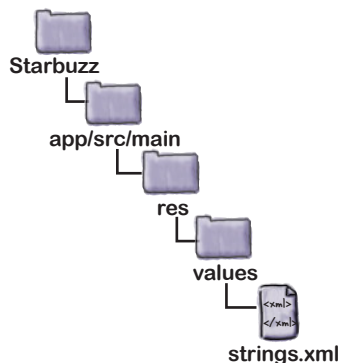
```
<ListView
  android:id="@+id/list_options"
  android:layout_width="match_parent"
  android:layout_height="wrap_content"
  android:entries="@array/options" />
```

← This defines the list view.

← The values in the list view are defined by the options array.

You define the array in exactly the same way that you did earlier in the book, by adding it to `strings.xml` like this:

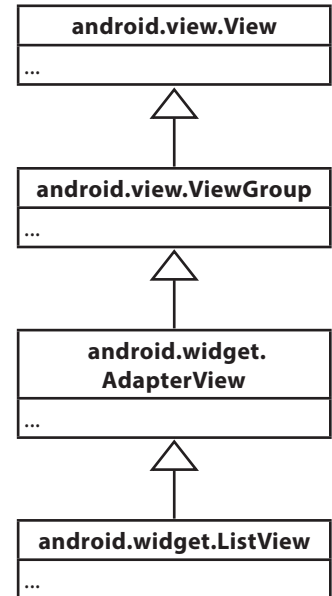
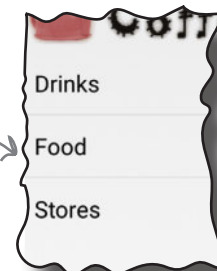
```
<resources>
  ...
  <string-array name="options">
    <item>Drinks</item>
    <item>Food</item>
    <item>Stores</item>
  </string-array>
</resources>
```



This populates the list view with three values: Drinks, Food, and Stores.



The entries attribute populates the list view with values from the options array. Each item in the list view is a text view.



The full top-level layout code

Here's the full layout code for our *activity_top_level.xml*; make sure you update your code to match ours:

```
<?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="com.hfad.starbuzz.TopLevelActivity" >

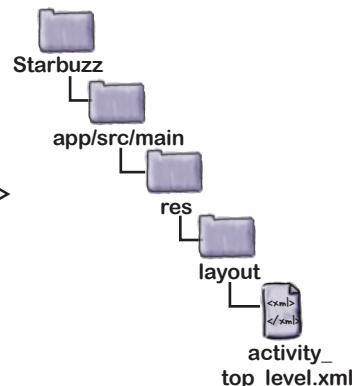
    <ImageView
        android:layout_width="200dp"
        android:layout_height="100dp"
        android:src="@drawable/starbuzz_logo"
        android:contentDescription="@string/starbuzz_logo" />

    <ListView
        android:id="@+id/list_options"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:entries="@array/options" />
</LinearLayout>
```

We're using a linear layout with a vertical orientation. This will display our list view directly underneath the Starbuzz logo.



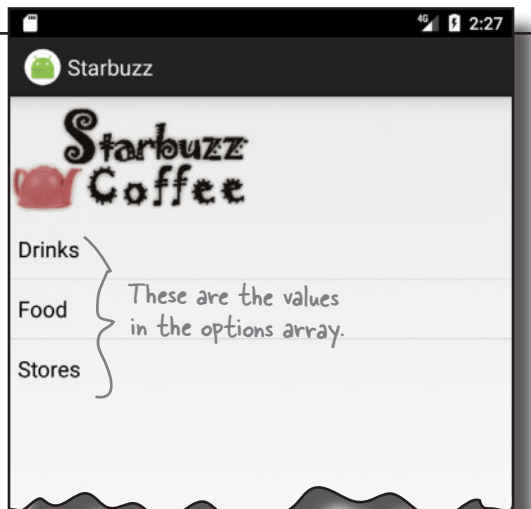
Add resources
TopLevelActivity
DrinkCategoryActivity
DrinkActivity



Test drive

Make sure you've applied all the changes to *activity_top_level.xml*, and also updated *strings.xml*. When you run the app, you should see the Starbuzz logo displayed on the device screen with the list view underneath it. The list view displays the three values from the options array.

If you click on any of the options in the list, nothing happens, as we haven't told the list view to respond to clicks yet. The next thing we'll do is see how you get list views to respond to clicks and launch a second activity.



Get list views to respond to clicks with a listener

You make the items in a list view respond to clicks by implementing an **event listener**.

An event listener allows you to listen for events that take place in your app, such as when views get clicked, when they receive or lose the focus, or when the user presses a hardware key on their device. By implementing an event listener, you can tell when your user performs a particular action—such as clicking on an item in a list view—and respond to it.

OnItemClickListener listens for item clicks

When you want to get items in a list view to respond to clicks, you need to create an `OnItemClickListener` and implement its `onItemClick()` method. The `OnItemClickListener` listens for when items are clicked, and the `onItemClick()` method lets you say how your activity should respond to the click. The `onItemClick()` method includes several parameters that you can use to find out which item was clicked, such as a reference to the view item that was clicked, the item's position in the list view (starting at 0), and the row ID of the underlying data.

We want to start `DrinkCategoryActivity` when the first item in the list view is clicked—the item at position 0. If the item at position 0 is clicked, we need to create an intent to start `DrinkCategoryActivity`. Here's the code to create the listener; we'll add it to `TopLevelActivity.java` on the next page:

```

AdapterView.OnItemClickListener itemClickListener = new AdapterView.OnItemClickListener() {
    public void onItemClick(AdapterView<?> listView,
                            View itemView,
                            int position,
                            long id) {
        if (position == 0) {
            Intent intent = new Intent(TopLevelActivity.this, DrinkCategoryActivity.class);
            startActivity(intent);
        }
    }
};

```

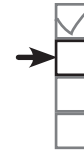
The diagram shows the relationship between the Activity and the ListView. The Activity is represented by a purple hexagon, and the ListView is represented by a purple circle. An arrow points from the Activity to the ListView with the text: "The ListView needs to know the Activity cares what happens to it." Another arrow points from the ListView back to the Activity with the text: "The ListView tells the Activity when an item gets clicked so the Activity can react."

The code is annotated with handwritten notes:

- `AdapterView.OnItemClickListener itemClickListener = new AdapterView.OnItemClickListener() {`: `OnItemClickListener` is a nested class within the `AdapterView` class. A `ListView` is a subclass of `AdapterView`.
- `AdapterView<?> listView,`: The view that was clicked (in this case, the list view).
- `View itemView,`: These parameters give you info about which item was clicked in the list view, such as the item's view and its position.
- `int position,`: These parameters give you info about which item was clicked in the list view, such as the item's view and its position.
- `long id) {`: These parameters give you info about which item was clicked in the list view, such as the item's view and its position.
- `if (position == 0) {`: Drinks is the first item in the list view, so it's at position 0.
- `Intent intent = new Intent(TopLevelActivity.this, DrinkCategoryActivity.class);`: The intent is coming from `TopLevelActivity`.
- `startActivity(intent);`: It needs to launch `DrinkCategoryActivity`.

Once you've created the listener, you need to add it to the list view.

Set the listener to the list view



Add resources
TopLevelActivity
DrinkCategoryActivity
DrinkActivity

Once you've created the `OnClickItemClickListener`, you need to attach it to the list view. You do this using the `ListView.setOnItemClickListener()` method, which takes one argument, the listener itself:

```

AdapterView.OnItemClickListener itemClickListener = new AdapterView.OnItemClickListener() {
    public void onItemClick(AdapterView<?> listView,
        ...
    }
};
ListView listView = (ListView) findViewById(R.id.list_options);
listView.setOnItemClickListener(itemClickListener);

```

We'll add this to `TopLevelActivity`. The full code is listed on the next couple of pages so you can see it in context.

This is the listener we just created.

Adding the listener to the list view is crucial, as it's this step that notifies the listener when the user clicks on items in the list view. If you don't do this, the items in your list view won't be able to respond to clicks.

You've now seen everything you need in order to get the `TopLevelActivity` list view to respond to clicks.

The full `TopLevelActivity.java` code

Here's the complete code for `TopLevelActivity.java`. Replace the code the wizard created for you with the code below and on the next page, then save your changes:

```

package com.hfad.starbuzz;

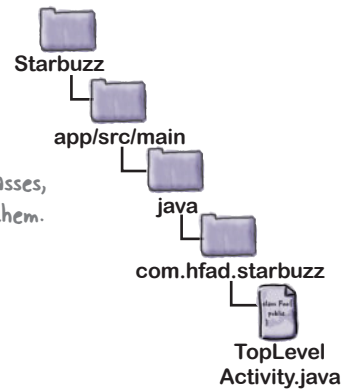
import android.app.Activity;
import android.os.Bundle;
import android.content.Intent;
import android.widget.AdapterView;
import android.widget.AdapterView.OnItemClickListener;
import android.widget.ListView;
import android.view.View;

public class TopLevelActivity extends Activity {

```

We're using all these classes, so we need to import them.

Make sure your activity extends the `Activity` class.



TopLevelActivity.java (continued)



Add resources
TopLevelActivity
DrinkCategoryActivity
DrinkActivity

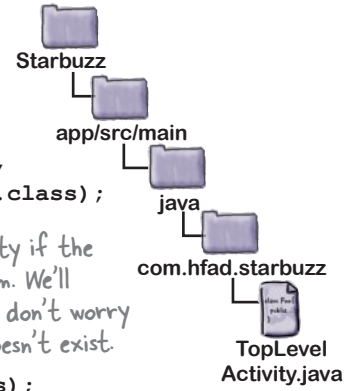
```

@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_top_level);
    //Create an OnItemClickListener
    AdapterView.OnItemClickListener itemClickListener =
        new AdapterView.OnItemClickListener() {
            public void onItemClick(AdapterView<?> listView,
                View itemView,
                int position,
                long id) {
                if (position == 0) {
                    Intent intent = new Intent(TopLevelActivity.this,
                        DrinkCategoryActivity.class);
                    startActivity(intent);
                }
            }
        };
    //Add the listener to the list view
    ListView listView = (ListView) findViewById(R.id.list_options);
    listView.setOnItemClickListener(itemClickListener);
}
}
    
```

← Create the listener.

← Implement its onItemClick() method.

← Launch DrinkCategoryActivity if the user clicks on the Drinks item. We'll create this activity next, so don't worry if Android Studio says it doesn't exist.



← Add the listener to the list view.

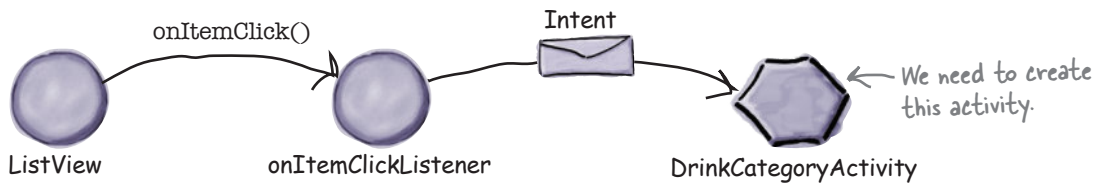
What the TopLevelActivity.java code does

- 1 The onCreate() method in TopLevelActivity creates an onItemClickListener and links it to the activity's ListView.



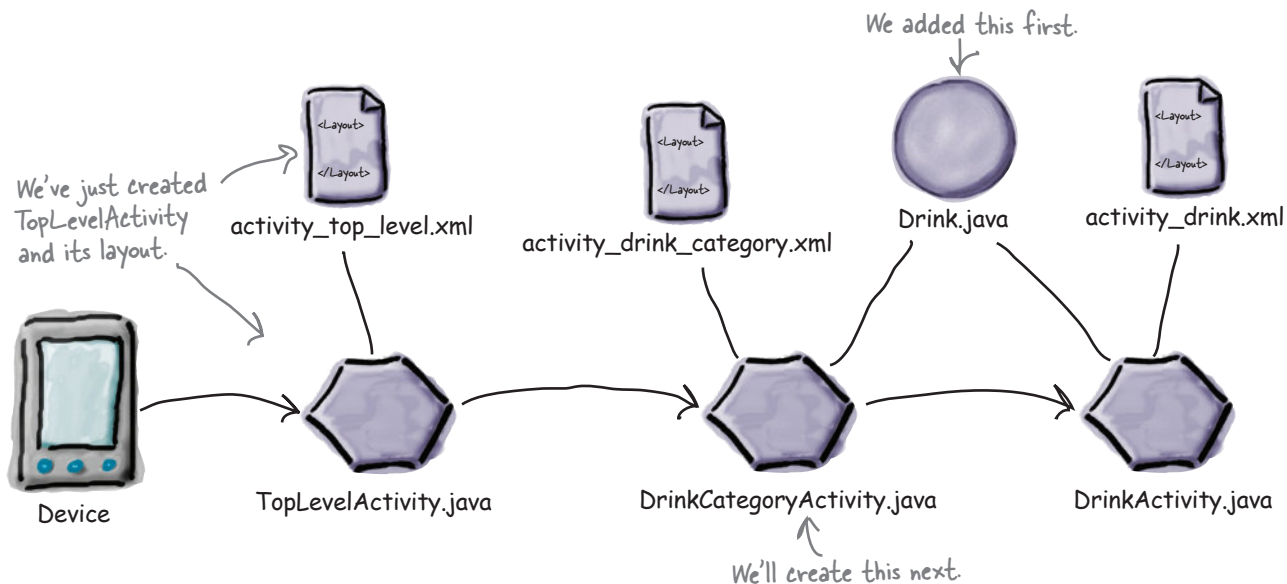
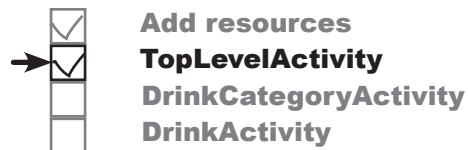
- 2 When the user clicks on an item in the list view, the onItemClickListener's onItemClick() method gets called.

If the Drinks item is clicked, the onItemClickListener creates an intent to start DrinkCategoryActivity.



Where we've got to

So far we've added *Drink.java* to our app and created *TopLevelActivity* and its layout.



The next thing we need to do is create *DrinkCategoryActivity* and its layout so that it gets launched when the user clicks on the Drinks option in *TopLevelActivity*.

there are no Dumb Questions

Q: Why did we have to create an event listener to get items in the list view to respond to clicks? Couldn't we have just used its `android:onClick` attribute in the layout code?

A: You can only use the `android:onClick` attribute in activity layouts for buttons, or any views that are subclasses of `Button` such as `CheckBoxes` and `RadioButtons`.

The `ListView` class isn't a subclass of `Button`, so using the `android:onClick` attribute won't work. That's why you have to implement your own listener.



Exercise

Here's some activity code from a separate project. When the user clicks on an item in a list view, the code is meant to display the text of that item in a text view (the text view has an ID of `text_view` and the list view has an ID of `list_view`). Does the code do what it's meant to? If not, why not?

```
package com.hfad.ch06ex;

import android.app.Activity;
import android.os.Bundle;
import android.widget.AdapterView;
import android.widget.AdapterView.OnItemClickListener;
import android.widget.AdapterView.OnItemSelectedListener;
import android.widget.AdapterView.OnItemClickListener;
import android.widget.AdapterView.OnItemSelectedListener;
import android.widget.AdapterView.OnItemClickListener;
import android.widget.AdapterView.OnItemSelectedListener;
import android.widget.AdapterView.OnItemClickListener;
import android.widget.AdapterView.OnItemSelectedListener;
import android.view.View;

public class MainActivity extends Activity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        final TextView textView = (TextView) findViewById(R.id.text_view);
        AdapterView.OnItemClickListener itemClickListener =
            new AdapterView.OnItemClickListener() {
                public void onItemClick(AdapterView<?> listView,
                    View v,
                    int position,
                    long id) {
                    TextView item = (TextView) v;
                    textView.setText(item.getText());
                }
            };
        ListView listView = (ListView) findViewById(R.id.list_view);
    }
}
```



Exercise Solution

Here's some activity code from a separate project. When the user clicks on an item in a list view, the code is meant to display the text of that item in a text view (the text view has an ID of `text_view` and the list view has an ID of `list_view`). Does the code do what it's meant to? If not, why not?

```
package com.hfad.ch06ex;

import android.app.Activity;
import android.os.Bundle;
import android.widget.AdapterView;
import android.widget.AdapterView.OnItemClickListener;
import android.widget.AdapterView.OnItemSelectedListener;
import android.widget.AdapterView.OnItemClickListener;
import android.widget.AdapterView.OnItemSelectedListener;
import android.widget.AdapterView.OnItemClickListener;
import android.widget.AdapterView.OnItemSelectedListener;
import android.view.View;

public class MainActivity extends Activity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        final TextView textView = (TextView) findViewById(R.id.text_view);
        AdapterView.OnItemClickListener itemClickListener =
            new AdapterView.OnItemClickListener() {
                public void onItemClick(AdapterView<?> listView,
                    View v,
                    int position,
                    long id) {
                    TextView item = (TextView) v;
                    textView.setText(item.getText());
                }
            };
        ListView listView = (ListView) findViewById(R.id.list_view);
    }
}
```

This is the item in the
ListView that was clicked.
It's a TextView, so we can
get its text using `getText()`.

The code doesn't work as intended because the line of code

```
listView.setOnItemClickListener(itemClickListener);
```

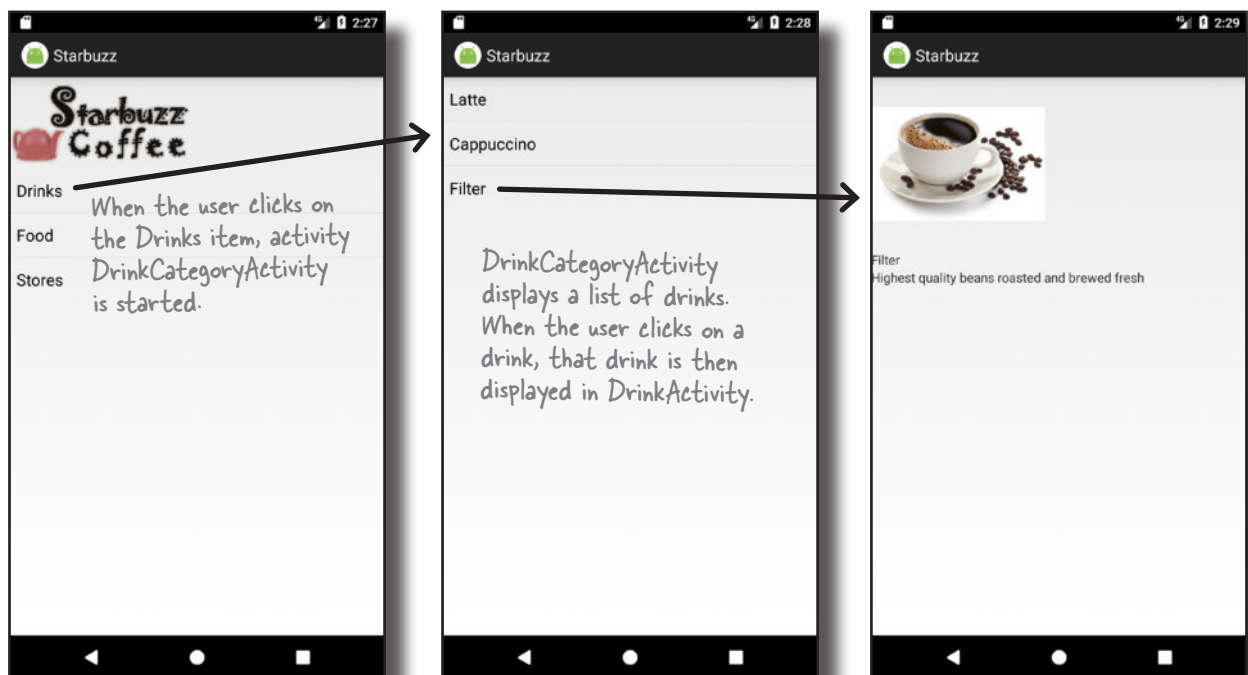
is missing from the end of the code. Apart from that, the code's fine.



A category activity displays the data for a single category

As we said earlier, `DrinkCategoryActivity` is an example of a category activity. A category activity is one that shows the data that belongs to a particular category, often in a list. You use the category activity to navigate to details of the data.

We're going to use `DrinkCategoryActivity` to display a list of drinks. When the user clicks on one of the drinks, we'll show them the details of that drink.



Create `DrinkCategoryActivity`

To work on the next step in our checklist, we'll create an activity with a single list view that displays a list of all the drinks. Select the `com.hfad.starbuzz` package in the `app/src/main/java` folder, then go to `File` → `New...` → `Activity` → `Empty Activity`. Name the activity "`DrinkCategoryActivity`", name the layout "`activity_drink_category`", make sure the package name is `com.hfad.starbuzz` and **uncheck the Backwards Compatibility (AppCompat) checkbox**.

We'll update the layout code on the next page.

Some versions of Android Studio may ask you what the source language of your activity should be. If prompted, select the option for Java.

Update activity_drink_category.xml

Here's the code for *activity_drink_category.xml*. As you can see, it's a simple linear layout with a list view. Update your version of *activity_drink_category.xml* to reflect ours 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="com.hfad.starbuzz.DrinkCategoryActivity">

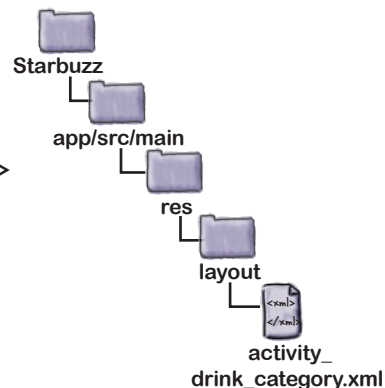
    <ListView
        android:id="@+id/list_drinks"
        android:layout_width="match_parent"
        android:layout_height="wrap_content" />

</LinearLayout>
```

This layout only needs
to contain a ListView.



Add resources
TopLevelActivity
DrinkCategoryActivity
DrinkActivity



There's one key difference between the list view we're creating here, and the one we created in *activity_top_activity.xml*: there's no `android:entries` attribute here. But why?

In *activity_top_activity.xml*, we used the `android:entries` attribute to bind data to the list view. This worked because the data was held as a static String array resource. The array was described in *strings.xml*, so we could easily refer to it using:

```
android:entries="@array/options"
```

where `options` is the name of the String array.

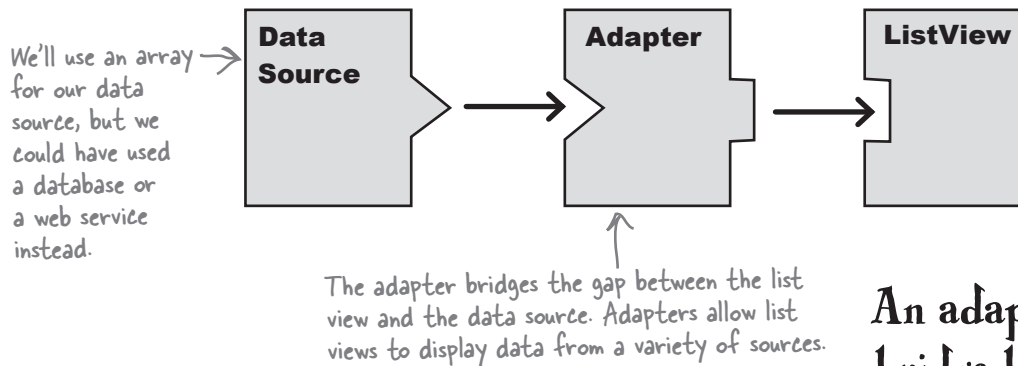
Using `android:entries` works fine if the data is a static array in *strings.xml*. But what if it isn't? What if the data is held in an array you've programmatically created in Java code, or held in a database? In that case, the `android:entries` attribute won't work.

If you need to bind your list view to data held in something other than a String array resource, you need to take a different approach; you need to write activity code to bind the data. In our case, we need to bind our list view to the `drinks` array in the `Drink` class.



For nonstatic data, use an adapter

If you need to display data in a list view that comes from a source other than *strings.xml* (such as a Java array or database), you need to use an **adapter**. An adapter acts as a bridge between the data source and the list view:

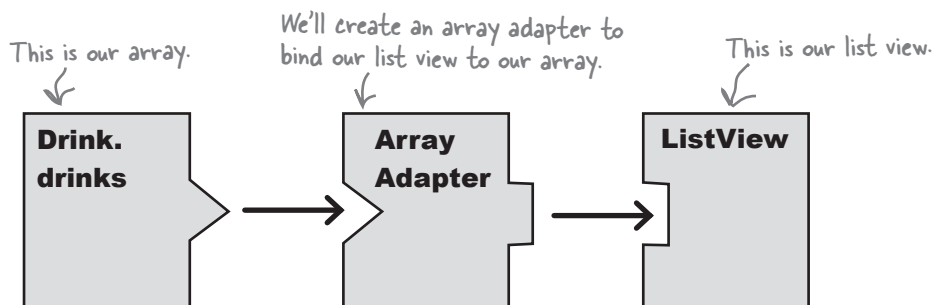


An adapter acts as a bridge between a view and a data source. An ArrayAdapter is a type of adapter that specializes in working with arrays.

There are several different types of adapter. For now, we're going to focus on **array adapters**.

An array adapter is a type of adapter that's used to bind arrays to views. You can use it with any subclass of the `AdapterView` class, which means you can use it with both list views and spinners.

In our case, we're going to use an array adapter to display data from the `Drink.drinks` array in the list view.



We'll see how this works on the next page.

Connect list views to arrays with an array adapter



Add resources
TopLevelActivity
DrinkCategoryActivity
DrinkActivity

You use an array adapter by initializing it and then attaching it to the list view.

To initialize the array adapter, you first specify what type of data is contained in the array you want to bind to the list view. You then pass the array adapter three parameters: a `Context` (usually the current activity), a layout resource that specifies how to display each item in the array, and the array itself.

Here's the code to create an array adapter that displays drink data from the `Drink.drinks` array (you'll add this code to `DrinkCategoryActivity.java` on the next page):

```

        ArrayAdapter<Drink> listAdapter = new ArrayAdapter<>(
            "this" refers to the current activity. The Activity class is a subclass of Context. → this,
            android.R.layout.simple_list_item_1, ← This is a built-in layout resource. It tells the array adapter to display each item in the array in a single text view.
            Drink.drinks); ← The array
    
```

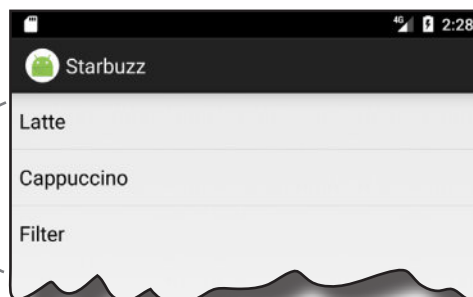
You then attach the array adapter to the list view using the `ListView.setAdapter()` method:

```

        ListView listDrinks = (ListView) findViewById(R.id.list_drinks);
        listDrinks.setAdapter(listAdapter);
    
```

Behind the scenes, the array adapter takes each item in the array, converts it to a `String` using its `toString()` method, and puts each result into a text view. It then displays each text view as a single row in the list view.

These are the drinks from the drinks array. Each row in the list view is a single text view, each one displaying a separate drink.





Add the array adapter to DrinkCategoryActivity

We'll change the *DrinkCategoryActivity.java* code so that the list view uses an array adapter to get drinks data from the *Drink* class. We'll put the code in the *onCreate()* method so that the list view gets populated when the activity gets created.

Here's the full code for the activity (update your copy of *DrinkCategoryActivity.java* to reflect ours, then save your changes):

```
package com.hfad.starbuzz;

import android.app.Activity;
import android.os.Bundle;
import android.widget.ArrayAdapter;
import android.widget.ListView;

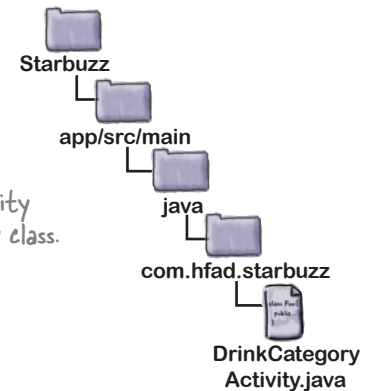
public class DrinkCategoryActivity extends Activity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_drink_category);
        ArrayAdapter<Drink> listAdapter = new ArrayAdapter<>(
            this,
            android.R.layout.simple_list_item_1,
            Drink.drinks);
        ListView listDrinks = (ListView) findViewById(R.id.list_drinks);
        listDrinks.setAdapter(listAdapter);
    }
}
```

We're using these classes so we need to import them.

Make sure your activity extends the Activity class.

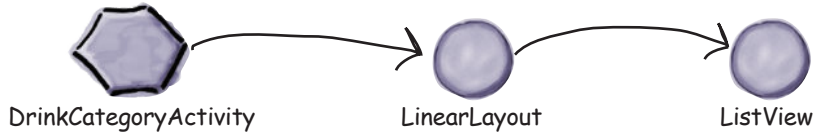
This populates the list view with data from the drinks array.



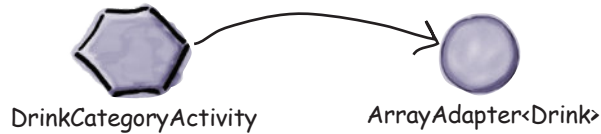
These are all the changes needed to get your list view to display a list of the drinks from the *Drink* class. Let's go through what happens when the code runs.

What happens when you run the code

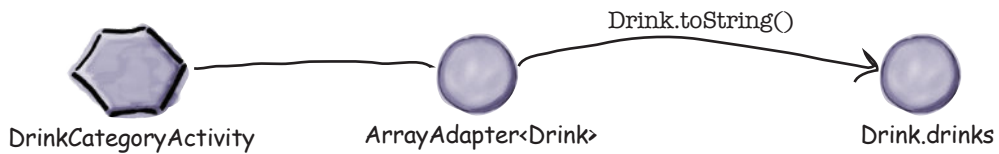
- 1 When the user clicks on the Drinks option, `DrinkCategoryActivity` is launched. Its layout has a `LinearLayout` that contains a `ListView`.



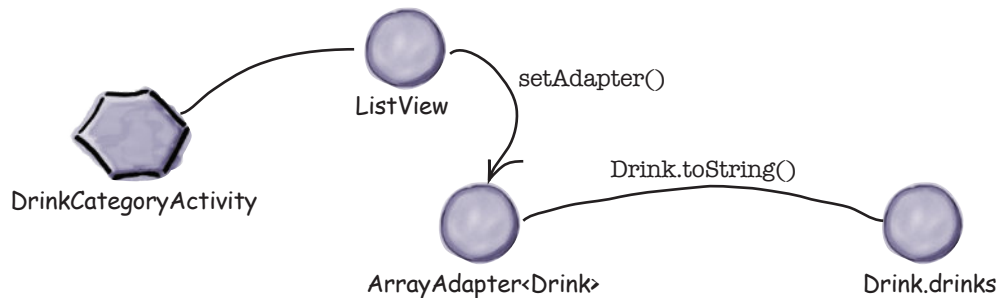
- 2 `DrinkCategoryActivity` creates an `ArrayAdapter<Drink>`, an array adapter that deals with arrays of `Drink` objects.



- 3 The array adapter retrieves data from the drinks array in the `Drink` class. It uses the `Drink.toString()` method to return the name of each drink.



- 4 `DrinkCategoryActivity` makes the `ListView` use the array adapter via the `setAdapter()` method. The list view uses the array adapter to display a list of the drink names.





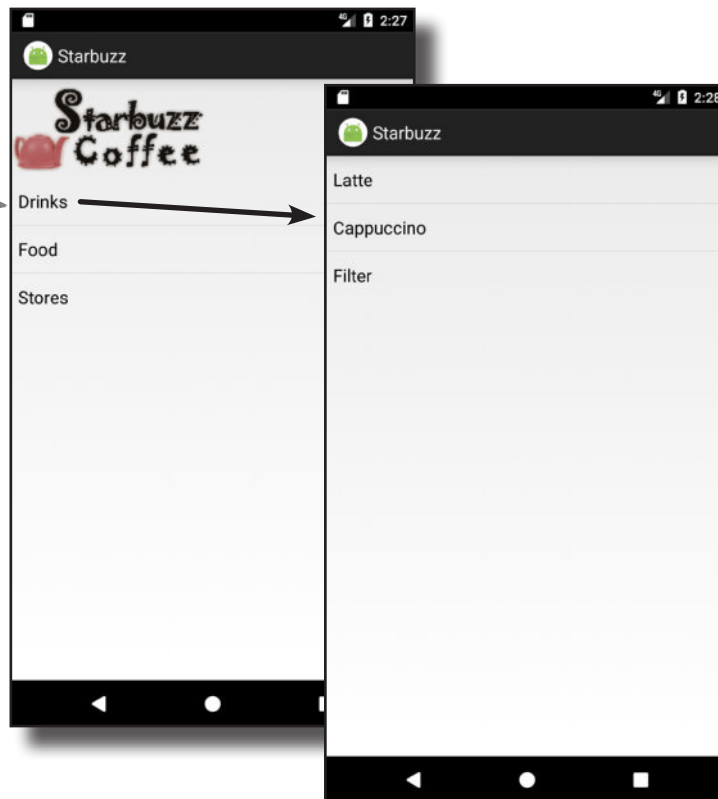
Test drive the app

When you run the app, `TopLevelActivity` gets displayed as before. When you click on the `Drinks` item, `DrinkCategoryActivity` is launched. It displays the names of all the drinks from the `Drink` Java class.



list views and adapters
Add resources
TopLevelActivity
DrinkCategoryActivity
DrinkActivity

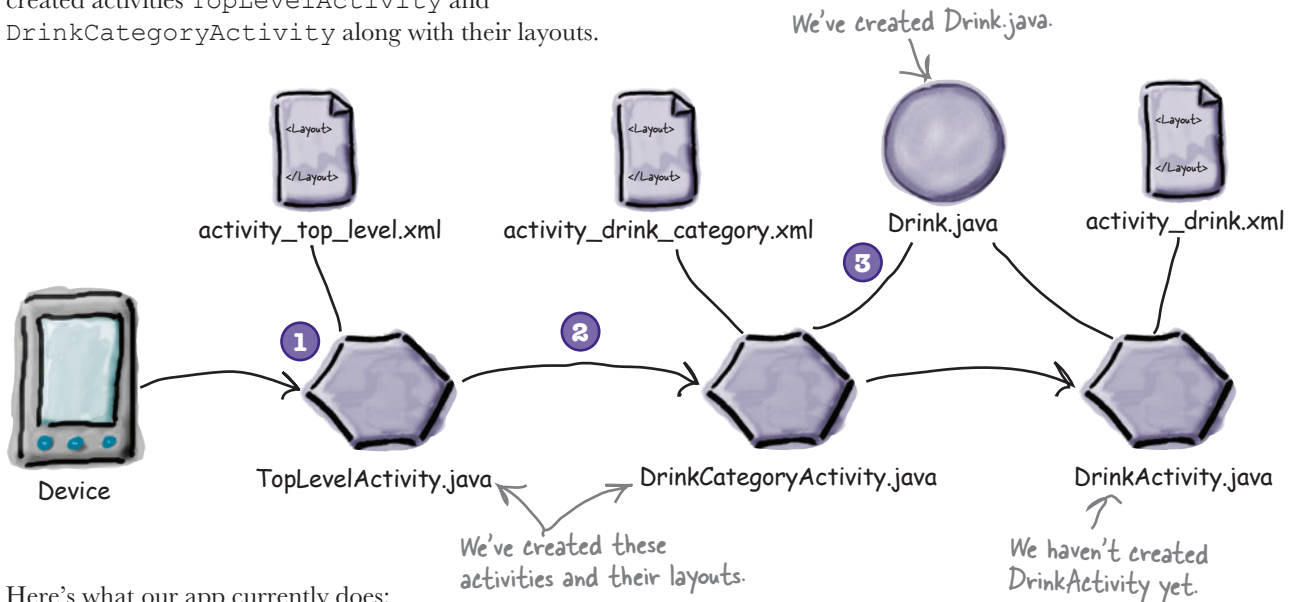
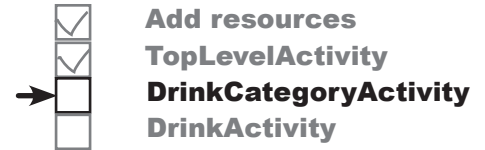
Click on the `Drinks` item →
to see a list of drinks.



On the next page we'll review what we've done in the app so far, and what's left for us to do.

App review: where we are

So far we've added *Drink.java* to our app, and created activities *TopLevelActivity* and *DrinkCategoryActivity* along with their layouts.



Here's what our app currently does:

- 1 When the app gets launched, it starts activity *TopLevelActivity*.**
The activity displays a list of options for Drinks, Food, and Stores.
- 2 The user clicks on Drinks in *TopLevelActivity*.**
This launches activity *DrinkCategoryActivity*, which displays a list of drinks.
- 3 *DrinkCategoryActivity* gets the values for its list of drinks from the *Drink.java* class file.**

The next thing we'll do is get *DrinkCategoryActivity* to launch *DrinkActivity*, passing it details of whichever drink was clicked.

Pool Puzzle



Your **goal** is to create an activity that binds a Java array of colors to a spinner. Take code snippets from the pool and place them into the blank lines in the activity. You may **not** use the same snippet more than once, and you won't need to use all the snippets.

Remember, we covered spinners in Chapter 5.

We're not using this activity in our app.

...

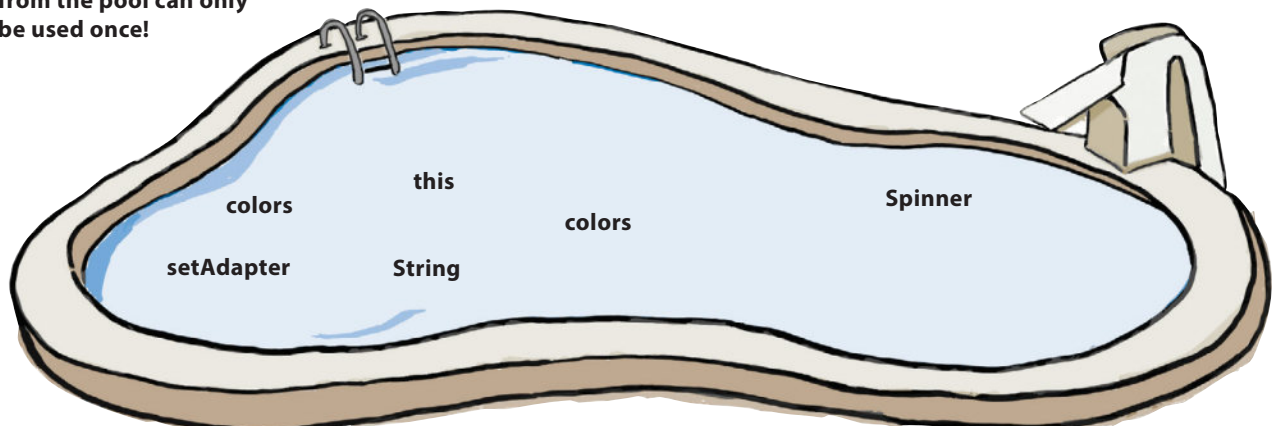
```
public class MainActivity extends Activity {

    String[] colors = new String[] {"Red", "Orange", "Yellow", "Green", "Blue"};

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        Spinner spinner = (.....) findViewById(R.id.spinner);
        ArrayAdapter<.....> adapter = new ArrayAdapter<>(
            .....,
            android.R.layout.simple_spinner_item,
            colors);
        spinner.....(adapter);
    }
}
```

This displays each value in the array as a single row in the spinner.

Note: each snippet from the pool can only be used once!



Answers on page 287.

you are here ▶ 275

How we handled clicks in TopLevelActivity



Add resources
TopLevelActivity
DrinkCategoryActivity
DrinkActivity

Earlier in this chapter, we needed to get TopLevelActivity to react to the user clicking on the first item in the list view, the Drinks option, by starting DrinkCategoryActivity. To do that, we had to create an OnItemClickListener, implement its onItemClick() method, and assign it to the list view. Here's a reminder of the code:

```

AdapterView.OnItemClickListener itemClickListener =
    new AdapterView.OnItemClickListener() {
        public void onItemClick(AdapterView<?> listView,
            View itemView,
            int position,
            long id) {
            if (position == 0) {
                Intent intent = new Intent(TopLevelActivity.this,
                    DrinkCategoryActivity.class);
                startActivity(intent);
            }
        }
    };
ListView listView = (ListView) findViewById(R.id.list_options);
listView.setOnItemClickListener(itemClickListener);

```

← Create the listener.

← The list view

} The item view that was clicked, its position in the list, and the row ID of the underlying data.

← Add the listener to the list view.

We had to set up an event listener in this way because list views aren't hardwired to respond to clicks in the way that buttons are.

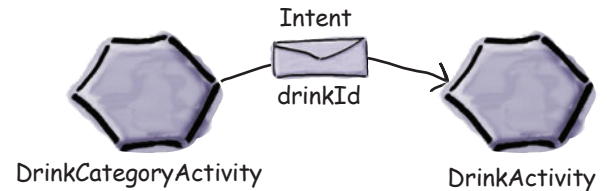
So how should we get DrinkCategoryActivity to handle user clicks?

Pass the ID of the item that was clicked by adding it to an intent



When you use a category activity to display items in a list view, you'll usually use the `onItemClick()` method to start another activity that displays details of the item the user clicked. To do this, you create an intent that starts the second activity. You then add the ID of the item that was clicked as extra information so that the second activity can use it when the activity starts.

In our case, we want to start `DrinkActivity` and pass it the ID of the drink that was selected. `DrinkActivity` will then be able to use this information to display details of the right drink. Here's the code for the intent:



```

Intent intent = new Intent(DrinkCategoryActivity.this, DrinkActivity.class);
intent.putExtra(DrinkActivity.EXTRA_DRINKID, (int) id);
startActivity(intent);
  
```

DrinkCategoryActivity needs to start DrinkActivity.

Add the ID of the item that was clicked to the intent. This is the index of the drink in the drinks array.

We're using a constant for the name of the extra information in the intent so that we know DrinkCategoryActivity and DrinkActivity are using the same String. We'll add this constant to DrinkActivity when we create the activity.

It's common practice to pass the ID of the item that was clicked because it's the ID of the underlying data. If the underlying data is an array, the ID is the index of the item in the array. If the underlying data comes from a database, the ID is the ID of the record in the table. Passing the ID of the item in this way means that it's easier for the second activity to get details of the data, and then display it.

That's everything we need to make `DrinkCategoryActivity` start `DrinkActivity` and tell it which drink was selected. The full activity code is on the next page.

The full DrinkCategoryActivity code

Here's the full code for *DrinkCategoryActivity.java* (add the new method to your code, then save your changes):

```

package com.hfad.starbuzz;

import android.app.Activity;
import android.os.Bundle;
import android.widget.AdapterView;
import android.widget.AdapterView.OnItemClickListener;
import android.widget.ArrayAdapter;
import android.widget.ListView;
import android.view.View;
import android.content.Intent;
import android.widget.AdapterView.OnItemClickListener;

public class DrinkCategoryActivity extends Activity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_drink_category);
        ArrayAdapter<Drink> listAdapter = new ArrayAdapter<>(
            this,
            android.R.layout.simple_list_item_1,
            Drink.drinks);
        ListView listDrinks = (ListView) findViewById(R.id.list_drinks);
        listDrinks.setAdapter(listAdapter);

        //Create the listener
        AdapterView.OnItemClickListener itemClickListener =
            new AdapterView.OnItemClickListener() {
                public void onItemClick(AdapterView<?> listDrinks,
                    View itemView,
                    int position,
                    long id) {
                    //Pass the drink the user clicks on to DrinkActivity
                    Intent intent = new Intent(DrinkCategoryActivity.this,
                        DrinkActivity.class);
                    intent.putExtra(DrinkActivity.EXTRA_DRINKID, (int) id);
                    startActivity(intent);
                }
            };

        //Assign the listener to the list view
        listDrinks.setOnItemClickListener(itemClickListener);
    }
}

```

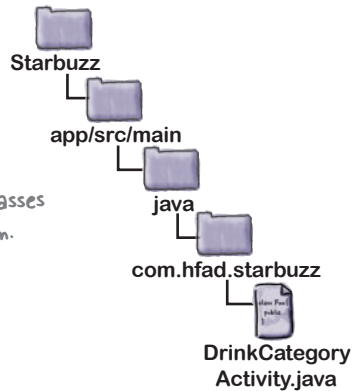
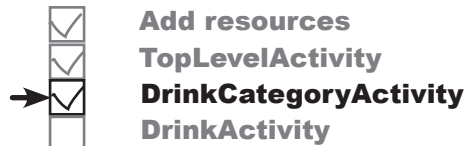
We're using these extra classes so we need to import them.

This gets called when an item in the list view is clicked.

When the user clicks on a drink, pass its ID to DrinkActivity and start it.

Create a listener to listen for clicks.

We'll add DrinkActivity next, so don't worry if Android Studio says it doesn't exist.



A detail activity displays data for a single record



As we said earlier, `DrinkActivity` is an example of a detail activity. A detail activity displays details for a particular record, and you generally navigate to it from a category activity.

We're going to use `DrinkActivity` to display details of the drink the user selects. The `Drink` class includes the drink's name, description, and image resource ID, so we'll display this data in our layout. We'll include an image view for the drink image resource, and text views for the drink name and description.

To create the activity, select the `com.hfad.starbuzz` package in the `app/src/main/java` folder, then go to `File→New...→Activity→Empty Activity`. Name the activity "DrinkActivity", name the layout "activity_drink", make sure the package name is `com.hfad.starbuzz`, and **uncheck the Backwards Compatibility (AppCompat) checkbox**. Then replace the contents of `activity_drink.xml` with this:

Make sure you create the new activity.

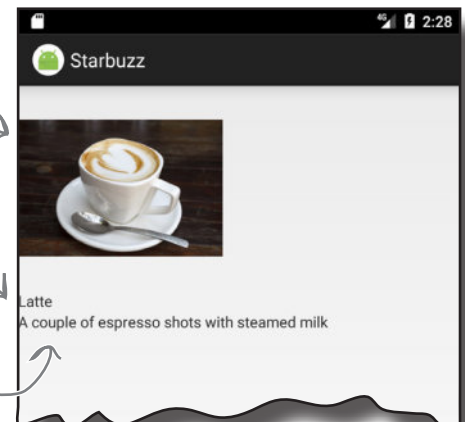
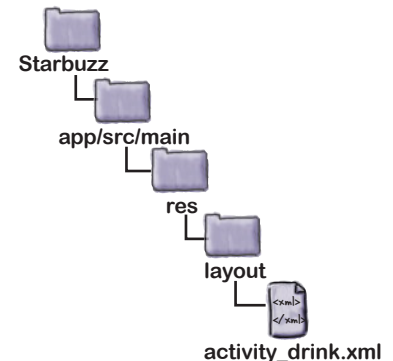
If prompted for the activity's source language, select the option for Java.

```
<?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="com.hfad.starbuzz.DrinkActivity" >

    <ImageView
        android:id="@+id/photo"
        android:layout_width="190dp"
        android:layout_height="190dp" />

    <TextView
        android:id="@+id/name"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" />

    <TextView
        android:id="@+id/description"
        android:layout_width="match_parent"
        android:layout_height="wrap_content" />
</LinearLayout>
```



Now that you've created the layout of your detail activity, we can populate its views.

Retrieve data from the intent

As you've seen, when you want a category activity to start a detail activity, you have to make items in the category activity list view respond to clicks. When an item is clicked, you create an intent to start the detail activity. You pass the ID of the item the user clicked as extra information in the intent.

When the detail activity is started, it can retrieve the extra information from the intent and use it to populate its views. In our case, we can use the information in the intent that started `DrinkActivity` to retrieve details of the drink the user clicked.

When we created `DrinkCategoryActivity`, we added the ID of the drink the user clicked as extra information in the intent. We gave it the label `DrinkActivity.EXTRA_DRINKID`, which we need to define as a constant in `DrinkActivity.java`:

```
public static final String EXTRA_DRINKID = "drinkId";
```

As you saw in Chapter 3, you can retrieve the intent that started an activity using the `getIntent()` method. If this intent has extra information, you can use the intent's `get*()` methods to retrieve it. Here's the code to retrieve the value of `EXTRA_DRINKID` from the intent that started `DrinkActivity`:

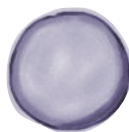
```
int drinkId = (Integer) getIntent().getExtras().get(EXTRA_DRINKID);
```

Once you've retrieved the information from the intent, you can use it to get the data you need to display in your detail record.

In our case, we can use `drinkId` to get details of the drink the user selected. `drinkId` is the ID of the drink, the index of the drink in the `drinks` array. This means that you can get details about the drink the user clicked on using:

```
Drink drink = Drink.drinks[drinkId];
```

This gives us a `Drink` object containing all the information we need to update the views attributes in the activity:



drink

```
name="Latte"
description="A couple of espresso shots with steamed milk"
imageResourceId=R.drawable.latte
```

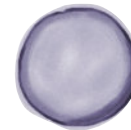


Add resources
TopLevelActivity
DrinkCategoryActivity
DrinkActivity

Update the views with the data

When you update the views in your detail activity, you need to make sure that the values they display reflect the data you've derived from the intent.

Our detail activity contains two text views and an image view. We need to make sure that each of these is updated to reflect the details of the drink.



drink

name
description
imageResourceId



Drink Magnets

See if you can use the magnets below to populate the `DrinkActivity` views with the correct data.

```
...
//Get the drink from the intent
int drinkId = (Integer) getIntent().getExtras().get(EXTRA_DRINKID);
Drink drink = Drink.drinks[drinkId];

//Populate the drink name
TextView name = (TextView) findViewById(R.id.name);

name.....(drink.getName());

//Populate the drink description
TextView description = (TextView) findViewById(R.id.description);

description.....(drink.getDescription());

//Populate the drink image
ImageView photo = (ImageView) findViewById(R.id.photo);

photo.....(drink.getImageResourceId());

photo.....(drink.getName());
...
```

setText

setContent

setContentDescription

setImageResourceId

setImageResource

setText



Drink Magnets Solution

See if you can use the magnets below to populate the DrinkActivity views with the correct data.

```
...
//Get the drink from the intent
int drinkId = (Integer) getIntent().getExtras().get(EXTRA_DRINKID);
Drink drink = Drink.drinks[drinkId];
```

```
//Populate the drink name
TextView name = (TextView) findViewById(R.id.name);
```

```
name.setText(drink.getName());
```

Use `setText()` to set the text in a text view.

```
//Populate the drink description
TextView description = (TextView) findViewById(R.id.description);
```

```
description.setText(drink.getDescription());
```

```
//Populate the drink image
ImageView photo = (ImageView) findViewById(R.id.photo);
```

You set the source of the image using `setImageResource()`.

```
photo.setImageResource(drink.getImageResourceId());
```

This is needed to make the app more accessible.

```
photo.setContentDescription(drink.getName());
```

```
...
```

You didn't need to use these.

setContent

setImageResourceId



The DrinkActivity code

Here's the full code for *DrinkActivity.java* (replace the code the wizard gave you with the code below, then save your changes):

```

package com.hfad.starbuzz;

import android.app.Activity;
import android.os.Bundle;
import android.widget.ImageView;
import android.widget.TextView;

public class DrinkActivity extends Activity {

    public static final String EXTRA_DRINKID = "drinkId";

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

        //Get the drink from the intent
        int drinkId = (Integer) getIntent().getExtras().get(EXTRA_DRINKID);
        Drink drink = Drink.drinks[drinkId];

        //Populate the drink name
        TextView name = (TextView) findViewById(R.id.name);
        name.setText(drink.getName());

        //Populate the drink description
        TextView description = (TextView) findViewById(R.id.description);
        description.setText(drink.getDescription());

        //Populate the drink image
        ImageView photo = (ImageView) findViewById(R.id.photo);
        photo.setImageResource(drink.getImageResourceId());
        photo.setContentDescription(drink.getName());
    }
}

```

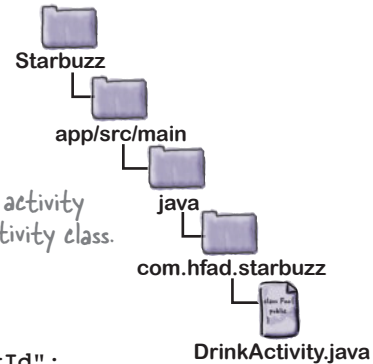
We're using these classes so we need to import them.

Make sure your activity extends the Activity class.

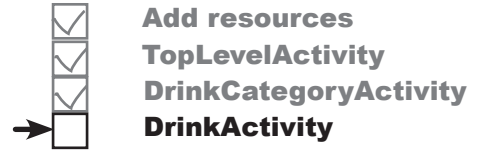
Add EXTRA_DRINKID as a constant.

Use the drinkId to get details of the drink the user chose.

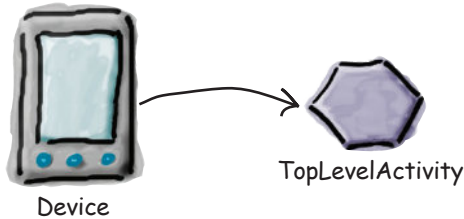
Populate the views with the drink data.



What happens when you run the app



1 When the user starts the app, it launches TopLevelActivity.

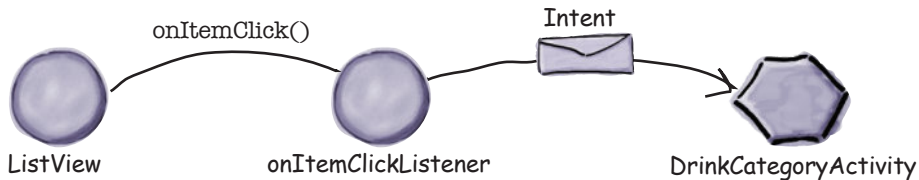


2 The onCreate() method in TopLevelActivity creates an onItemClickListener and links it to the activity's ListView.



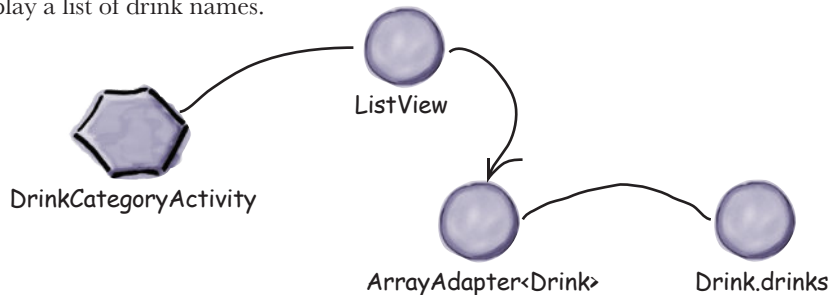
3 When the user clicks on an item in the ListView, the onItemClickListener's onItemClick() method gets called.

If the Drinks item was clicked, the onItemClickListener creates an intent to start DrinkCategoryActivity.



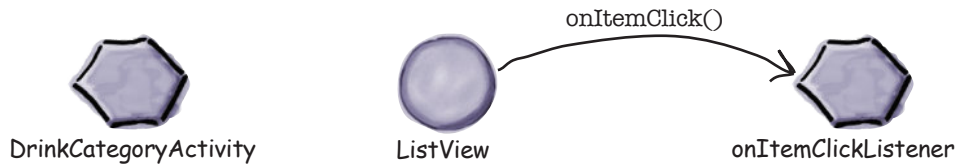
4 DrinkCategoryActivity displays a single ListView.

The DrinkCategoryActivity list view uses an ArrayAdapter<Drink> to display a list of drink names.

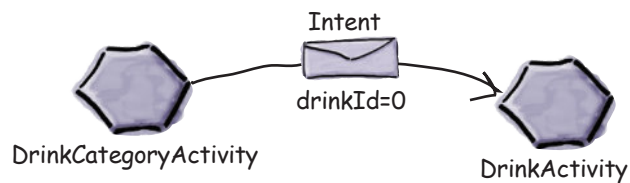


The story continues

- 5 When the user chooses a drink from `DrinkCategoryActivity`'s `ListView`, `onItemClickListener`'s `onItemClick()` method gets called.

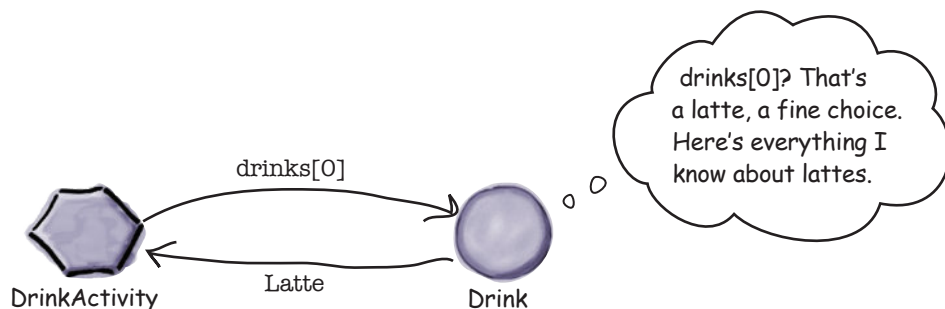


- 6 The `onItemClick()` method creates an intent to start `DrinkActivity`, passing along the drink ID as extra information.



- 7 **DrinkActivity launches.**

It retrieves the drink ID from the intent, and gets details for the correct drink from the `Drink` class. It uses this information to update its views.





Test drive the app

When you run the app, `TopLevelActivity` gets displayed.

We've implemented the Drinks part of the app. The other items won't do anything if you click on them.

When you click on the Drinks item, `DrinkCategoryActivity` is launched. It displays all the drinks from the `Drink` java class.

When you click on one of the drinks, `DrinkActivity` is launched and details of the selected drink are displayed.

Using these three activities, you can see how to structure your app into top-level activities, category activities, and detail/edit activities. In Chapter 15, we'll revisit the Starbuzz app to explain how you can retrieve the drinks from a database.



Pool Puzzle Solution



Your **goal** is to create an activity that binds a Java array of colors to a spinner. Take code snippets from the pool and place them into the blank lines in the activity. You may **not** use the same snippet more than once, and you won't need to use all the snippets.

...

```
public class MainActivity extends Activity {

    String[] colors = new String[] {"Red", "Orange", "Yellow", "Green", "Blue"};

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        Spinner spinner = (Spinner) findViewById(R.id.spinner);
        ArrayAdapter<String> adapter = new ArrayAdapter<>(
            this,
            android.R.layout.simple_spinner_item,
            colors);
        spinner.setAdapter(adapter);
    }
}
```

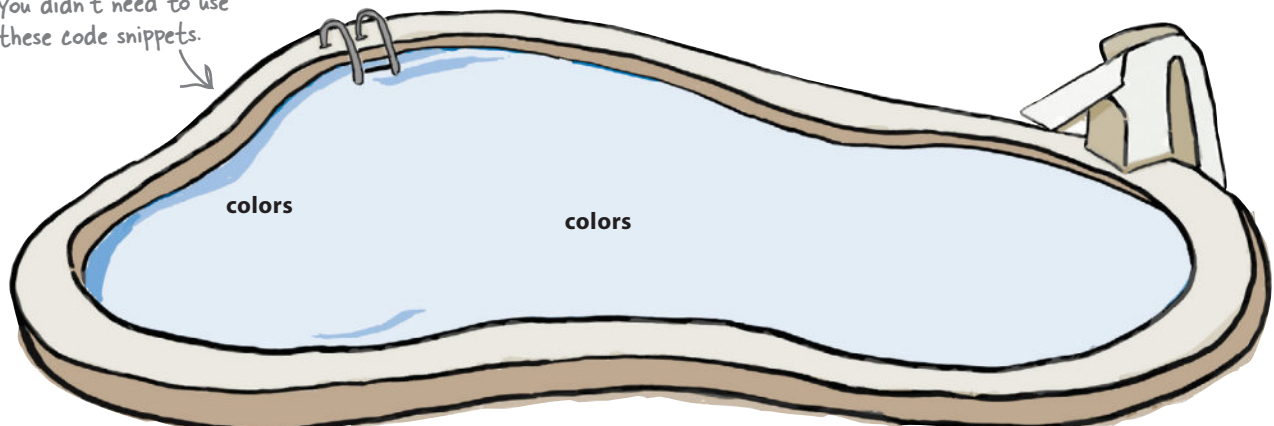
We're using an array of type String.

this,
android.R.layout.simple_spinner_item,
colors);

spinner.**setAdapter**(adapter);

Use setAdapter() to get the spinner to use the array adapter.

You didn't need to use these code snippets.





Your Android Toolbox

You've got **Chapter 7** under your belt and now you've added **list views and app design** to your toolbox.

You can download the full code for the chapter from <https://tinyurl.com/HeadFirstAndroid>.



BULLET POINTS

- Sort your ideas for activities into top-level activities, category activities, and detail/edit activities. Use the category activities to navigate from the top-level activities to the detail/edit activities.
- A list view displays items in a list. Add it to your layout using the `<ListView>` element.
- Use `android:entries` in your layout to populate the items in your list views from an array defined in `strings.xml`.
- An adapter acts as a bridge between an `AdapterView` and a data source. `ListView`s and `Spinners` are both types of `AdapterView`.
- An `ArrayAdapter` is an adapter that works with arrays.
- Handle click events on `Buttons` using `android:onClick` in the layout code. Handle click events elsewhere by creating a listener and implementing its click event.

8 support libraries and app bars

Taking Shortcuts



Everybody likes a shortcut.

And in this chapter you'll see how to add shortcuts to your apps using **app bars**. We'll show you how to start activities by *adding actions* to your app bar, how to share content with other apps using the *share action provider*, and how to navigate up your app's hierarchy by implementing *the app bar's Up button*. Along the way we'll introduce you to the powerful **Android Support Libraries**, which are key to making your apps look fresh on older versions of Android.

Great apps have a clear structure

In the previous chapter, we looked at ways of structuring an app to create the best user experience. Remember that one way of creating an app is to organize the screens into three types:

Top-level screens

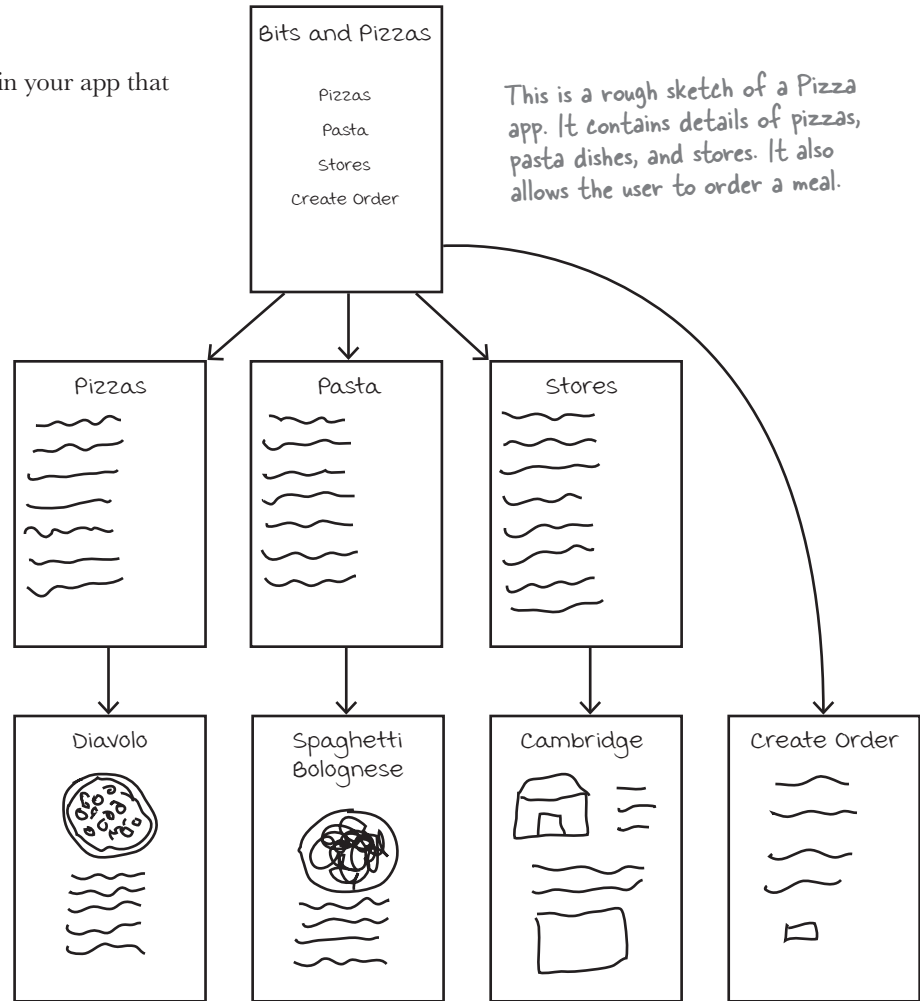
This is usually the first activity in your app that your user sees.

Category screens

Category screens show the data that belongs to a particular category, often in a list. They allow the user to navigate to detail/edit screens.

Detail/edit screens

These display details for a particular record, let the user edit the record, or allow the user to enter new records.



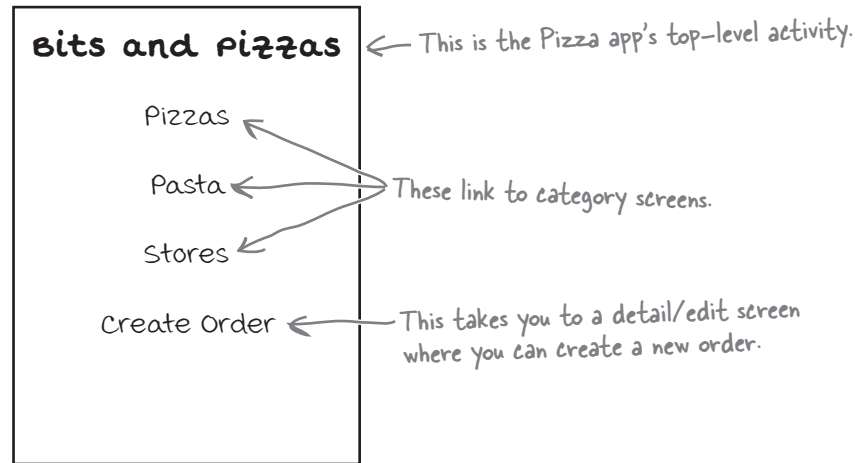
This is a rough sketch of a Pizza app. It contains details of pizzas, pasta dishes, and stores. It also allows the user to order a meal.

They also have great shortcuts

If a user's going to use your app a lot, they'll want quick ways to get around. We're going to look at navigational views that will give your user shortcuts around your app, providing more space in your app for actual content. Let's begin by taking a closer look at the top-level screen in the above Pizza app.

Different types of navigation

In the top-level screen of the Pizza app, there's a list of options for places in the app the user can go to.



The top three options link to category activities; the first presents the user with a list of pizzas, the second a list of pasta, and the third a list of stores. They allow the user to navigate around the app.

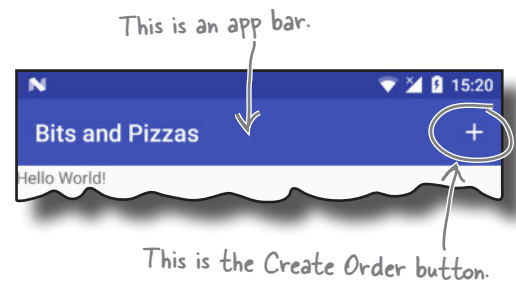
The fourth option links to a detail/edit activity that allows the user to create an order. This option enables the user to perform an **action**.

In Android apps, you can add actions to the **app bar**. The app bar is the bar you often see at the top of activities; it's sometimes known as the **action bar**. You generally put your app's most important actions in the app bar so that they're prominent at the top of the screen.

In the Pizza app, we can make it easy for the user to place an order wherever they are in the app by making sure there's an app bar at the top of every activity that includes a Create Order button. This way the user will have access to it wherever they are.

Let's look at how you create app bars.

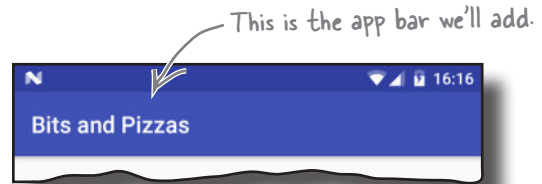
← These are like the navigation options we looked at in Chapter 7.



Here's what we're going to do

There are a few things we're going to cover in this chapter.

- 1 Add a basic app bar.**
 We'll create an activity called `MainActivity` and add a basic app bar to it by applying a theme.

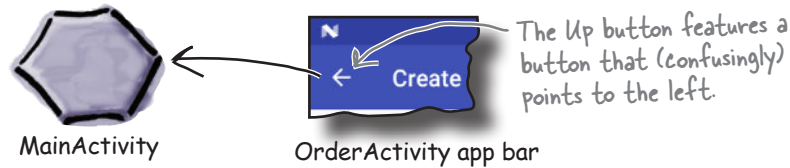


- 2 Replace the basic app bar with a toolbar.**
 To use the latest app bar features, you need to replace the basic app bar with a toolbar. This looks the same as the basic app bar, but you can use it to do more things.

- 3 Add a Create Order action.**
 We'll create a new activity called `OrderActivity`, and add an action to `MainActivity`'s app bar that opens it.

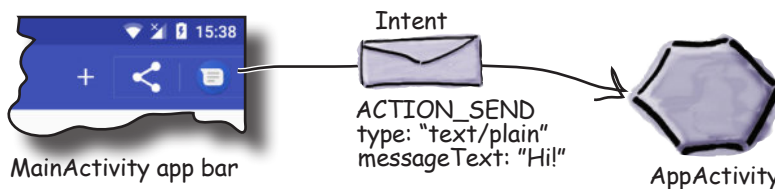


- 4 Implement the Up button.**
 We'll implement the Up button on `OrderActivity`'s app bar so that users have an easy way of navigating back to `MainActivity`.



- 5 Add a share action provider.**
 We'll add a share action provider to `MainActivity`'s app bar so that users can share text with other apps and invite their friends to join them for pizza.

← You'll find out what action providers are later in the chapter.



Let's start by looking at how you add a basic app bar.



- Basic app bar**
- Toolbar**
- Action**
- Up button**
- Share action**

Add an app bar by applying a theme

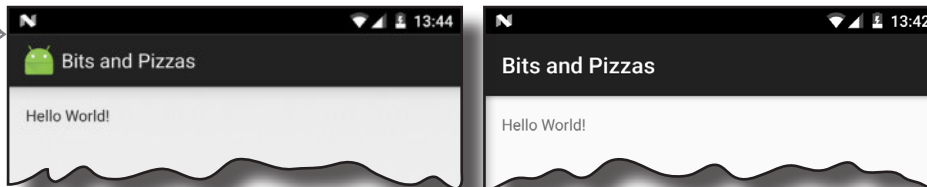
An app bar has a number of uses:

- ★ Displaying the app or activity name so that the user knows where in the app they are. As an example, an email app might use the app bar to indicate whether the user is in their inbox or junk folder.
- ★ Making key actions prominent in a way that's predictable—for example, sharing content or performing searches.
- ★ Navigating to other activities to perform an action.

To add a basic app bar, you need to use a **theme** that includes an app bar. A theme is a style that's applied to an activity or application so that your app has a consistent look and feel. It controls such things as the color of the activity background and app bar, and the style of the text.

Android comes with a number of built-in themes that you can use in your apps. Some of these, such as the Holo themes, were introduced in early releases of Android, and others, such as the Material themes, were introduced much later to give apps a more modern appearance.

The Holo themes have been in Android since API level 11.



The Material themes were introduced in API level 21.

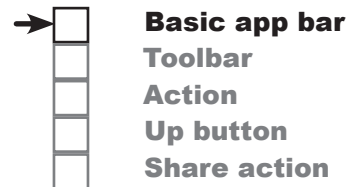
These themes look a bit different from the one on the previous page, as they haven't had any extra styling applied to them. You'll find out how to add styling later in the chapter.

But there's a problem. You want your apps to look as modern and up-to-date as possible, but you can only use themes from the version of Android they were released in. As an example, you can't use the native Material themes on devices that are running a version of Android older than Lollipop, as the Material themes were introduced with API level 21.

The problem isn't just limited to themes. Every new release of Android introduces new features that people want to see in their apps, such as new GUI components. But not everyone upgrades to the latest version of Android as soon as it comes out. In fact, most people are at least one version of Android behind.

So how can you use the latest Android features and themes in your apps if most people aren't using the latest version? How can you give your users a consistent user experience irrespective of what version of Android they're using without making your app look old-fashioned?

Support libraries allow you to use new features in older versions of Android



The Android team solved this problem by coming up with the idea of **Support Libraries**.

The Android Support Libraries provide backward compatibility with older versions of Android. They sit outside the main release of Android, and contain new Android features that developers can use in the apps they're building. The Support Libraries mean that you can give users on older devices the same experience as users on newer devices *even if they're using different versions of Android*.

Here are some of the Support Libraries that are available for you to use:

v4 Support Library

Includes the largest set of features, such as support for application components and user interface features.

v7 AppCompat Library

Includes support for app bars.

v7 Cardview Library

Adds support for the CardView widget, allowing you to show information inside cards.



These are just some of the Support Libraries.

Constraint Layout Library

Allows you to create constraint layouts. You used features from this library in Chapter 6.

v7 RecyclerView Library

Adds support for the RecyclerView widget.

Design Support Library

Adds support for extra components such as tabs and navigation drawers.

Each library includes a specific set of features.

The v7 AppCompat Library contains a set of up-to-date themes that can be used with older versions of Android: in practice, they can be used with nearly all devices, as most people are using API level 19 or above. We're going to use the v7 AppCompat Library by applying one of the themes it contains to our app. This will add an app bar that will look up-to-date and work the same on all versions of Android that we're targeting. Whenever you want to use one of the Support Libraries, you first need to add it to your app. We'll look at how you do this after we've created the project.

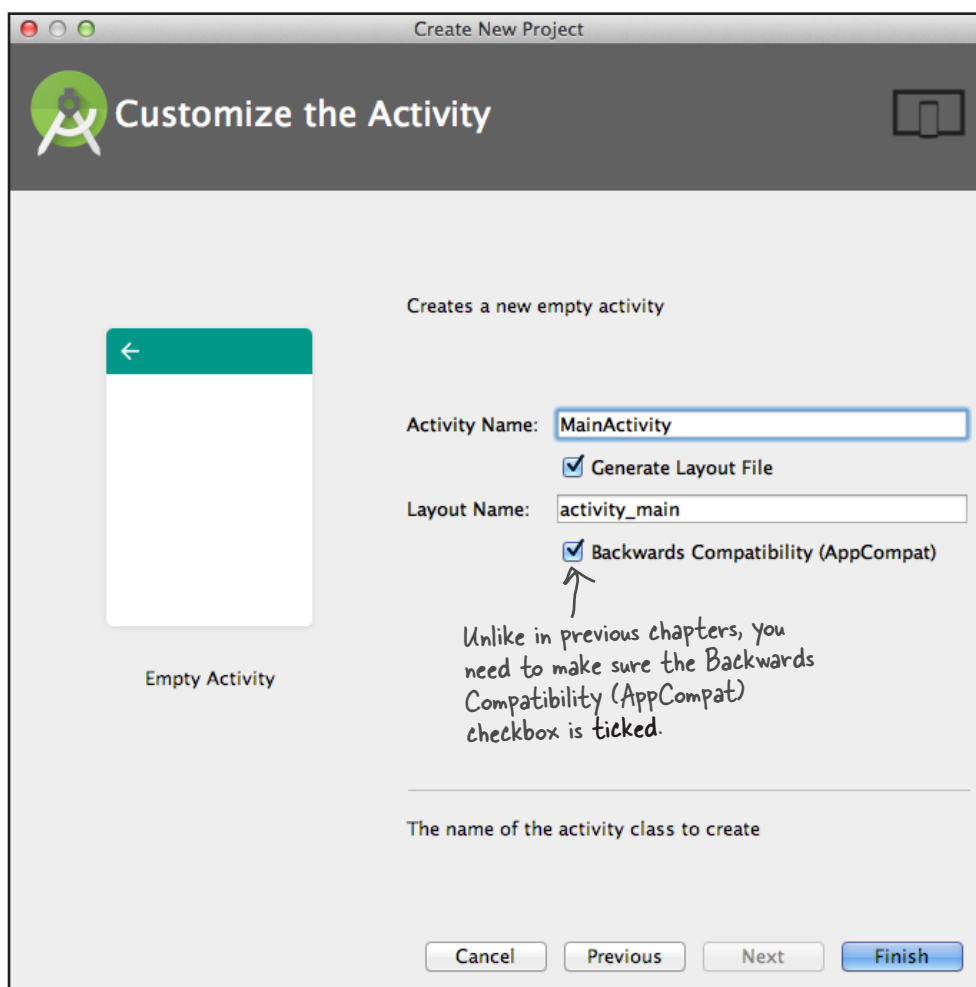
Create the Pizza app

We'll start by creating a prototype of the Pizza app. Create a new Android project for an application named "Bits and Pizzas" with a company domain of "hfad.com", making the package name `com.hfad.bitsandpizzas`. The minimum SDK should be API level 19 so that it works with most devices. You'll need an empty activity called "MainActivity" and a layout called "activity_main". Make sure you **check the Backwards Compatibility (AppCompat) checkbox** (you'll see why a few pages ahead).

support libraries and app bars



- Basic app bar
- Toolbar
- Action
- Up button
- Share action

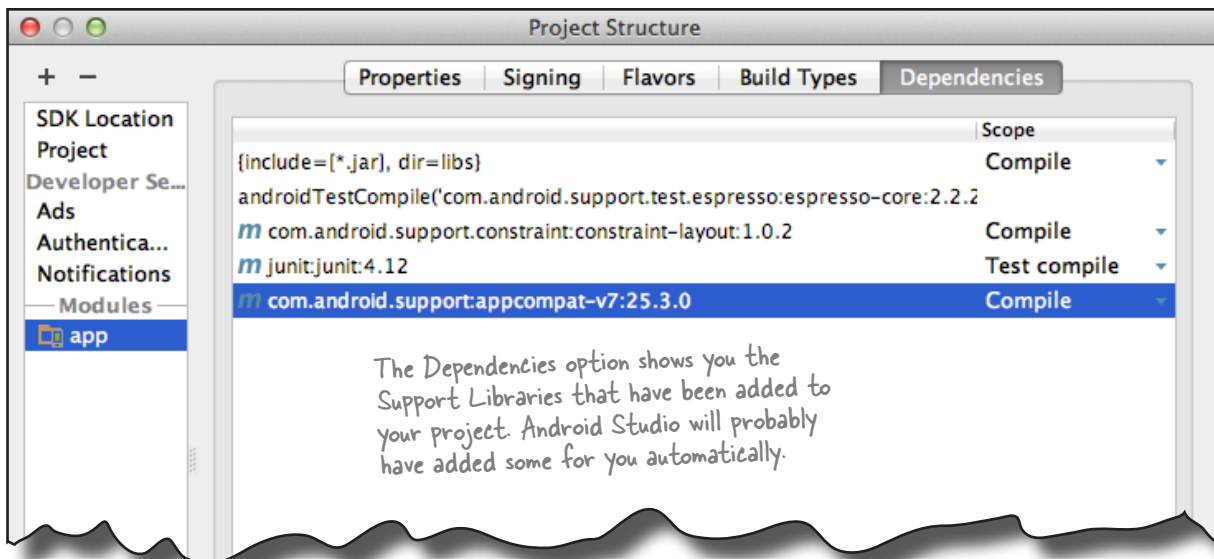
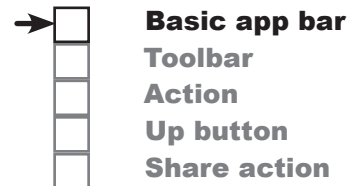


Next, we'll look at how you add a Support Library to the project.

Add the v7 AppCompat Support Library

We're going to use one of the themes from the v7 AppCompat Library, so we need to add the library to our project as a dependency. Doing so means that the library gets included in your app, and downloaded to the user's device.

To manage the Support Library files that are included in your project, choose File→Project Structure. Then click on the app module and choose Dependencies. You'll be presented with the following screen:



Android Studio may have already added the AppCompat Support Library for you automatically. If so, you will see it listed as appcompat-v7, as shown above.

If the AppCompat Library hasn't been added for you, you will need to add it yourself. Click on the "+" button at the bottom or right side of the Project Structure screen. Choose the Library Dependency option, select the appcompat-v7 library, then click on the OK button. Click on OK again to save your changes and close the Project Structure window.

Once the AppCompat Support Library has been added to your project, you can use its resources in your app. In our case, we want to apply one of its themes in order to give MainActivity an app bar. Before we do that, however, we need to look at the type of activity we're using for MainActivity.

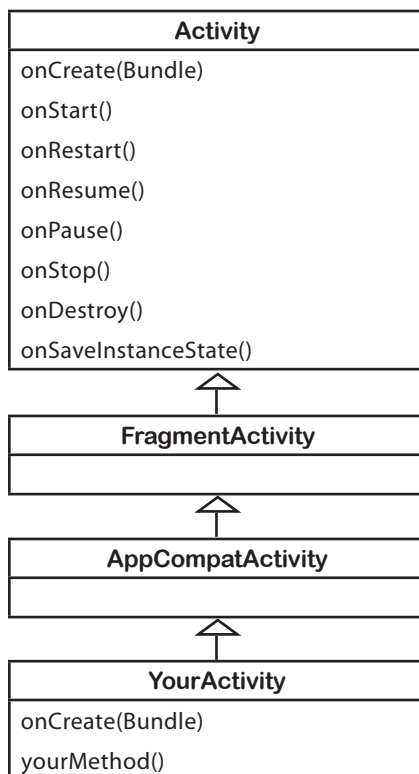
AppCompatActivity lets you use AppCompatActivity themes

So far, all of the activities we've created have extended the `Activity` class. This is the base class for all activities, and it's what makes your activity an activity. If you want to use the AppCompatActivity themes, however, you need to use a special kind of activity, called an **AppCompatActivity**, instead.

The AppCompatActivity class is a subclass of `Activity`. It lives in the AppCompatActivity Support Library, and it's designed to work with the AppCompatActivity themes. **Your activity needs to extend the AppCompatActivity class instead of the Activity class whenever you want an app bar that provides backward compatibility with older versions of Android.**

As AppCompatActivity is a subclass of the `Activity` class, everything you've learned about activities so far still applies. AppCompatActivity works with layouts in just the same way, and inherits all the lifecycle methods from the `Activity` class. The main difference is that, compared to `Activity`, AppCompatActivity contains extra smarts that allow it to work with the themes from the AppCompatActivity Support Library.

Here's a diagram showing the AppCompatActivity class hierarchy:



Activity class

(`android.app.Activity`)

The `Activity` class implements default versions of the lifecycle methods.

FragmentActivity class

(`android.support.v4.app.FragmentActivity`)

The base class for activities that need to use support fragments. You'll find out about fragments in the next chapter.

AppCompatActivity class

(`android.support.v7.app.AppCompatActivity`)

The base class for activities that use the Support Library app bar.

YourActivity class

(`com.hfad.foo`)

We'll make sure `MainActivity` extends `AppCompatActivity` on the next page.

MainActivity needs to be an AppCompatActivity

We want to use one of the AppCompatActivity themes, so we need to make sure our activities extend the AppCompatActivity class instead of the Activity class. Happily, this should already be the case if you checked the Backwards Compatibility (AppCompat) checkbox when you first created the activity. Open the file *MainActivity.java*, then make sure your code matches ours below:

```
package com.hfad.bitsandpizzas;

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

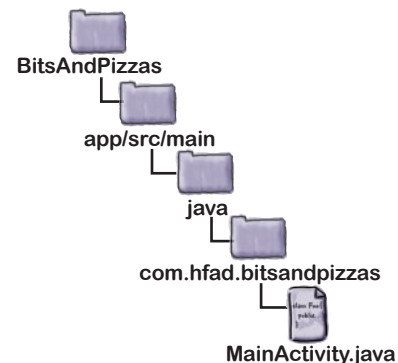
public class MainActivity extends AppCompatActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }
}
```

The AppCompatActivity class lives in the v7 AppCompat Support Library.

Make sure your activity extends AppCompatActivity.



Basic app bar
Toolbar
Action
Up button
Share action



Now that we've confirmed that our activity extends AppCompatActivity, we can add an app bar by applying a theme from the AppCompatActivity Support Library. You apply a theme in the app's *AndroidManifest.xml* file, so we'll look at this file next.

there are no Dumb Questions

Q: What versions of Android can the Support Libraries be used with?

A: It depends on the version of the Support Library. Prior to version 24.2.0, Libraries prefixed with v4 could be used with API level 4 and above, and those prefixed with v7 could be used with API level 7 and above. When version 24.2.0 of the Support Libraries was released, the minimum API for all Support Libraries became API level 9. The minimum API level is likely to increase in the future.

Q: In earlier chapters, Android Studio gave me activities that already extended AppCompatActivity. Why's that?

A: When you create an activity in Android Studio, the wizard includes a checkbox asking if you want to create a Backwards Compatible (AppCompat) activity. If you left this checked in earlier chapters, Android Studio would have generated activities that extend AppCompatActivity.

Q: I've seen code that extends ActionBarActivity. What's that?

A: In older versions of the AppCompatActivity Support Library, you used the ActionBarActivity class to add app bars. This was deprecated in version 22.1 in favor of AppCompatActivity.

AndroidManifest.xml can change your app bar's appearance

As you've seen earlier in the book, an app's *AndroidManifest.xml* file provides essential information about the app, such as what activities it contains. It also includes a number of attributes that have a direct impact on your app bars.

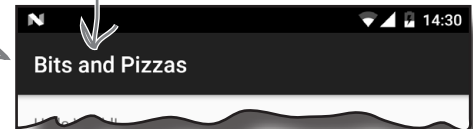
Here's the *AndroidManifest.xml* code Android Studio created for us (we've highlighted the key areas):

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.hfad.bitsandpizzas">
    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:label="@string/app_name"
        android:supportsRtl="true"
        android:theme="@style/AppTheme">
        <activity android:name=".MainActivity">
            ...
        </activity>
    </application>
</manifest>
```

App icons. Android Studio provides icons by default.

The user-friendly name of the app

The theme



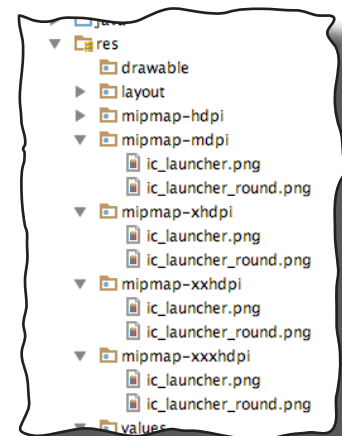
The **android:icon** attribute assigns an icon to the app. The icon is used as the launcher icon for the app, and if the theme you're using displays an icon in the app bar, it will use this icon. **android:roundIcon** may be used instead on devices running Android 7.1 or above.

The icon is a **mipmap** resource. A mipmap is an image that can be used for application icons, and they're held in *mipmap** folders in *app/src/main/res*. Just as with drawables, you can add different images for different screen densities by adding them to an appropriately named *mipmap* folder. As an example, an icon in the *mipmap-hdpi* folder will be used by devices with high-density screens. You refer to mipmap resources in your layout using @mipmap.

The **android:label** attribute describes a user-friendly label that gets displayed in the app bar. In the code above, it's used in the `<application>` tag to apply a label to the entire app. You can also add it to the `<activity>` tag to assign a label to a single activity.

The **android:theme** attribute specifies the theme. Using this attribute in the `<application>` element applies the theme to the entire app. Using it in the `<activity>` element applies the theme to a single activity.

We'll look at how you apply the theme on the next page.



Android Studio automatically added icons to our *mipmap** folders when we created the project.

How to apply a theme

When you want to apply a theme to your app, you have two main options:

- ★ Hardcode the theme in *AndroidManifest.xml*.
- ★ Apply the theme using a style.

Let's look at these two approaches.

1. Hardcoding the theme

To hardcode the theme in *AndroidManifest.xml*, you update the `android:theme` attribute in the file to specify the name of the theme you want to use. As an example, to apply a theme with a light background and a dark app bar, you'd use:

```
<application
    ...
    android:theme="Theme.AppCompat.Light.DarkActionBar">
```

This approach works well if you want to apply a basic theme without making any changes to it.

↑ This is a simple way of applying a basic theme, but it means you can't, for example, change its colors.

2. Using a style to apply the theme

Most of the time, you'll want to apply the theme using a style, as this approach enables you to tweak the theme's appearance. You may want to override the theme's main colors to reflect your app's brand, for example.

To apply a theme using a style, you update the `android:theme` attribute in *AndroidManifest.xml* to the name of a style resource (which you then need to create). In our case, we're going to use a style resource named `AppTheme`, so update the `android:theme` attribute in your version of *AndroidManifest.xml* to the following:

```
<application
    ...
    android:theme="@style/AppTheme">
```

↓ Android Studio may have already added this to your version of *AndroidManifest.xml*.



The `@style` prefix tells Android that the theme the app's using is a style that's defined in a **style resource file**. We'll look at this next.

Define styles in a style resource file

The style resource file holds details of any themes and styles you want to use in your app. When you create a project in Android Studio, the IDE will usually create a default style resource file for you called *styles.xml* located in the *app/src/main/res/values* folder.

If Android Studio hasn't created the file, you'll need to add it yourself. Switch to the Project view of Android Studio's explorer, highlight the *app/src/main/res/values* folder, go to the File menu, and choose New. Then choose the option to create a new Values resource file, and when prompted, name the file "styles". When you click on OK, Android Studio will create the file for you.

A basic style resource file looks like this:

```
<resources>
  <style name="AppTheme" parent="Theme.AppCompat.Light.DarkActionBar">
  </style>
</resources>
```

← There may be extra code here to customize the theme. We'll look at this a couple of pages ahead.

A style resource file can contain one or more styles. Each style is defined through the `<style>` element.

Each style must have a name, which you define with the name attribute; for example:

```
name="AppTheme"
```

In the code above, the style has a name of "AppTheme", and *AndroidManifest.xml* can refer to it using "@style/AppTheme".

The `parent` attribute specifies where the style should inherit its properties from; for example:

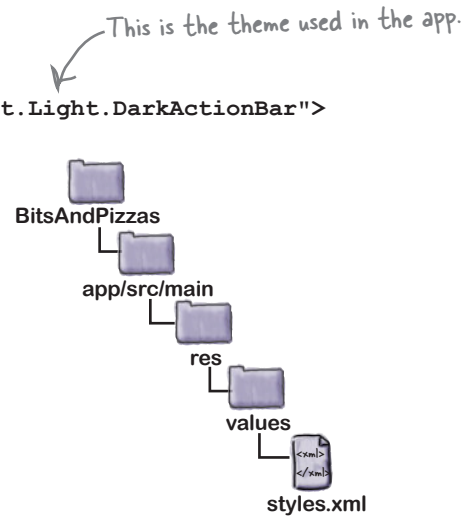
```
parent="Theme.AppCompat.Light.DarkActionBar"
```

This gives the app a theme of "Theme.AppCompat.Light.DarkActionBar", which gives activities a light background, with a dark app bar. We'll look at some more of Android's available themes on the next page.

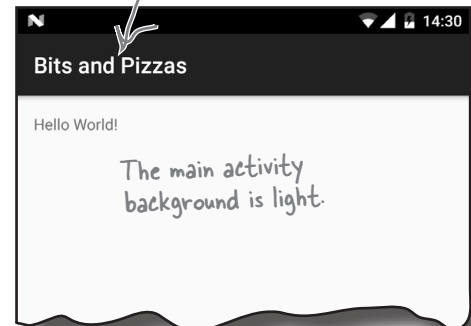
support libraries and app bars



- Basic app bar
- Toolbar
- Action
- Up button
- Share action

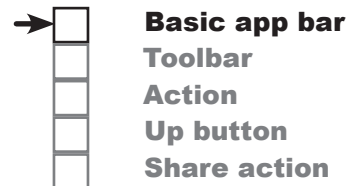


The app bar has a dark background with white text.

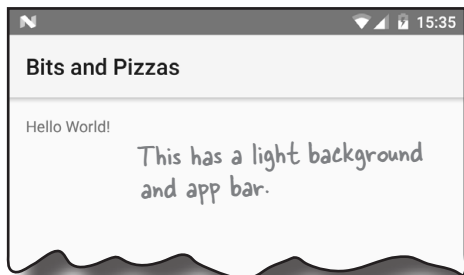


Theme gallery

Android comes with a whole bunch of built-in themes that you can use in your apps. Here are just a few of them:



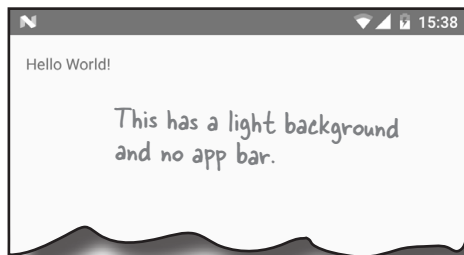
Theme.AppCompat.Light



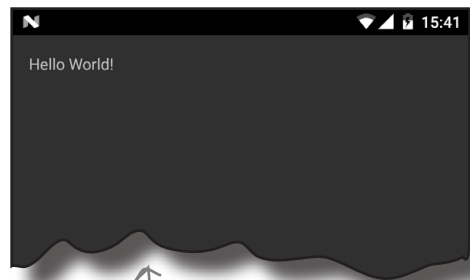
Theme.AppCompat



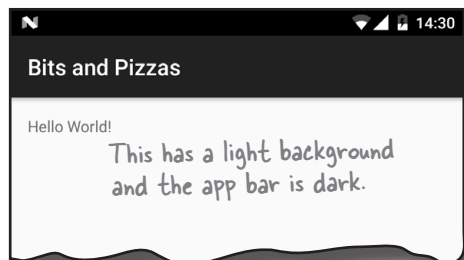
Theme.AppCompat.Light.NoActionBar



Theme.AppCompat.NoActionBar



Theme.AppCompat.Light.DarkActionBar



This has a dark background and no app bar.

There's also a DayNight theme, which uses one set of colors in the day, and another set at night.

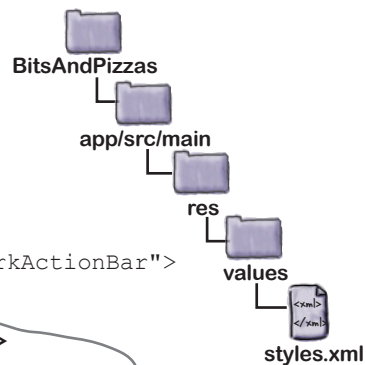
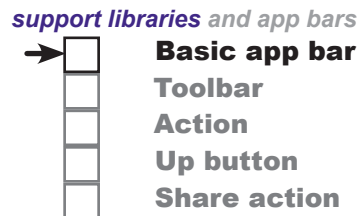
The theme determines the basic appearance of the app, such as the color of the app bar and any views. But what if you want to modify the app's appearance?

Customize the look of your app

You can use customize the look of your app by overriding the properties of an existing theme in the style resource file. For example, you can change the color of the app bar, the status bar, and any UI controls. You override the theme by adding `<item>` elements to the `<style>` to describe each modification you want to make.

We're going to override three of the colors used by our theme. To do this, make sure that your version of `styles.xml` matches ours below:

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



These three lines of code modify the theme by changing three of the colors.

The above code includes three modifications, each one described by a separate `<item>`. Each `<item>` has a name attribute that indicates what part of the theme you want to change, and a value that specifies what you want to change it to, like this:

```
<item name="colorPrimary">@color/colorPrimary</item>
```

This will change the `colorPrimary` part of the theme so it has a value of `@color/colorPrimary`.

name="colorPrimary" refers to the main color you want to use for your app. This color gets used for your app bar, and to "brand" your app with a particular color.

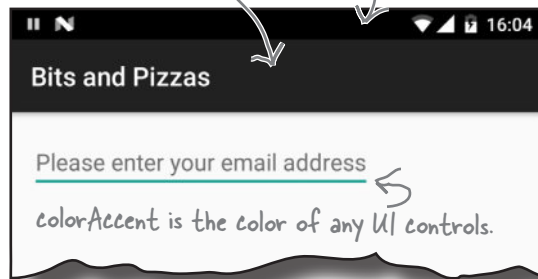
name="colorPrimaryDark" is a darker variant of your main color. It gets used as the color of the status bar.

name="colorAccent" refers to the color of any UI controls such as editable text views or checkboxes.

You set a new color for each of these areas by giving each `<item>` a value. The value can either be a hardcoded hexadecimal color value, or a reference to a color resource. We'll look at color resources on the next page.

`colorPrimary` is the color of the app bar.

`colorPrimaryDark` is the color of the status bar.



There are a whole host of other theme properties you can change, but we're not going to cover them here. To find out more, visit <https://developer.android.com/guide/topics/ui/themes.html>.

Define colors in a color resource file

A color resource file is similar to a String resource file except that it contains colors instead of Strings. Using a color resource file makes it easy to make changes to the color scheme of your app, as all the colors you want to use are held in one place.

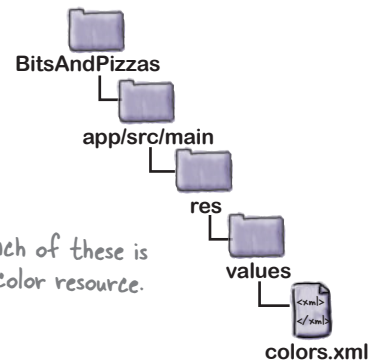
The color resource file is usually called `colors.xml`, and it's located in the `app/src/main/res/values` folder. When you create a project in Android Studio, the IDE will usually create this file for you.

If Android Studio hasn't created the file, you'll need to add it yourself. Switch to the Project view of Android Studio's explorer, highlight the `app/src/main/res/values` folder, go to the File menu, and choose New. Then choose the option to create a new Values resource file, and when prompted, name the file "colors". When you click on OK, Android Studio will create the file for you.

Next, open `colors.xml` and make sure that your version of the file matches ours below:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <color name="colorPrimary">#3F51B5</color>
  <color name="colorPrimaryDark">#303F9F</color>
  <color name="colorAccent">#FF4081</color>
</resources>
```

Each of these is a color resource.



The code above defines three color resources. Each one has a name and a value. The value is a hexadecimal color value:

This says it's a color resource.

```
<color name="colorPrimary">#3F51B5</color>
```

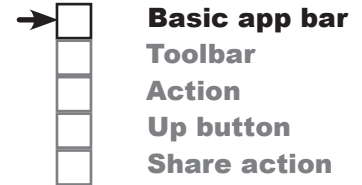
The color resource has a name of "colorPrimary", and a value of #3F51B5 (blue).

The style resource file looks up colors from the color resource file using `@color/colorName`. For example:

```
<item name="colorPrimary">@color/colorPrimary</item>
```

overrides the primary color used in the theme with the value of `colorPrimary` in the color resource file.

Now that we've seen how to add an app bar by applying a theme, let's update `MainActivity`'s layout and take the app for a test drive.



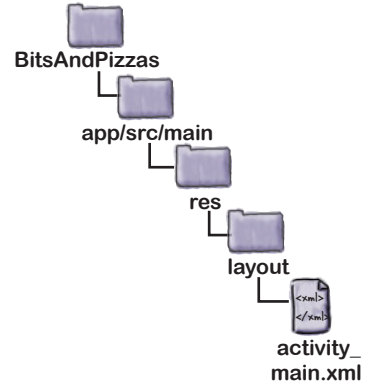
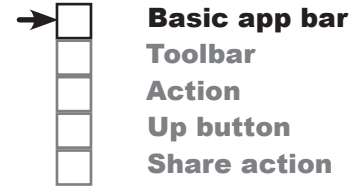
The code for activity_main.xml

For MainActivity's layout, we're going to display some default text in a linear layout. Here's the code to do that; update your version of *activity_main.xml* to match ours 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="com.hfad.bitsandpizzas.MainActivity">

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

support libraries and app bars



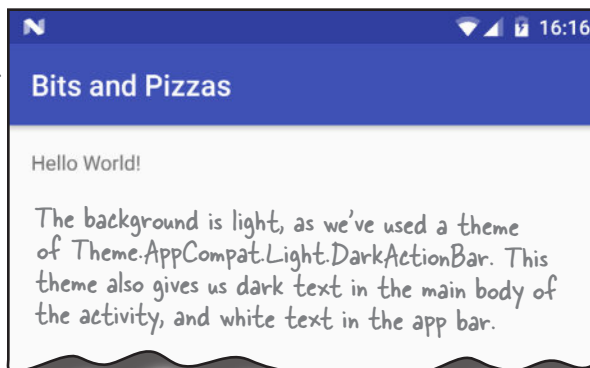
We're just displaying some basic placeholder text in MainActivity's layout because right now we want you to focus on app bars.



Test drive the app

When you run the app, MainActivity gets displayed. At the top of the activity there's an app bar.

This is the app bar. The default color's been overridden so that it's blue.



The status bar's default color has been overridden so it's a darker shade of blue than the app bar.

That's everything you need to apply a basic app bar in your activities. Why not experiment with changing the theme and colors? Then when you're ready, turn the page and we'll move on to the next step.

ActionBar vs. Toolbar

So far, you've seen how to add a basic app bar to the activities in your app by applying a theme that includes an app bar. Adding an app bar in this way is easy, but it has one disadvantage: *it doesn't necessarily include all the latest app bar features.*

Behind the scenes, any activity that acquires an app bar via a theme uses the `ActionBar` class for its app bar. The most recent app bar features, however, have been added to the `Toolbar` class in the AppCompat Support Library instead. This means that if you want to use the most recent app bar features in your app, you need to use the `Toolbar` class from the Support Library.

Using the `Toolbar` class also gives you more flexibility. A toolbar is a type of view that you add to your layout just as you would any other type of view, and this makes it much easier to position and control than a basic app bar.

How to add a toolbar

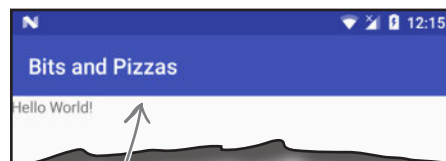
We're going to change our activity so that it uses a toolbar from the Support Library for its app bar. Whenever you want to use the `Toolbar` class from the Support Library, there are a number of steps you need to perform:

- 1 **Add the v7 AppCompat Support Library as a dependency.**
This is necessary because the `Toolbar` class lives in this library.
- 2 **Make sure your activity extends the `AppCompatActivity` class.**
Your activity must extend `AppCompatActivity` (or one of its subclasses) in order to use the Support Library toolbar.
- 3 **Remove the existing app bar.**
You do this by changing the theme to one that doesn't include an app bar.
- 4 **Add a toolbar to the layout.**
The toolbar is a type of view, so you can position it where you want and control its appearance.
- 5 **Update the activity to set the toolbar as the activity's app bar.**
This allows the activity to respond to the toolbar.

We'll go through these steps now.



Basic app bar
Toolbar
Action
Up button
Share action

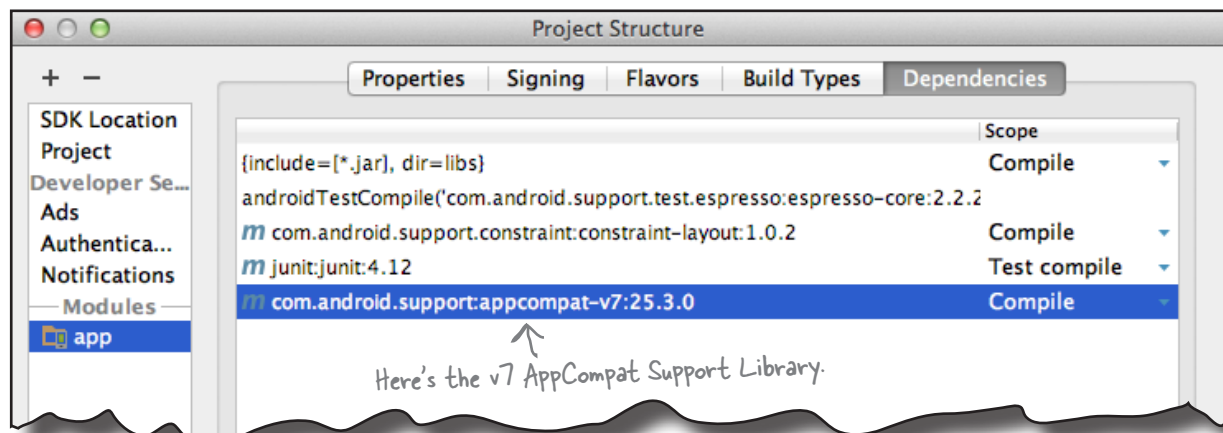


A toolbar looks just like the app bar you had previously, but it gives you more flexibility and includes the most recent app bar features.

1. Add the AppCompatActivity Support Library

Before you can use the `Toolbar` class from the Support Library in your activities, you need to make sure that the v7 AppCompatActivity Support Library has been added to your project as a dependency. In our particular case, the library has already been added to our project, as we needed it for the AppCompatActivity themes.

To double-check that the Support Library is there, in Android Studio choose `File`→`Project Structure`, click on the app module, and choose `Dependencies`. You should see the v7 AppCompatActivity Library listed as shown below:



2. Extend the AppCompatActivity class

When you want to use a theme from the AppCompatActivity Library, you have to make sure that your activities extend the AppCompatActivity class. This is also the case if you want to use a toolbar from the Support Library as your app bar.

We've already completed this step because, earlier in this chapter, we changed `MainActivity.java` to use AppCompatActivity:

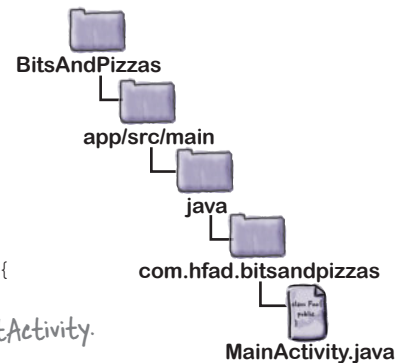
```

...
import android.support.v7.app.AppCompatActivity;

public class MainActivity extends AppCompatActivity {
    ...
}

```

Our MainActivity already extends AppCompatActivity.



The next thing we need to do is remove the existing app bar.

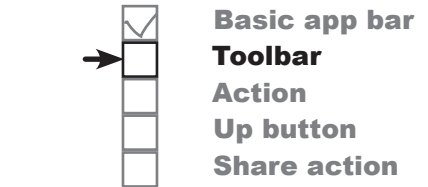
3. Remove the app bar

You remove the existing app bar in exactly the same way that you add one—by applying a **theme**.

When we wanted to add an app bar to our app, we applied a theme that displayed one. To do this, we used the `theme` attribute in *AndroidManifest.xml* to apply a style called `AppTheme`:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.hfad.bitsandpizzas">
    <application
        ...
        android:theme="@style/AppTheme">
        ...
    </application>
</manifest>
```

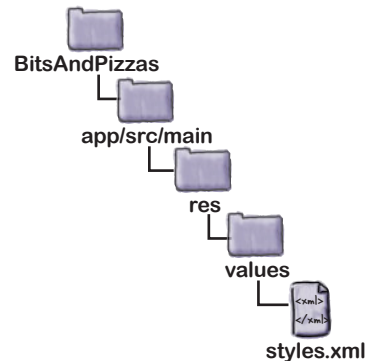
This looks up the theme from *styles.xml*.



The theme was then defined in *styles.xml* like this:

```
<resources>
    <style name="AppTheme" parent="Theme.AppCompat.Light.DarkActionBar">
        ...
    </style>
</resources>
```

This is the theme we're using. It displays a dark app bar.



The theme `Theme.AppCompat.Light.DarkActionBar` gives your activity a light background with a dark app bar. To remove the app bar, we're going to change the theme to `Theme.AppCompat.Light.NoActionBar` instead. Your activity will look the same as it did before except that no app bar will be displayed.

To change the theme, update *styles.xml* like this:

```
<resources>
    <style name="AppTheme" parent="Theme.AppCompat.Light.NoActionBar">
        ...
    </style>
</resources>
```

We customized the theme by overriding some of the colors. You can leave this code in place.

Change the theme from `DarkActionBar` to `NoActionBar`. This removes the app bar.

Now that we've removed the current app bar, we can add the toolbar.

4. Add a toolbar to the layout

As we said earlier, a toolbar is a view that you add to your layout. Toolbar code looks like this:



```
<android.support.v7.widget.Toolbar ← This defines the toolbar.  
    android:id="@+id/toolbar" ← Give the toolbar an ID so you can refer to it in your activity code.  
    android:layout_width="match_parent" ← Set the toolbar's size.  
    android:layout_height="?attr/actionBarSize" ←  
    android:background="?attr/colorPrimary" ← These control the app bar's appearance.  
    android:theme="@style/ThemeOverlay.AppCompat.Dark.ActionBar" /> ←
```

You start by defining the toolbar using:

```
<android.support.v7.widget.Toolbar ← This is the full path of the Toolbar  
    ... /> class in the Support Library.
```

where `android.support.v7.widget.Toolbar` is the fully qualified path of the `Toolbar` class in the Support Library.

Once the toolbar has been defined, you then use other view attributes to give it an ID, and specify its appearance. As an example, to make the toolbar as wide as its parent and as tall as the default app bar size from the underlying theme, you'd use:

```
    android:layout_width="match_parent" ← The toolbar is as wide as its parent,  
    android:layout_height="?attr/actionBarSize" ← and as tall as the default app bar.
```

The `?attr` prefix means that you want to use an attribute from the current theme. In this particular case, `?attr/actionBarSize` is the height of an app bar that's specified in our theme.

You can also change your toolbar's appearance so that it has a similar appearance to the app bar that we had before. To do this, you can change the background color, and apply a **theme overlay** like this:

```
    android:background="?attr/colorPrimary" ← Make the toolbar's background the same  
    android:theme="@style/ThemeOverlay.AppCompat.Dark.ActionBar" ← color as the app bar we had previously.
```

A theme overlay is a special type of theme that alters the current theme by overwriting some of its attributes. We want our toolbar to look like our app bar did when we used a theme of `Theme.AppCompat.Light.DarkActionBar`, so we're using a theme overlay of `ThemeOverlay.AppCompat.Dark.ActionBar`.

↑ This gives the toolbar the same appearance as the app bar we had before. We have to use a theme overlay, as the `NoActionBar` theme doesn't style app bars in the same way as the `DarkActionBar` theme did.

On the next page we'll add the toolbar to the layout.

we're not doing this (in our app)

Add the toolbar to the layout...

If your app contains a single activity, you can add the toolbar to your layout just as you would any other view. Here is an example of the sort of code you would use in this situation (we're using a different approach, so don't update your layout with the code below):

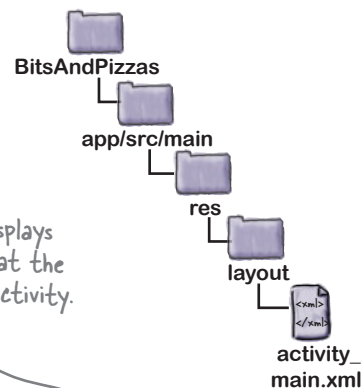
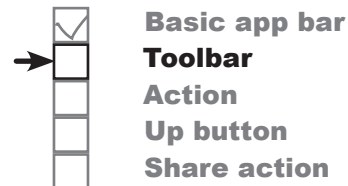
```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    tools:context="com.hfad.bitsandpizzas.MainActivity">
    <android.support.v7.widget.Toolbar
        android:id="@+id/toolbar"
        android:layout_width="match_parent"
        android:layout_height="?attr/actionBarSize"
        android:background="?attr/colorPrimary"
        android:theme="@style/ThemeOverlay.AppCompat.Dark.ActionBar" />
    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hello World!" />
</LinearLayout>
```

The code here doesn't include any padding. This is so that the toolbar fills the screen horizontally.

This code displays the toolbar at the top of the activity.

Later in the chapter we'll add a second activity to our app, so we're not using this approach. So you don't need to change your layout code to match this example.

We're using a linear layout, so the text view will be positioned below the toolbar.



This code displays the toolbar at the top of the activity. We've positioned the text view Android Studio gave us so that it's displayed underneath the toolbar. Remember that a toolbar is a view like any other view, so you need to take this into account when you're positioning your other views.

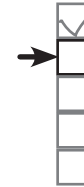
Adding the toolbar code to your layout works well if your app contains a single activity, as it means that all the code relating to your activity's appearance is in a single file. It works less well, however, if your app contains multiple activities. If you wanted to display a toolbar in multiple activities, you would need to define the toolbar in the layout of each activity. This means that if you wanted to change the style of the toolbar in some way, you'd need to edit *every single layout file*.

So what's the alternative?

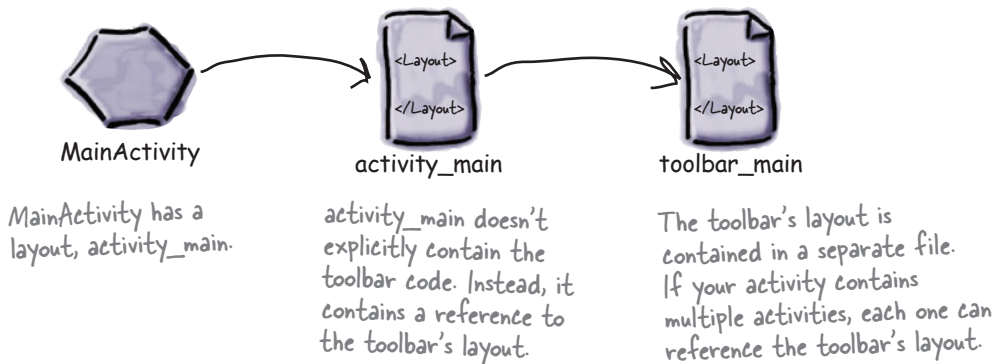
...or define the toolbar as a separate layout

An alternative approach is to define the toolbar in a separate layout, and then include the toolbar layout in each activity. This means that you only need to define the toolbar once, and if you want to change the style of your toolbar, you only need to edit one file.

support libraries and app bars



Basic app bar
 Toolbar
 Action
 Up button
 Share action



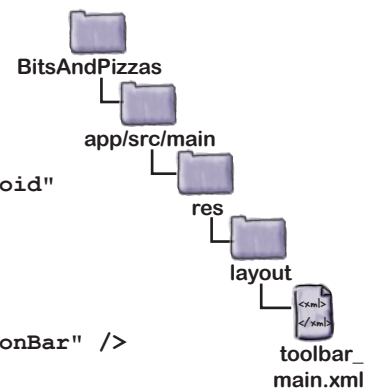
We're going to use this approach in our app. Start by creating a new layout file. Switch to the Project view of Android Studio's explorer, highlight the *app/src/res/main/layout* folder in Android Studio, then go to the File menu and choose New → Layout resource file. When prompted, give the layout file a name of "toolbar_main" and then click on OK. This creates a new layout file called *toolbar_main.xml*.

Next, open *toolbar_main.xml*, and replace any code Android Studio has created for you with the following:

```
<android.support.v7.widget.Toolbar
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="?attr/actionBarSize"
    android:background="?attr/colorPrimary"
    android:theme="@style/ThemeOverlay.AppCompat.Dark.ActionBar" />
```

This code is almost identical to the toolbar code you've already seen. The main difference is that we've left out the toolbar's *id* attribute, as we'll define this in the activity's main layout file *activity_main.xml* instead.

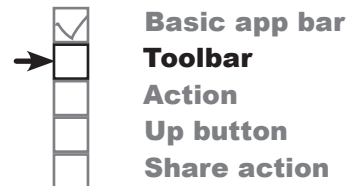
On the next page we'll look at how you include the toolbar layout in *activity_main.xml*.



This toolbar code goes in a separate layout file so multiple activities can reference it.

Include the toolbar in the activity's layout

You can display one layout inside another using the `<include>` tag. This tag must contain a `layout` attribute that specifies the name of the layout you want to include. As an example, here's how you would use the `<include>` tag to include the layout `toolbar_main.xml`:



```
<include
    layout="@layout/toolbar_main" />
```

The `@layout` tells Android to look for a layout called `toolbar_main`.

We want to include the `toolbar_main` layout in `activity_main.xml`. Here's our code; update your version of `activity_main.xml` to match ours:

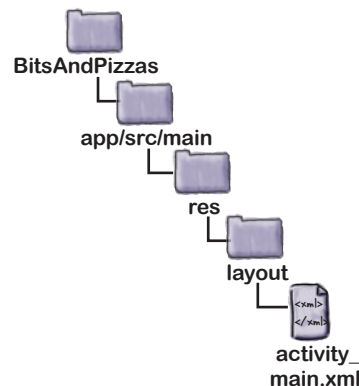
```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    android:padding="16dp"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

Remove the padding so that the toolbar fills the screen horizontally.

```
<include
    layout="@layout/toolbar_main"
    android:id="@+id/toolbar" />
<TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Hello World!" />
</LinearLayout>
```

Include the `toolbar_main` layout.

We're giving the toolbar an ID so we can refer to it in our activity code.



Now that we've added the toolbar to the layout, there's one more change we need to make.

5. Set the toolbar as the activity's app bar

The final thing we need to do is tell `MainActivity` to use the toolbar as its app bar.

So far we've only added the toolbar to the layout. While this means that the toolbar gets displayed at the top of the screen, the toolbar doesn't yet have any app bar functionality. As an example, if you were to run the app at this point, you'd find that the title of the app isn't displayed in the toolbar as it was in the app bar we had previously.

To get the toolbar to behave like an app bar, we need to call the `AppCompatActivity`'s `setSupportActionBar()` method in the activity's `onCreate()` method, which takes one parameter: the toolbar you want to set as the activity's app bar.

Here's the code for `MainActivity.java`; update your code to match ours:

```
package com.hfad.bitsandpizzas;

import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.support.v7.widget.Toolbar;

public class MainActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        Toolbar toolbar = (Toolbar) findViewById(R.id.toolbar);
        setSupportActionBar(toolbar);
    }
}
```

We need to use `setSupportActionBar()`, as we're using the toolbar from the Support Library.

We're using the `Toolbar` class, so we need to import it.

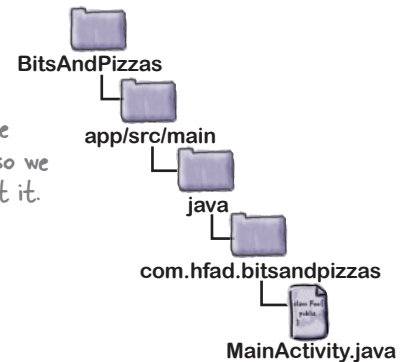
Get a reference to the toolbar, and set it as the activity's app bar.

support libraries and app bars



Basic app bar
Toolbar
Action
Up button
Share action

If you don't update your activity code after adding a toolbar to your layout, your toolbar will just appear as a plain strip with nothing in it.



That's all the code that you need to replace the activity's basic app bar with a toolbar. Let's see how it looks.

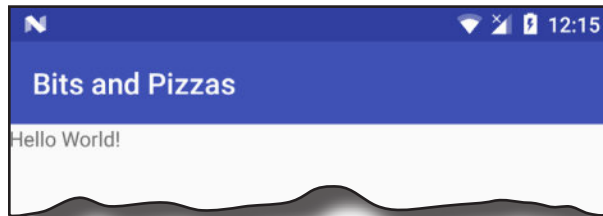


Test drive the app

When you run the app, a new toolbar is displayed in place of the basic app bar we had before. It looks similar to the app bar, but as it's based on the Support Library `Toolbar` class, it includes all the latest Android app bar functionality.



Basic app bar
Toolbar
Action
Up button
Share action



← Here's our new toolbar. It looks like the app bar we had before, but it gives you more flexibility.

You've seen how to add an app bar, and how to replace the basic app bar with a toolbar. Over the next few pages we'll look at how to add extra functionality to the app bar.

there are no Dumb Questions

Q: You've mentioned app bars, action bars, and toolbars. Is there a difference?

A: An app bar is the bar that usually appears at the top of your activities. It's sometimes called an action bar because in earlier versions of Android, the only way of implementing an app bar was via the `ActionBar` class.

The `ActionBar` class is used behind the scenes when you add an app bar by applying a theme. If your app doesn't rely on any new app bar features, this may be sufficient for your app.

An alternative way of adding an app bar is to implement a toolbar using the `Toolbar` class. The result looks similar to the default theme-based app bar, but it includes newer features of Android.

Q: I've added a toolbar to my activity, but when I run the app, it just looks like a band across the top of the screen. It doesn't even include the app name. Why's that?

A: First, check `AndroidManifest.xml` and make sure that your app has been given a label. This is where the app bar gets the app's name from.

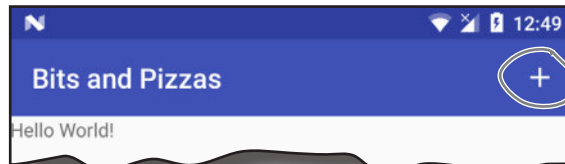
Also, check that your activity calls the `setSupportActionBar()` method in its `onCreate()` method, as this sets the toolbar as the activity's app bar. Without it, the name of the app or activity won't get displayed in the toolbar.

Q: I've seen the `<include>` tag in some of the code that Android Studio has created for me. What does it do?

A: The `<include>` tag is used to include one layout inside another. Depending on what version of Android Studio you're using and what type of project you create, Android Studio may split your layout code into one or more separate layouts.

Add actions to the app bar

In most of the apps you create, you'll probably want to add actions to the app bar. These are buttons or text in the app bar that you click on to make something happen. We're going to add a "Create Order" button to the app bar. When you click on it, it will start a new activity we'll create called `OrderActivity`:



We'll create a new `Create Order` action that will start `OrderActivity`.

Create `OrderActivity`

We'll start by creating `OrderActivity`. Select the `com.hfad.bitsandpizzas` package in the `app/src/main/java` folder, then go to `File` → `New...` → `Activity` → `Empty Activity`. Name the activity "`OrderActivity`", name the layout "`activity_order`", make sure the package name is `com.hfad.bitsandpizzas`, and **check the Backwards Compatibility (AppCompat) checkbox.**

If prompted for the activity's source language, select the option for Java.

Sharpen your pencil



We want `OrderActivity` to display the same toolbar as `MainActivity`. See if you can complete the code for `activity_order.xml` below to display the toolbar.

```
<?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="com.hfad.bitsandpizzas.OrderActivity">
    .....
    .....
    .....
</LinearLayout>
```

The code for adding the toolbar needs to go here.



We want `OrderActivity` to display the same toolbar as `MainActivity`. See if you can complete the code for `activity_order.xml` below to display the toolbar.

```
<?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="com.hfad.bitsandpizzas.OrderActivity">

    <include
        layout="@layout/toolbar_main"
        android:id="@+id/toolbar" />

</LinearLayout>
```

← This is the same code that we had in `MainActivity`. It includes the `toolbar_main` layout in `activity_order`.

Update `activity_order.xml`

We'll start by updating `activity_order.xml` so that it displays a toolbar. The toolbar will use the same layout we created earlier.

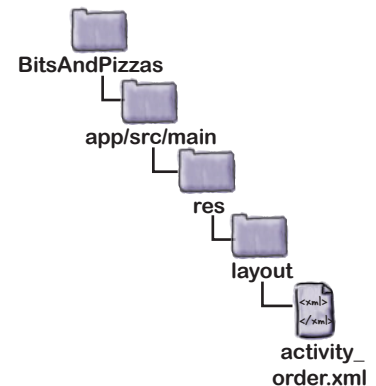
Here's our code; update yours so that it matches ours:

```
<?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="com.hfad.bitsandpizzas.OrderActivity">

    <include
        layout="@layout/toolbar_main"
        android:id="@+id/toolbar" />

</LinearLayout>
```

← Add the toolbar layout we created earlier.



Update OrderActivity.java

Next we'll update `OrderActivity` so that it uses the toolbar we set up in the layout as its app bar. To do this, we need to call the `setSupportActionBar()` method, passing in the toolbar as a parameter, just as we did before.

Here's the full code for `OrderActivity.java`; update your version of the code so that it matches ours:

```
package com.hfad.bitsandpizzas;

import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.support.v7.widget.Toolbar;

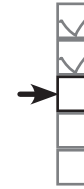
public class OrderActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_order);
        Toolbar toolbar = (Toolbar) findViewById(R.id.toolbar);
        setSupportActionBar(toolbar);
    }
}
```

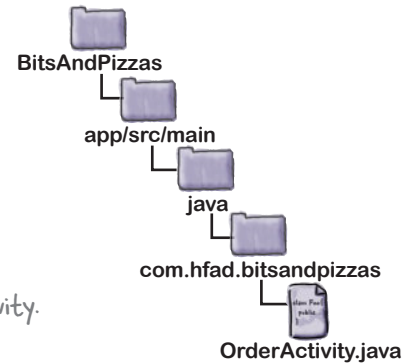
Make sure the activity extends AppCompatActivity.

Set the toolbar as the activity's app bar.

support libraries and app bars



- Basic app bar
- Toolbar
- Action
- Up button
- Share action



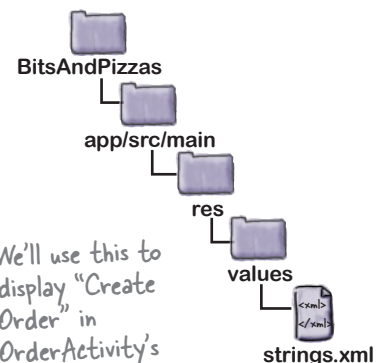
Add a String resource for the activity's title

Before we move on to creating an action to start `OrderActivity`, there's one more change we're going to make. We want to make it obvious to users when `OrderActivity` gets started, so we're going to change the text that's displayed in `OrderActivity`'s app bar to make it say "Create Order" rather than the name of the app.

To do this, we'll start by adding a String resource for the activity's title. Open the file `strings.xml` in the `app/src/main/res/values` folder, then add the following resource:

```
<string name="create_order">Create Order</string>
```

We'll use this to display "Create Order" in OrderActivity's app bar.



We'll update the text that gets displayed in the app bar on the next page.

Change the app bar text by adding a label

As you saw earlier in the chapter, you tell Android what text to display in the app bar by using the `label` attribute in file `AndroidManifest.xml`.

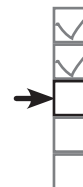
Here's our current code for `AndroidManifest.xml`. As you can see, the code includes a `label` attribute of `@string/app_name` inside the `<application>` element. This means that the name of the app gets displayed in the app bar for the entire app.

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.hfad.bitsandpizzas">
    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:label="@string/app_name"
        android:supportRtl="true"
        android:theme="@style/AppTheme">

        <activity android:name=".MainActivity">
            ...
        </activity>

        <activity android:name=".OrderActivity">
        </activity>

    </application>
</manifest>
```



Basic app bar
Toolbar
Action
Up button
Share action



← The label attribute tells Android what text to display in the app bar.

← This is the entry for MainActivity that we had before.

← This is the entry for OrderActivity. Android added this for us when we created the new activity.

We want to override the label for `OrderActivity` so that the text “Create Order” gets displayed in the app bar whenever `OrderActivity` has the focus. To do this, we’ll add a new `label` attribute to `OrderActivity`’s `<activity>` element to display the new text:

```
<activity
    android:name=".OrderActivity"
    android:label="@string/create_order">
</activity>
```

← Adding a label to an activity means that for this activity, the activity’s label gets displayed in its app bar instead of the app’s label.

We’ll show you this code in context on the next page.

The code for AndroidManifest.xml

Here's our code for *AndroidManifest.xml*. Update your code to reflect our changes.

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.hfad.bitsandpizzas">
    <application
        ...
        android:label="@string/app_name"
        ...>
        <activity android:name=".MainActivity">
            ...
        </activity>

        <activity android:name=".OrderActivity"
            android:label="@string/create_order">
        </activity>
    </application>
</manifest>
```

The app's label is the default label for the entire app.

We don't need to change the code for MainActivity. MainActivity has no label of its own, so it will use the label in the <application> element.

Adding a label to OrderActivity overrides the app's label for this activity. It means that different text gets displayed in the app bar.

support libraries and app bars



Basic app bar
Toolbar
Action
Up button
Share action



That's everything we need for *OrderActivity*. Next we'll look at how you add an action to the app bar so that we can start it.

How to add an action to an app bar

To add an action to the app bar, you need to do four things:

- 1 **Add resources for the action's icon and text.**
- 2 **Define the action in a menu resource file.**
This tells Android what actions you want on the app bar.
- 3 **Get the activity to add the menu resource to the app bar.**
You do this by implementing the `onCreateOptionsMenu()` method.
- 4 **Add code to say what the action should do when clicked.**
You do this by implementing the `onOptionsItemSelected()` method.

We'll start by adding the action's icon and text resources.

1. Add the action's resources

When you add an action to an app bar, you generally assign it an icon and a short text title. The icon usually gets displayed if the action appears in the main area of the app bar. If the action doesn't fit in the main area, it's automatically moved to the app bar overflow, and the title appears instead.

We'll start with the icon.

Add the icon

If you want to display your action as an icon, you can either create your own icon from scratch or use one of the icons provided by Google. You can find the Google icons here: <https://material.io/icons/>.

We're going to use the "add" icon `ic_add_white_24dp`, and we'll add a version of it to our project's `drawable*` folders, one for each screen density. Android will decide at runtime which version of the icon to use depending on the screen density of the device.

First, switch to the Project view of Android Studio's explorer if you haven't done so already, highlight the `app/src/main/res` folder, and then create folders called `drawable-hdpi`, `drawable-mdpi`, `drawable-xhdpi`, `drawable-xxhdpi`, and `drawable-xxxhdpi` if they're not already there. Then go to <https://git.io/v9oet>, and download the `ic_add_white_24dp.png` Bits and Pizzas images. Add the image in the `drawable-hdpi` folder to the `drawable-hdpi` folder in your project, then repeat this process for the other folders.

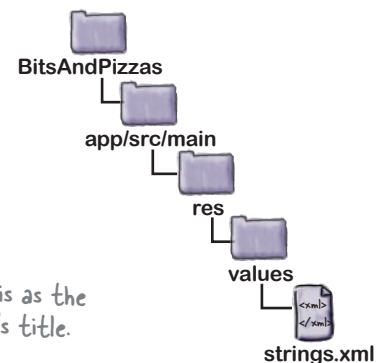
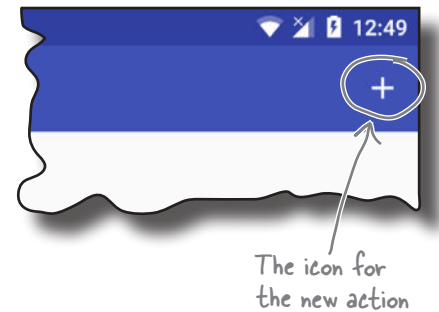
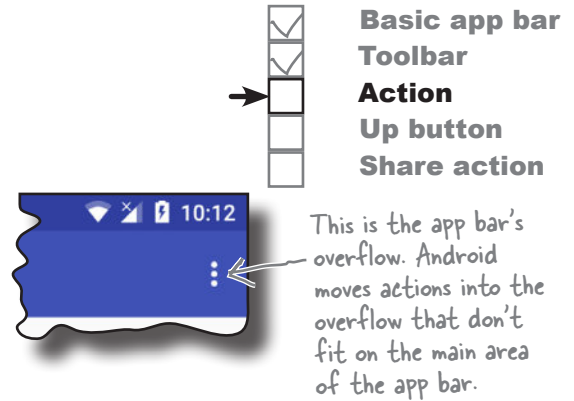
Add the action's title as a String resource

In addition to adding an icon for the action, we'll also add a title. This will get used if Android displays the action in the overflow area of the app bar, for example if there's no space for the action in the main area of the app bar.

We'll create the title as a String resource. Open the file `strings.xml` in the `app/src/main/res/values` folder, then add the following String resource:

```
<string name="create_order_title">Create Order</string>
```

Now that we've added resources for the action's icon and title, we can create the menu resource file.



We'll use this as the action item's title.

2. Create the menu resource file

A menu resource file tells Android what actions you want to appear on the app bar. Your app can contain multiple menu resource files. For example, you can create a separate menu resource file for each set of actions; this is useful if you want different activities to display different actions on their app bars.

We're going to create a new menu resource file called `menu_main.xml` in the folder `app/src/main/res/menu`. All menu resource files go in this folder.

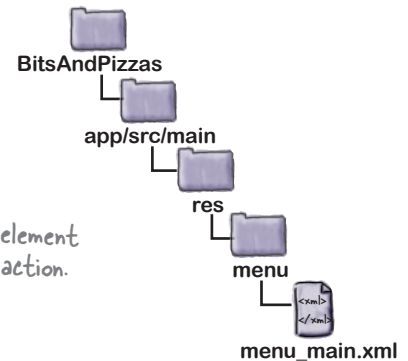
To create the menu resource file, select the `app/src/main/res` folder, go to the File menu, and choose New. Then choose the option to create a new Android resource file. You'll be prompted for the name of the resource file and the type of resource. Give it a name of "menu_main" and a resource type of "Menu", and make sure that the directory name is `menu`. When you click on OK, Android Studio will create the file for you, and add it to the `app/src/main/res/menu` folder.

Here's the code to add the new action. Replace the contents of `menu_main.xml` with the code below:

```
<?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/action_create_order"
        android:title="@string/create_order_title"
        android:icon="@drawable/ic_add_white_24dp"
        android:orderInCategory="1"
        app:showAsAction="ifRoom" />
</menu>
```

The `<menu>` element identifies the file as a menu resource file.

The `<item>` element defines the action.

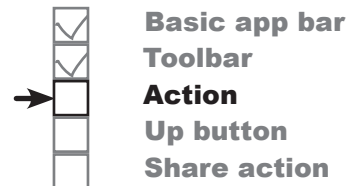


The menu resource file has a `<menu>` element at its root. Inside the `<menu>` element, you get a number of `<item>` elements, each one describing a separate action. In this particular case, we have a single action.

You use attributes of `<item>` to describe each action. The code creates an action with an id of `action_create_order`. This is so that we can refer to the action in our activity code, and respond to the user clicking on it.

The action includes a number of other attributes that determine how the action appears on the app bar, such as its icon and text. We'll look at these on the next page.

Control the action's appearance



Whenever you create an action to be displayed on the app bar, it's likely you'll want to display it as an icon. The icon can be any drawable resource. You set the icon using the `icon` attribute:

```
android:icon="@drawable/ic_add_white_24dp" ← This is the name of the drawable resource we want to use as the icon.
```

Sometimes Android can't display the action's icon. This may be because the action has no icon, or because the action is displayed in the app bar overflow instead of in the main area. For this reason, it's a good idea to set an action's title so that the action can display a short piece of text instead of an icon. You set the action's title using the `title` attribute:

```
android:title="@string/create_order_title" ← The title doesn't always get displayed, but it's a good idea to include it in case the action appears in the overflow.
```

If your app bar contains multiple actions, you might want to specify the order in which they appear. To do this, you use the `orderInCategory` attribute, which takes an integer value that reflects the action's order. Actions with a lower number will appear before actions with a higher number.

```
android:orderInCategory="1" ← An action with an orderInCategory of 1 will appear before an action with an orderInCategory of 10.
```

Finally, the `showAsAction` attribute is used to say how you want the item to appear in the app bar. As an example, you can use it to get an item to appear in the overflow area rather than the main part of the app bar, or to place an item on the main app bar only if there's room. The `showAsAction` attribute can take the following values:

"ifRoom"	Place the item in the app bar if there's space. If there's not space, put it in the overflow.
"withText"	Include the item's title text.
"never"	Put the item in the overflow area, and never in the main app bar.
"always"	Always place the item in the main area of the app bar. This value should be used sparingly; if you apply this to many items, they may overlap each other.

In our example, we want the action to appear on the main area of the app bar if there's room, so we're using:

```
app:showAsAction="ifRoom"
```

↑ There are other attributes for controlling an action's appearance, but these are the most common ones.

Our menu resource file is now complete. The next thing we need to do is implement the `onCreateOptionsMenu()` method in our activity.

3. Add the menu to the app bar with the onCreateOptionsMenu() method



Once you've created the menu resource file, you add the actions it contains to an activity's app bar by implementing the activity's `onCreateOptionsMenu()` method. This method runs when the app bar's menu gets created. It takes one parameter, a `Menu` object that's a Java representation of the menu resource file.

Here's our `onCreateOptionsMenu()` method for `MainActivity.java` (update your code to reflect our changes):

```
package com.hfad.bitsandpizzas;

import android.view.Menu; ← The onCreateOptionsMenu()
...                          method uses the Menu class.

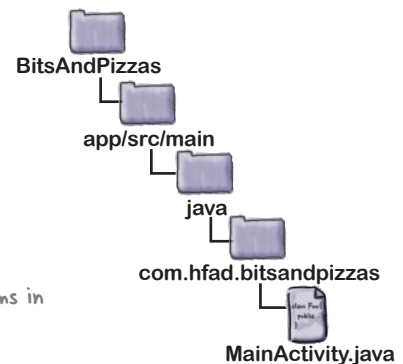
public class MainActivity extends AppCompatActivity {

    ...

    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
        // Inflate the menu; this adds items to the app bar.
        getMenuInflater().inflate(R.menu.menu_main, menu);
        return super.onCreateOptionsMenu(menu);
    }
}
```

Implementing this method adds any items in the menu resource file to the app bar.

All onCreateOptionsMenu() methods generally look like this.



The line:

```
getMenuInflater().inflate(R.menu.menu_main, menu);
```

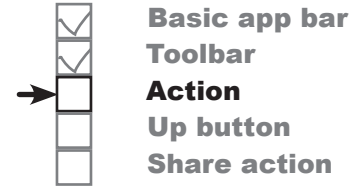
This is the menu resource file.

This is a Menu object that's a Java representation of the menu resource file.

inflates your menu resource file. This means that it creates a `Menu` object that's a Java representation of your menu resource file, and any actions the menu resource file contains are translated to `MenuItem`s. These are then added to the app bar.

There's one more thing we need to do: get our action to start `OrderActivity` when it's clicked. We'll do that on the next page.

4. React to action item clicks with the onOptionsItemSelected() method



To make your activity react when an action in the app bar is clicked, you implement the onOptionsItemSelected() method in your activity:

```

@Override
public boolean onOptionsItemSelected(MenuItem item) {
    switch (item.getItemId()) {
        ...
        default:
            return super.onOptionsItemSelected(item);
    }
}

```

The MenuItem object is the action on the app bar that was clicked.

Get the action's ID.

The onOptionsItemSelected() method runs whenever an action gets clicked. It takes one parameter, a MenuItem object that represents the action on the app bar that was clicked. You can use the MenuItem's getItemId() method to get the ID of the action so that you can perform an appropriate action, such as starting a new activity.

We want to start OrderActivity when our action is clicked. Here's the code for the onOptionsItemSelected() method that will do this:

```

@Override
public boolean onOptionsItemSelected(MenuItem item) {
    switch (item.getItemId()) {
        case R.id.action_create_order:
            //Code to run when the Create Order item is clicked
            Intent intent = new Intent(this, OrderActivity.class);
            startActivity(intent);
            return true;
        default:
            return super.onOptionsItemSelected(item);
    }
}

```

This intent is used to start OrderActivity when the Create Order action is clicked.

Returning true tells Android you've dealt with the item being clicked.

The full code for MainActivity.java is on the next page.

The full MainActivity.java code

Here's the full code for *MainActivity.java*. Update your code so that it matches ours. We've highlighted our changes.

```
package com.hfad.bitsandpizzas;

import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.support.v7.widget.Toolbar;
import android.view.Menu;
import android.view.MenuItem;
import android.content.Intent;

public class MainActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        Toolbar toolbar = (Toolbar) findViewById(R.id.toolbar);
        setSupportActionBar(toolbar);
    }

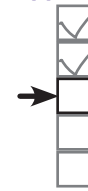
    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
        getMenuInflater().inflate(R.menu.menu_main, menu);
        return super.onCreateOptionsMenu(menu);
    }

    @Override
    public boolean onOptionsItemSelected(MenuItem item) {
        switch (item.getItemId()) {
            case R.id.action_create_order:
                Intent intent = new Intent(this, OrderActivity.class);
                startActivity(intent);
                return true;
            default:
                return super.onOptionsItemSelected(item);
        }
    }
}
```

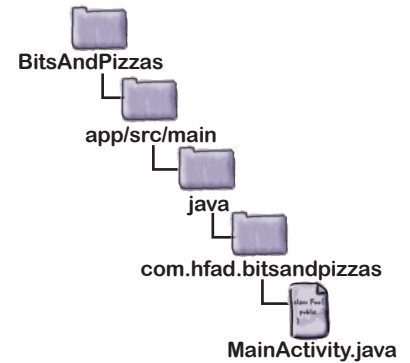
These classes are used by the `onOptionsItemSelected()` method so we need to import them.

This method gets called when an action on the app bar is clicked.

support libraries and app bars



Basic app bar
Toolbar
Action
Up button
Share action

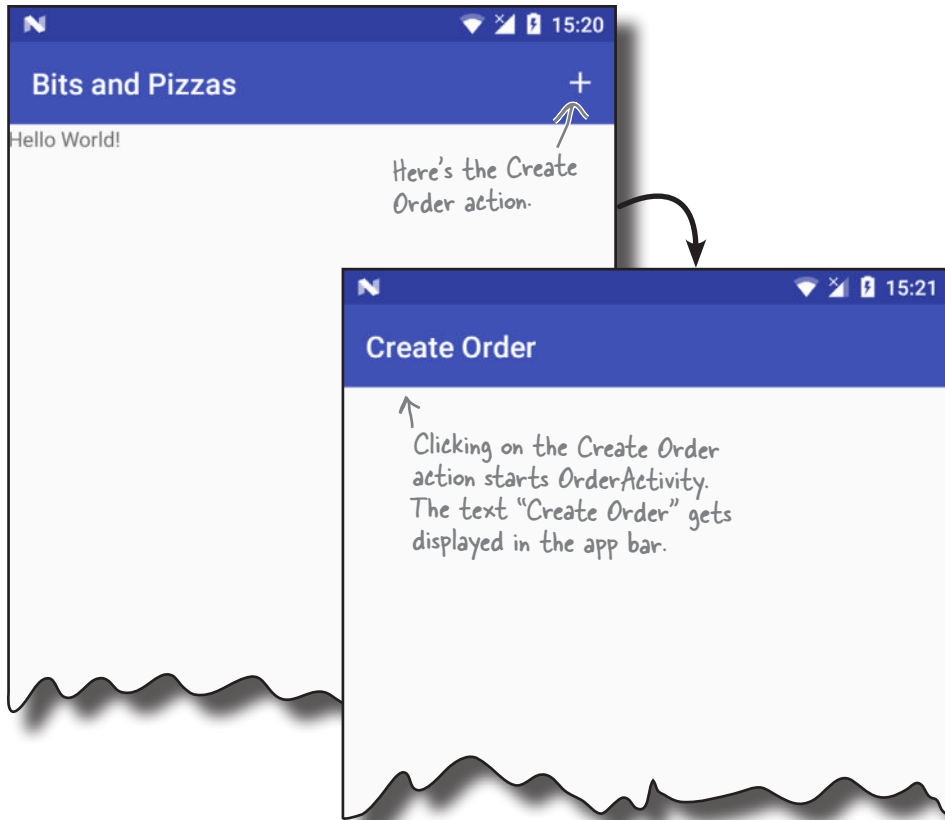
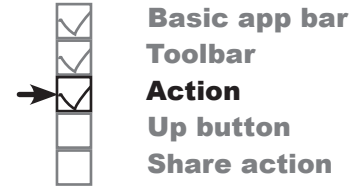


Let's see what happens when we run the app.



Test drive the app

When you run the app, a new Create Order action is displayed in the MainActivity app bar. When you click on the action item, it starts OrderActivity.



But how do we get back to MainActivity?

To return to MainActivity from OrderActivity, we currently need to click on the Back button on our device. But what if we want to get back to it from the app bar?

One option would be to add an action to OrderActivity's app bar that starts MainActivity, but there's a better way. We can get OrderActivity to return to MainActivity by enabling the Up button on OrderActivity's app bar.

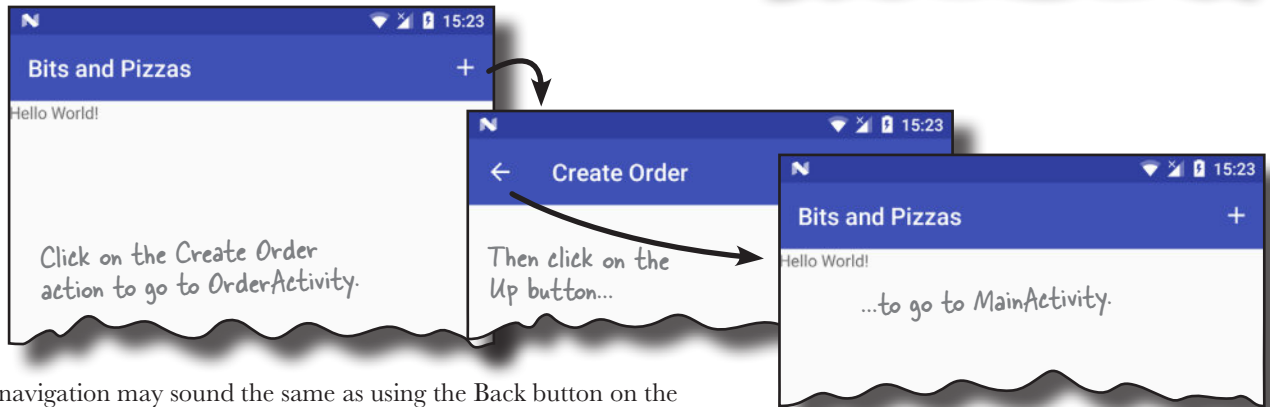
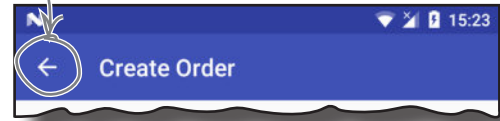
Enable Up navigation

If you have an app that contains a hierarchy of activities, you can enable the Up button on the app bar to let users navigate through the app using hierarchical relationships. As an example, MainActivity in our app includes an action on its app bar that starts a second activity, OrderActivity. If we enable the Up button on OrderActivity's app bar, the user will be able to return to MainActivity by clicking on this button.

support libraries and app bars

- Basic app bar
- Toolbar
- Action
- Up button
- Share action

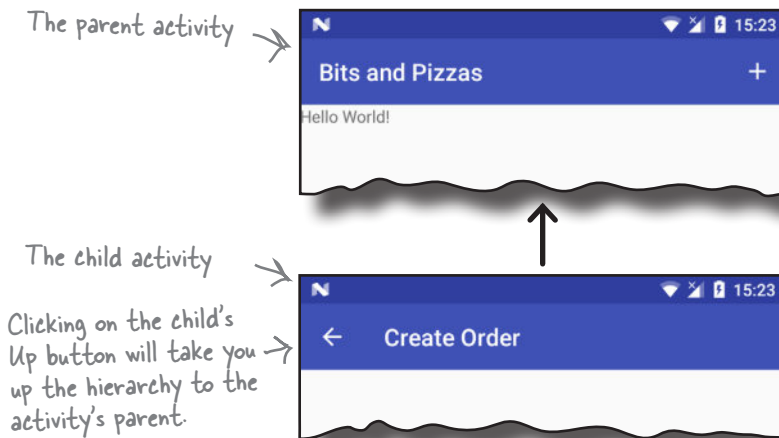
This is the Up button.



Up navigation may sound the same as using the Back button on the device, but it's different. The Back button allows users to work their way back through the history of activities they've been to. The Up button, on the other hand, is purely based on the app's hierarchical structure. If your app contains a lot of activities, implementing the Up button gives your users a quick and easy way to return to an activity's parent without having to keep pressing the Back button.

Use the Back button to navigate back to the previous activity.

Use the Up button to navigate up the app's hierarchy.



We're going to enable the Up button on OrderActivity's app bar. When you click on it, it will display MainActivity.

Set an activity's parent

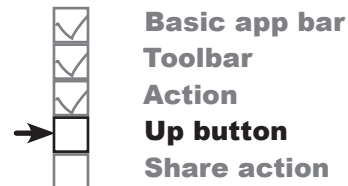
The Up button enables the user to navigate up a hierarchy of activities in the app. You declare this hierarchy in *AndroidManifest.xml* by specifying the parent of each activity. As an example, we want the user to be able to navigate from *OrderActivity* to *MainActivity* when they press the Up button, so this means that *MainActivity* is the parent of *OrderActivity*.

For API level 16 and above, you specify the parent activity using the `android:parentActivityName` attribute. For older versions of Android, you need to include a `<meta-data>` element that includes the name of the parent activity. Here are both approaches in our *AndroidManifest.xml*:

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

    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:label="@string/app_name"
        android:supportsRtl="true"
        android:theme="@style/AppTheme">
        <activity android:name=".MainActivity">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
        <activity
            android:name=".OrderActivity"
            android:label="@string/create_order"
            android:parentActivityName=".MainActivity">
            <meta-data
                android:name="android.support.PARENT_ACTIVITY"
                android:value=".MainActivity" />
        </activity>
    </application>

</manifest>
```



Apps at API level 16 or above use this line. It says that *OrderActivity*'s parent is *MainActivity*.

You only need to add the `<meta-data>` element if you're supporting apps below API level 16. We've only included it so you can see what it looks like, but including it in your code won't do any harm.

Finally, we need to enable the Up button in *OrderActivity*.

Adding the Up button

You enable the Up button from within your activity code. You first get a reference to the app bar using the activity's `getSupportActionBar()` method. This returns an object of type `ActionBar`. You then call the `ActionBar` `setDisplayHomeAsUpEnabled()` method, passing it a value of `true`.

```
ActionBar actionBar = getSupportActionBar();
actionBar.setDisplayHomeAsUpEnabled(true);
```

We want to enable the Up button in `OrderActivity`, so we'll add the above code to the `onCreate()` method in `OrderActivity.java`. Here's our full activity code:

```
package com.hfad.bitsandpizzas;

import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.support.v7.widget.Toolbar;
import android.support.v7.app.ActionBar;

public class OrderActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_order);
        Toolbar toolbar = (Toolbar) findViewById(R.id.toolbar);
        setSupportActionBar(toolbar);
        ActionBar actionBar = getSupportActionBar();
        actionBar.setDisplayHomeAsUpEnabled(true);
    }
}
```

This enables the Up button. Even though we're using a toolbar for our app bar, we need to use the `ActionBar` class for this method.

That's all the changes we need to make to enable the Up button. Let's see what happens when we run the app.

support libraries and app bars



Basic app bar
Toolbar
Action
Up button
Share action

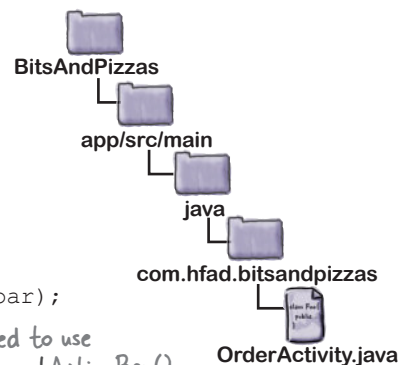


Watch it!

If you enable the Up button for an activity, you **MUST** specify its parent.

If you don't, you'll get a null pointer exception when you call the `setDisplayHomeAsUpEnabled()` method.

We're using the `ActionBar` class, so we need to import it. It comes from the `AppCompatActivity` Support Library.



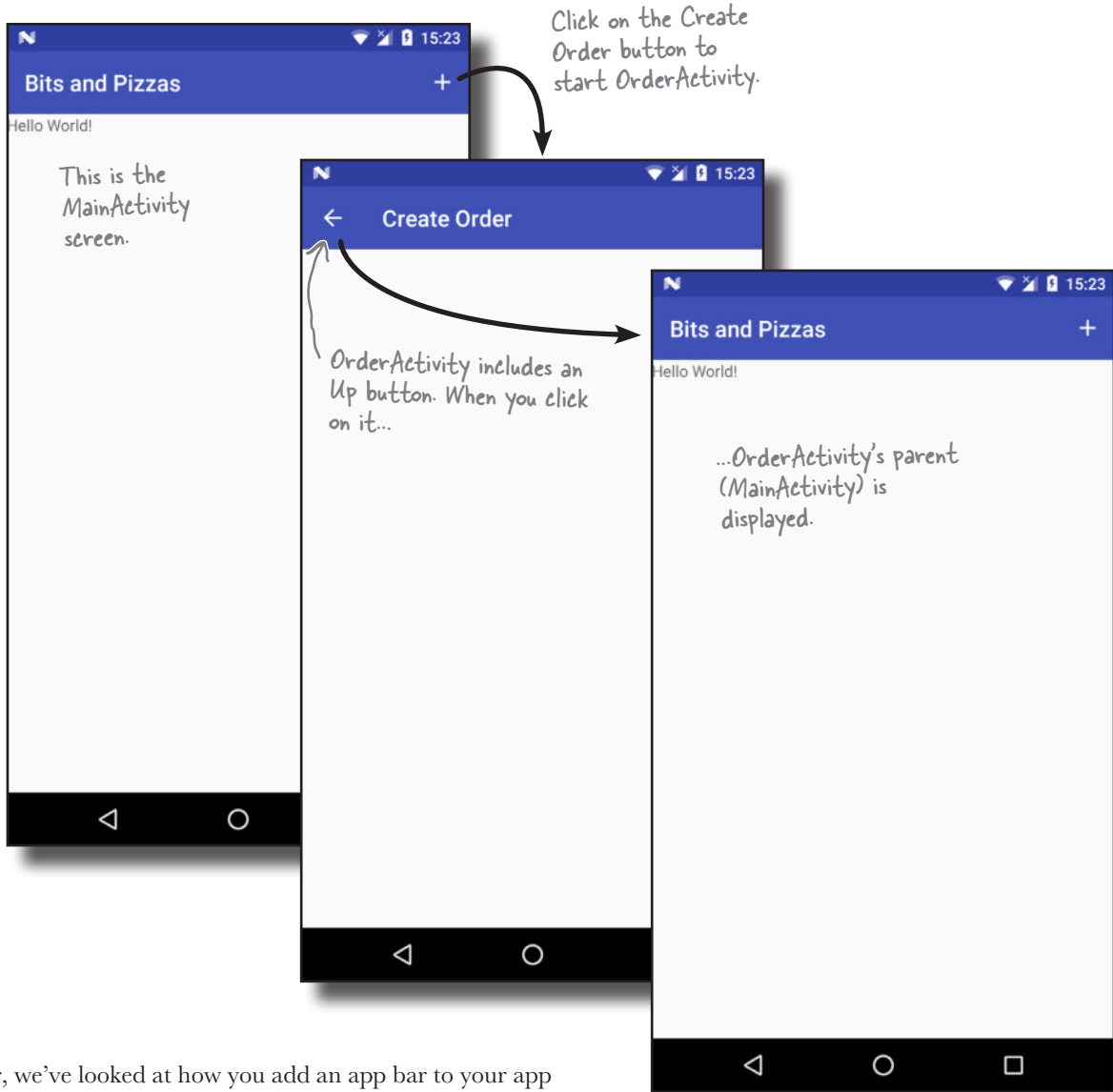
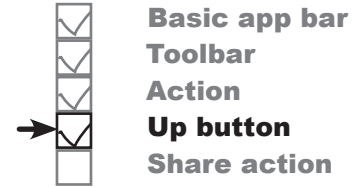
You need to use `getSupportActionBar()`, as we're using the toolbar from the Support Library.



Test drive the app

When you run your app and click on the Create Order action item, `OrderActivity` is displayed as before.

`OrderActivity` displays an Up button in its app bar. When you click on the Up button, it displays its hierarchical parent `MainActivity`.



So far, we've looked at how you add an app bar to your app and add basic actions to it. Next we'll look at how you add more sophisticated actions using **action providers**.

Sharing content on the app bar

The next thing we'll look at is how to add an action provider to your app bar. An action provider is an action that defines its own appearance and behavior.

We're going to concentrate on using the share action provider, which allows users to share content in your app with other apps such as Gmail. As an example, you could use it to let users send details of a particular pizza to one of their contacts.

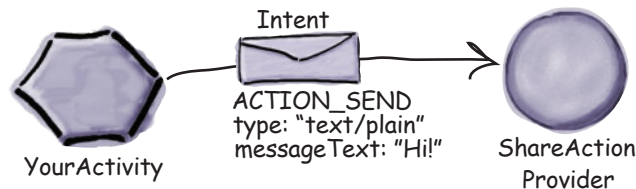
The share action provider defines its own icon, so you don't have to add it yourself. When you click on it, it provides you with a list of apps you can use to share content. It adds a separate icon for the most commonly used app you choose to share content with.

You share the content with an intent

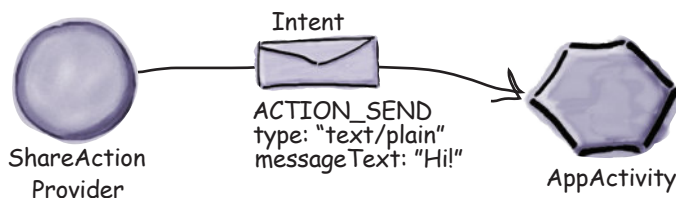
To get the share action provider to share content, you pass it an intent that defines the content you want to share, and its type. As an example, if you define an intent that passes text with an ACTION_SEND action, the share action will offer you a list of apps on your device that are capable of sharing text.

Here's how the share action works (you'll see this in action over the next two pages):

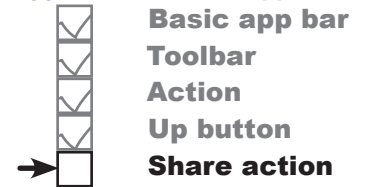
- 1 Your activity creates an intent and passes it to the share action provider.**
The intent describes the content that needs to be shared, its type, and an action.



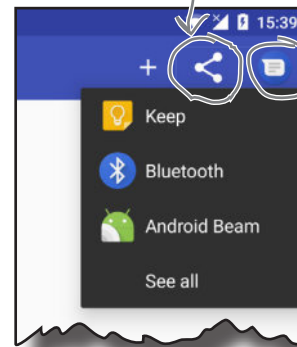
- 2 When the user clicks on the share action, the share action uses the intent to present the user with a list of apps that can deal with the intent.**
The user chooses an app, and the share action provider passes the intent to the app's activity that can handle it.



support libraries and app bars



This is what the share action looks like on the app bar. When you click on it, it gives you a list of apps that you can use to share content.



The share action also displays an icon for the app you most commonly share content with, in this case the Messenger app. This may not be visible at first.

Add a share action provider to menu_main.xml

You add a share action to the app bar by including it in the menu resource file.

To start, add a new `action_share` String to `strings.xml`. We'll use this String to add a title to the share action in case it appears in the overflow:

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

You add the share action to the menu resource file using the `<item>` element as before. This time, however, you need to specify that you're using a share action provider. You do this by adding an attribute of `app:actionProviderClass` and setting it to `android.support.v7.widget.ShareActionProvider`.

Here's the code to add the share action; update your copy of `menu_main.xml` to match ours:

```
<?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">

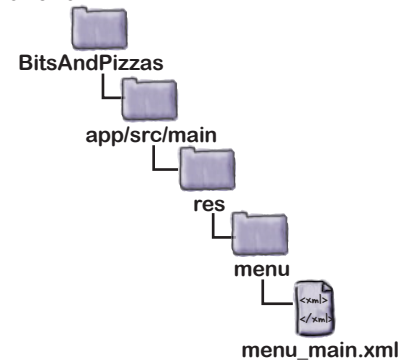
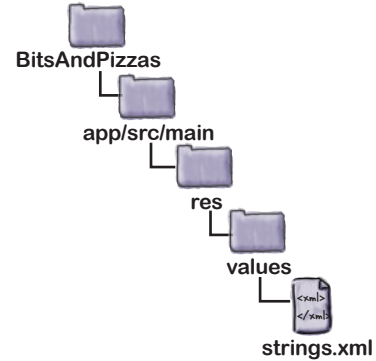
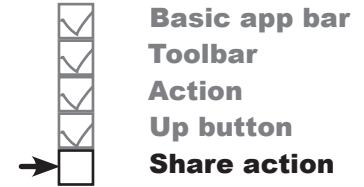
  <item android:id="@+id/action_create_order"
        android:title="@string/create_order_title"
        android:icon="@drawable/ic_add_white_24dp"
        android:orderInCategory="1"
        app:showAsAction="ifRoom" />

  <item android:id="@+id/action_share"
        android:title="@string/action_share"
        android:orderInCategory="2"
        app:showAsAction="ifRoom"
        app:actionProviderClass="android.support.v7.widget.ShareActionProvider" />

</menu>
```

As we mentioned earlier, when you add a share action to your menu resource file, there's no need to include an icon. The share action provider already defines one.

Now that we've added the share action to the app bar, let's specify what content to share.



Display the share action provider in the app bar if there's room.

This is the share action provider class. It comes from the AppCompatActivity class.

Specify the content with an intent

To get the share action to share content when it's clicked, you need to tell it what to share in your activity code. You do this by passing the share action provider an intent using its `setShareIntent()` method. Here's how you'd get the share action to share some default text when it's clicked:



```

package com.hfad.bitsandpizzas;
...
import android.support.v7.widget.ShareActionProvider;
import android.support.v4.view.MenuItemCompat;

public class MainActivity extends AppCompatActivity {

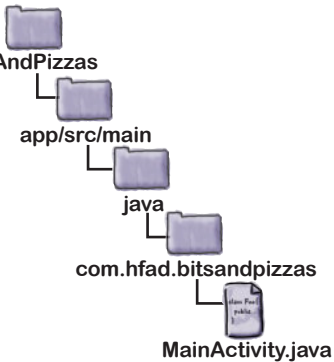
    private ShareActionProvider shareActionProvider;
    ...
    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
        getMenuInflater().inflate(R.menu.menu_main, menu);
        MenuItem menuItem = menu.findItem(R.id.action_share);
        shareActionProvider =
            (ShareActionProvider) MenuItemCompat.getActionProvider(menuItem);
        setShareActionIntent("Want to join me for pizza?");
        return super.onCreateOptionsMenu(menu);
    }

    private void setShareActionIntent(String text) {
        Intent intent = new Intent(Intent.ACTION_SEND);
        intent.setType("text/plain");
        intent.putExtra(Intent.EXTRA_TEXT, text);
        shareActionProvider.setShareIntent(intent);
    }
}

```

We're using these extra classes, so we need to import them.

Add a ShareActionProvider private variable.



Get a reference to the share action provider and assign it to the private variable. Then call the `setShareActionIntent()` method.

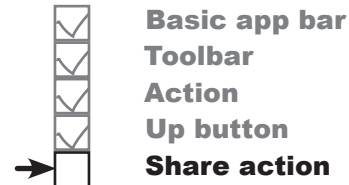
We created the `setShareActionIntent()` method. It creates an intent, and passes it to the share action provider using its `setShareIntent()` method.

You need to call the share action provider's `setShareIntent()` method whenever the content you wish to share has changed. As an example, if you're flicking through images in a photo app, you need to make sure you share the current photo.

We'll show you our full activity code on the next page, and then we'll see what happens when the app runs.

The full MainActivity.java code

Here's the full activity code for *MainActivity.java*. Update your code to reflect ours.



```
package com.hfad.bitsandpizzas;

import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.support.v7.widget.Toolbar;
import android.view.Menu;
import android.view.MenuItem;
import android.content.Intent;
import android.support.v7.widget.ShareActionProvider;
import android.support.v4.view.MenuItemCompat;

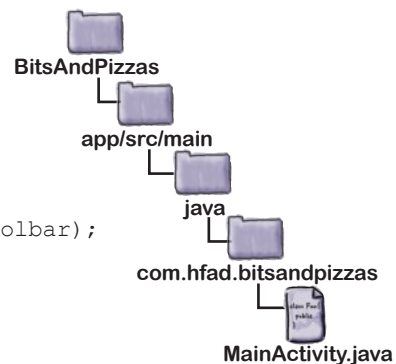
public class MainActivity extends AppCompatActivity {
```

We're using these extra classes, so we need to import them.

```
    private ShareActionProvider shareActionProvider;
```

```
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        Toolbar toolbar = (Toolbar) findViewById(R.id.toolbar);
        setSupportActionBar(toolbar);
    }
```

```
    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
        getMenuInflater().inflate(R.menu.menu_main, menu);
        MenuItem menuItem = menu.findItem(R.id.action_share);
        shareActionProvider =
            (ShareActionProvider) MenuItemCompat.getActionProvider(menuItem);
        setShareActionIntent("Want to join me for pizza?");
        return super.onCreateOptionsMenu(menu);
    }
```



← This is the default text that the share action should share.

The code continues on the next page. →

The MainActivity.java code (continued)

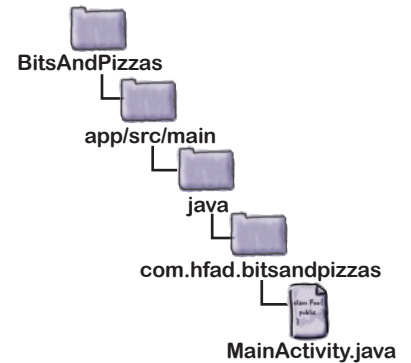
```

private void setShareActionIntent(String text) {
    Intent intent = new Intent(Intent.ACTION_SEND);
    intent.setType("text/plain");
    intent.putExtra(Intent.EXTRA_TEXT, text);
    shareActionProvider.setShareIntent(intent);
}

@Override
public boolean onOptionsItemSelected(MenuItem item) {
    switch (item.getItemId()) {
        case R.id.action_create_order:
            //Code to run when the Create Order item is clicked
            Intent intent = new Intent(this, OrderActivity.class);
            startActivity(intent);
            return true;
        default:
            return super.onOptionsItemSelected(item);
    }
}
}

```

↑
This sets the default text
in the share action provider.

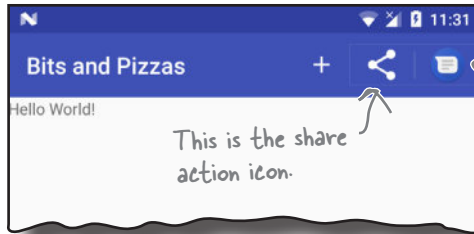
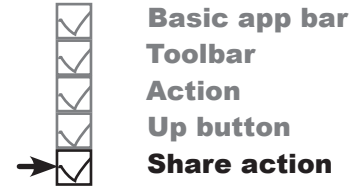


On the next page we'll check what happens when the code runs by taking the app for a test drive.



Test drive the app

When you run the app, the share action is displayed in the app bar:



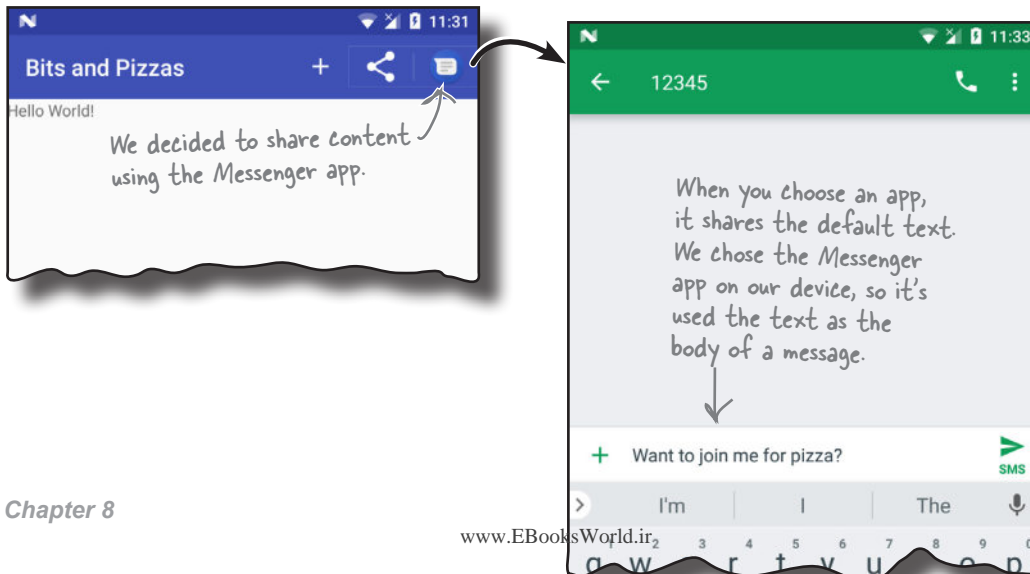
The share action provider also added the Messenger icon to our app bar. We usually share this app's content with the Messenger app, so the share action gave us a shortcut.

When you click on the share action, it gives you a list of apps to choose from that can accept the intent you want to share:



The intent we passed to the share action provider says we want to share text using ACTION_SEND. It displays a list of apps that can do this.

When you choose an app to share content with, the app gets launched and the default text is shared with it:





Your Android Toolbox

You've got Chapter 8 under your belt and now you've added Android Support Libraries and app bars to your toolbox.

You can download the full code for the chapter from <https://tinyurl.com/HeadFirstAndroid>.



BULLET POINTS

- You add a basic app bar by applying a theme that contains one.
- The Android Support Libraries provide backward compatibility with older versions of Android.
- The `AppCompatActivity` class is a type of activity that resides in the v7 AppCompat Support Library. In general, your activity needs to extend the `AppCompatActivity` class whenever you want an app bar that provides backward compatibility with older versions of Android.
- The `android:theme` attribute in `AndroidManifest.xml` specifies which theme to apply.
- You define styles in a style resource file using the `<style>` element. The `name` attribute gives the style a name. The `parent` attribute specifies where the style should inherit its properties from.
- The latest app bar features are in the `Toolbar` class in the v7 AppCompat Support Library. You can use a toolbar as your app bar.
- Add actions to your app bar by adding them to a menu resource file.
- Add the items in the menu resource file to the app bar by implementing the activity's `onCreateOptionsMenu()` method.
- You determine what items should do when clicked by implementing the activity's `onOptionsItemSelected()` method.
- Add an Up button to your app bar to navigate up the app's hierarchy. Specify the hierarchy in `AndroidManifest.xml`. Use the `ActionBar.setDisplayHomeAsUpEnabled()` method to enable the Up button.
- You can share content by adding the share action provider to your app bar. Add it by including it in your menu resource file. Call its `setShareIntent()` method to pass it an intent describing the content you wish to share.

9 fragments

Make It Modular

Doing the same job in different places... I guess that makes me a fragment.



You've seen how to create apps that work in the same way no matter what device they're running on.

But what if you want your app to look and behave differently depending on whether it's running on a *phone* or a *tablet*? In this case you need **fragments**, modular code components that can be **reused by different activities**. We'll show you how to create **basic fragments** and **list fragments**, how to **add them to your activities**, and how to get your fragments and activities to **communicate** with one another.

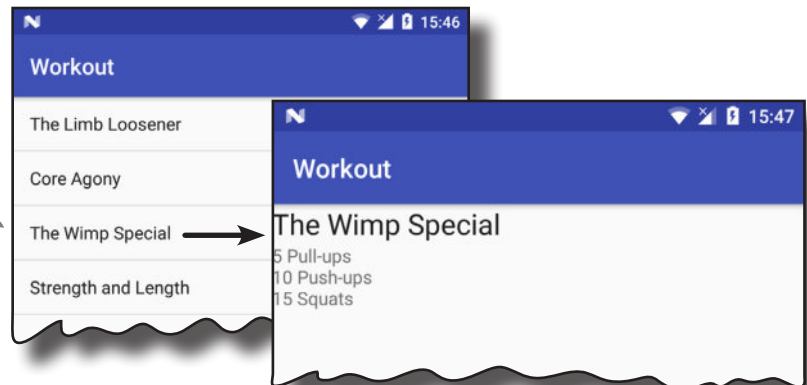
Your app needs to look great on ALL devices

One of the great things about Android development is that you can put the exact same app on devices with completely different screen sizes and processors, and have them run in exactly the same way. But that doesn't mean that they always have to *look* exactly the same.

On a phone:

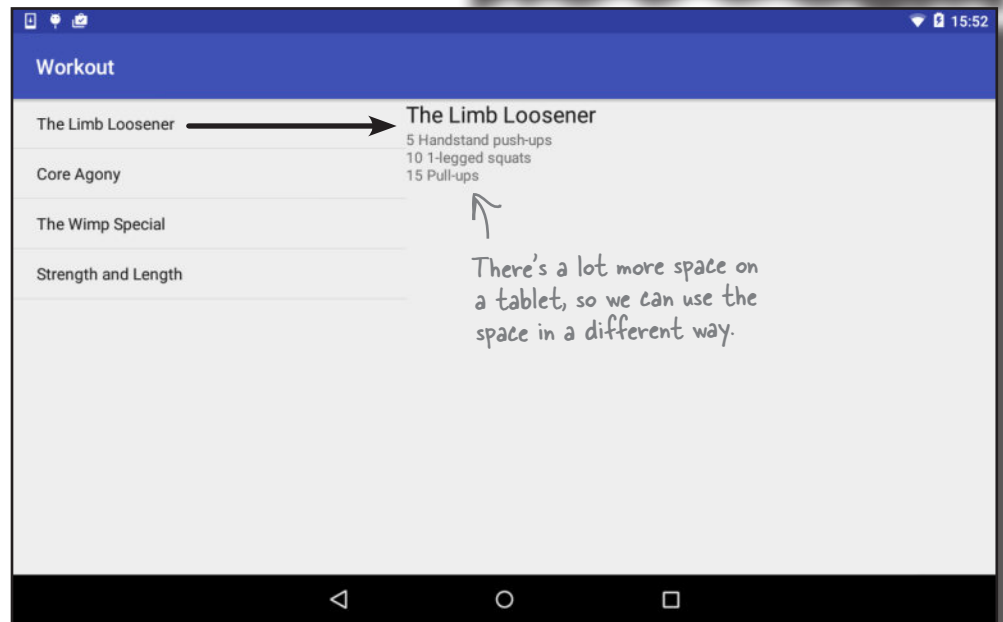
Take a look at this image of an app on a phone. It displays a list of workouts, and when you click on one, you are shown the details of that workout.

Click on an item in a list, and it launches a second activity.



On a tablet:

On a larger device, like a tablet, you have a lot more screen space available, so it would be good if all the information appeared on the same screen. On the tablet, the list of workouts only goes partway across the screen, and when you click on an item, the details appear on the right.



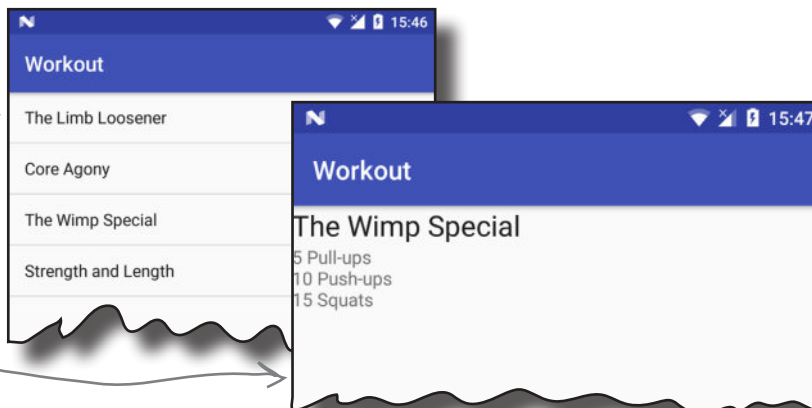
To make the phone and tablet user interfaces look different from each other, you can use separate layouts for large devices and small devices.

Your app may need to behave differently too

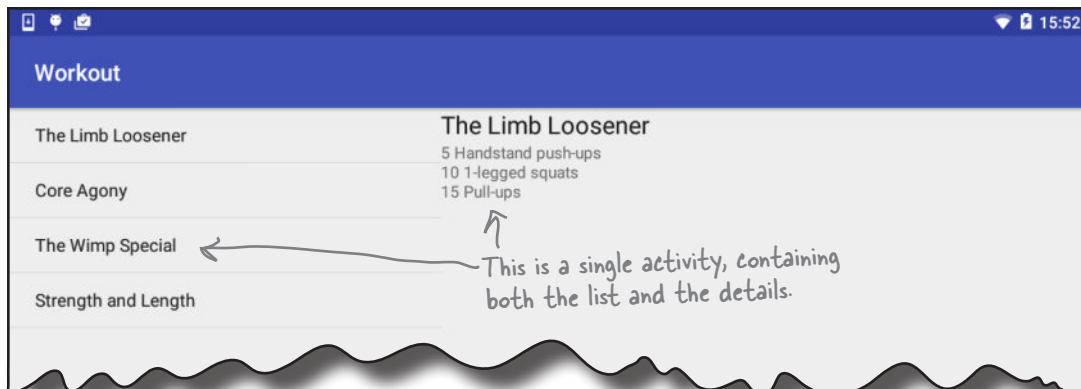
It's not enough to simply have different layouts for different devices. You also need *different Java code* to run alongside the layouts so that the app can behave differently depending on the device. In our Workout app, for instance, we need to provide **one activity for tablets**, and **two activities for phones**.

On a phone:

Here we have two activities: one for the list and one for the details.



On a tablet:



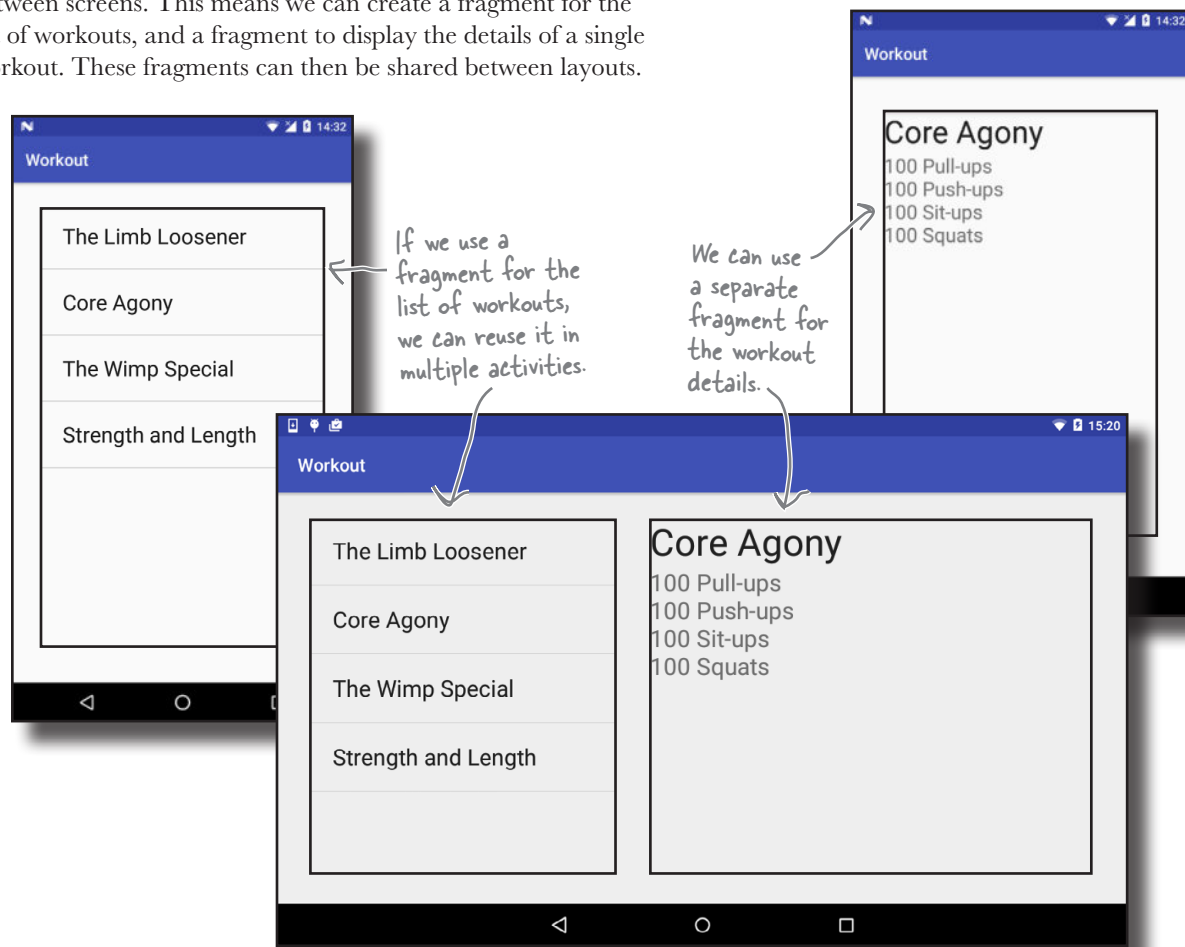
But that means you might duplicate code

The second activity that runs only on phones will need to insert the details of a workout into the layout. But that code will also need to be available in the main activity for when the app is running on a tablet. *The same code needs to be run by multiple activities.*

Rather than duplicate the code in the two activities, we can use **fragments**. So what's a fragment?

Fragments allow you to reuse code

Fragments are like reusable components or subactivities. A fragment is used to control part of a screen, and can be reused between screens. This means we can create a fragment for the list of workouts, and a fragment to display the details of a single workout. These fragments can then be shared between layouts.



A fragment has a layout

Just like an activity, a fragment has an associated layout. If you design it carefully, the fragment's Java code can be used to control everything within the interface. If the fragment code contains all that you need to control its layout, it greatly increases the chances that you'll be able to reuse it elsewhere in the app.

We're going to show you how to create and use fragments by building the Workout app.

The phone version of the app

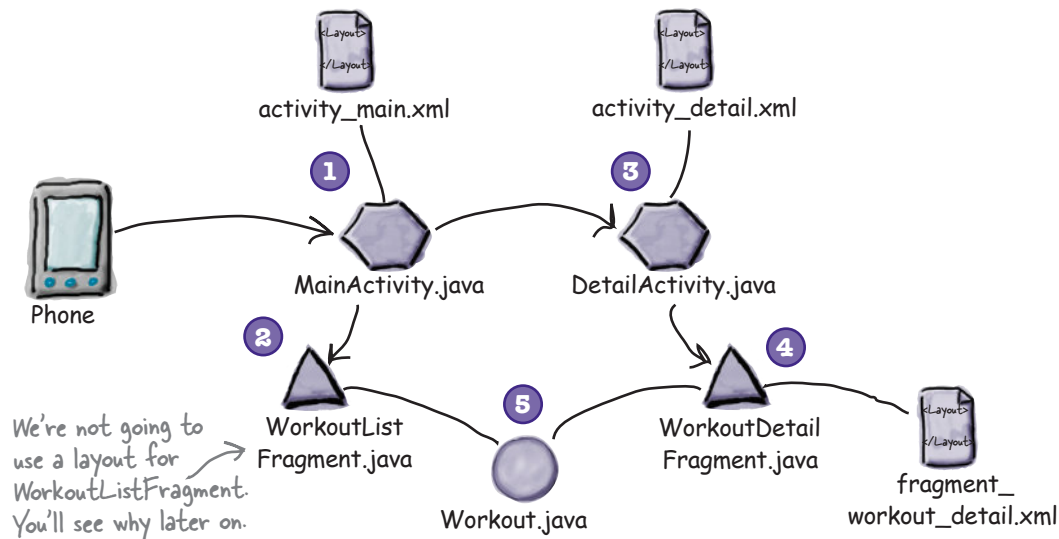
We're going to build the phone version of the app in this chapter, and reuse the fragments we create to build the tablet version of the app in Chapter 10. Here's how the phone version of the app will work:

- 1 **When the app gets launched, it starts MainActivity.**
MainActivity uses *activity_main.xml* for its layout, and contains a fragment called *WorkoutListFragment*.
- 2 **WorkoutListFragment displays a list of workouts.**
- 3 **When the user clicks on one of the workouts, DetailActivity starts.**
DetailActivity uses *activity_detail.xml* for its layout, and contains a fragment called *WorkoutDetailFragment*.
- 4 **WorkoutDetailFragment uses *fragment_workout_detail.xml* for its layout.**
It displays the details of the workout the user has selected.
- 5 **WorkoutListFragment and WorkoutDetailFragment get their workout data from *Workout.java*.**
Workout.java contains an array of *Workouts*.



Don't worry if the app structure looks complex.

We'll work through it step by step over the course of this chapter.



We'll go through the steps for creating this app on the next page.

Here's what we're going to do

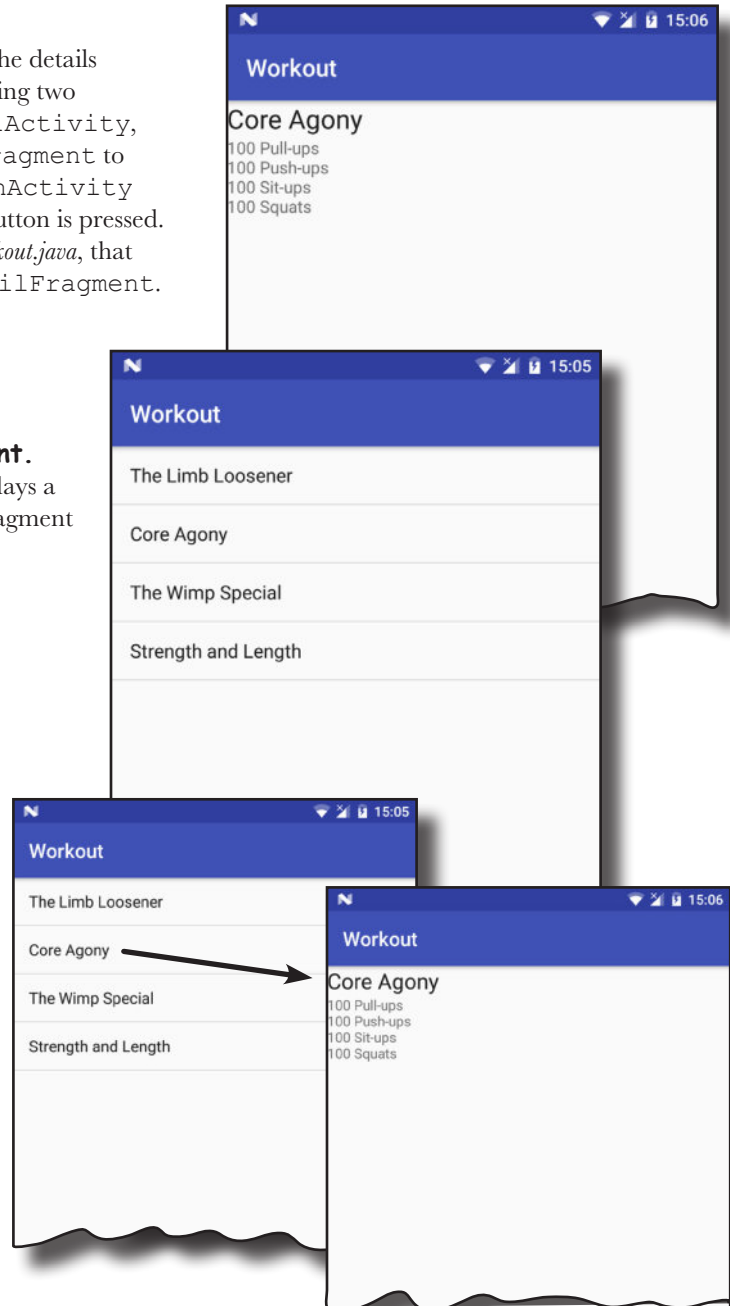
There are three main steps we'll go through to build the app:

- 1 Create WorkoutDetailFragment.**
 WorkoutDetailFragment displays the details of a specific workout. We'll start by creating two activities, MainActivity and DetailActivity, and then we'll add WorkoutDetailFragment to DetailActivity. We'll also get MainActivity to launch DetailActivity when a button is pressed. We'll also add a plain old Java class, *Workout.java*, that will provide the data for WorkoutDetailFragment.

- 2 Create WorkoutListFragment.**
 WorkoutListFragment displays a list of workouts. We'll add this fragment to MainActivity.

- 3 Coordinate the fragments to display the correct workout.**
 When the user clicks on an item in WorkoutListFragment, we'll start DetailActivity and get WorkoutDetailFragment to display details of the workout the user selected.

Let's get started.



Create the project and activities

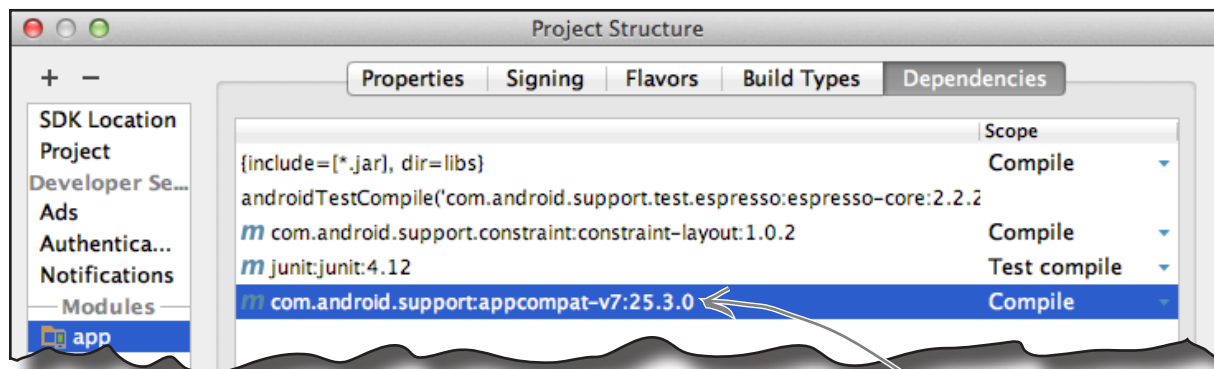
We're going to start by creating a project that contains two activities, `MainActivity` and `DetailActivity`. `MainActivity` will be used for the fragment that displays a list of workouts, and `DetailActivity` will be used for the fragment that displays details of one particular workout.

To do this, first create a new Android project with an empty activity for an application named “Workout” with a company domain of “hfad.com”, making the package name `com.hfad.workout`. The minimum SDK should be API 19 so that it works on most devices. Name the activity “`MainActivity`” and name the layout “`activity_main`”. **Make sure you check the Backwards Compatibility (AppCompat) checkbox.**

Next, create a second empty activity by highlighting the `com.hfad.workout` package in the `app/src/main/java` folder, and going to `File`→`New...`→`Activity`→`Empty Activity`. Name the activity “`DetailActivity`”, name the layout “`activity_detail`”, make sure the package name is `com.hfad.workout`, and **check the Backwards Compatibility (AppCompat) checkbox.**

Add the AppCompat Support Library

We're going to be using activities and fragments from the v7 AppCompat Library, which means you need to make sure the library has been added to your project as a dependency. To do this, go to the `File` menu and choose `Project Structure`. Then click on the `app` module and choose `Dependencies`.

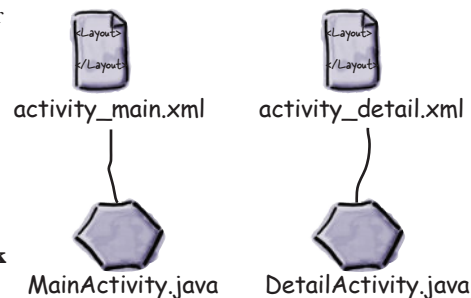


If Android Studio has already added the v7 AppCompat Support Library to your project, you'll see it listed in the list of dependencies. If it's not there, you'll need to add it yourself. To do this, click on the “+” button at the bottom or right side of the screen. When prompted, choose the `Library Dependency` option, then select the `appcompat-v7` library from the list of options. Finally, use the `OK` buttons to save your changes.

Once you've made sure the v7 AppCompat Support Library has been added, you can close the `Project Structure` window. On the next page we'll update `MainActivity`.



WorkoutDetailFragment
WorkoutListFragment
 Coordinate fragments



If prompted for the activity's source language, select the option for Java.

Add a button to MainActivity's layout

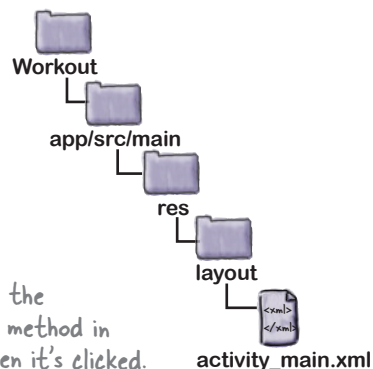
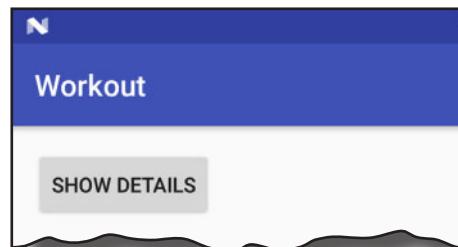
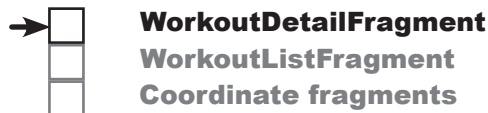
We're going to add a button to MainActivity that will start DetailActivity. This is because we're going to work on the fragment for DetailActivity first, and adding a button to MainActivity will give us an easy way of navigating from MainActivity to DetailActivity.

We'll start by adding the button to the layout. Open file `activity_main.xml`, then update your code so that it matches ours 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:padding="16dp"
    android:orientation="vertical"
    tools:context="com.hfad.workout.MainActivity">
    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:onClick="onShowDetails"
        android:text="@string/details_button" />
</LinearLayout>
```

This is the button we're adding.

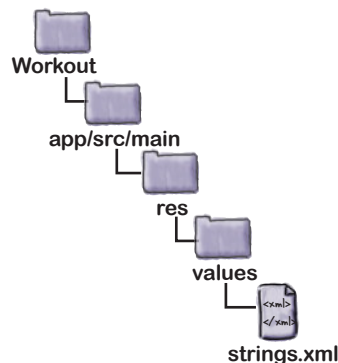
The button calls the `onShowDetails()` method in MainActivity when it's clicked. We need to write this method.



The button uses a String resource for its text, so we need to add it to the String resource file. Open file `strings.xml`, then add the following String resource:

```
<resources>
    ...
    <string name="details_button">Show details</string>
</resources>
```

This text will be displayed on the button.



When the button is clicked, we've specified that MainActivity's `onShowDetails()` method should be called. We'll write the code for this method next.

Make the button respond to clicks

We need to get `MainActivity`'s button to start `DetailActivity` when it's clicked. To do this, we'll add a method called `onShowDetails()` to `MainActivity`. The method will start `DetailActivity` using an intent, just as we've done in previous chapters.

Here's the full code for `MainActivity.java`. Update your code so that it matches ours.

```
package com.hfad.workout;

import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.view.View;
import android.content.Intent;

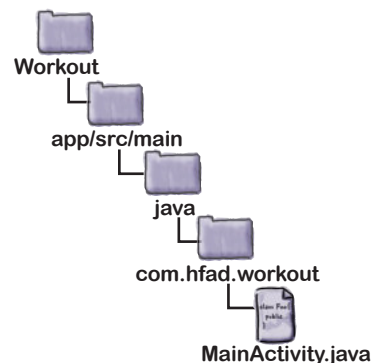
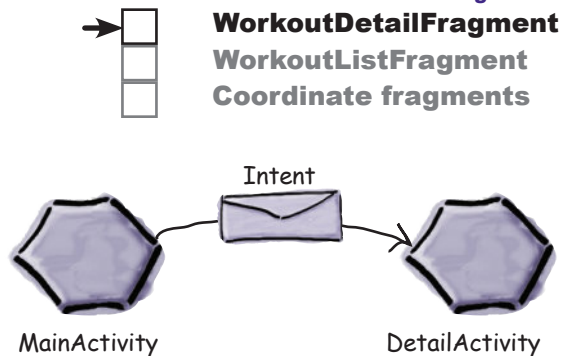
public class MainActivity extends AppCompatActivity {

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

    public void onShowDetails(View view) {
        Intent intent = new Intent(this, DetailActivity.class);
        startActivity(intent);
    }
}
```

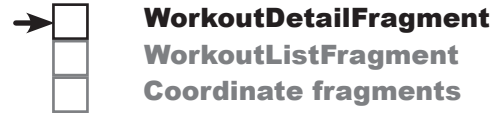
The activity extends AppCompatActivity.

This method is called when the button is clicked. It starts DetailActivity.



That's everything we need to get `MainActivity` to start `DetailActivity`. On the next page we'll add a new fragment to our project called `WorkoutDetailFragment` that we'll then add to `DetailActivity`.

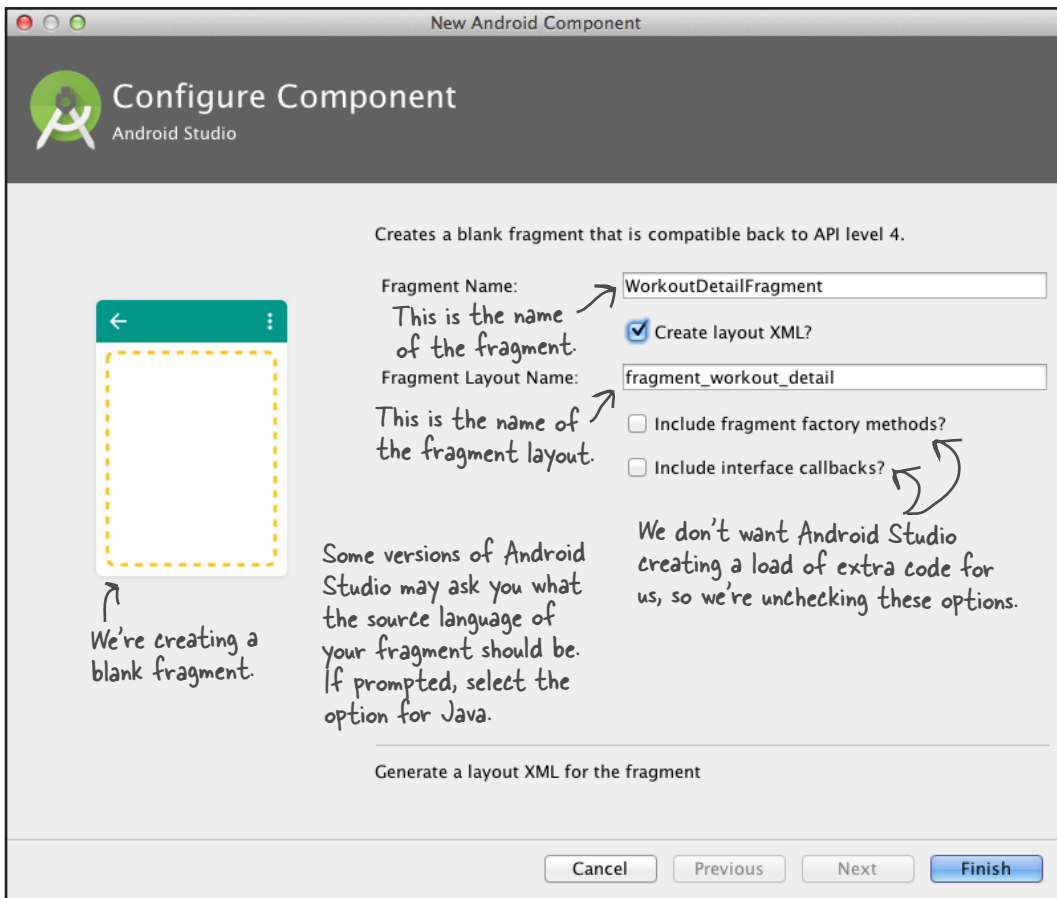
How to add a fragment to your project



We're going to add a new fragment called `WorkoutDetailFragment` to the project to display details of a single workout. You add a new fragment in a similar way to how you add a new activity: by switching to the Project view of Android Studio's explorer, highlighting the `com.hfad.workout` package in the `app/src/main/java` folder, going to the File menu, and choosing `New... → Fragment → Fragment (Blank)`.

You will be asked to choose options for your new fragment. Name the fragment "WorkoutDetailFragment", check the option to create layout XML for it, and give the fragment layout a name of "fragment_workout_detail". Uncheck the options to include fragment factory methods and interface callbacks; these options generate extra code that you don't need to use. When you're done, click on the Finish button.

We suggest looking at the extra code Android generates for you after you've finished this book. You might find some of it useful depending on what you want to do.



When you click on the Finish button, Android Studio creates your new fragment and adds it to the project.

What fragment code looks like



When you create a fragment, Android Studio creates two files for you: Java code for the fragment itself, and XML code for the fragment's layout. The Java code describes the fragment's behavior, and the layout describes the fragment's appearance.

We'll look at the Java code first. Go to the `com.hfad.workout` package in the `app/src/main/java` folder and open the file `WorkoutDetailFragment.java` that Android Studio just created for us. Then replace the code that Android Studio generated with the code below:

```

package com.hfad.workout;

import android.support.v4.app.Fragment;
import android.os.Bundle;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;

public class WorkoutDetailFragment extends Fragment {

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

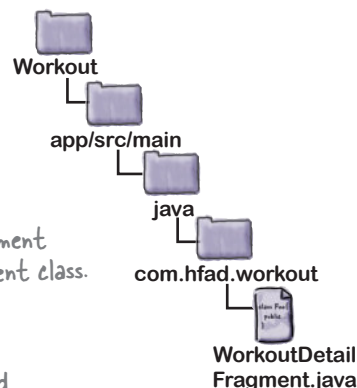
```

We're using the `Fragment` class from the Android Support Library.

`WorkoutDetailFragment` extends the `Fragment` class.

This is the `onCreateView()` method. It's called when Android needs the fragment's layout.

This tells Android which layout the fragment uses (in this case, it's `fragment_workout_detail`).



The above code creates a basic fragment. As you can see, the code for a fragment looks very similar to activity code.

To create a fragment, you first need to extend the `Fragment` class or one of its subclasses. We're using fragments from the Support Library, so our fragment needs to extend the `android.support.v4.app.Fragment` class. This is because the Support Library fragments are backward compatible with earlier versions of Android, and contain the latest fragment features.

The fragment implements the `onCreateView()` method, which gets called each time Android needs the fragment's layout, and it's where you say which layout the fragment uses. The `onCreateView()` method is optional, but as you need to implement it whenever you're creating a fragment with a layout, you'll need to implement it nearly every time you create a fragment. We'll look at this method in more detail on the next page.

The fragment's onCreateView() method

The onCreateView() method returns a View object that represents the fragment's user interface. It gets called when Android is ready to instantiate the user interface, and it takes three parameters:

```
public View onCreateView(LayoutInflater inflater,
                        ViewGroup container,
                        Bundle savedInstanceState) {
}
```

The first parameter is a LayoutInflater that you can use to inflate the fragment's layout. Inflating the layout turns your XML views into Java objects.

The second parameter is a ViewGroup. This is the ViewGroup in the activity's layout that will contain the fragment.

The final parameter is a Bundle. This is used if you've previously saved the fragment's state, and want to reinstate it.

You specify the fragment's layout using the LayoutInflater's inflate() method:

```
public View onCreateView(LayoutInflater inflater,
                        ViewGroup container,
                        Bundle savedInstanceState) {
    return inflater.inflate(R.layout.fragment_workout_detail,
                          container,
                          false);
}
```

This is the fragment equivalent of calling an activity's setContentView() method. You use it to say what layout the fragment should use, in this case R.layout.fragment_workout_detail.

The inflate() method's container argument specifies the ViewGroup in the activity that the fragment's layout needs to be inserted into. It gets passed to the fragment as the second parameter in the fragment's onCreateView() method.

Now that you've seen the fragment's code, let's have a look at its layout.



WorkoutDetailFragment
WorkoutListFragment
 Coordinate fragments

The onCreateView() method gets called when Android needs the fragment's layout.



All fragments must have a public no-argument constructor.

Android uses it to reinstantiate the fragment when needed, and if it's not there, you'll get a runtime exception.

In practice, you only need to add a public no-argument constructor to your fragment code if you include **another** constructor with one or more arguments. This is because if a Java class contains no constructors, the Java compiler automatically adds a public no-argument constructor for you.

This inflates the fragment's layout from XML to Java objects.

Fragment layout code looks just like activity layout code

As we said earlier, fragments use layout files to describe their appearance. Fragment layout code looks just like activity layout code, so when you write your own fragment layout code, you can use any of the views and layouts you've already been using to write activity layout code.

We're going to update our layout code so that our fragment contains two text views, one for the workout title and one for the workout description.

Open the file `fragment_workout_detail.xml` in the `app/src/res/layout` folder, and replace its contents with the code below:

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

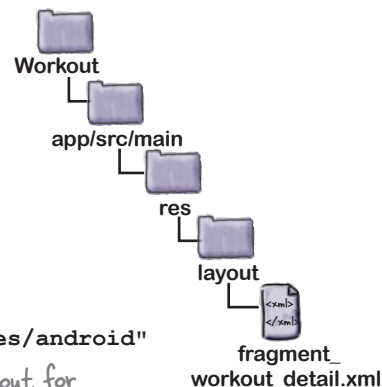
We're using a `LinearLayout` for our fragment, but we could have used any of the other layout types we've looked at instead.

```
<TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:textAppearance="?android:attr/textAppearanceLarge"
    android:text="@string/workout_title"
    android:id="@+id/textTitle" />
<TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/workout_description"
    android:id="@+id/textDescription" />
</LinearLayout>
```

We'll display the workout title and description in two separate `TextView`s.

This makes the text large.

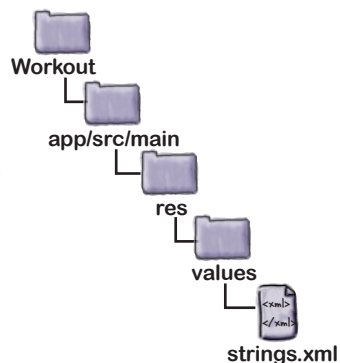
Static String resources.



For now we're using static String resources for the title and description of the workout so that we can see our fragment working. Open `strings.xml`, and add the following String resources:

```
<resources>
  ...
  <string name="workout_title">Title</string>
  <string name="workout_description">Description</string>
</resources>
```

We'll use these to display default text in our fragment.



That's everything we need for our fragment. On the next page we'll look at how you add the fragment to an activity.

Add a fragment to an activity's layout



WorkoutDetailFragment
WorkoutListFragment
 Coordinate fragments

We're going to add our `WorkoutDetailFragment` to `DetailActivity` so that the fragment gets displayed in the activity's layout. To do this, we're going to add a `<fragment>` element to `DetailActivity`'s layout.

The `<fragment>` element is a view that specifies the name of the fragment you want to display. It looks like this:

```
<fragment
    android:name="com.hfad.workout.WorkoutDetailFragment"
    android:layout_width="match_parent"
    android:layout_height="match_parent" />
```

← This is the full name of the fragment class.

You specify the fragment using the `android:name` attribute and giving it a value of the fully qualified name of the fragment. In our case, we want to display a fragment called `WorkoutDetailFragment` that's in the `com.hfad.workout` package, so we use:

```
android:name="com.hfad.workout.WorkoutDetailFragment"
```

When Android creates the activity's layout, it replaces the `<fragment>` element with the `View` object returned by the fragment's `onCreateView()` method. This view is the fragment's user interface, so the `<fragment>` element is really a placeholder for where the fragment's layout should be inserted.

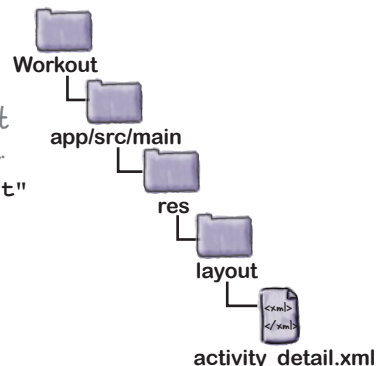
You add the `<fragment>` element to your layout in the same way that you add any other element. As an example, here's how you'd add the fragment to a linear layout:

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

    <fragment
        android:name="com.hfad.workout.WorkoutDetailFragment"
        android:layout_width="match_parent"
        android:layout_height="match_parent" />

</LinearLayout>
```

← This adds the fragment to the activity's layout.



If your layout only contains a single fragment with no other views, however, you can simplify your layout code.



WorkoutDetailFragment
WorkoutListFragment
Coordinate fragments

Simplify the layout

If the layout code for your activity comprises a single fragment contained within a layout element with no other views, you can simplify your layout code by removing the root layout element.

Each layout file you create must have a single root element that's either a view or view group. This means that if your layout contains multiple items, these items must be contained within a view group such as a linear layout.

If your layout contains a single fragment, the `<fragment>` element can be the layout file's root. This is because the `<fragment>` element is a type of view, and Android replaces it with the layout of the fragment at runtime.

In our code example on the previous page, we showed you a fragment contained within a linear layout. As there are no other views in the layout, we can remove the linear layout so that our code looks like this:

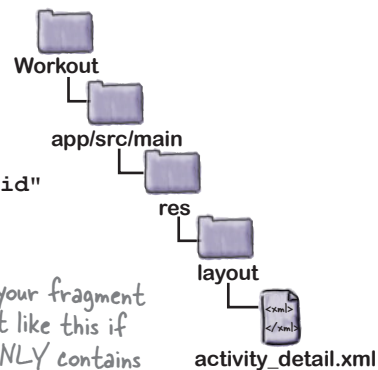
```
<?xml version="1.0" encoding="utf-8"?>
<fragment
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:name="com.hfad.workout.WorkoutDetailFragment"
    android:layout_width="match_parent"
    android:layout_height="match_parent" />
```

This code does exactly the same thing as the code on the previous page, but it's much shorter.

That's the only code we need for `DetailActivity`'s layout, so replace the code you have in your version of `activity_detail.xml`, with the code above and save your changes.

On the next page we'll look at the code for the activity itself.

A layout file requires a single view or view group as its root element. If your activity only contains a fragment, the fragment itself can be the root element.



Support Library fragments need activities that extend FragmentActivity

When you add a fragment to an activity, you usually need to write code that controls any interactions between the fragment and the activity. You'll see examples of this later in the chapter.

Currently, `WorkoutDetailFragment` only contains static data. `DetailActivity` only has to display the fragment, and doesn't need to interact with it, so this means that we don't need to write any extra activity code to control the interaction.

There's one important point to be aware of, however. When you're using fragments from the Support Library, as we are here, you must make sure that **any activity you want to use them with extends the `FragmentActivity` class, or one of its subclasses**. The `FragmentActivity` class is designed to work with Support Library fragments, and if your activity doesn't extend this class, your code will break.

In practice, this isn't a problem. This is because the `AppCompatActivity` class is a subclass of `FragmentActivity`, so as long as your activity extends the `AppCompatActivity` class, your Support Library fragments will work.

Here's the code for `DetailActivity.java`. Update your code so that it matches ours below:

```
package com.hfad.workout;

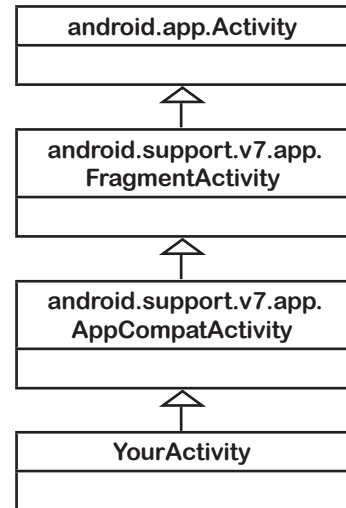
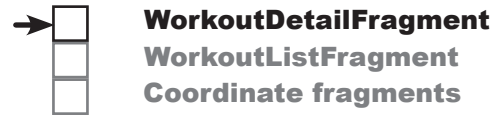
import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;

public class DetailActivity extends AppCompatActivity {

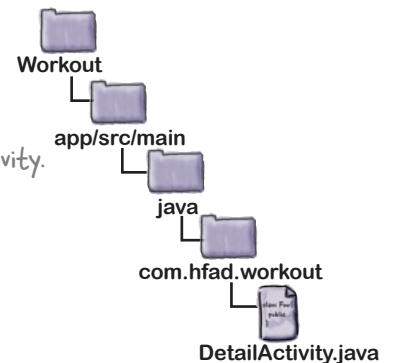
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_detail);
    }
}
```

DetailActivity extends AppCompatActivity.

That's everything we need to display the `WorkoutDetailFragment` in our activity. Let's see what happens when we run the app.



As long as your activity extends `FragmentActivity`, or one of its subclasses such as `AppCompatActivity`, you can use fragments from the Support Library.





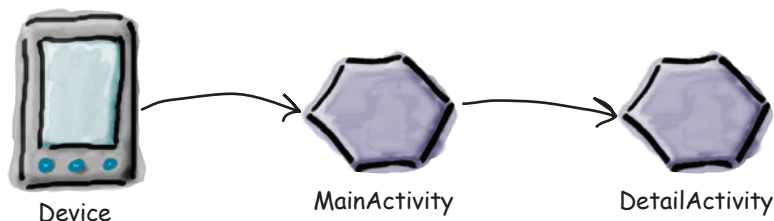
WorkoutDetailFragment
WorkoutListFragment
 Coordinate fragments

What the code does

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

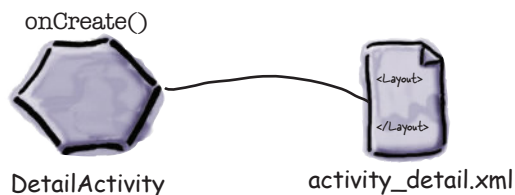
1 When the app is launched, activity MainActivity gets created.

The user clicks on the button in MainActivity to start DetailActivity.

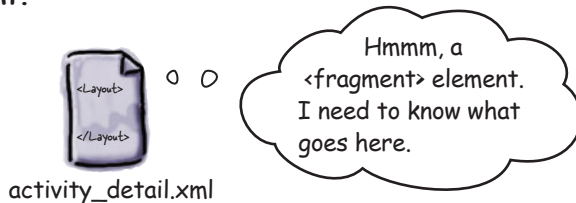


2 DetailActivity's onCreate() method runs.

The onCreate() method specifies that *activity_detail.xml* should be used for DetailActivity's layout.

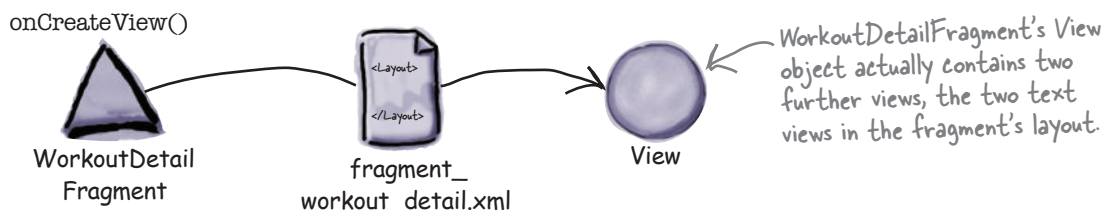


3 activity_detail.xml sees that it includes a <fragment> element that refers to WorkoutDetailFragment.

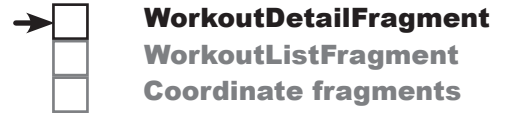


4 WorkoutDetailFragment's onCreateView() method is called.

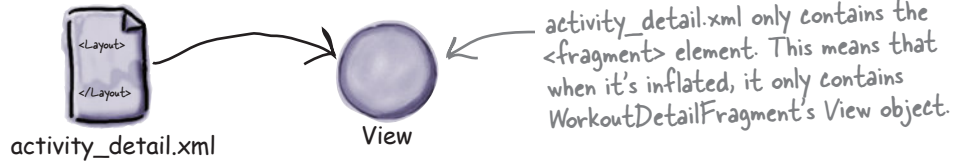
The onCreateView() method specifies that *fragment_workout_detail.xml* should be used for WorkoutDetailFragment's layout. It inflates the layout to a View object.



The story continues



- 5 **activity_detail.xml's Views are inflated to View Java objects.**
DetailActivity layout uses WorkoutDetailFragment's View object in place of the <fragment> element in its layout's XML.

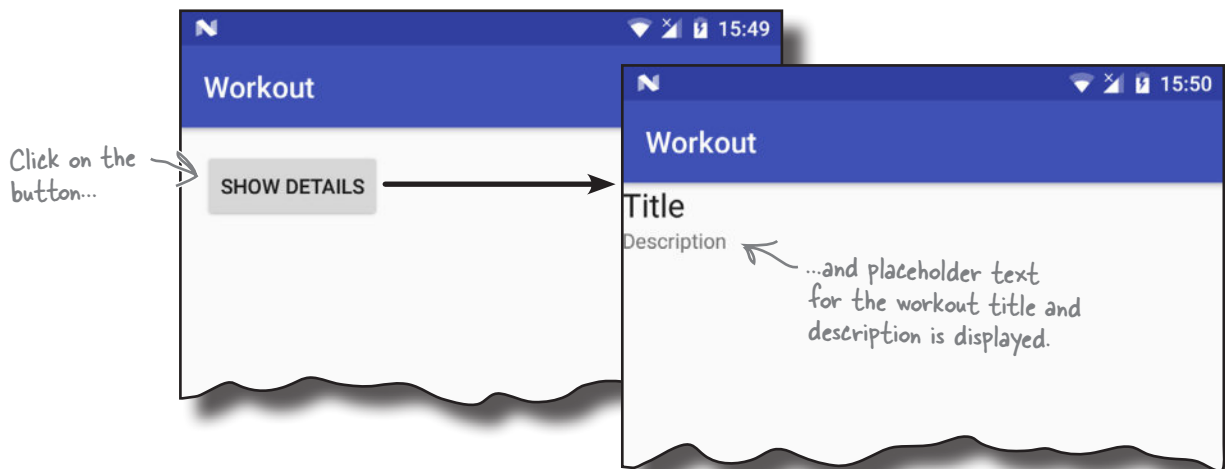


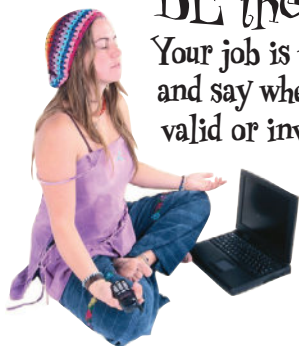
- 6 **Finally, DetailActivity is displayed on the device.**
Its layout contains the fragment WorkoutDetailFragment.



Test drive the app

When we run our app, MainActivity gets launched.
When we click on MainActivity's button, it starts DetailActivity. DetailActivity contains WorkoutDetailFragment, and we see this on the device.





BE the Layout

Your job is to play like you're the layout and say whether each of these layouts is valid or invalid and why. Assume that any fragments or String resources referred to in the layout already exist.

A

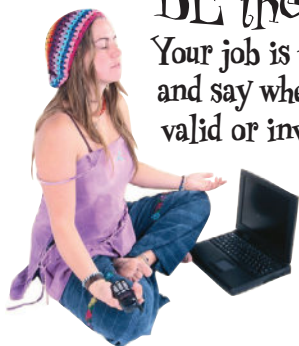
```
<?xml version="1.0" encoding="utf-8"?>
<fragment
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:name="com.hfad.workout.WorkoutDetailFragment"
    android:layout_width="match_parent"
    android:layout_height="match_parent" />
```

B

```
<?xml version="1.0" encoding="utf-8"?>
<fragment
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:name="com.hfad.workout.WorkoutDetailFragment"
    android:layout_width="match_parent"
    android:layout_height="match_parent" />
<Button
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/details_button" />
```

C

```
<?xml version="1.0" encoding="utf-8"?>
<Button
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/details_button" />
```



BE the Layout Solution
 Your job is to play like you're the layout and say whether each of these layouts is valid or invalid and why. Assume that any fragments or String resources referred to in the layout already exist.

A `<?xml version="1.0" encoding="utf-8"?>`
`<fragment`
`xmlns:android="http://schemas.android.com/apk/res/android"`
`android:name="com.hfad.workout.WorkoutDetailFragment"`
`android:layout_width="match_parent"`
`android:layout_height="match_parent" />`

This layout is valid, as it consists of a single fragment.

B `<?xml version="1.0" encoding="utf-8"?>`
`<fragment`
`xmlns:android="http://schemas.android.com/apk/res/android"`
`android:name="com.hfad.workout.WorkoutDetailFragment"`
`android:layout_width="match_parent"`
`android:layout_height="match_parent" />`
`<Button`
`android:layout_width="wrap_content"`
`android:layout_height="wrap_content"`
`android:text="@string/details_button" />`

This layout is invalid. A layout must have a single View or ViewGroup as its root element. To make this layout valid, you would need to put the fragment and Button in a ViewGroup.

C `<?xml version="1.0" encoding="utf-8"?>`
`<Button`
`xmlns:android="http://schemas.android.com/apk/res/android"`
`android:layout_width="wrap_content"`
`android:layout_height="wrap_content"`
`android:text="@string/details_button" />`

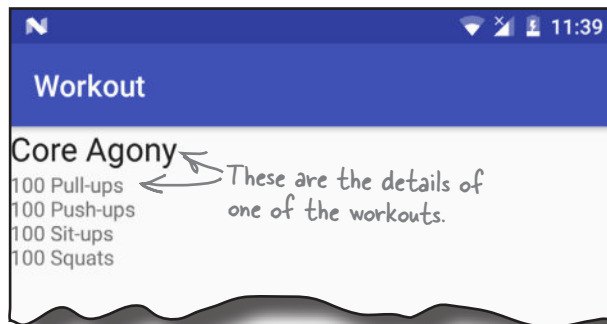
This layout is valid as it has a single View, in this case a Button, as its root element.

Get the fragment and activity to interact



So far we've looked at how you add a basic fragment to an activity. The next thing we'll look at is how you get the fragment and activity to interact.

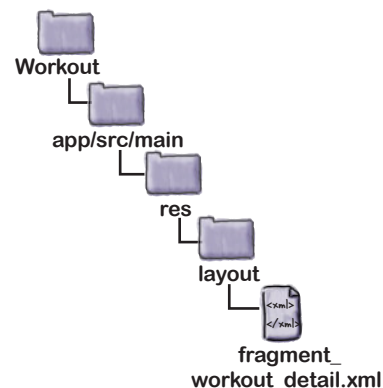
To do this, we'll start by changing `WorkoutDetailFragment` so that it displays details of a workout instead of the placeholder text we have currently.



The first thing we'll do is update the fragment's layout to remove the static text that's currently displayed. Open file `fragment_workout_detail.xml`, then update the code to match our changes below:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    ...
    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:textAppearance="?android:attr/textAppearanceLarge"
        android:text="@string/workout_title"
        android:id="@+id/textTitle" />
    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/workout_description"
        android:id="@+id/textDescription" />
</LinearLayout>
```

Delete both these lines.



On the next page we'll add a new class to our project to hold the workout data.

The Workout class

We're going to hold our workout data in a file called *Workout.java*, which is a pure Java class file that the app will get workout data from. The class defines an array of four workouts, where each workout is composed of a name and description. Select the *com.hfad.workout* package in the *app/src/main/java* folder in your project, then go to File→New...→Java Class. When prompted, name the class "Workout", and make sure the package name is *com.hfad.workout*. Then replace the code in *Workout.java* with the following, and save your changes.

```
package com.hfad.workout;

public class Workout {
    private String name;
    private String description;

    public static final Workout[] workouts = {
        new Workout("The Limb Loosener",
            "5 Handstand push-ups\n10 1-legged squats\n15 Pull-ups"),
        new Workout("Core Agony",
            "100 Pull-ups\n100 Push-ups\n100 Sit-ups\n100 Squats"),
        new Workout("The Wimp Special",
            "5 Pull-ups\n10 Push-ups\n15 Squats"),
        new Workout("Strength and Length",
            "500 meter run\n21 x 1.5 pood kettleball swing\n21 x pull-ups")
    };

    //Each Workout has a name and description
    private Workout(String name, String description) {
        this.name = name;
        this.description = description;
    }

    public String getDescription() {
        return description;
    }

    public String getName() {
        return name;
    }

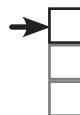
    public String toString() {
        return this.name;
    }
}
```

Each Workout has a name and description.

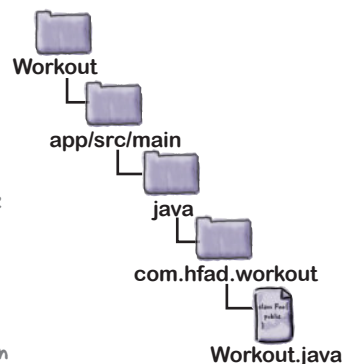
workouts is an array of four Workouts.

These are getters for the private variables.

The String representation of a Workout is its name.



WorkoutDetailFragment
WorkoutListFragment
 Coordinate fragments



The data will be used by the fragment to display details of a particular workout. We'll look at this next.



WorkoutDetailFragment
WorkoutListFragment
 Coordinate fragments

Pass the workout ID to the fragment

When you have an activity that uses a fragment, the activity will usually need to talk to the fragment in some way. As an example, if you have a fragment that displays detail records, you need the activity to tell the fragment which record it needs to display details of.

In our case, we need `WorkoutDetailFragment` to display details of a particular workout. To do this, we'll add a simple setter method to the fragment that sets the value of the workout ID. The activity will then be able to use this method to set the workout ID. Later on, we'll use the workout ID to update the fragment's views.

Here's the revised code for `WorkoutDetailFragment` (update your code with our changes):

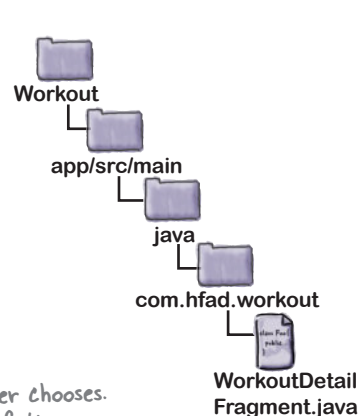
```
package com.hfad.workout;

import android.support.v4.app.Fragment;
import android.os.Bundle;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;

public class WorkoutDetailFragment extends Fragment {
    private long workoutId; ← This is the ID of the workout the user chooses.
                            Later, we'll use it to set the values of the
                            fragment's views with the workout details.

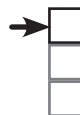
    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup container,
                            Bundle savedInstanceState) {
        return inflater.inflate(R.layout.fragment_workout_detail, container, false);
    }

    public void setWorkout(long id) { ← This is a setter method for the workout
                                      ID. The activity will use this method to
                                      set the value of the workout ID.
        this.workoutId = id;
    }
}
```



We need to get the `DetailActivity` to call the fragment's `setWorkout()` method and pass it the ID of a particular workout. In order to do this, the activity must get a reference to the fragment. But how?

Use the fragment manager to manage fragments



WorkoutDetailFragment
WorkoutListFragment
 Coordinate fragments

Before an activity can talk to its fragment, the activity first needs to get a reference to the fragment. You get a reference to an activity's fragments using the activity's **fragment manager**. The fragment manager is used to keep track of and deal with any fragments used by the activity.

There are two methods for getting a reference to the fragment manager, `getFragmentManager()` and `getSupportFragmentManager()`. The `getSupportFragmentManager()` method gets a reference to the fragment manager that deals with fragments from the Support Library like ours, and the `getFragmentManager()` method gets a reference to the fragment manager that deals with fragments that use the native Android fragment class instead. You then use the fragment manager's `findFragmentById()` method to get a reference to the fragment.

We're using fragments from the Support Library, so we're going to use the `getSupportFragmentManager()` method like this:

```
getSupportFragmentManager().findFragmentById(R.id.fragment_id)
```

This is the ID of the fragment in the activity's layout.

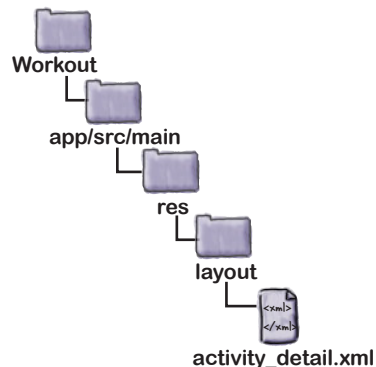
`findFragmentById()` is a bit like `findViewById()` except you use it to get a reference to a fragment.

We're going to use `DetailActivity`'s fragment manager to get a reference to its `WorkoutDetailFragment`. In order to do this, we first need to assign an ID to the fragment.

You assign an ID to an activity's fragment in the activity's layout. Open the file `activity_detail.xml`, then add an ID to the activity's fragment by adding the line of code highlighted below:

```
<?xml version="1.0" encoding="utf-8"?>
<fragment xmlns:android="http://schemas.android.com/apk/res/android"
    android:name="com.hfad.workout.WorkoutDetailFragment"
    android:id="@+id/detail_frag"
    android:layout_width="match_parent"
    android:layout_height="match_parent" />
```

← Add an ID to the fragment.



The above code gives the fragment an ID of `detail_frag`. On the next page we'll use the ID to get a reference to the fragment.



Get the activity to set the workout ID

To get a reference to the fragment, we need to add the following code:

```
WorkoutDetailFragment frag = (WorkoutDetailFragment)
    getSupportFragmentManager().findFragmentById(R.id.detail_frag);
```

We can then call the fragment's `setWorkout()` method to tell the fragment which workout we want it to display details for. For now, we'll hardcode which workout we want it to display so that we can see it working. Later on, we'll change the code so that the user can select which workout she wants to see.

Here's our revised code for `DetailActivity.java`. Update your code to reflect our changes:

```
package com.hfad.workout;

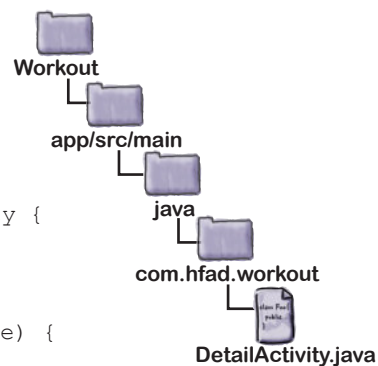
import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;

public class DetailActivity extends AppCompatActivity {

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

        WorkoutDetailFragment frag = (WorkoutDetailFragment)
            getSupportFragmentManager().findFragmentById(R.id.detail_frag);
        frag.setWorkout(1);
    }
}
```

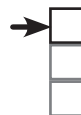
We're going to get WorkoutDetailFragment to display details of a workout here to check that it's working.



↑
This gets us a reference to WorkoutDetailFragment. Its id in the activity's layout is detail_frag.

As you can see, we've got a reference to the fragment after calling `setContentView()`. Getting the reference here is really important, because before this point, the fragment won't have been created.

The next thing we need to do is get the fragment to update its views when the fragment is displayed to the user. Before we can do this, we need to understand the fragment's lifecycle so that we add our code to the correct method in the fragment.

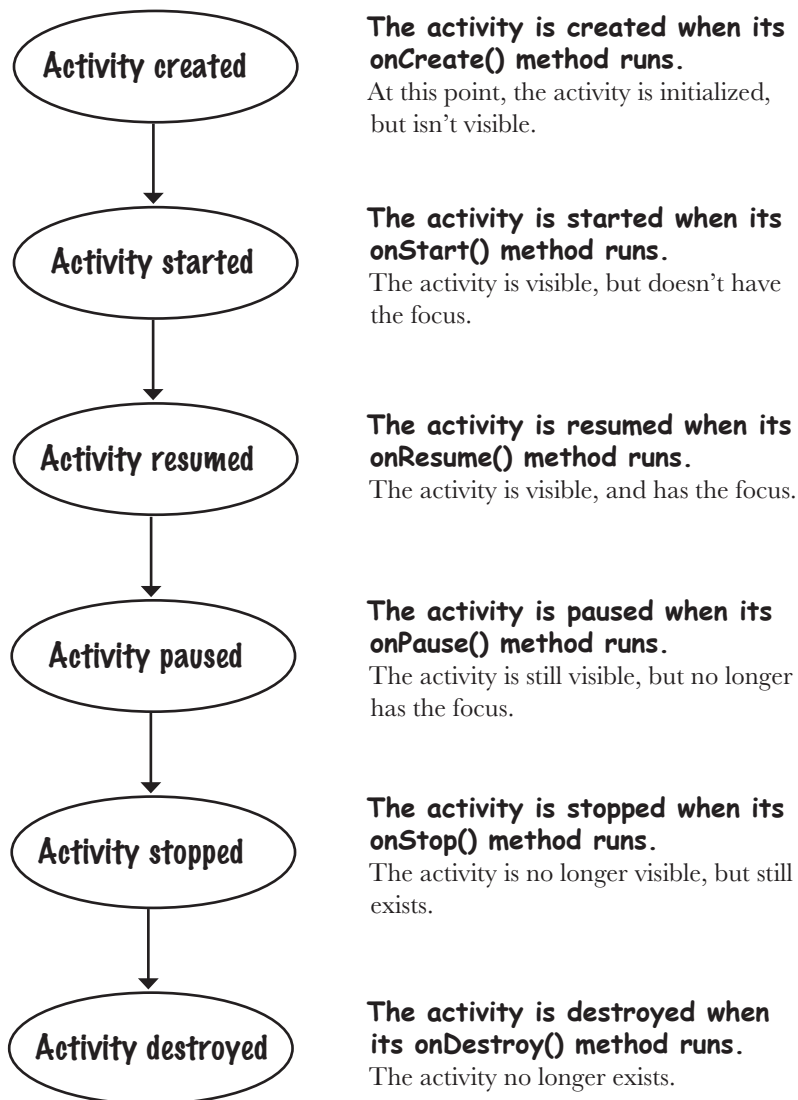


WorkoutDetailFragment
WorkoutListFragment
 Coordinate fragments

Activity states revisited

Just like an activity, a fragment has a number of key lifecycle methods that get called at particular times. It's important to know what these methods do and when they get called so your fragment works in just the way you want.

Fragments are contained within and controlled by activities, so the fragment lifecycle is closely linked to the activity lifecycle. Here's a reminder of the different states an activity goes through, and on the next page we'll show you how these relate to fragments.





The fragment lifecycle

A fragment's lifecycle is very similar to an activity's, but it has a few extra steps. This is because it needs to interact with the lifecycle of the activity that contains it. Here are the fragment lifecycle methods, along with where they fit in with the different activity states.

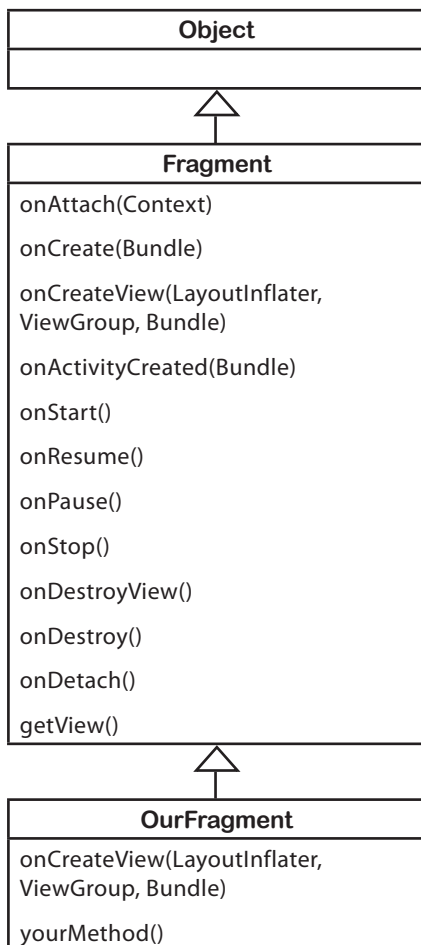
Activity states	Fragment callbacks	
Activity created	onAttach()	onAttach(Context) This happens when the fragment is associated with a context, in this case an activity.
	onCreate()	onCreate(Bundle) This is very similar to the activity's onCreate () method. It can be used to do the initial setup of the fragment.
	onCreateView()	onCreateView(LayoutInflater, ViewGroup, Bundle) Fragments use a layout inflater to create their view at this stage.
	onActivityCreated()	onActivityCreated(Bundle) Called when the onCreate () method of the activity has completed.
	Activity started	onStart()
Activity resumed		onResume()
	Activity paused	onPause()
Activity stopped	onStop()	onStop() Called when the fragment is no longer visible to the user.
Activity destroyed	onDestroyView()	onDestroyView() Gives the fragment the chance to clear away any resources that were associated with its view.
	onDestroy()	onDestroy() In this method, the fragment can clear away any other resources it created.
	onDetach()	onDetach() Called when the fragment finally loses contact with the activity.



WorkoutDetailFragment
WorkoutListFragment
 Coordinate fragments

Fragments inherit lifecycle methods

As you saw earlier, our fragment extends the Android `Fragment` class. This class gives our fragment access to the fragment lifecycle methods. Here's a diagram showing the class hierarchy.



Object class
 (`java.lang.Object`)

Fragment class
 (`android.support.v4.app.Fragment`)
 The `Fragment` class implements default versions of the lifecycle methods. It also defines other methods that fragments need, such as `getView()`.

OurFragment class
 (`com.hfad.foo`)
 Most of the behavior of our fragment is handled by superclass methods our fragment inherits. All you do is override the methods you need.

Even though fragments have a lot in common with activities, the `Fragment` class doesn't extend the `Activity` class. This means that some methods that are available to activities aren't available to fragments.

Note that the `Fragment` class doesn't implement the `Context` class. Unlike an activity, a fragment isn't a type of context and therefore doesn't have direct access to global information about the application environment. Instead, fragments must access this information using the context of other objects such as its parent activity.

Now that you understand the fragment's lifecycle better, let's get back to getting `WorkoutDetailFragment` to update its views.

Set the view's values in the fragment's onStart() method

We need to get `WorkoutDetailFragment` to update its views with details of the workout. We need to do this when the activity becomes visible, so we'll use the fragment's `onStart()` method. Update your code to match ours:

```
package com.hfad.workout;

import android.support.v4.app.Fragment;
import android.os.Bundle;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;
import android.widget.TextView;

public class WorkoutDetailFragment extends Fragment {
    private long workoutId;

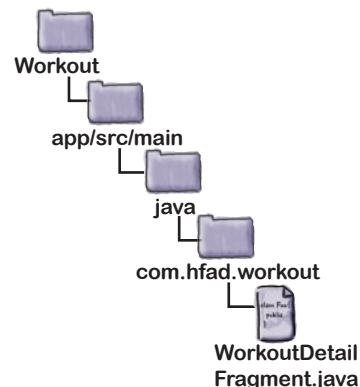
    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup container,
        Bundle savedInstanceState) {
        return inflater.inflate(R.layout.fragment_workout_detail, container, false);
    }

    @Override
    public void onStart() {
        super.onStart();
        View view = getView();
        if (view != null) {
            TextView title = (TextView) view.findViewById(R.id.textTitle);
            Workout workout = Workout.workouts[(int) workoutId];
            title.setText(workout.getName());
            TextView description = (TextView) view.findViewById(R.id.textDescription);
            description.setText(workout.getDescription());
        }
    }

    public void setWorkout(long id) {
        this.workoutId = id;
    }
}
```

We're using this class in the `onStart()` method.

The `getView()` method gets the fragment's root View. We can then use this to get references to the workout title and description text views.



You should always call up to the superclass when you implement any fragment lifecycle methods.

As we said on the previous page, fragments are distinct from activities, and therefore don't have all the methods that an activity does. Fragments don't include a `findViewById()` method, for instance. To get a reference to a fragment's views, we first have to get a reference to the fragment's root view using the `getView()` method, and use that to find its child views.

Now that we've got the fragment to update its views, let's take the app for a test drive.



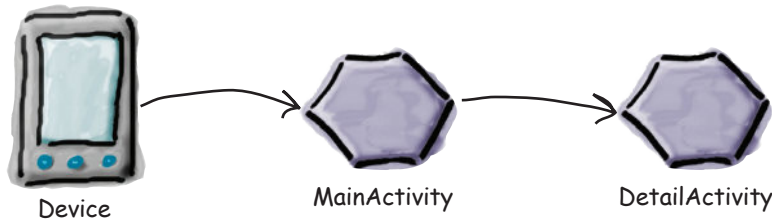
WorkoutDetailFragment
WorkoutListFragment
Coordinate fragments

What happens when the code runs

Before we run the app, let's go through what happens when the code runs.

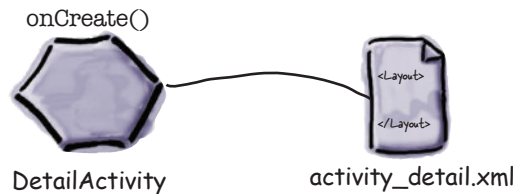
1 When the app is launched, MainActivity gets created.

The user clicks on the button in MainActivity to start DetailActivity.



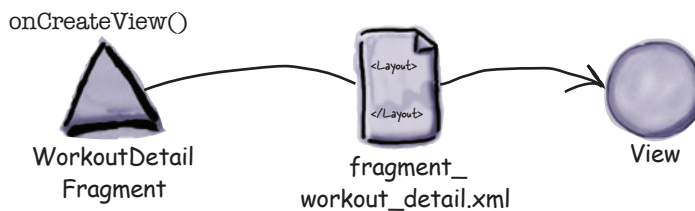
2 DetailActivity's onCreate() method runs.

The onCreate() method specifies that *activity_detail.xml* should be used for DetailActivity's layout. *activity_detail.xml* includes a <fragment> element with an ID of detail_frag that refers to the fragment WorkoutDetailFragment.



3 WorkoutDetailFragment's onCreateView() method runs.

The onCreateView() method specifies that *fragment_workout_detail.xml* should be used for WorkoutDetailFragment's layout. It inflates the layout to a View object.

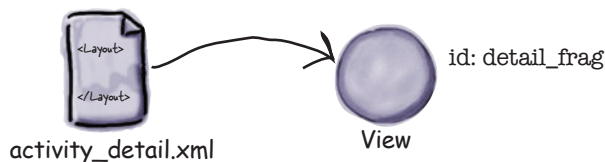


The story continues

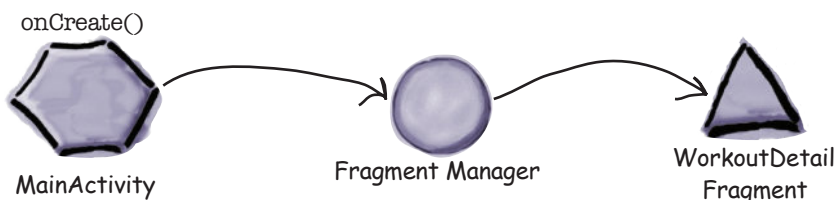


WorkoutDetailFragment
WorkoutListFragment
 Coordinate fragments

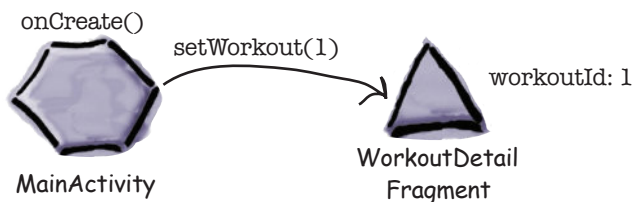
- 4 **activity_detail.xml's Views are inflated to View Java objects.**
 MainActivity uses WorkoutDetailFragment's View object in place of the <fragment> element in its layout's XML, and gives it an ID of detail_frag.



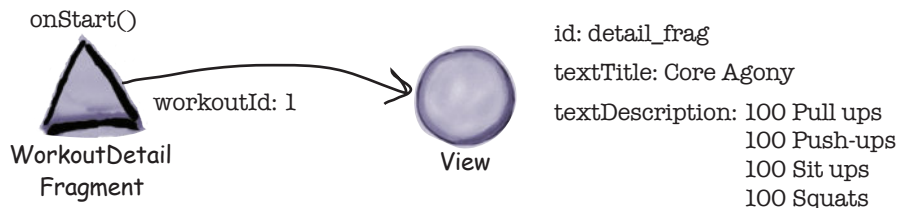
- 5 **MainActivity's onCreate() method continues to run.**
 MainActivity gets a reference to WorkoutDetailFragment by asking the fragment manager for the fragment with an ID of detail_frag.



- 5 **MainActivity calls WorkoutDetailFragment's setWorkout() method.**
 MainActivity passes WorkoutDetailFragment a workout ID of 1. The fragment sets its workoutId variable to 1.



- 6 **The fragment uses the value of the workout ID in its onStart() method to set the values of its views.**



Let's take the app for a test drive.



Test drive the app

When we run the app, `MainActivity` is launched.

When we click on `MainActivity`'s button, it starts `DetailActivity`. `DetailActivity` contains `WorkoutDetailFragment`, and the fragment displays details of the Core Agony workout.



WorkoutDetailFragment
WorkoutListFragment
 Coordinate fragments



there are no Dumb Questions

Q: Why can't an activity get a fragment by calling the `findViewById()` method?

A: Because `findViewById()` always returns a `View` object and, surprisingly, fragments aren't views.

Q: Why isn't `findFragmentById()` an activity method like `findViewById()` is?

A: That's a good question. Fragments weren't available before API 11, so it uses the fragment manager as a way to add a whole bunch of useful code for managing fragments, without having to pack lots of extra code into the activity base class.

Q: Why don't fragments have a `findViewById()` method?

A: Because fragments aren't views or activities. Instead, you need to use the fragment's `getView()` method to get a reference to the fragment's root view, and then call the view's `findViewById()` method to get its child views.

Q: Activities need to be registered in `AndroidManifest.xml` so that the app can use them. Do fragments?

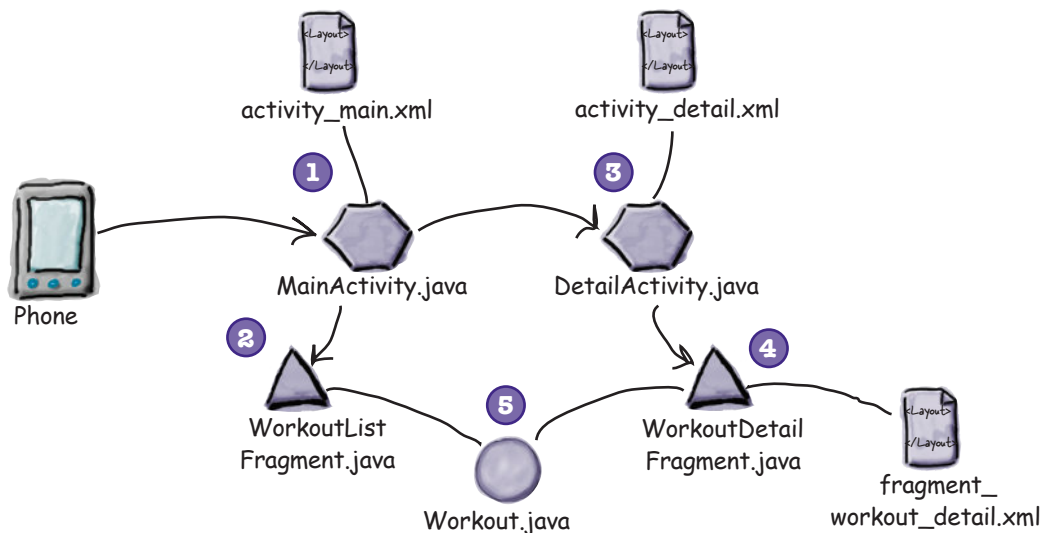
A: No. Activities need to be registered in `AndroidManifest.xml`, but fragments don't.



Where we've got to

Here's a reminder of the structure of the app, and what we want it to do:

- 1 **When the app gets launched, it starts MainActivity.**
MainActivity uses *activity_main.xml* for its layout, and contains a fragment called *WorkoutListFragment*.
- 2 **WorkoutListFragment displays a list of workouts.**
- 3 **When the user clicks on one of the workouts, DetailActivity starts.**
DetailActivity uses *activity_detail.xml* for its layout, and contains a fragment called *WorkoutDetailFragment*.
- 4 **WorkoutDetailFragment uses *fragment_workout_detail.xml* for its layout.**
It displays the details of the workout the user has selected.
- 5 **WorkoutListFragment and WorkoutDetailFragment get their workout data from *Workout.java*.**
Workout.java contains an array of *Workouts*.



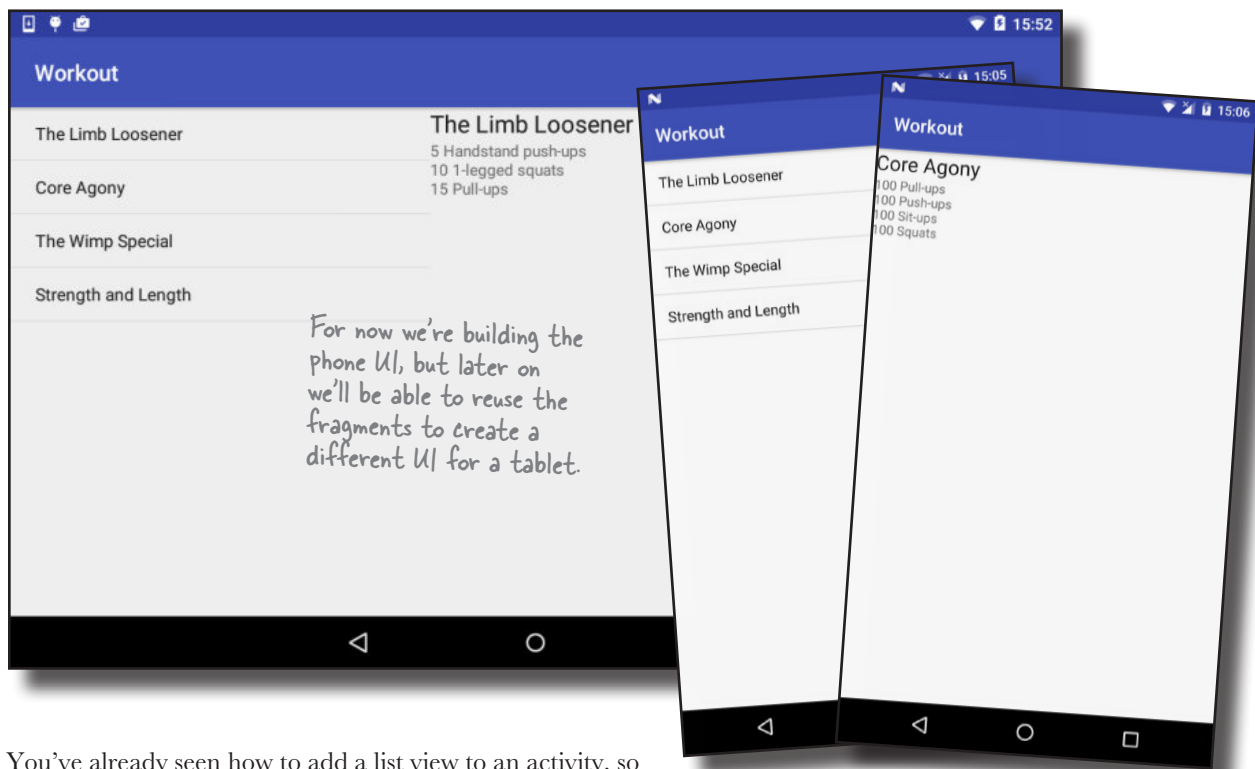
So far we've created both activities and their layouts, *WorkoutDetailFragment.java* and its layout, and also *Workout.java*. The next thing we need to look at is *WorkoutListFragment*.

We need to create a fragment with a list

We need to create a second fragment, `WorkoutListFragment`, that contains a list of the different workouts that the user can choose from. Using a fragment for this means that later on, we'll be able to use it to create different user interfaces for phones and tablets.



WorkoutDetailFragment
WorkoutListFragment
Coordinate fragments



You've already seen how to add a list view to an activity, so we could do something similar for the fragment. But rather than create a new fragment with a layout that contains a single list view, we're going to use a different approach that involves a new type of fragment called a **list fragment**.

WorkoutDetailFragment
 WorkoutListFragment
 Coordinate fragments



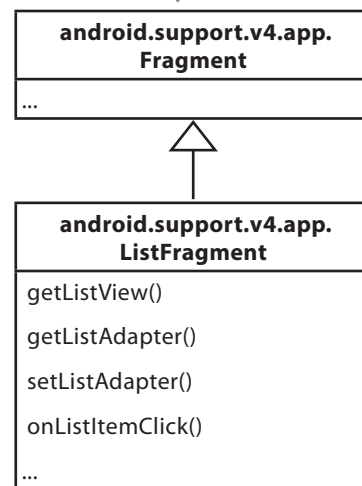
A list fragment is a fragment that contains only a list

A list fragment is a type of fragment that specializes in working with a list. It's automatically bound to a list view, so you don't need to create one yourself. Here's what one looks like:

A list fragment comes complete with its own list view so you don't need to add it yourself. You just need to provide the list fragment with data.



ListFragment is a subclass of Fragment.



There are a couple of major advantages in using a list fragment to display categories of data:



You don't need to create your own layout.

List fragments define their own layout programmatically, so there's no XML layout for you to create or maintain. The layout the list fragment generates includes a single list view. You access this list view in your fragment code using the list fragment's `getListView()` method. You need this in order to specify what data should be displayed in the list view.



You don't have to implement your own event listener.

The `ListFragment` class automatically implements an event listener that listens for when items in the list view are clicked. Instead of creating your own event listener and binding it to the list view, you just need to implement the list fragment's `onItemClickListener()` method. This makes it easier to get your fragment to respond when the user clicks on items in the list view. You'll see this in action later on.

So what does the list fragment code look like?

A list fragment is a type of fragment that specializes in working with a list view. It has a default layout that contains the list view.

How to create a list fragment

You add a list fragment to your project in the same way you add a normal fragment. Highlight the `com.hfad.workout` package in the `app/src/main/java` folder, then go to File→New...→Fragment→Fragment (Blank). Name the fragment “WorkoutListFragment”, and then uncheck the options to create layout XML, and also the options to include fragment factory methods and interface callbacks (list fragments define their own layouts programmatically, so you don’t need Android Studio to create one for you). When you click on the Finish button, Android Studio creates a new list fragment in a file called `WorkoutListFragment.java` in the `app/src/main/java` folder.

Here’s what the basic code looks like to create a list fragment. As you can see, it’s very similar to that of a normal fragment. Replace the code in `WorkoutListFragment.java` with the code below:

```
package com.hfad.workout;

import android.os.Bundle;
import android.support.v4.app.ListFragment;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;

public class WorkoutListFragment extends ListFragment {

    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup container,
        Bundle savedInstanceState) {
        return super.onCreateView(inflater, container, savedInstanceState);
    }
}
```

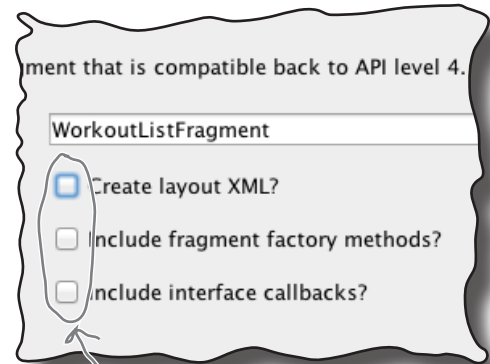
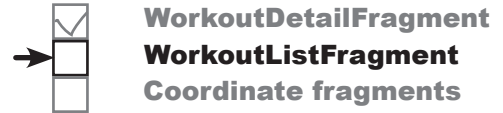
The activity needs to extend ListFragment, not Fragment.

Calling the superclass onCreateView() method gives you the default layout for the ListFragment.

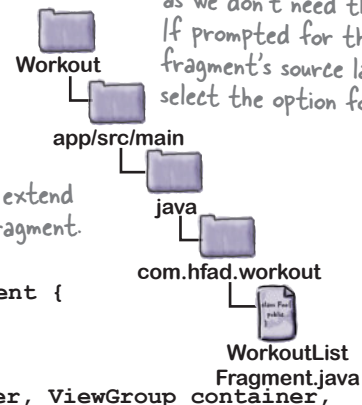
The above code creates a basic list fragment called `WorkoutListFragment`. As it’s a list fragment, it needs to extend the `ListFragment` class rather than `Fragment`.

The `onCreateView()` method is optional. It gets called when the fragment’s view gets created. We’re including it in our code as we want to populate the fragment’s list view with data as soon as it gets created. If you don’t need your code to do anything at this point, you don’t need to include the `onCreateView()` method.

The next thing we need to do is add data to the list view in the `onCreateView()` method.



Uncheck these options, as we don’t need them. If prompted for the fragment’s source language, select the option for Java.



there are no Dumb Questions

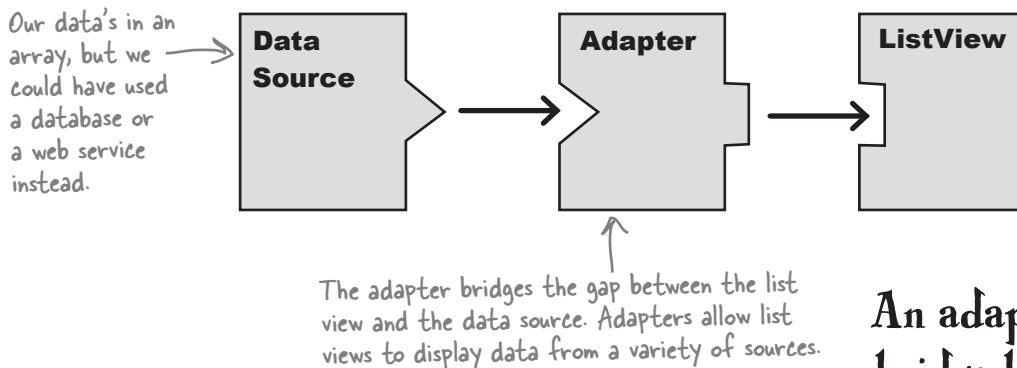
Q: When we create a list fragment, why do we choose the option for Fragment (Blank) instead of Fragment (List)?

A: The Fragment (List) option produces code that’s more complex, most of which we don’t need to use. The code generated by the Fragment (Blank) is simpler.



Adapters revisited

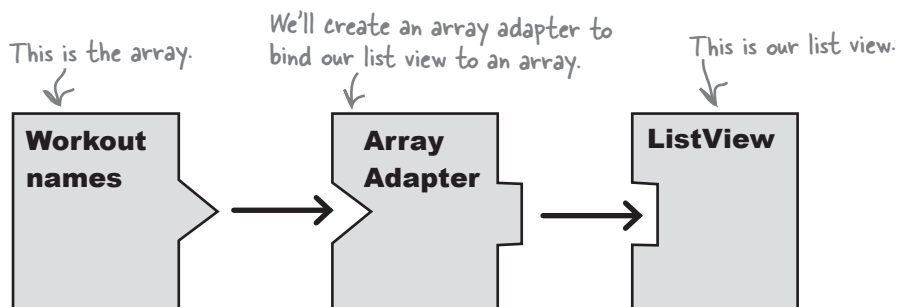
As we said in Chapter 7, you can connect data to a list view using an adapter. The adapter acts as a bridge between the data and the list view. This is still the case when your list view is in a fragment, or a list fragment:



We want to supply the list view in `WorkoutListFragment` with an array of workout names, so we'll use an array adapter to bind the array to the list view as before. As you may recall, an array adapter is a type of adapter that's used to bind arrays to views. You can use it with any subclass of the `AdapterView` class, which means you can use it with both list views and spinners.

In our case, we're going to use an array adapter to display an array of data from the `Workout` class in the list view.

An adapter acts as a bridge between a view and a data source. An array adapter is a type of adapter that specializes in working with arrays.



We'll see how this works on the next page.

Our previous array adapter

As we said in Chapter 7, you use an array adapter by initializing it and attaching it to the list view.

To initialize the array adapter, you first specify what type of data is contained in the array you want to bind to the list view. You then pass it three parameters: a `Context` (usually the current activity), a layout resource that specifies how to display each item in the array, and the array itself.

Here's the code we used in Chapter 7 to create an array adapter to displays `Drink` data from the `Drink.drinks` array:

```
ArrayAdapter<Drink> listAdapter = new ArrayAdapter<>(
```

```

    this,
    android.R.layout.simple_list_item_1,
    Drink.drinks);

```

The current context. In our Chapter 7 scenario, it was the current activity. → *this*,

This is a built-in layout resource. It tells the array adapter to display each item in the array in a single text view. ← `android.R.layout.simple_list_item_1`,

← The array `Drink.drinks`);

There's a big difference between the situation we had back in Chapter 7 and the situation we have now. Back in Chapter 7, we used the array adapter to display data in an activity. But this time, we want to display data in a fragment. What difference does this make?

A fragment isn't a subclass of Context

As you saw earlier in the book, the `Activity` class is a subclass of the `Context` class. This means that all of the activities you create have access to global information about the app's environment.

But the `Fragment` class *isn't* a subclass of the `Context` class. It has no access to global information, and you can't use `this` to pass the current context to the array adapter. Instead, you need to get the current context in some other way.

One way is to use another object's `getContext()` method to get a reference to the current context. If you create the adapter in the fragment's `onCreateView()` method, you can use the `getContext()` method of the `onCreateView()` `LayoutInflater` parameter to get the context instead.

Once you've created the adapter, you bind it to the `ListView` using the fragment's `setListAdapter()` method:

```
setListAdapter(listAdapter);
```

We'll show you the full code on the next page.



WorkoutDetailFragment
WorkoutListFragment
 Coordinate fragments

The updated WorkoutListFragment code



We've updated our `WorkoutListFragment.java` code so that it populates the list view with the names of the workouts. Apply these changes to your code, then save your changes:

```
package com.hfad.workout;

import android.os.Bundle;
import android.support.v4.app.ListFragment;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;
import android.widget.AdapterView;

public class WorkoutListFragment extends ListFragment {
```

```
    @Override
```

```
    public View onCreateView(LayoutInflater inflater, ViewGroup container,
        Bundle savedInstanceState) {
```

```
        String[] names = new String[Workout.workouts.length];
        for (int i = 0; i < names.length; i++) {
            names[i] = Workout.workouts[i].getName();
        }
```

Create an array adapter.

Create a String array of the workout names.

```
        ArrayAdapter<String> adapter = new ArrayAdapter<>(
```

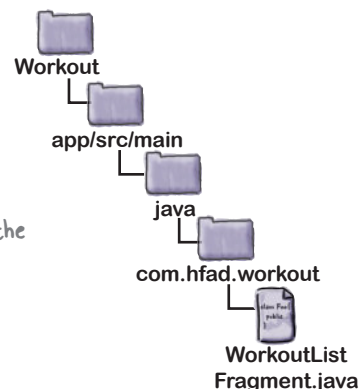
Get the context from the layout inflater. → `inflater.getContext()`, `android.R.layout.simple_list_item_1`, `names`;

```
        setListAdapter(adapter); ← Bind the array adapter to the list view.
```

```
        return super.onCreateView(inflater, container, savedInstanceState);
```

```
    }
```

```
}
```



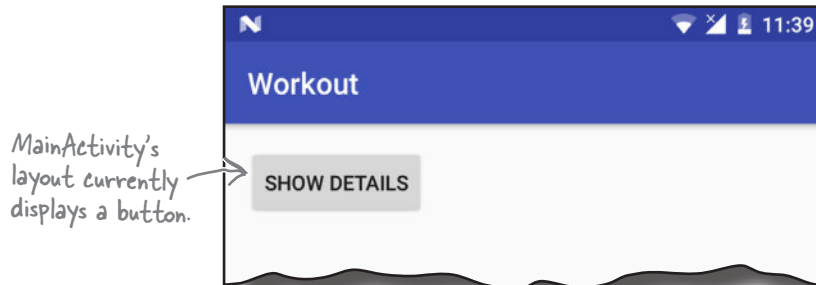
Now that the `WorkoutListFragment` contains a list of workouts, let's add it to `MainActivity`.



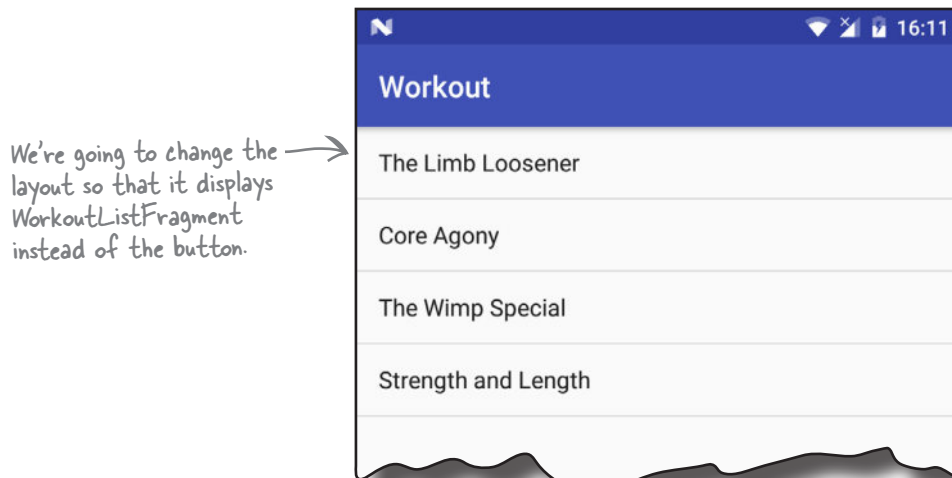
WorkoutDetailFragment
WorkoutListFragment
Coordinate fragments

Display `WorkoutListFragment` in the `MainActivity` layout

We're going to add our new `WorkoutListFragment` to `MainActivity`'s layout `activity_main.xml`. The layout currently displays a button that we're using to navigate from `MainActivity` to `DetailActivity`:



We want to remove the button, and display `WorkoutListFragment` in its place. Here's what the new version of the layout will look like:



What will the code be like? Have a go at the exercise on the next page.



Layout Magnets

Somebody put a new version of *activity_main.xml* on our fridge door. Unfortunately some of the magnets fell off when we shut the door too hard. Can you piece the layout back together again? (You won't need to use all of the magnets below.)

The layout needs to display `WorkoutListFragment`.

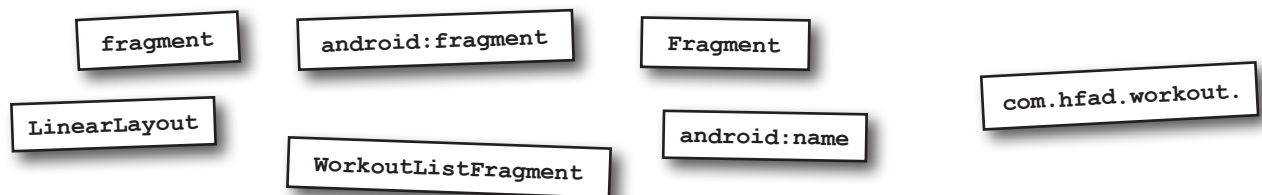
```
<?xml version="1.0" encoding="utf-8"?>

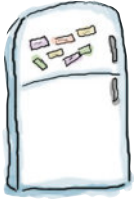
<..... xmlns:android="http://schemas.android.com/apk/res/android"

..... ="....."

android:layout_width="match_parent"

android:layout_height="match_parent"/>
```





Layout Magnets Solution

Somebody put a new version of `activity_main.xml` on our fridge door. Unfortunately some of the magnets fell off when we shut the door too hard. Can you piece the layout back together again? (You won't need to use all of the magnets below.)

The layout needs to display `WorkoutListFragment`.

```
<?xml version="1.0" encoding="utf-8"?>
```

```
<fragment xmlns:android="http://schemas.android.com/apk/res/android"
    android:name="com.hfad.workout.WorkoutListFragment"
    android:layout_width="match_parent"
    android:layout_height="match_parent"/>
```

Annotations for the XML code:

- An arrow points to the `<fragment>` tag with the text: "You declare a fragment with the `<fragment>` element."
- An arrow points to the `xmlns:android="http://schemas.android.com/apk/res/android"` attribute.
- An arrow points to the `android:name="com.hfad.workout.WorkoutListFragment"` attribute with the text: "You need to give the full name of the fragment."

Annotations for the magnets:

- An arrow points from the `LinearLayout` magnet to the text: "You didn't need to use these magnets."
- An arrow points from the `android:fragment` magnet to the text: "You didn't need to use these magnets."
- An arrow points from the `Fragment` magnet to the text: "You didn't need to use these magnets."



The code for activity_main.xml

As we want MainActivity's layout to only contain a single fragment, we can replace nearly all of the code we currently have.

Here's the updated code for *activity_main.xml*. As you can see, it's much shorter than the original version. Update your version of the code to reflect our changes.

```

<?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="com.hfad.workout.MainActivity">

  <Button
  android:layout_width="wrap_content"
  android:layout_height="wrap_content"
  android:onClick="onShowDetails"
  android:text="@string/details_button" />
</LinearLayout>

```

Our layout only contains a single fragment, so we can get rid of the LinearLayout.

We no longer need this button.

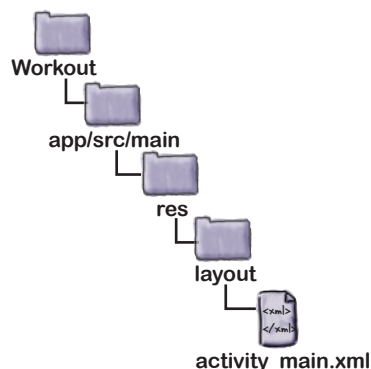
Here's the fragment.

```

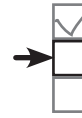
<fragment xmlns:android="http://schemas.android.com/apk/res/android"
  android:name="com.hfad.workout.WorkoutListFragment"
  android:layout_width="match_parent"
  android:layout_height="match_parent"/>

```

We'll go through what happens when this code runs over the next couple of pages.



What happens when the code runs

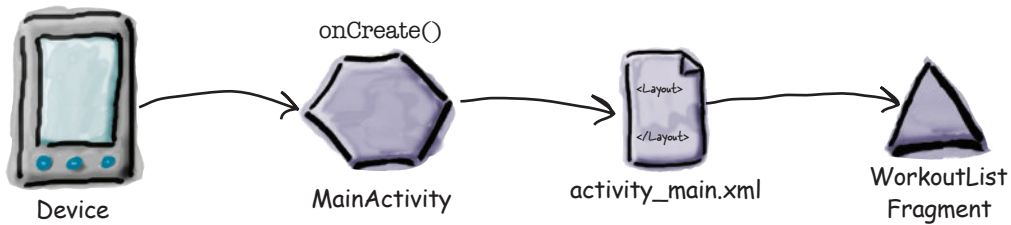


WorkoutDetailFragment
WorkoutListFragment
Coordinate fragments

Here's a runthrough of what happens when we run the app.

1 When the app is launched, MainActivity gets created.

MainActivity's onCreate() method runs. This specifies that *activity_main.xml* should be used for MainActivity's layout. *activity_main.xml* includes a <fragment> element that refers to WorkoutListFragment.



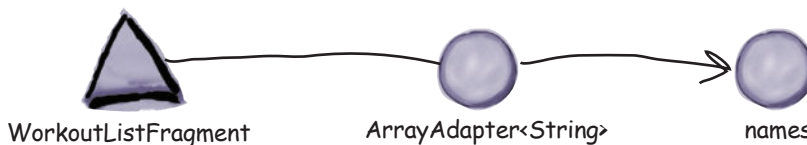
2 WorkoutListFragment is a ListFragment, so it uses a ListView as its layout.



3 WorkoutListFragment creates an ArrayAdapter<String>, an array adapter that deals with arrays of String objects.



4 The ArrayAdapter<String> retrieves data from the names array.

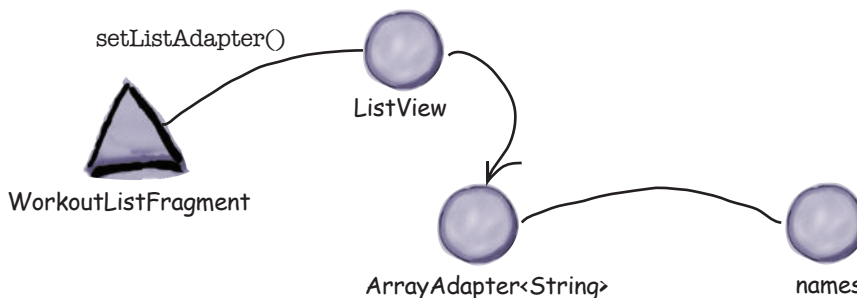




The story continues

- 5 **WorkoutListFragment attaches the array adapter to the ListView using the `setListAdapter()` method.**

The list view uses the array adapter to display a list of the workout names.



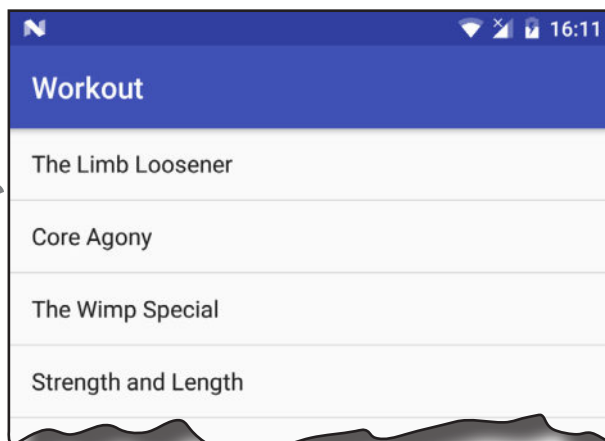
Test drive the app

When we run the app, MainActivity gets launched.

MainActivity's layout contains the fragment WorkoutListFragment. The fragment contains a list of the workout names, and this is displayed in the activity.



Here's a list of all the workout titles from the Workout class.



That looks great, but when we click on one of the workouts, nothing happens. We need to update the code so that when we click on one of the workouts, details of that workout are displayed.

Connect the list to the detail



WorkoutDetailFragment
WorkoutListFragment
Coordinate fragments

There are a few ways that we can start `DetailActivity` and display the details of the workout that was clicked. We'll use this technique:

- 1 Add code to `WorkoutListFragment` that waits for a workout to be clicked.
- 2 When that code runs, call some code in `MainActivity.java` that will start `DetailActivity`, passing it the ID of the workout.
- 3 Get `DetailActivity` to pass the ID to `WorkoutDetailFragment` so that the fragment can display details of the correct workout.

We don't want to write code in `WorkoutListFragment` that talks *directly* to `MainActivity`. Can you think why?

The answer is *reuse*. We want our fragments to know as little as possible about the environment that contains them so that we can reuse them elsewhere. The more a fragment needs to know about the activity using it, the less reusable it is.



Wait a minute! You're saying you don't want the fragment to know about the activity that contains it? But I thought you said the fragment has to call code in `MainActivity`. Won't that mean we can't use it in another activity?

We need to use an *interface* to decouple the fragment from the activity.

We have two objects that need to talk to each other—the fragment and the activity—and we want them to talk without one side knowing too much about the other. The way we do that in Java is with an *interface*. When we define an interface, we're saying *what the minimum requirements are for one object to talk usefully to another*. That means that we'll be able to get the fragment to talk to any activity, so long as that activity implements the interface.

We need to decouple the fragment by using an interface

We're going to create an interface called **Listener**. If `MainActivity` implements the interface, `WorkoutListFragment` will be able to tell `MainActivity` when one of its items has been clicked. To do this, we'll need to make changes to `WorkoutListFragment` and `MainActivity`.

What `WorkoutListFragment` needs to do

We'll start with the code for `WorkoutListFragment`. There are a few changes we need to make, in this order.

- 1 **Define the interface.**
We'll define the listener's interface in `WorkoutListFragment`. We're defining the interface here, as its purpose is to allow `WorkoutListFragment` to communicate with any activity.
- 2 **Register the listener (in this case `MainActivity`) when `WorkoutListFragment` gets attached to it.**
This will give `WorkoutListFragment` a reference to `MainActivity`.
- 3 **Tell the listener when an item gets clicked.**
`MainActivity` will then be able to respond to the click.

You need to go through similar steps to these whenever you have a fragment that needs to communicate with the activity it's attached to.

We'll go through each change individually, then show you the full code.

1. Define the listener interface

We want any activities that implement the listener interface to respond to item clicks, so we'll define a method for the interface, `itemClicked()`. The `itemClicked()` method has one parameter, the ID of the item that's clicked.

Here's the interface:

```
interface Listener {
    void itemClicked(long id);
};
```

We'll call the interface `Listener`.

Any activities that implement the `Listener` interface must include this method. We'll use it to get the activity to respond to items in the fragment being clicked.

Next we'll look at how you register the listener on the next page.

2. Register the listener

We need to save a reference to the activity `WorkoutListFragment` gets attached to. This activity will implement the `Listener` interface, so we'll add the following private variable to `WorkoutListFragment`:

```
private Listener listener;
```

We need to set this variable when `WorkoutListFragment` gets attached to an activity. If you look back at the fragment lifecycle, when a fragment gets attached to an activity, the fragment's `onAttach()` method is called. We'll use this method to set the value of the listener:

```
public void onAttach(Context context) {
    super.onAttach(context);
    this.listener = (Listener) context;
}
```

This is the context (in this case, the activity) the fragment is attached to.

3. Respond to clicks

When an item in `WorkoutListFragment` gets clicked, we want to call the listener's `itemClicked()` method. This is the method we defined in the interface on the previous page. But how can we tell when an item's been clicked?

Whenever an item gets clicked in a list fragment, the list fragment's `onListItemClick()` method gets called. Here's what it looks like:

```
public void onListItemClick(ListView listView,
    View itemView,
    int position,
    long id) {
    //Do something
}
```

The list view

The item in the list view that was clicked, its position, and its ID

The `onListItemClick()` method has four parameters: the list view, the item in the list that was clicked, its position, and the row ID of the underlying data. This means we can use the method to pass the listener the ID of the workout the user clicked on:

```
public void onListItemClick(ListView listView, View itemView, int position, long id) {
    if (listener != null) {
        listener.itemClicked(id);
    }
}
```

Call the itemClicked() method in the activity, passing it the ID of the workout the user selected.



WorkoutDetailFragment
WorkoutListFragment
Coordinate fragments

The code for WorkoutListFragment.java



Here's the full code for *WorkoutListFragment.java* code (apply these changes to your code, then save your work):

```

package com.hfad.workout;

import android.os.Bundle;
import android.support.v4.app.ListFragment;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;
import android.widget.AdapterView;
import android.widget.AdapterView.OnItemClickListener;
import android.widget.ArrayAdapter;
import android.widget.ListView;
import android.content.Context;

public class WorkoutListFragment extends ListFragment {

    static interface Listener {
        void itemClicked(long id);
    };

    private Listener listener;

    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup container,
        Bundle savedInstanceState) {
        String[] names = new String[Workout.workouts.length];
        for (int i = 0; i < names.length; i++) {
            names[i] = Workout.workouts[i].getName();
        }
        ArrayAdapter<String> adapter = new ArrayAdapter<>(
            inflater.getContext(), android.R.layout.simple_list_item_1,
            names);
        setListAdapter(adapter);
        return super.onCreateView(inflater, container, savedInstanceState);
    }

    @Override
    public void onAttach(Context context) {
        super.onAttach(context);
        this.listener = (Listener) context;
    }

    @Override
    public void onItemClick(ListView listView, View itemView, int position, long id) {
        if (listener != null) {
            listener.itemClicked(id);
        }
    }
}

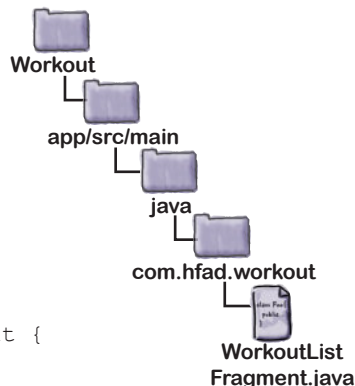
```

Import these classes.

Add the listener to the fragment.

This is called when the fragment gets attached to the activity. Remember, the Activity class is a subclass of Context.

Tell the listener when an item in the ListView is clicked.



MainActivity needs to implement the interface

Next we need to make MainActivity implement the Listener interface we just created. The interface specifies an `itemClicked()` method, so we'll make the method start `DetailActivity`, passing it the ID of the workout the user selected.

Here's the full code for `MainActivity.java`. Update your code so that it matches ours.

```

package com.hfad.workout;

import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.view.View;
import android.content.Intent;

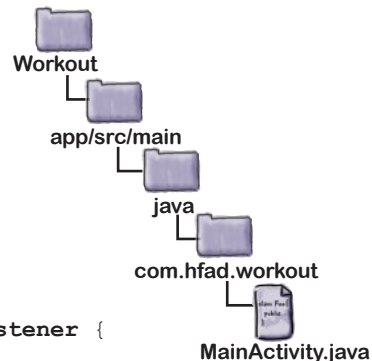
public class MainActivity extends AppCompatActivity
    implements WorkoutListFragment.Listener {

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

    public void onShowDetails(View view) {
        Intent intent = new Intent(this, DetailActivity.class);
        startActivity(intent);
    }

    @Override
    public void itemClicked(long id) {
        Intent intent = new Intent(this, DetailActivity.class);
        intent.putExtra(DetailActivity.EXTRA_WORKOUT_ID, (int)id);
        startActivity(intent);
    }
}

```



Implement the listener interface defined in `WorkoutListFragment`.

This is the method called by MainActivity's button. We've removed the button, so we no longer need this method.

This method is defined by the interface, so we need to implement it.

Pass the ID of the workout to `DetailActivity`. `EXTRA_WORKOUT_ID` is the name of a constant we'll define in `DetailActivity`.

Those are all the changes we need to make to `MainActivity`. There's just one more code change we need to make to our app.



DetailActivity needs to pass the ID to WorkoutDetailFragment

So far, `WorkoutListFragment` passes the ID of the workout that was clicked to `MainActivity`, and `MainActivity` passes it to `DetailActivity`. We need to make one more change, which is to pass the ID from `DetailActivity` to `WorkoutDetailActivity`.

Here's the updated code for `DetailActivity` that does this. Update your version of `DetailActivity.java` to reflect our changes:

```
package com.hfad.workout;

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

public class DetailActivity extends AppCompatActivity {

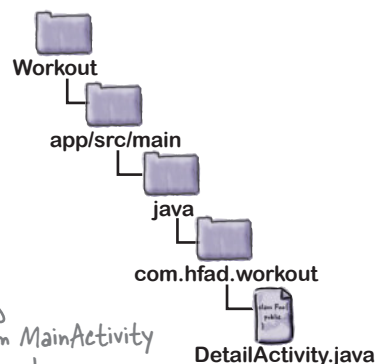
    public static final String EXTRA_WORKOUT_ID = "id";

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_detail);
        WorkoutDetailFragment frag = (WorkoutDetailFragment)
            getSupportFragmentManager().findFragmentById(R.id.detail_frag);
        frag.setWorkout(1);
        int workoutId = (int) getIntent().getExtras().get(EXTRA_WORKOUT_ID);
        frag.setWorkout(workoutId);
    }
}
```

We're no longer hardcoding an ID of 1, so remove this line.

We're using a constant to pass the ID from `MainActivity` to `DetailActivity` to avoid hardcoding this value.

Get the ID from the intent, and pass it to the fragment via its `setWorkout()` method.



Over the next couple of pages we'll examine what happens when the code runs.

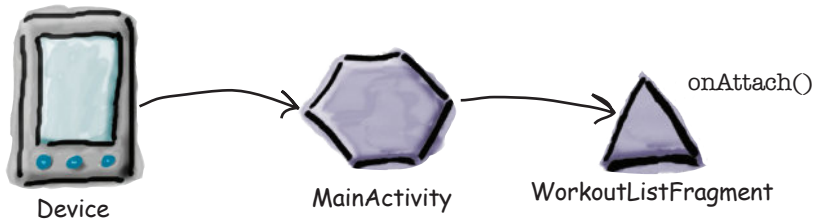
What happens when the code runs



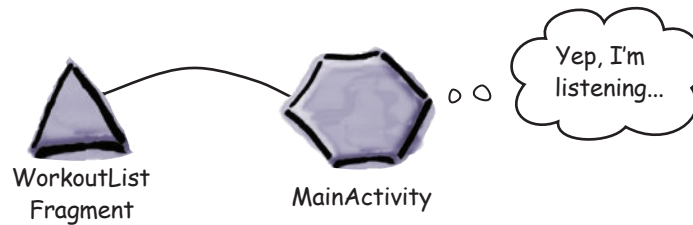
WorkoutDetailFragment
WorkoutListFragment
Coordinate fragments

Here's a runthrough of what happens when we run the app.

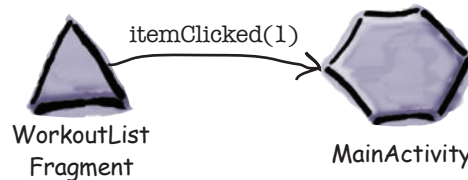
- 1 **When the app is launched, MainActivity gets created.** WorkoutListFragment is attached to MainActivity, and WorkoutListFragment's `onAttach()` method runs.



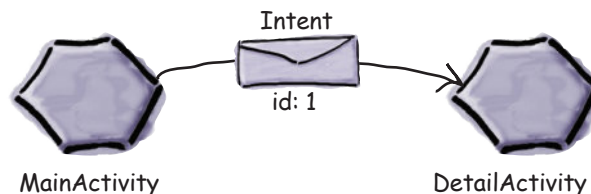
- 2 **WorkoutListFragment registers MainActivity as a Listener.**



- 3 **When an item is clicked in WorkoutListFragment, the fragment's `onListItemClick()` method is called.** This calls MainActivity's `itemClicked()` method, passing it the ID of the workout that was clicked, in this example 1.



- 4 **MainActivity's `itemClicked()` method starts DetailActivity, passing it the value of the workout ID in an intent.**



The story continues...



WorkoutDetailFragment
WorkoutListFragment
Coordinate fragments

- 5 **DetailActivity calls WorkoutDetailFragment's setWorkout() method, passing it the value of the workout ID.**

WorkoutDetailFragment uses the workout ID, in this case 1, to display the workout title and description in its views.



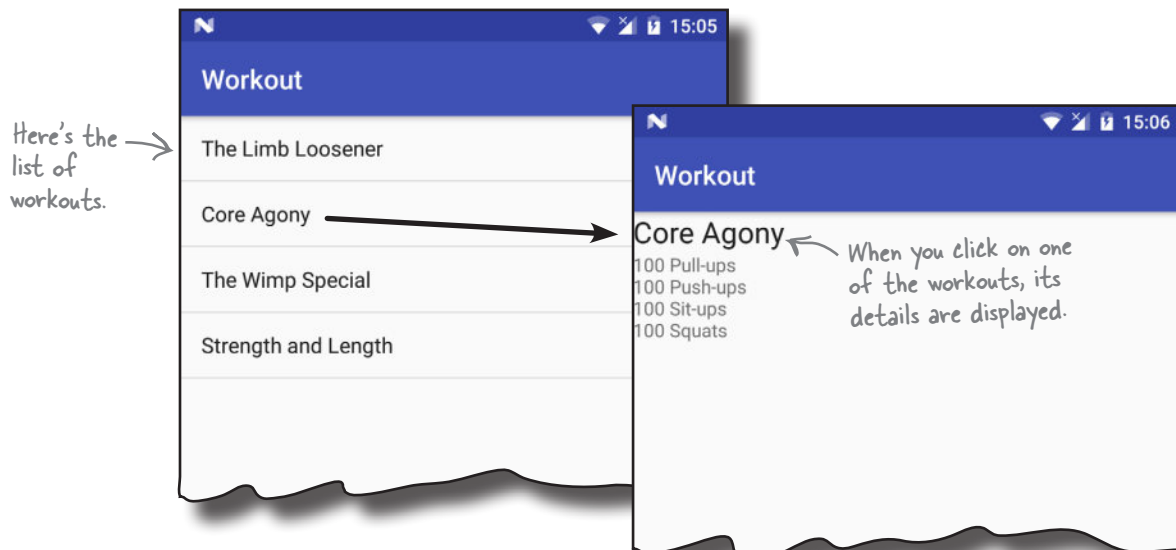
Test drive the app

When we run the app, MainActivity gets launched. It displays a list of workouts in its fragment, WorkoutListFragment.

When you click on one of the workouts, DetailActivity is displayed. It shows details of the workout that we selected.



WorkoutDetailFragment
WorkoutListFragment
Coordinate fragments



That's everything we need to do to use the fragments we've created in a user interface for a phone. In the next chapter, you'll see how to reuse the fragments, and create a different user interface that will work better for tablets.



Your Android Toolbox

You've got Chapter 9 under your belt and now you've added fragments to your toolbox.

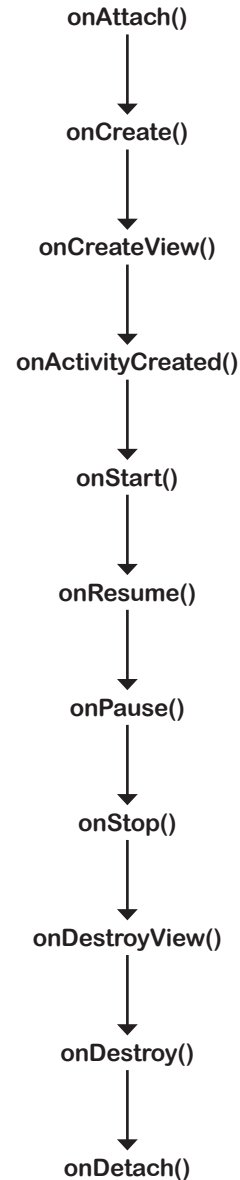


BULLET POINTS

- A fragment is used to control part of a screen. It can be reused across multiple activities.
- A fragment has an associated layout.
- The `onCreateView()` method gets called each time Android needs the fragment's layout.
- Add a fragment to an activity's layout using the `<fragment>` element and adding a name attribute.
- The fragment lifecycle methods tie in with the states of the activity that contains the fragment.
- The `Fragment` class doesn't extend the `Activity` class or implement the `Context` class.
- Fragments don't have a `findViewById()` method. Instead, use the `getView()` method to get a reference to the root view, then call the view's `findViewById()` method.
- A list fragment is a fragment that comes complete with a `ListView`. You create one by subclassing `ListFragment`.

You can download the full code for the chapter from <https://tinyurl.com/HeadFirstAndroid>.

Fragment lifecycle methods



10 fragments for larger interfaces

Different Size, Different Interface

They're using a tablet?
Maybe we should
rethink the UI...



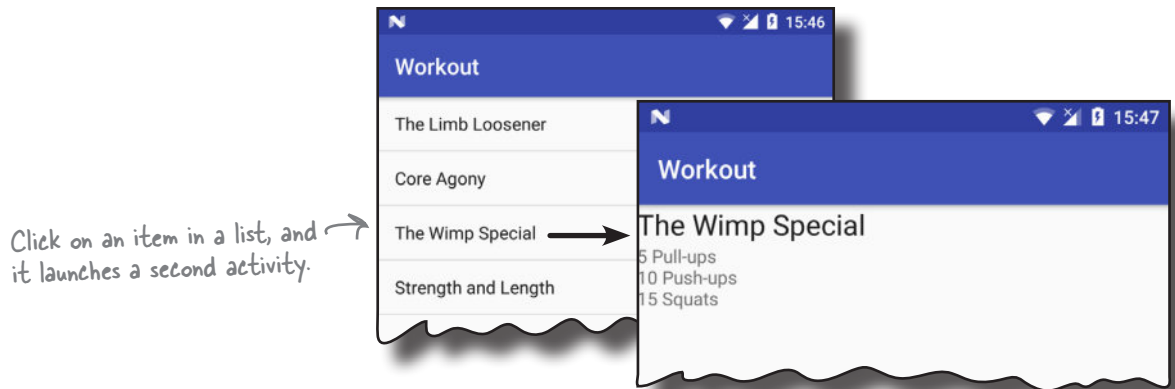
So far we've only run our apps on devices with a small screen.

But what if your users have tablets? In this chapter you'll see how to create **flexible user interfaces** by making your app **look and behave differently** depending on the device it's running on. We'll show you how to control the behavior of your app when you press the Back button by introducing you to the **back stack** and **fragment transactions**. Finally, you'll find out how to **save and restore the state** of your fragment.

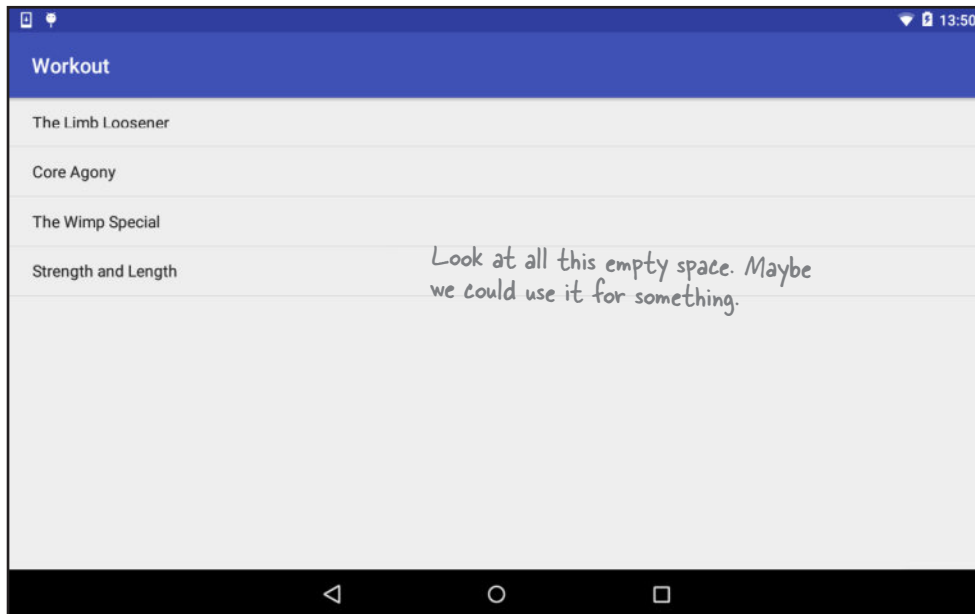
The Workout app looks the same on a phone and a tablet

In the previous chapter, we created a version of the Workout app designed to work on a phone.

As a reminder, when the app launches, it displays `MainActivity`. This contains a fragment, `WorkoutListFragment`, that displays a list of workouts. When the user clicks on one of the workouts, `DetailActivity` starts, and displays details of the workout in its fragment, `WorkoutDetailFragment`.

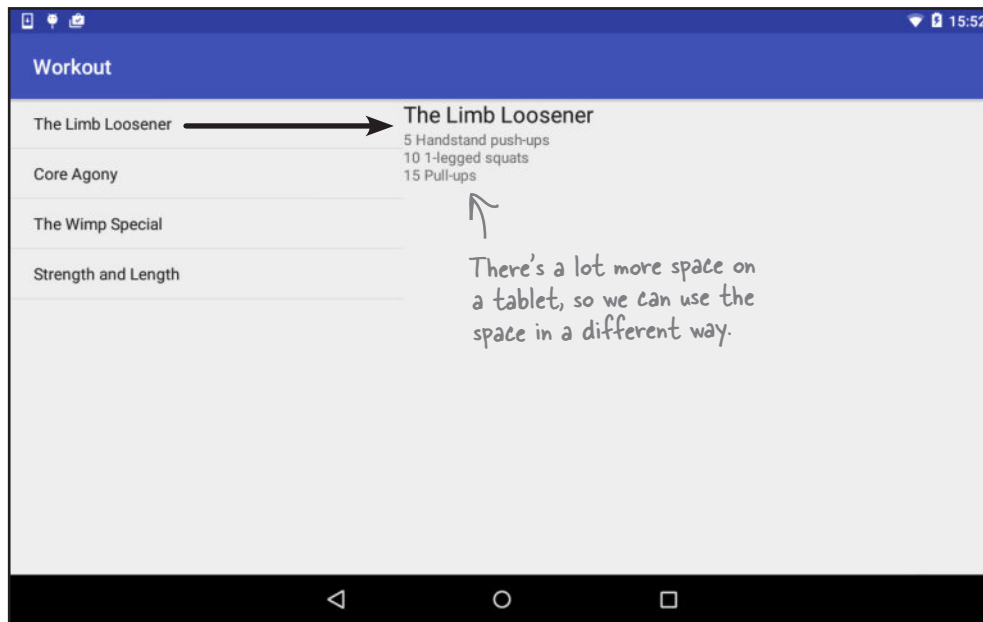


When we run the app on a tablet, the app works in exactly the same way. As the screen size is larger, however, there's lots of empty space in the user interface that we could make better use of.



Designing for larger interfaces

One way in which we could better use the empty space is to display details of the workout to the right of the list of workouts. When the user clicks on one of the workouts, details of that workout could be displayed on the same screen without us having to start a second activity:



We don't want to change our app completely though. We still want our app to work as it does currently if it's running on a phone.

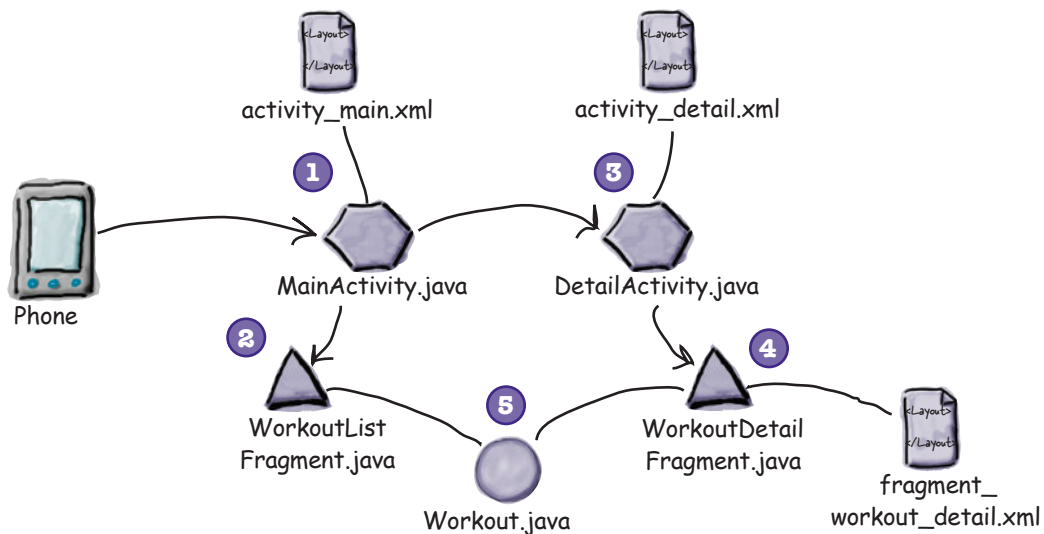
We're going to get our app to adapt to the type of device it's running on. If the app's running on a phone, we'll display details of the workout in a separate activity (this is the app's current behavior). If the app's running on a tablet, we'll display details of the workout next to the list of workouts.

Before we get started, let's remind ourselves how the app's currently structured.

The phone version of the app

The phone version of the app we built in Chapter 9 works in the following way:

- 1 When the app gets launched, it starts MainActivity.**
MainActivity uses *activity_main.xml* for its layout, and contains a fragment called *WorkoutListFragment*.
- 2 WorkoutListFragment displays a list of workouts.**
- 3 When the user clicks on one of the workouts, DetailActivity starts.**
DetailActivity uses *activity_detail.xml* for its layout, and contains a fragment called *WorkoutDetailFragment*.
- 4 WorkoutDetailFragment uses *fragment_workout_detail.xml* for its layout.**
It displays the details of the workout the user has selected.
- 5 WorkoutListFragment and WorkoutDetailFragment get their workout data from *Workout.java*.**
Workout.java contains an array of *Workouts*.

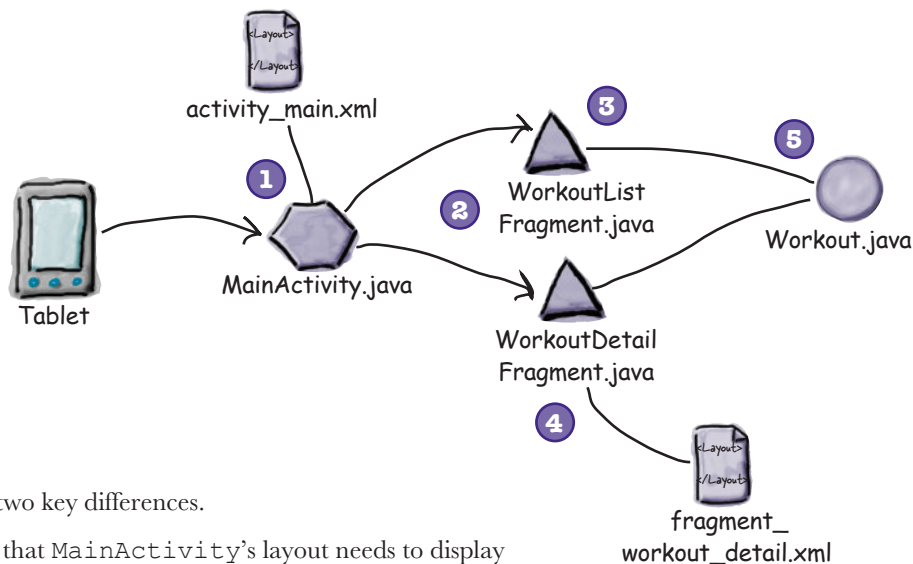


So how does it need to work differently on a tablet?

The tablet version of the app

Here's how the app will work when it runs on a tablet:

- 1 **When the app gets launched, it starts MainActivity as before.**
MainActivity uses *activity_main.xml* for its layout.
- 2 **MainActivity's layout displays two fragments, WorkoutListFragment and WorkoutDetailFragment.**
- 3 **WorkoutListFragment displays a list of workouts.**
It's a list fragment, so it has no extra layout file.
- 4 **When the user clicks on one of the workouts, its details are displayed in WorkoutDetailFragment.**
WorkoutDetailFragment uses *fragment_workout_detail.xml* for its layout.
- 5 **Both fragments get their workout data from Workout.java as before.**



There are two key differences.

The first is that MainActivity's layout needs to display both fragments, not just WorkoutListFragment.

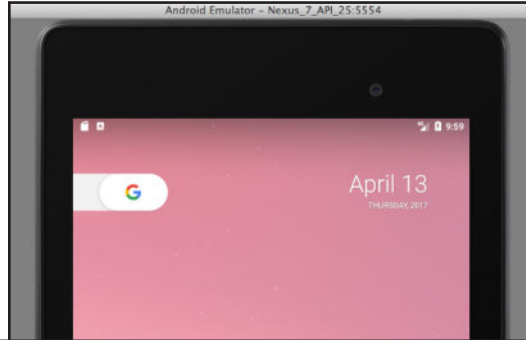
The second difference is that we no longer need to start DetailActivity when the user clicks on one of the workouts. Instead, we need to display WorkoutDetailFragment in MainActivity.

We'll go through the steps for how to change the app on the next page.

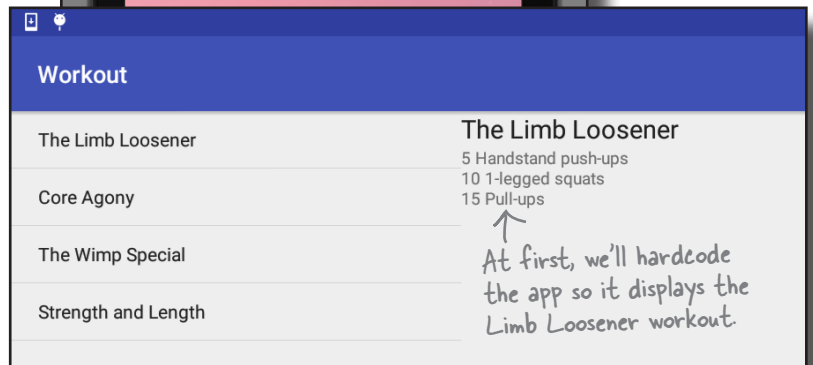
Here's what we're going to do

There are a number of steps we'll go through to change the app:

1 Create a tablet AVD (Android Virtual Device).
 We're going to create a new UI for a tablet, so we'll create a new tablet AVD to run it on. This will allow us to check how the app looks and behaves on a device with a larger screen.



2 Create a new tablet layout.
 We'll reuse the fragments we've already created in a new layout that's designed to work on devices with larger screens. We'll display details of the first workout in the first instance so that we can see the fragments side by side.



3 Display details of the workout the user selects.
 We'll update the app so that when the user clicks on one of the workouts, we'll display the details of the workout the user selected.



We're going to update the Workout app in this chapter, so open your original Workout project from Chapter 9 in Android Studio.

Create a tablet AVD

fragments for larger interfaces



Create AVD
Create layout
Show workout

Before we get into changing the app, we're going to create a new Nexus 7 AVD running API level 25 so that you can see how the app looks and behaves when it's running on a tablet. The steps are nearly the same as when you created a Nexus 5X AVD back in Chapter 1.

Open the Android Virtual Device Manager

You create AVDs using the AVD Manager. Open the AVD Manager by selecting Android on the Tools menu and choosing AVD Manager.

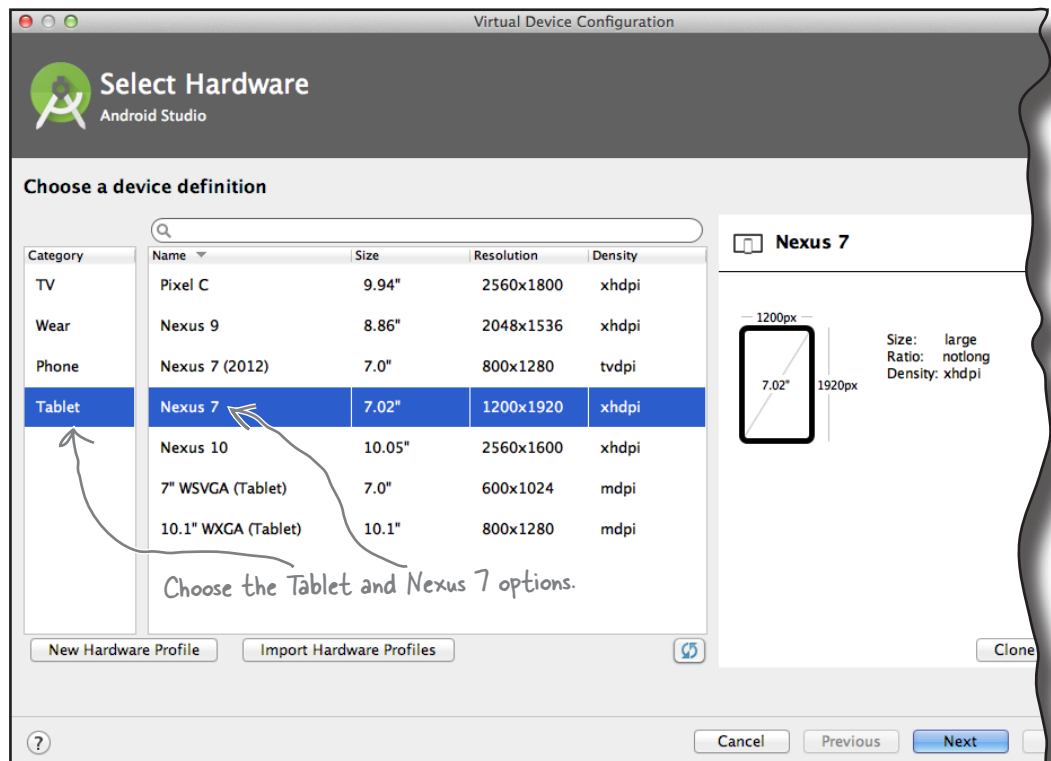
You'll be presented with a screen showing you a list of the AVDs you've already set up. Click on the Create Virtual Device button at the bottom of the screen.

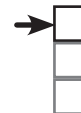


Select the hardware

On the next screen, you'll be prompted to choose a device definition, the type of device your AVD will emulate.

We're going to see what our app looks like running on a Nexus 7 tablet. Choose Tablet from the Category menu and Nexus 7 from the list. Then click the Next button.





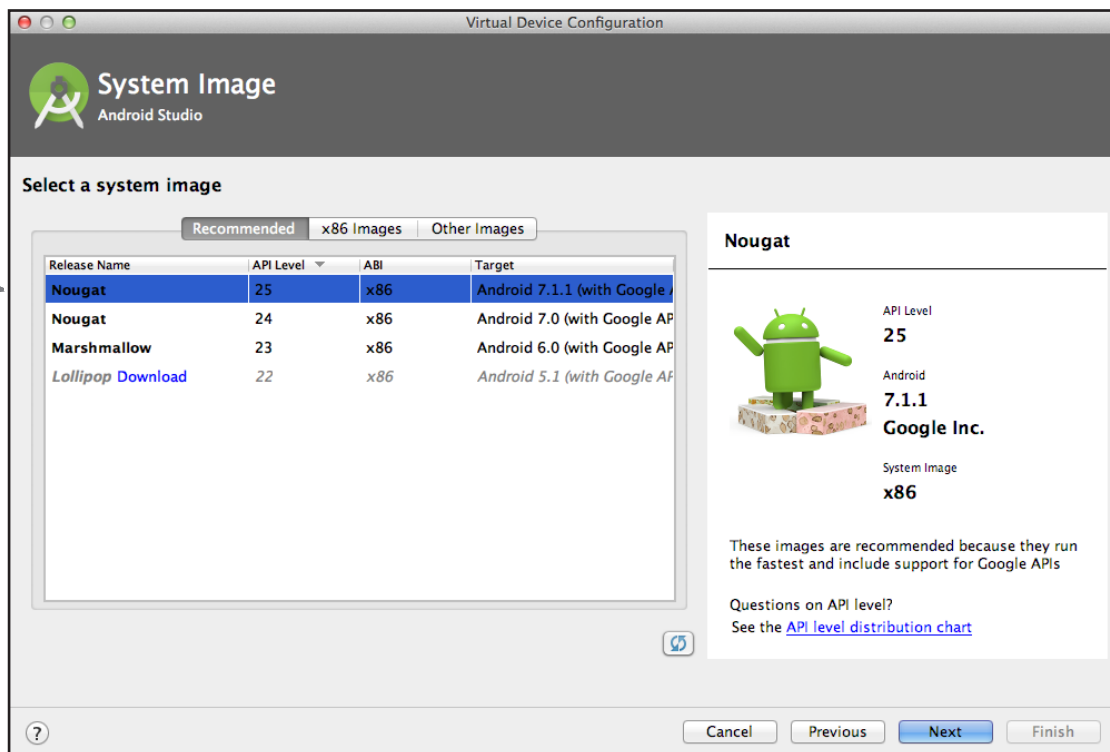
Creating a tablet AVD (continued)

Select a system image

Next, you need to select a system image. The system image gives you an installed version of the Android operating system. You can choose the version of Android you want to be on your AVD.

You need to choose a system image for an API level that's compatible with the app you're building. As an example, if you want your app to work on a minimum of API level 19, choose a system image for *at least* API level 19. As in Chapter 1, we want our AVD to run API level 25, so choose the system image with a release name of Nougat and a target of Android 7.1.1, the version number of API level 25. Then click on the Next button.

We'll choose the same system image as we did in Chapter 1. →

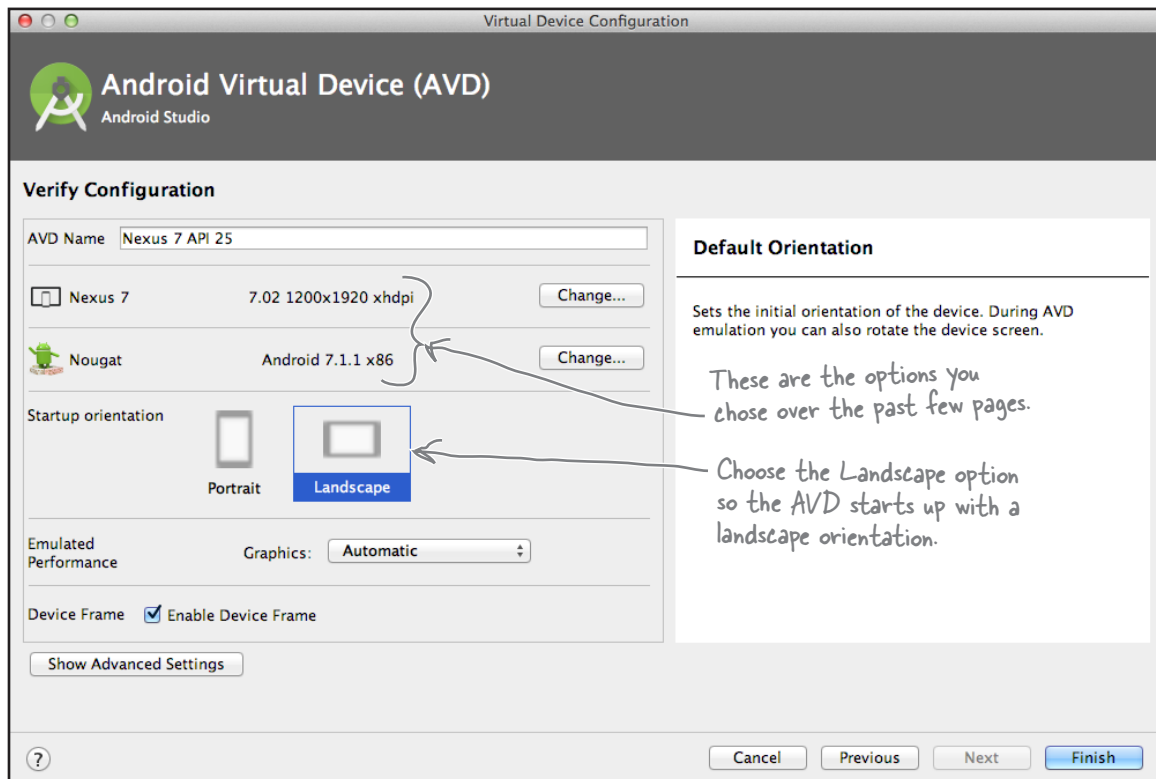




Creating a tablet AVD (continued)

Verify the AVD configuration

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



The AVD Manager will create the Nexus 7 AVD for you, and when it's done, display it in its list of devices. You may now close the AVD Manager.

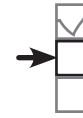
Now that we've created our tablet AVD, we can get to work on updating the Workout app. We want to change the app so that `MainActivity` uses one layout when it's running on a phone, and another layout when it's running on a tablet. But how can we do this?

Put screen-specific resources in screen-specific folders

Earlier in the book, you saw how you could get different devices to use image resources appropriate to their screen size by putting different-sized images in the different *drawable** folders. As an example, you put images intended for devices with high-density screens in the *drawable-hdpi* folder.

You can do something similar with other resources such as layouts, menus, and values. If you want to create multiple versions of the same resource for different screen specs, you need to create multiple resource folders with an appropriate name, then add the resource to that folder. The device will then load the resource at runtime from the folder that's the closest match to its screen spec.

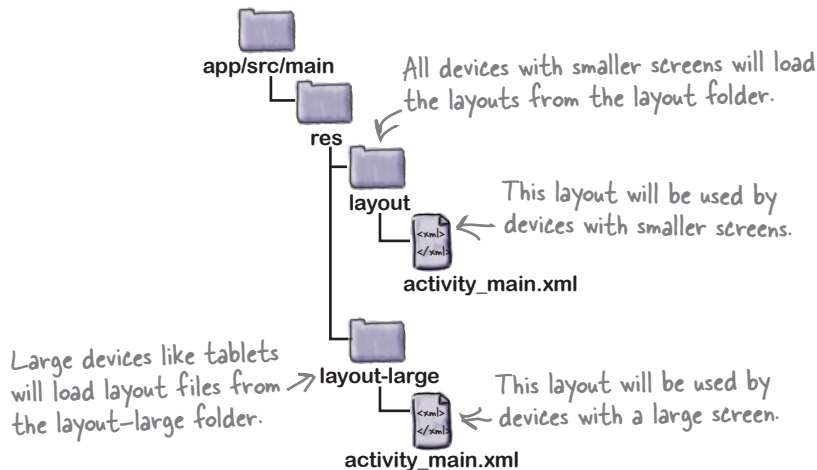
If you want to have one layout for large screen devices such as tablets, and another layout for smaller devices such as phones, you put the layout for the tablet in the *app/src/main/res/layout-large* folder, and the layout for the phone in the *app/src/main/res/layout* folder. When the app runs on a phone, it will use the layout in the *layout* folder. If it's run on a tablet, it will use the layout in the *layout-large* folder instead.



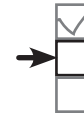
Create AVD
Create layout
Show workout

Android uses the names of your resource folders to decide which resources it should use at runtime.

Layouts in the layout folder can be used by any device, but layouts in the layout-large folder will only be used by devices with a large screen.



On the next page, we'll show you all the different options you can use for your resource folder names.



The different folder options

You can put all kinds of resources (drawables or images, layouts, menus, and values) in different folders to specify which types of device they should be used with. The screen-specific folder name can include screen size, density, orientation and aspect ratio, with each part separated by hyphens. As an example, if you want to create a layout that will only be used by very large tablets in landscape mode, you would create a folder called *layout-xlarge-land* and put the layout file in that folder. Here are the different options you can use for the folder names:

You must specify a resource type.

Resource type	Screen size	Screen density	Orientation	Aspect ratio
drawable	-small	-ldpi	-land	-long
layout	-normal	-mdpi	-port	-notlong
menu	-large	-hdpi		
mipmap	-xlarge	-xhdpi		
values		-xxhdpi		
		-xxxhdpi		
		-nodpi		
		-tvdpi		

Screen density is based on dots per inch.

long is for screens that have a very high value for height.

This is for density-independent resources. Use `-nodpi` for any image resources you don't want to scale (e.g., a folder called `drawable-nodpi`).

A mipmap resource is used for application icons. Older versions of Android Studio use drawables instead.

Android decides at runtime which resources to use by checking the spec of the device and looking for the best match. If there's no exact match, it will use resources designed for a smaller screen than the current one. If resources are only available for screens *larger* than the current one, Android won't use them and the app will crash.

If you only want your app to work on devices with particular screen sizes, you can specify this in *AndroidManifest.xml* using the `<supports-screens>` attribute. As an example, if you don't want your app to run on devices with small screens, you'd use:

```
<supports-screens android:smallScreens="false"/>
```

Using the different folder names above, you can create layouts that are tailored for phones and tablets.





BE the Folder Structure

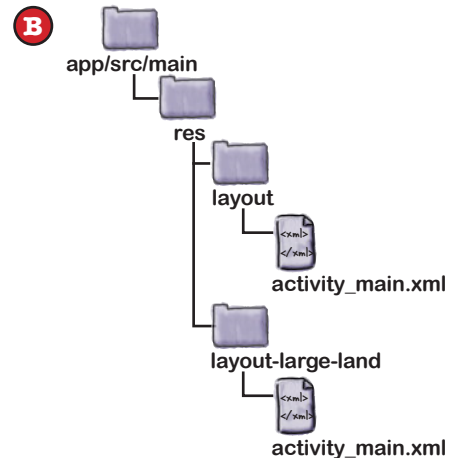
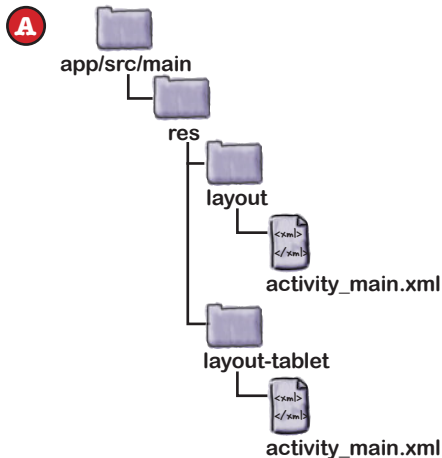
Below you'll see the code for an activity. You want to display one layout when it runs on devices with large-sized screens, and another layout when it runs on devices with smaller-sized screens. Which of these folder structures will allow you to do that?

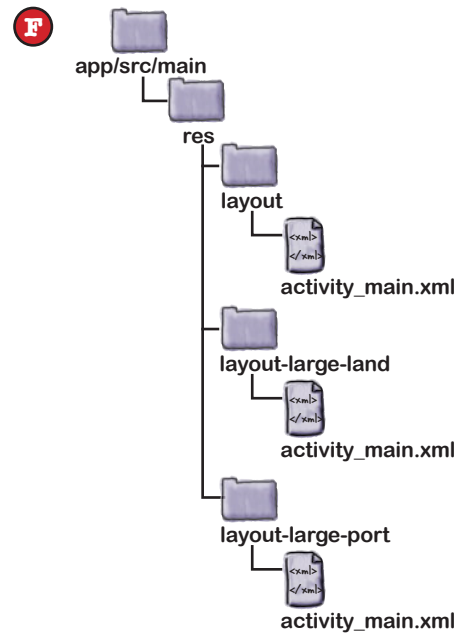
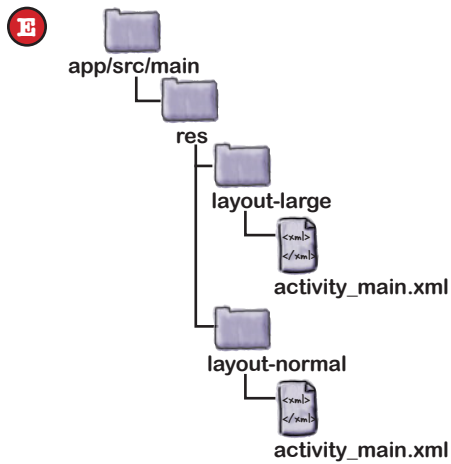
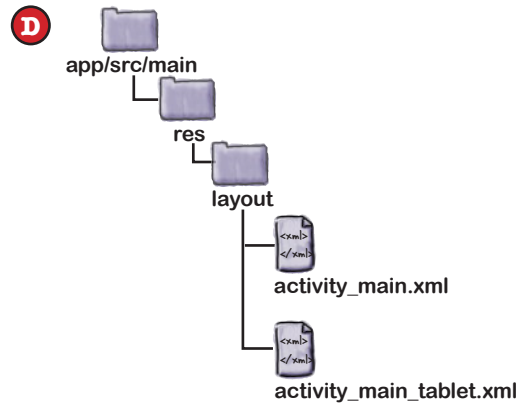
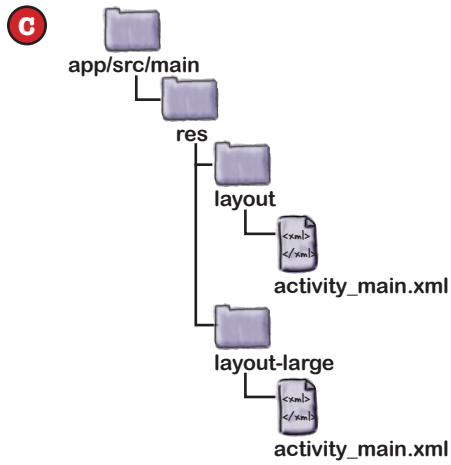
```
import android.app.Activity;
import android.os.Bundle;

public class MainActivity extends Activity {

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

← Here's the activity.







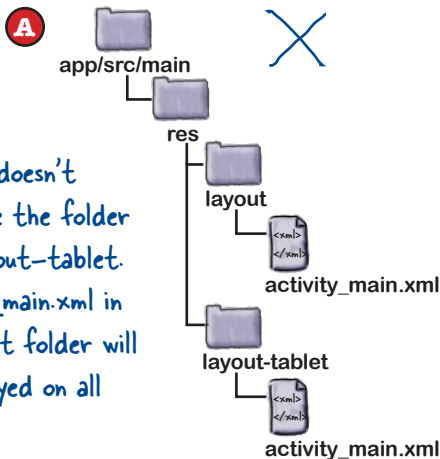
BE the Folder Structure Solution

Below you'll see the code for an activity. You want to display one layout when it runs on devices with large-sized screens, and another layout when it runs on devices with smaller-sized screens. Which of these folder structures will allow you to do that?

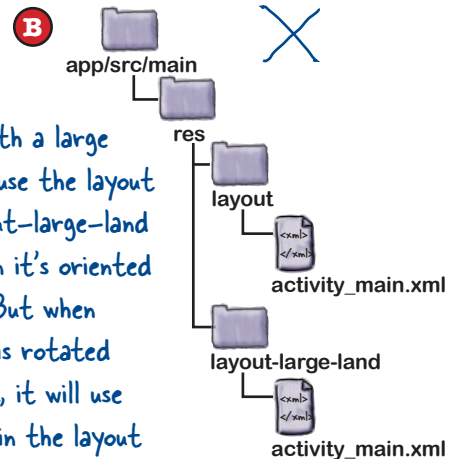
```
import android.app.Activity;
import android.os.Bundle;

public class MainActivity extends Activity {

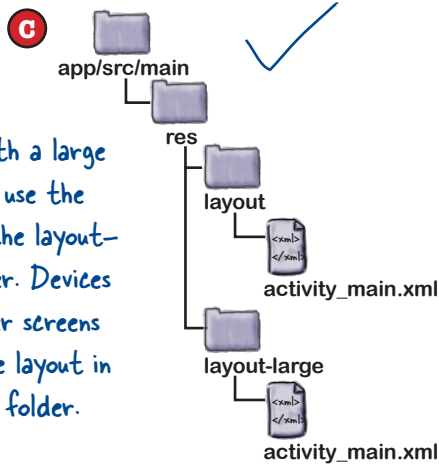
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        ...
    }
}
```



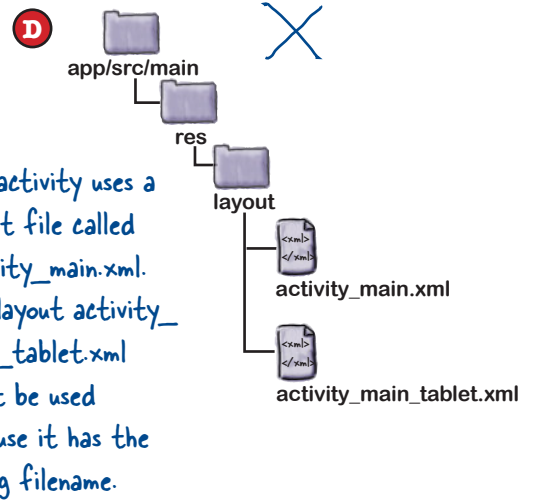
Android doesn't recognize the folder name `layout-tablet`. `activity_main.xml` in the `layout` folder will be displayed on all devices.



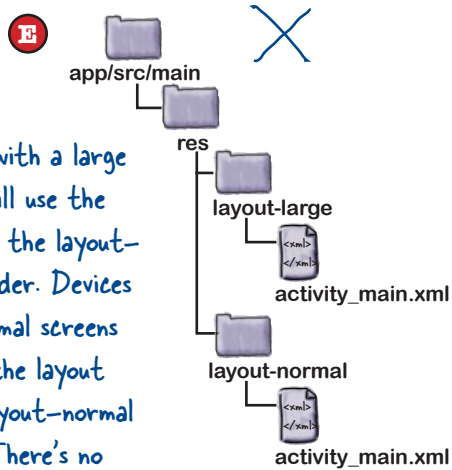
A device with a large screen will use the layout in the `layout-large-land` folder when it's oriented landscape. But when the device is rotated to portrait, it will use the layout in the `layout` folder—and that's not what you want.



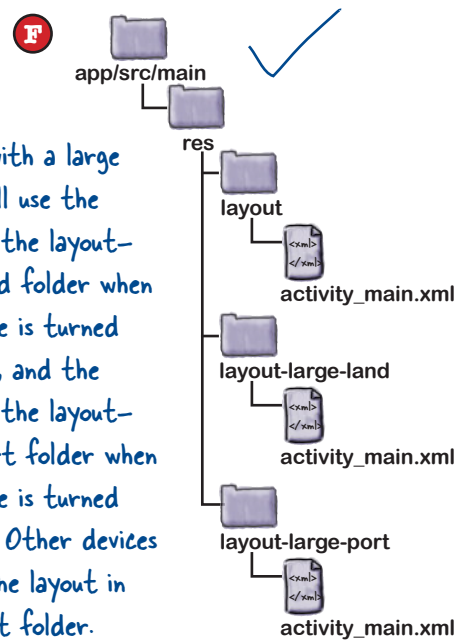
Devices with a large screen will use the layout in the layout-large folder. Devices with smaller screens will use the layout in the layout folder.



The activity uses a layout file called `activity_main.xml`. The layout `activity_main_tablet.xml` won't be used because it has the wrong filename.

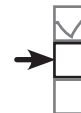


Devices with a large screen will use the layout in the layout-large folder. Devices with normal screens will use the layout in the layout-normal folder. There's no layout for devices with a small screen, so they won't be able to run the app.



Devices with a large screen will use the layout in the layout-large-land folder when the device is turned landscape, and the layout in the layout-large-port folder when the device is turned portrait. Other devices will use the layout in the layout folder.

Tablets use layouts in the layout-large folder



Create AVD
Create layout
Show workout

To get the tablet version of our app up and running, we need to copy our existing activity layout file *activity_main.xml* into the *app/src/main/res/layout-large* folder and then update that version of the file. This layout will then only be used by devices with a large screen.

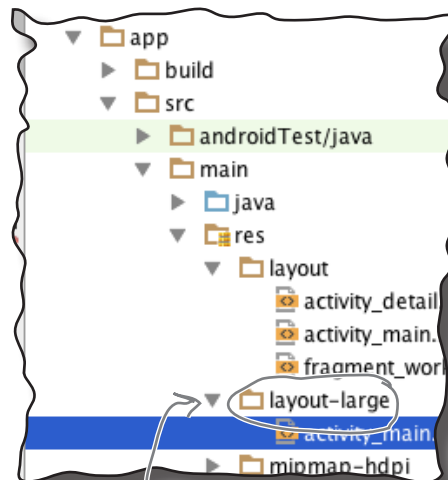
If the *app/src/main/res/layout-large* folder doesn't exist in your Android Studio project, you'll need to create it. To do this, switch to the Project view of Android Studio's explorer, highlight the *app/src/main/res* folder, and choose File→New...→Directory. When prompted, give the folder a name of "layout-large". When you click on the OK button, Android Studio will create the new *app/src/main/res/layout-large* folder.

To copy the *activity_main.xml* layout file, highlight the file in the explorer, and choose the Copy command from the Edit menu. Then highlight the new *layout-large* folder, and choose the Paste command from the Edit menu. Android Studio will copy the *activity_main.xml* file into the *app/src/main/res/layout-large* folder.

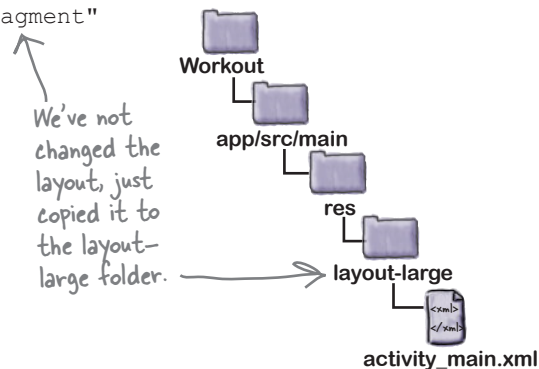
If you open the file you just pasted, it should look like this:

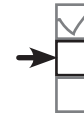
```
<?xml version="1.0" encoding="utf-8"?>
<fragment xmlns:android="http://schemas.android.com/apk/res/android"
    android:name="com.hfad.workout.WorkoutListFragment"
    android:layout_width="match_parent"
    android:layout_height="match_parent"/>
```

This is exactly the same layout that we had before. It contains a single fragment, *WorkoutListFragment*, that displays a list of workouts. The next thing we need to do is update the layout so that it displays two fragments side by side, *WorkoutListFragment* and *WorkoutDetailFragment*.



Here's the folder
Android Studio created.





- Create AVD
- Create layout
- Show workout

The layout-large version of the layout needs to display two fragments

We're going to change the version of `activity_main.xml` in the `layout-large` folder so that it contains the two fragments. To do this, we'll add the fragments to a linear layout with the orientation set to `horizontal`. We'll adjust the width of the fragments so that `WorkoutListFragment` takes up two-fifths of the available space, and `WorkoutDetailFragment` takes up three-fifths.

Our version of `activity_main.xml` is below. Update your code to reflect our changes. Make sure that you only edit the tablet version of the layout that's in the `layout-large` folder.

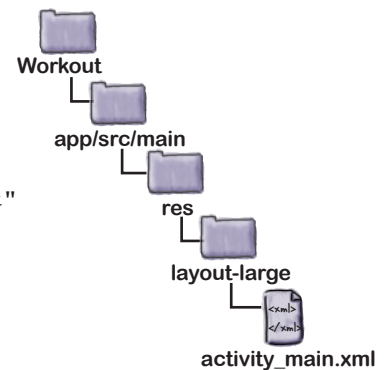
We're putting the fragments in a `LinearLayout` with a horizontal orientation so the two fragments will be displayed alongside each other.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="horizontal"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
    <fragment
        android:name="com.hfad.workout.WorkoutListFragment"
        android:id="@+id/list_frag"
        android:layout_width="0dp"
        android:layout_weight="2"
        android:layout_height="match_parent"/>
    <fragment
        android:name="com.hfad.workout.WorkoutDetailFragment"
        android:id="@+id/detail_frag"
        android:layout_width="0dp"
        android:layout_weight="3"
        android:layout_height="match_parent"/>
</LinearLayout>
```

Our layout already includes `WorkoutListFragment`.

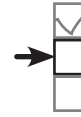
The fragments need IDs so that Android doesn't lose track of where to put each fragment.

We're adding `WorkoutDetailFragment` to `MainActivity`'s layout.



We'll run through what happens when the code runs on the next page.

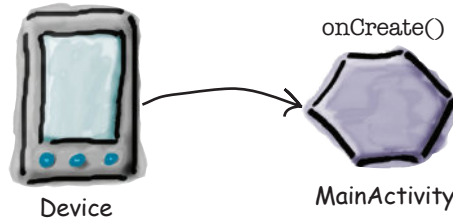
What the updated code does



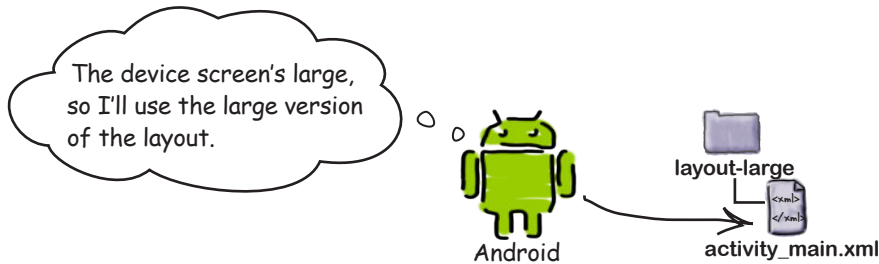
Create AVD
Create layout
Show workout

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

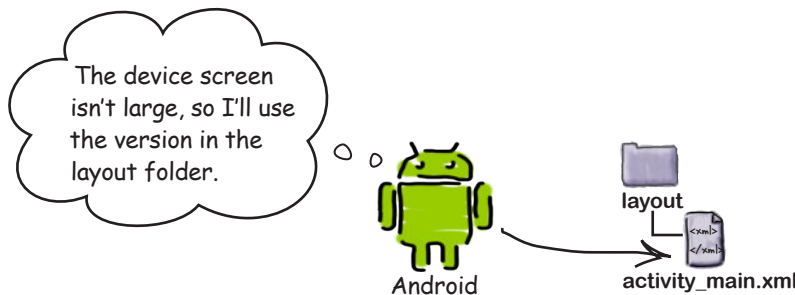
- 1** **When the app is launched, MainActivity gets created.** MainActivity's onCreate() method runs. This specifies that activity_main.xml should be used for MainActivity's layout.



- 2a** **If the app's running on a tablet, it uses the version of activity_main.xml that's in the layout-large folder.** The layout displays WorkoutListFragment and WorkoutDetailFragment side by side.



- 2b** **If the app's running on a device with a smaller screen, it uses the version of activity_main.xml that's in the layout folder.** The layout displays WorkoutListFragment on its own.





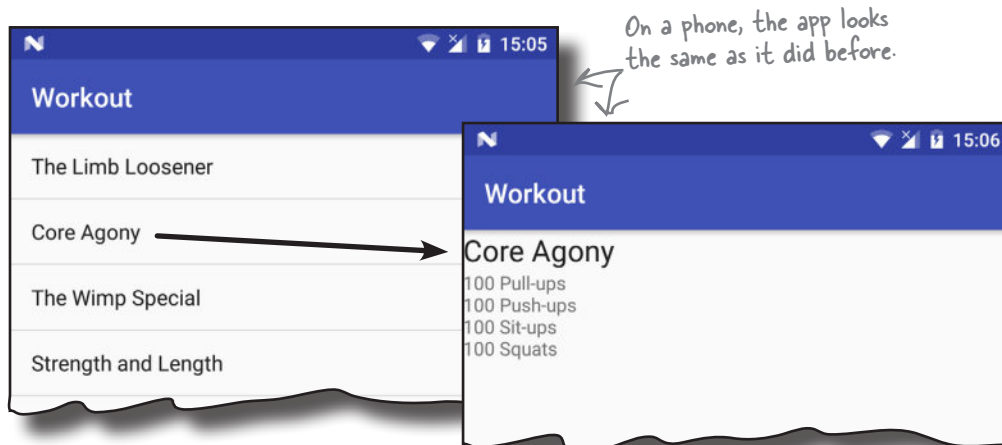
Test drive the app

fragments for larger interfaces

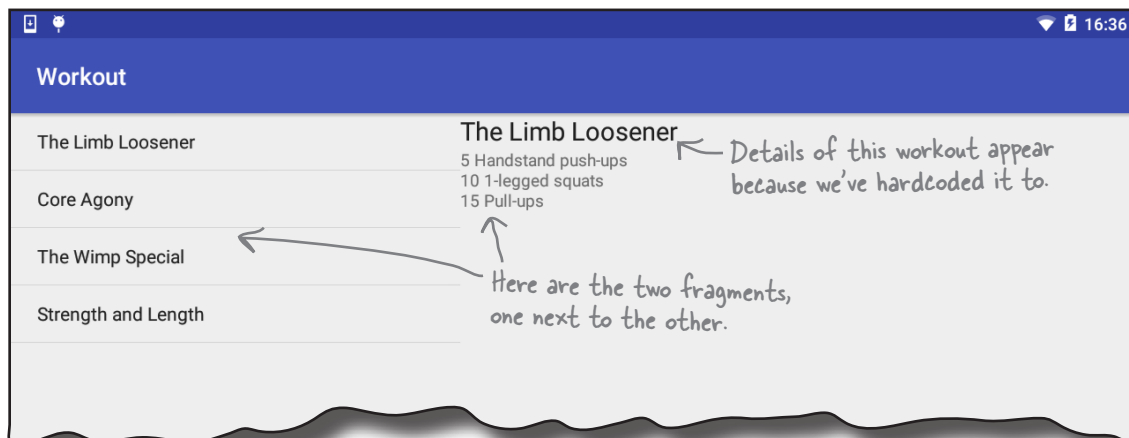


Create AVD
Create layout
Show workout

When you run the app on a phone, the app looks just as it did before. MainActivity displays a list of workout names, and when you click on one of the workouts, DetailActivity starts and displays its details.



When you run the app on a tablet, MainActivity displays a list of workout names on the left, and details of the first workout appear next to it.



When you click on one of the workouts, DetailActivity still gets displayed. We need to change our code so that if the app's running on a tablet, DetailActivity no longer starts. Instead, we need to display details of the workout the user selects in MainActivity, and not just the first workout.

We need to change the itemClicked() code

We need to change the code that decides what to do when items in `WorkoutListFragment` are clicked. This means that we need to change the `itemClicked()` method in `MainActivity`. Here's the current code:

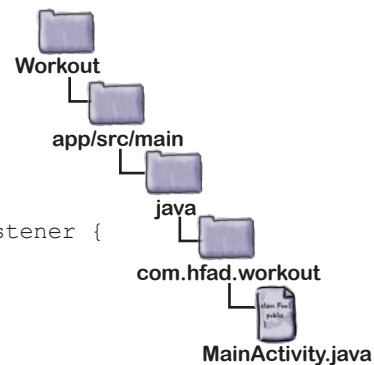
```
...

public class MainActivity extends AppCompatActivity
    implements WorkoutListFragment.Listener {
...

    @Override
    public void itemClicked(long id) {
        Intent intent = new Intent(this, DetailActivity.class);
        intent.putExtra(DetailActivity.EXTRA_WORKOUT_ID, (int)id);
        startActivity(intent);
    }
}
```



Create AVD
Create layout
Show workout



↑ This is the `itemClicked()` method we wrote in the previous chapter. It starts `DetailActivity`, and passes it the ID of the workout that was clicked.

The current code starts `DetailActivity` whenever the user clicks on one of the workouts. We need to change the code so that this only happens if the app's running on a device with a small screen such as a phone. If the app's running on a device with a large screen, when the user picks a workout we need to display the details of the workout shown to the right of the list of workouts in `WorkoutDetailFragment`.

But how do we update the workout details?

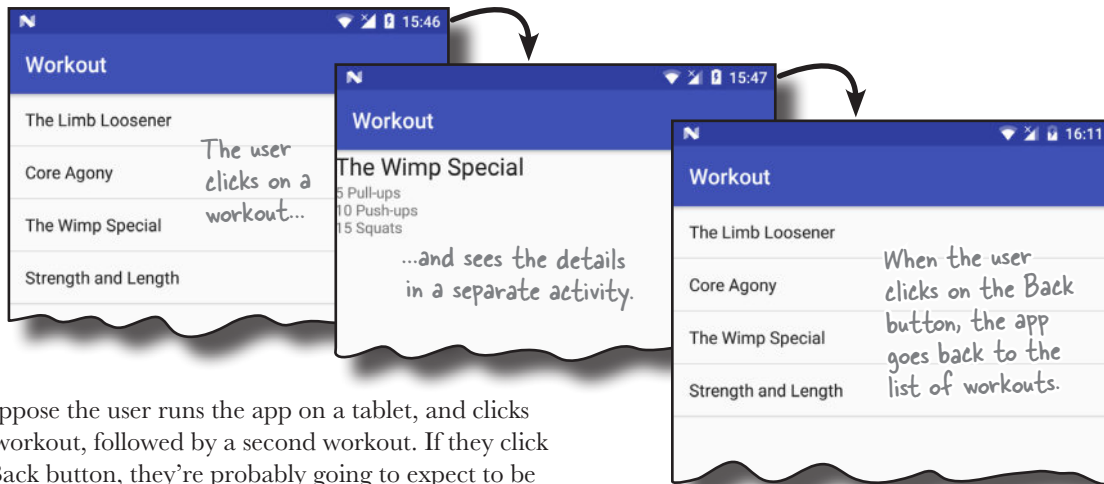
The `WorkoutDetailFragment` updates its views when it is started. But once the fragment is displayed onscreen, how do we get the fragment to update the details?

You might be thinking that we could play with the fragment's lifecycle so that we get it to update. Instead, **we'll replace the detail fragment with a brand-new detail fragment, each time we want its text to change.**

There's a really good reason why...

You want fragments to work with the Back button

Suppose you have a user that runs the app on a phone. When they click on a workout, details of that workout are displayed in a separate activity. If the user clicks on the Back button, they're returned to the list of workouts:



Then suppose the user runs the app on a tablet, and clicks on one workout, followed by a second workout. If they click on the Back button, they're probably going to expect to be returned to the first workout they chose:



In every app we've built so far, when we've clicked on the Back button, we've been returned to the previous activity. This is standard Android behavior, and something that Android has handled for us automatically. If we're running this particular app on a tablet, however, we don't want the Back button to return us to the previous *activity*. We want it to return us to the previous *fragment state*.

Welcome to the back stack

When you go from activity to activity in your app, Android keeps track of each activity you've visited by adding it to the **back stack**. The back stack is a log of the places you've visited on the device, each place recorded as a separate transaction.



Create AVD
Create layout
Show workout

A back stack scenario

- 1 Suppose you start by visiting a fictitious activity in your app, `Activity1`. Android records your visit to `Activity1` on the back stack as a transaction.
- 2 You then go to `Activity2`. Your visit to `Activity2` is added to the top of the back stack as a separate transaction.
- 3 You then go to `Activity3`. `Activity3` is added to the top of the back stack.
- 4 When you click on the Back button, `Activity3` pops off the top of the back stack. Android displays `Activity2`, as this activity is now at the top of the back stack.
- 5 If you click on the Back button again, `Activity2` pops off the top of the back stack, and `Activity1` is displayed.





Create AVD
Create layout
Show workout

Back stack transactions don't have to be activities

We've shown you how the back stack works with activities, but the truth is, it doesn't just apply to activities. It applies to any sort of transaction, including changes to fragments.



These are two different fragment transactions for `WorkoutDetailFragment`. The top one displays details of the Core Agony workout, and the bottom one displays details of the Wimp Special.

This means that *fragment* changes can be reversed when you click on the Back button, just like *activity* changes can.



When you click on the Back button, the transaction that contains details of the Core Agony is popped off the top of the back stack. Details of the Wimp Special are displayed.

So how can we record changes to fragments as separate transactions on the back stack?

Don't update—instead, replace

We're going to replace the entire `WorkoutDetailFragment` with a new instance of it each time the user selects a different workout. Each new instance of `WorkoutDetailFragment` will be set up to display details of the workout the user selects. That way, we can add each fragment replacement to the back stack as a separate transaction. Each time the user clicks on the Back button, the most recent transaction will be popped off the top of the stack, and the user will see details of the previous workout they selected.

To do this, we first need to know how to replace one fragment with another. We'll look at this on the next page.

Android builds the back stack as you navigate from one activity to another. Each activity is recorded in a separate transaction.

Use a frame layout to replace fragments programmatically

To replace one fragment with another in MainActivity's tablet user interface, we need to begin by making a change to the `activity_main.xml` layout file in the `layout-large` folder. Instead of inserting `WorkoutDetailFragment` directly using the `<fragment>` element, we'll use a frame layout. ←

We'll add the fragment to the frame layout programmatically. Whenever an item in the `WorkoutListFragment` list view gets clicked, we'll replace the contents of the frame layout with a new instance of `WorkoutDetailFragment` that displays details of the correct workout.

Here's our new version of the code for `activity_main.xml` in the `layout-large` folder. Update your code to include our changes.

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

    <fragment
        android:name="com.hfad.workout.WorkoutListFragment"
        android:id="@+id/list_frag"
        android:layout_width="0dp"
        android:layout_weight="2"
        android:layout_height="match_parent"/>


```

~~<fragment~~

<FrameLayout

```

    android:name="com.hfad.workout.WorkoutDetailFragment"
    android:id="@+id/detail_frag"
    android:id="@+id/fragment_container"
    android:layout_width="0dp"
    android:layout_weight="3"
    android:layout_height="match_parent"/>

```

```
</LinearLayout>
```

We're going to display the fragment inside a `FrameLayout`.

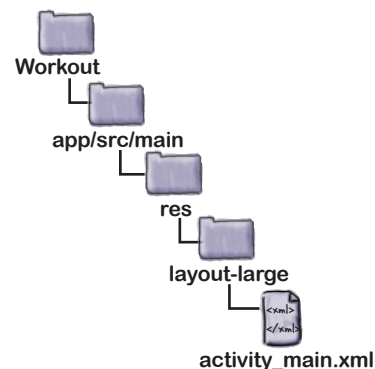
We'll add the fragment to the frame layout programmatically.

We'll give the `FrameLayout` an ID of `fragment_container` so we can refer to it in our activity code.



Create AVD
Create layout
Show workout

Add a fragment using a `<FrameLayout>` whenever you need to replace fragments programmatically, such as when you need to add fragment changes to the back stack.





- Create AVD
- Create layout
- Show workout

Use layout differences to tell which layout the device is using

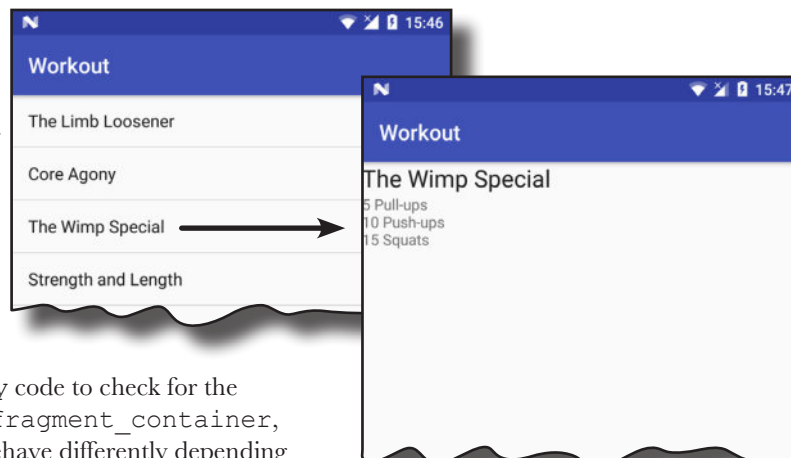
We want `MainActivity` to perform different actions when the user clicks on a workout depending on whether the device is running on a phone or a tablet. We can tell which version of the layout's being used by checking whether or not the layout includes the frame layout we added on the previous page.

If the app is running on a tablet, the device will be using the version of `activity_main.xml` that's in the `layout-large` folder. This layout includes a frame layout with an ID of `fragment_container`. When the user clicks on a workout, we want to display a new instance of `WorkoutDetailFragment` in the frame layout.



If the app's running on a phone, the device will be using `activity_main.xml` in the `layout` folder. This layout doesn't include the frame layout. If the user clicks on a workout, we want `MainActivity` to start `DetailActivity` as it does currently.

MainActivity doesn't include the frame layout if it's running on a phone.



If we can get our `MainActivity` code to check for the existence of a view with an ID of `fragment_container`, we can get `MainActivity` to behave differently depending on whether the app's running on a phone or a tablet.

The revised MainActivity code



Create AVD
Create layout
Show workout

We've updated MainActivity so that the `itemClicked()` method looks for a view with an ID of `fragment_container`. We can then perform different actions depending on whether or not the view is found.

Here's our full code for *MainActivity.java*; update your version of the code to match ours:

```
package com.hfad.workout;

import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.view.View;
import android.content.Intent;

public class MainActivity extends AppCompatActivity
    implements WorkoutListFragment.Listener {

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

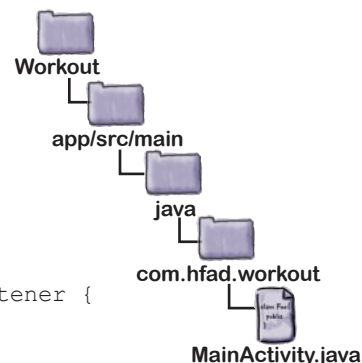
    @Override
    public void itemClicked(long id) {
        View fragmentContainer = findViewById(R.id.fragment_container);
        if (fragmentContainer != null) {
            //Add the fragment to the FrameLayout
        } else {
            Intent intent = new Intent(this, DetailActivity.class);
            intent.putExtra(DetailActivity.EXTRA_WORKOUT_ID, (int) id);
            startActivity(intent);
        }
    }
}
```

We've not changed this method.

Get a reference to the frame layout that will contain WorkoutDetailFragment. This will only exist if the app is being run on a device with a large screen.

We need to write code that will run if the frame layout exists.

If the frame layout doesn't exist, the app must be running on a device with a smaller screen. In that case, start DetailActivity and pass it the ID of the workout as before.



The next thing we need to do is see how we can add `WorkoutDetailFragment` to the frame layout programmatically.



Using fragment transactions

You can programmatically add a fragment to an activity's layout so long as the activity's running. All you need is a view group in which to place the fragment, such as a frame layout.

You add, replace, or remove fragments at runtime using a **fragment transaction**. A fragment transaction is a set of changes relating to the fragment that you want to apply, all at the same time.

When you create a fragment transaction, you need to do three things:

- 1 Begin the transaction.**
This tells Android that you're starting a series of changes that you want to record in the transaction.
- 2 Specify the changes.**
These are all the actions you want to group together in the transaction. This can include adding, replacing, or removing a fragment, updating its data, and adding it to the back stack.
- 3 Commit the transaction.**
This finishes the transaction and applies the changes.

1. Begin the transaction

You begin the transaction by first getting a reference to the activity's fragment manager. As you may remember from the previous chapter, the fragment manager is used to manage any fragments used by the activity. If you're using fragments from the Support Library as we are here, you get a reference to the fragment manager using the following method:

```
getSupportFragmentManager();
```

← This returns the fragment manager that deals with fragments from the Support Library.

Once you have a reference to the fragment manager, you call its `beginTransaction()` method to begin the transaction:

```
FragmentManager transaction = getSupportFragmentManager().beginTransaction();
```

That's all you need to do to begin the transaction. On the next page we'll look at how you specify the changes you want to make.

*↑
The start of the fragment transaction*

2. Specify the changes

After beginning the transaction, you need to say what changes the transaction should include.

If you want to add a fragment to your activity's layout, you call the fragment transaction's `add()` method. This takes two parameters, the resource ID of the view group you want to add the fragment to, and the fragment you want to add. The code looks like this:

```
WorkoutDetailFragment fragment = new WorkoutDetailFragment(); ← Create the fragment.
transaction.add(R.id.fragment_container, fragment); ← Add the fragment to the ViewGroup.
```

To replace the fragment, you use the `replace()` method:

```
transaction.replace(R.id.fragment_container, fragment); ← Replace the fragment.
```

To remove the fragment completely, you use the `remove()` method:

```
transaction.remove(fragment); ← Remove the fragment.
```

You can optionally use the `setTransition()` method to say what sort of transition animation you want for this transaction:

```
transaction.setTransition(transition); ← You don't have to set a transition.
```

`transition` is the type of animation. Options for this are `TRANSIT_FRAGMENT_CLOSE` (a fragment is being removed from the stack), `TRANSIT_FRAGMENT_OPEN` (a fragment is being added), `TRANSIT_FRAGMENT_FADE` (the fragment should fade in and out), and `TRANSIT_NONE` (no animation). By default, there are no animations.

Once you've specified all the actions you want to take as part of the transaction, you can use the `addToBackStack()` method to add the transaction to the back stack. This method takes one parameter, a `String` name you can use to label the transaction. This parameter is needed if you need to programmatically retrieve the transaction. Most of the time you won't need to do this, so you can pass in a `null` value like this:

```
transaction.addToBackStack(null); ← Most of the time you won't need to retrieve
the transaction, so it can be set to null.
```



Create AVD
Create layout
Show workout

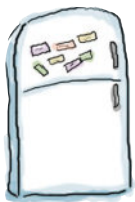


3. Commit the transaction

Finally, you need to commit the transaction. This finishes the transaction, and applies the changes you specified. You commit the transaction by calling the transaction's `commit()` method like this:

```
transaction.commit();
```

That's everything we need to know in order to create fragment transactions, so let's put it into practice by getting our `MainActivity` code to display an updated version of `WorkoutDetailFragment` every time the user clicks on a workout.



Activity Magnets

We want to write a new version of the `MainActivity`'s `itemClicked()` method. It needs to change the workout details that are displayed in `WorkoutDetailFragment` each time the user clicks on a new workout. See if you can finish the code below.

```
public void itemClicked(long id) {
    View fragmentContainer = findViewById(R.id.fragment_container);
    if (fragmentContainer != null) {
        WorkoutDetailFragment details = new WorkoutDetailFragment();

        FragmentTransaction ft = getSupportFragmentManager(). .....;
        details.setWorkout(id);

        ft. .... (R.id.fragment_container, .....);

        ft. .... (FragmentTransaction.TRANSIT_FRAGMENT_FADE);

        ft. .... (null);

        ft. ....;
    } else {
        Intent intent = new Intent(this, DetailActivity.class);
        intent.putExtra(DetailActivity.EXTRA_WORKOUT_ID, (int) id);
        startActivity(intent);
    }
}
```

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

replace

commit()

beginTransaction()

setTransition

startTransaction()

details

endTransaction()

addToBackStack



Activity Magnets Solution

We want to write a new version of the MainActivity's `itemClicked()` method. It needs to change the workout details that are displayed in `WorkoutDetailFragment` each time the user clicks on a new workout. See if you can finish the code below.

```

public void itemClicked(long id) {
    View fragmentContainer = findViewById(R.id.fragment_container);
    if (fragmentContainer != null) {
        WorkoutDetailFragment details = new WorkoutDetailFragment();

        FragmentTransaction ft = getSupportFragmentManager().beginTransaction();
        details.setWorkout(id);

        ft.replace(R.id.fragment_container, details);
        ft.setTransition(FragmentTransaction.TRANSIT_FRAGMENT_FADE);
        ft.addToBackStack(null);
        ft.commit();
    } else {
        Intent intent = new Intent(this, DetailActivity.class);
        intent.putExtra(DetailActivity.EXTRA_WORKOUT_ID, (int) id);
        startActivity(intent);
    }
}

```

Each time the user clicks on a workout, we'll replace the fragment with a new instance of it.

beginTransaction(); This begins the transaction.

replace (R.id.fragment_container, **details**); This is a new instance of `WorkoutDetailFragment`. It displays details of the workout the user selected.

setTransition (FragmentTransaction.TRANSIT_FRAGMENT_FADE); Set the fragment to fade in and out.

addToBackStack (null); Add the transaction to the back stack.

commit(); Commit the transaction.

endTransaction() ← You didn't need to use these magnets. → **startTransaction()**

The updated MainActivity code

fragments for larger interfaces



Create AVD
Create layout
Show workout

We're going to get a new instance of `WorkoutDetailFragment` (one that displays the correct workout), display the fragment in the activity, and then add the transaction to the back stack. Here's the full code. Update your version of `MainActivity.java` to reflect our changes:

```
package com.hfad.workout;
```

```
import android.support.v4.app.FragmentTransaction;
```

```
...
```

```
public class MainActivity extends AppCompatActivity
    implements WorkoutListFragment.Listener {
```

```
@Override
```

```
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
}
```

We haven't changed this method.

```
@Override
```

```
public void itemClicked(long id) {
    View fragmentContainer = findViewById(R.id.fragment_container);
    if (fragmentContainer != null) {
```

```
        WorkoutDetailFragment details = new WorkoutDetailFragment();
        FragmentTransaction ft = getSupportFragmentManager().beginTransaction();
        details.setWorkout(id);
```

```
        ft.replace(R.id.fragment_container, details);
```

```
        ft.setTransition(FragmentTransaction.TRANSIT_FRAGMENT_FADE);
```

```
        ft.addToBackStack(null);
```

```
        ft.commit();
```

```
    } else {
```

```
        Intent intent = new Intent(this, DetailActivity.class);
```

```
        intent.putExtra(DetailActivity.EXTRA_WORKOUT_ID, (int) id);
```

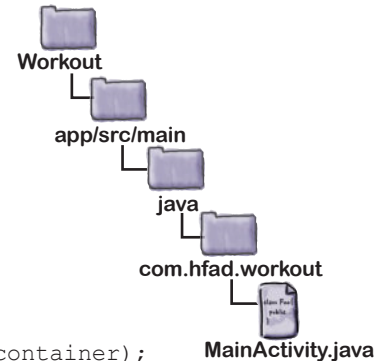
```
        startActivity(intent);
```

```
    }
```

```
}
```

```
}
```

We're using a `FragmentTransaction` from the Support Library as we're using Support Library Fragments.



Start the fragment transaction.

Add the transaction to the back stack.

Replace the fragment.

Get the new and old fragments to fade in and out.

On the next page we'll see what happens when the code runs.

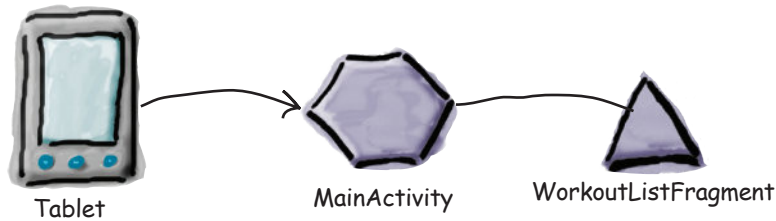
What happens when the code runs



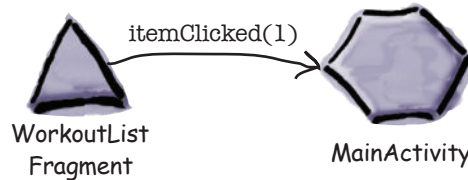
Create AVD
Create layout
Show workout

Here's a runthrough of what happens when we run the app.

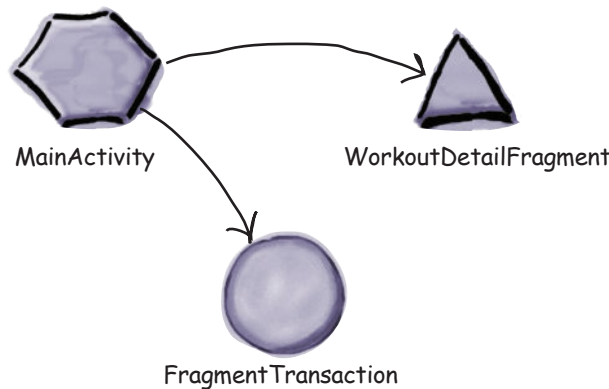
- 1 **The app is launched on a tablet and MainActivity starts.**
WorkoutListFragment is attached to MainActivity, and MainActivity is registered as a listener on WorkoutListFragment.



- 2 **When an item is clicked in WorkoutListFragment, the fragment's onItemClick() method is called.**
This calls MainActivity's itemClicked() method, passing it the ID of the workout that was clicked; in this example, the ID is 1.



- 3 **MainActivity's itemClicked() method sees that the app is running on a tablet.**
It creates a new instance of WorkoutDetailFragment, and begins a new fragment transaction.



The story continues...

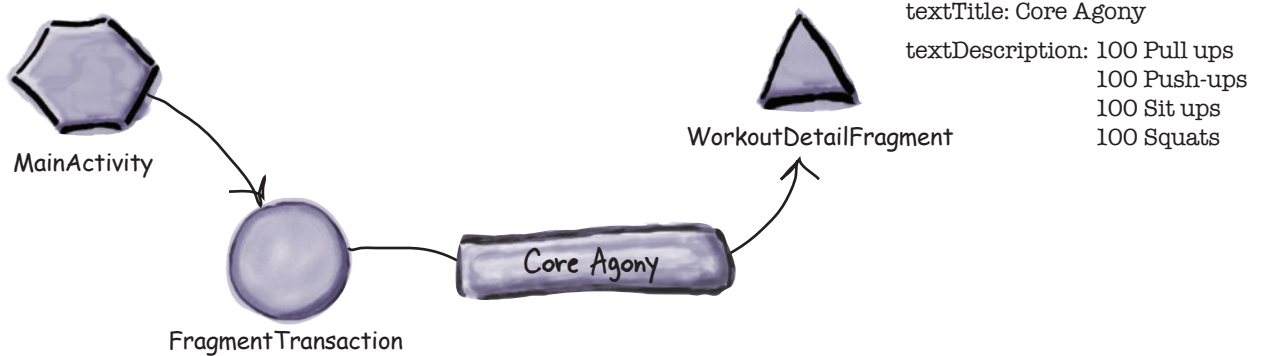
fragments for larger interfaces



Create AVD
Create layout
Show workout

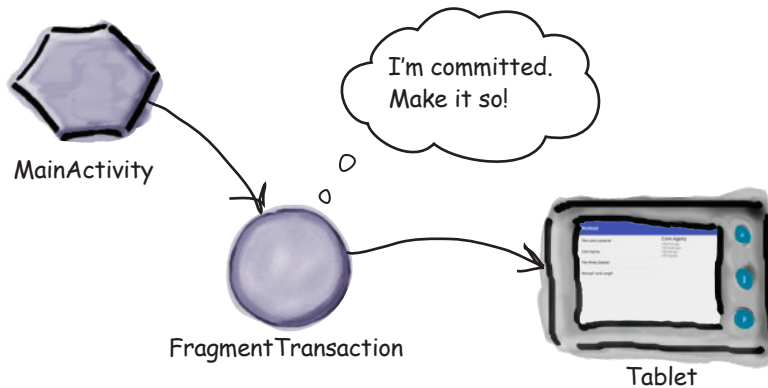
- 4 **As part of the transaction, WorkoutDetailFragment's views are updated with details of the workout that was selected, in this case the one with ID 1.**

The fragment is added to the `FrameLayout` `fragment_container` in `MainActivity`'s layout, and the whole transaction is added to the back stack.



- 5 **MainActivity commits the transaction.**

All of the changes specified in the transaction take effect, and the `WorkoutDetailFragment` is displayed next to `WorkoutListFragment`.



Let's take the app for a test drive.



Test drive the app



Create AVD
Create layout
Show workout

When we run the app, a list of the workouts appears on the left side of the screen. When we click on one of the workouts, details of that workout appear on the right. If we click on another workout and then click on the Back button, details of the workout we chose previously appear on the screen.



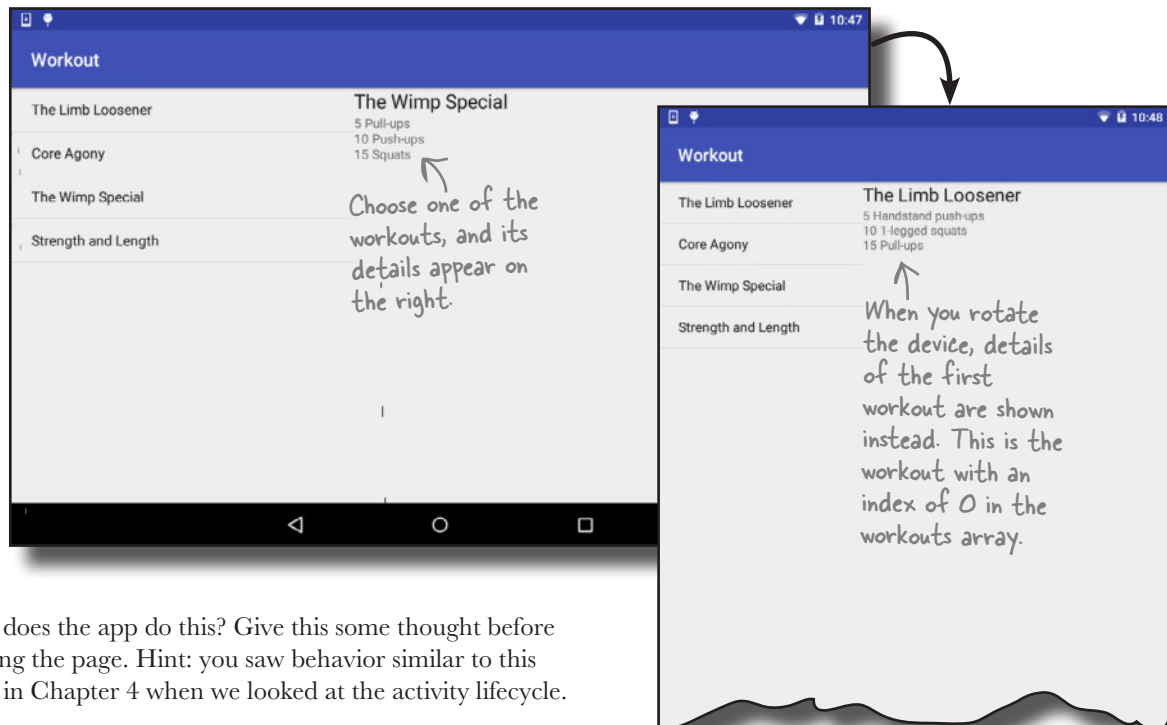
The app seems to be working fine as long as we don't rotate the screen. If we change the screen orientation, there's a problem. Let's see what happens.

Rotating the tablet breaks the app

When you run the app on a phone and rotate the device, the app works as you'd expect. Details of the workout the user selected continue to be displayed on the screen:



But when you run the app on a tablet, there's a problem. Regardless of which workout you've chosen, when you rotate the device, the app displays details of the first workout in the list:



Why does the app do this? Give this some thought before turning the page. Hint: you saw behavior similar to this back in Chapter 4 when we looked at the activity lifecycle.

Saving an activity's state (revisited)

When we first looked at the activity lifecycle back in Chapter 4, you saw how when you rotate the device, Android destroys and recreates the activity. When this happens, local variables used by the activity can get lost. To prevent this from happening, we saved the state of our local variables in the activity's `onSaveInstanceState()` method:

```
public void onSaveInstanceState(Bundle savedInstanceState) {
    savedInstanceState.putInt("seconds", seconds);
    savedInstanceState.putBoolean("running", running);
}
```

Earlier in the book, we used the `onSaveInstanceState()` method to save the state of these two variables.

We then restored the state of the variables in the activity's `onCreate()` method:

```
protected void onCreate(Bundle savedInstanceState) {
    ...
    if (savedInstanceState != null) {
        seconds = savedInstanceState.getInt("seconds");
        running = savedInstanceState.getBoolean("running");
    }
    ...
}
```

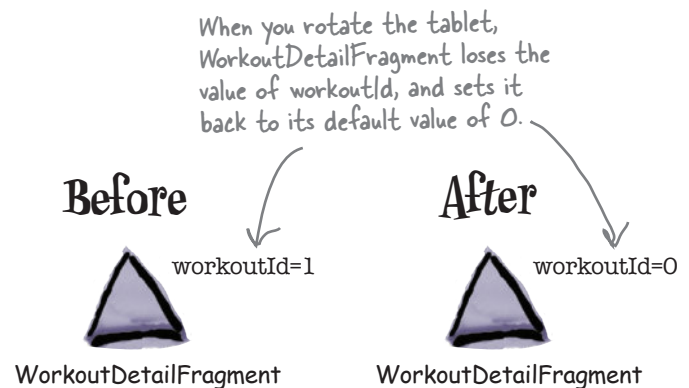
We restored the state of the variables in the `onCreate()` method.

So what does this have to do with our current problem?

Fragments can lose state too

If the activity uses a fragment, **the fragment gets destroyed and recreated along with the activity**. This means that any local variables used by the fragment can also lose their state.

In our `WorkoutDetailFragment` code, we use a local variable called `workoutId` to store the ID of the workout the user clicks on in the `WorkoutListFragment` list view. When the user rotates the device, `workoutId` loses its current value and it's set to 0 by default. The fragment then displays details of the workout with an ID of 0—the first workout in the list.



Save the fragment's state...

You deal with this problem in a fragment in a similar way to how you deal with it in an activity.

You first override the fragment's `onSaveInstanceState()` method. This method works in a similar way to an activity's `onSaveInstanceState()` method. It gets called before the fragment gets destroyed, and it has one parameter: a `Bundle`. You use the `Bundle` to save the values of any variables whose state you need to keep.

In our case, we want to save the state of our `workoutId` variable, so we'd use code like this:

```
public void onSaveInstanceState(Bundle savedInstanceState) {
    savedInstanceState.putLong("workoutId", workoutId);
}
```

The `onSaveInstanceState()` method gets called before the fragment is destroyed.

Once you've saved the state of any variables, you can restore it when the fragment is recreated.

...then use `onCreate()` to restore the state

Just like an activity, a fragment has an `onCreate()` method that has one parameter, a `Bundle`. This is the `Bundle` to which you saved the state of your variables in the fragment's `onSaveInstanceState()` method, so you can use it to restore the state of those variables in your fragment's `onCreate()` method.

In our case, we want to restore the state of the `workoutId` variable, so we can use code like this:

```
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    if (savedInstanceState != null) {
        workoutId = savedInstanceState.getLong("workoutId");
    }
}
```

We can use this `Bundle` to get the previous state of the `workoutId` variable.

We'll show you the full code on the next page.

The updated code for WorkoutDetailFragment.java

We've updated our code for *WorkoutDetailFragment.java* to save the state of the `workoutId` variable before the fragment is destroyed, and restore it if the fragment is recreated. Here's our code; update your version of *WorkoutDetailFragment.java* to reflect our changes.

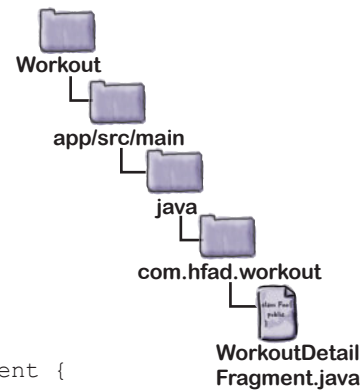
```
package com.hfad.workout;

import android.support.v4.app.Fragment;
import android.os.Bundle;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;
import android.widget.TextView;

public class WorkoutDetailFragment extends Fragment {
    private long workoutId;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        if (savedInstanceState != null) {
            workoutId = savedInstanceState.getLong("workoutId");
        }
    }

    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup container,
        Bundle savedInstanceState) {
        return inflater.inflate(R.layout.fragment_workout_detail, container, false);
    }
}
```



↙ Add the onCreate() method.

↑ Set the value of the workoutId.

The code continues →
on the next page.

WorkoutDetailFragment.java (continued)

```

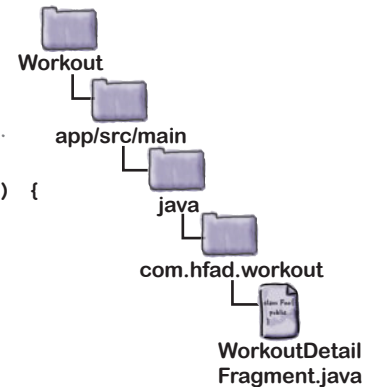
@Override
public void onStart() {
    super.onStart();
    View view = getView();
    if (view != null) {
        TextView title = (TextView) view.findViewById(R.id.textTitle);
        Workout workout = Workout.workouts[(int) workoutId];
        title.setText(workout.getName());
        TextView description = (TextView) view.findViewById(R.id.textDescription);
        description.setText(workout.getDescription());
    }
}

@Override
public void onSaveInstanceState(Bundle savedInstanceState) {
    savedInstanceState.putLong("workoutId", workoutId);
}

public void setWorkout(long id) {
    this.workoutId = id;
}
}

```

Save the value of the workoutId in the savedInstanceState Bundle before the fragment gets destroyed. We're retrieving it in the onCreate() method.



Test drive the app

Now, when you run the app on a tablet and rotate the device, details of the workout the user selected continue to be displayed on the screen.





Your Android Toolbox

You've got Chapter 10 under your belt and now you've added fragments for larger interfaces to your toolbox.

You can download the full code for the chapter from <https://tinyurl.com/HeadFirstAndroid>.



BULLET POINTS

- Make apps look different on different devices by putting separate layouts in device-appropriate folders.
- Android keeps track of places you've visited within an app by adding them to the back stack as separate transactions. Pressing the Back button pops the last transaction off the back stack.
- Use a frame layout to add, replace, or remove fragments programmatically using fragment transactions.
- Begin the transaction by calling the `FragmentManager.beginTransaction()` method. This creates a `FragmentTransaction` object.
- Add, replace, and delete fragments using the `FragmentTransaction.add()`, `replace()`, and `remove()` methods.
- Add a transaction to the back stack using the `FragmentTransaction.addToBackStack()` method.
- Commit a transaction using the `FragmentTransaction.commit()` method. This applies all the updates in the transaction.
- Save the state of a fragment's variables in the `Fragment onSaveInstanceState()` method.
- Restore the state of a fragment's variables in the `Fragment onCreate()` method.

11 dynamic fragments

Nesting Fragments



The Back button was going crazy, transactions everywhere. So I hit them with the getChildFragmentManager() method and BAM! Everything went back to normal.

So far you've seen how to create and use static fragments.

But what if you want your fragments to be more **dynamic**? Dynamic fragments have a lot in common with dynamic activities, but there are crucial differences you need to be able to deal with. In this chapter you'll see how to **convert dynamic activities** into **working dynamic fragments**. You'll find out how to use **fragment transactions** to help **maintain your fragment state**. Finally, you'll discover how to **nest one fragment inside another**, and how the **child fragment manager** helps you control unruly back stack behavior.

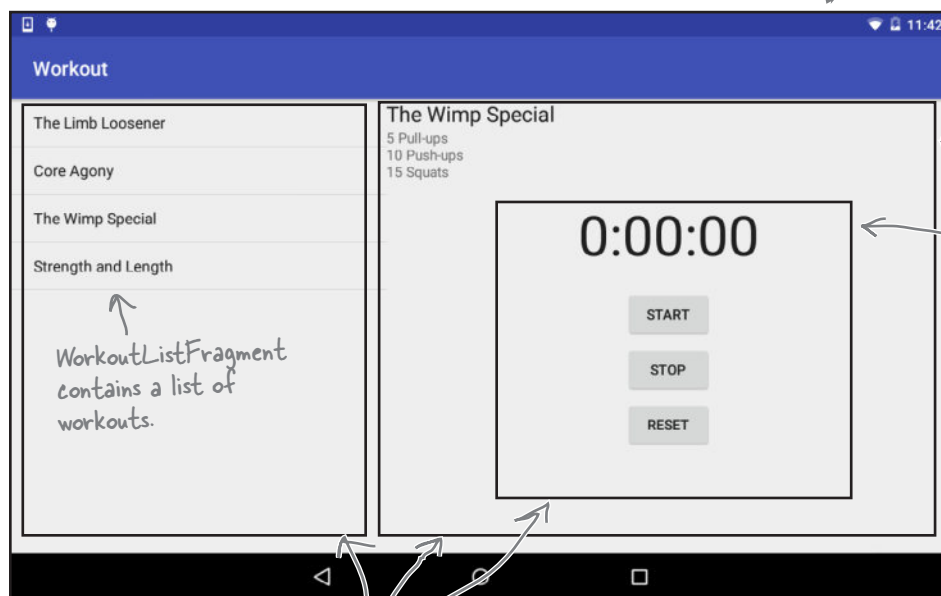
Adding dynamic fragments

In Chapters 9 and 10, you saw how to create fragments, how to include them in activities, and how to connect them together. To do this, we created a list fragment displaying a list of workouts, and a fragment displaying details of a single workout.

These fragments we've created so far have both been static. Once the fragments are displayed, their contents don't change. We may completely replace the fragment that's displayed with a new instance, but we can't update the contents of the fragment itself.

In this chapter we're going to look at how you deal with a fragment that's more dynamic. By this, we mean a fragment whose views gets updated after the fragment is displayed. To learn how to do, we're going to change the stopwatch *activity* we created in Chapter 4 into a stopwatch *fragment*. We're going to add our new stopwatch fragment to `WorkoutDetailFragment` so that it's displayed underneath the details of the workout.

We're only showing the tablet version of the app here, but the new stopwatch fragment will appear in the phone version too.



`WorkoutDetailFragment` displays details of the workout the user clicks on.

We're going to add a stopwatch fragment to `WorkoutDetailFragment`.

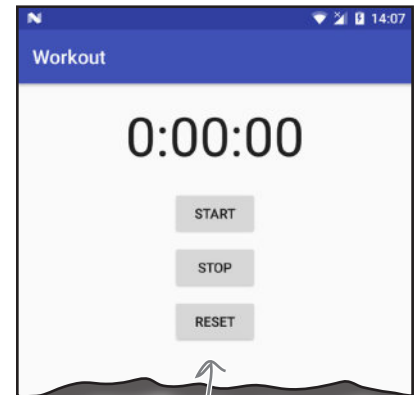
These lines won't appear in the actual app. We've added them here to show you each of the fragments.

Here's what we're going to do

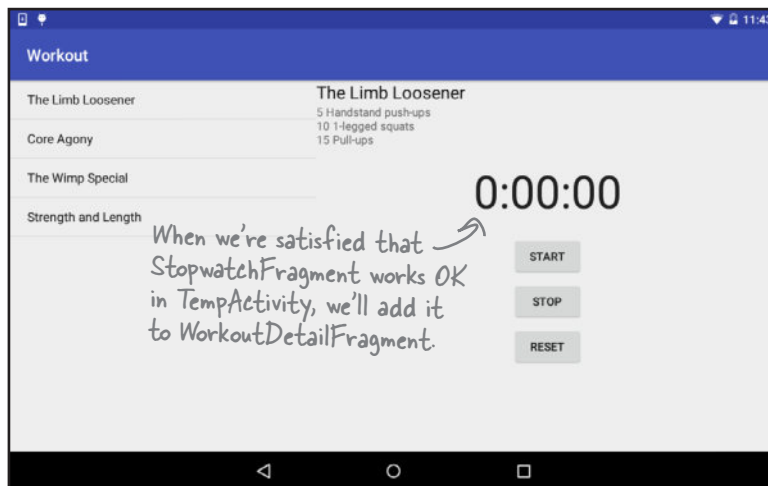
There are a number of steps we'll go through to change the app to display the stopwatch:

- 1 **Convert StopwatchActivity into StopwatchFragment.**
We'll take the `StopwatchActivity` code we created in Chapter 4, and change it into fragment code. We'll also display it in a new temporary activity called `TempActivity` so that we can check that it works. We'll temporarily change the app so that `TempActivity` starts when the app gets launched.
- 2 **Test StopwatchFragment.**
The `StopwatchActivity` included Start, Stop, and Reset buttons. We need to check that these still work when the stopwatch code is in a fragment.

We also need to test what happens to `StopwatchFragment` when the user rotates the device.
- 3 **Add StopwatchFragment to WorkoutDetailFragment.**
Once we're satisfied that `StopwatchFragment` works, we'll add it to `WorkoutDetailFragment`.



We'll start by adding `StopwatchFragment` to a new activity called `TempActivity`.



Let's get started.

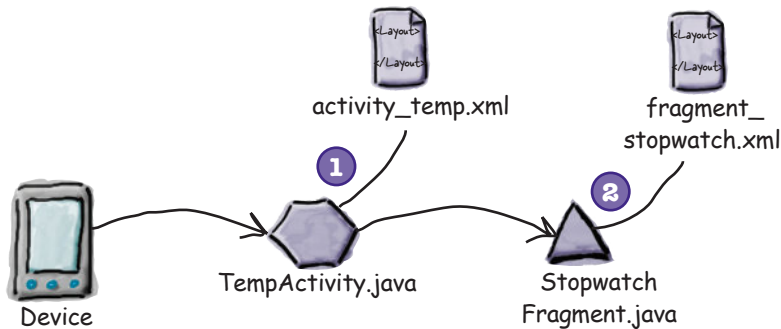
We're going to update the Workout app in this chapter, so open your original Workout project from Chapter 9 in Android Studio.

The new version of the app

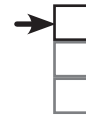
We're going to change our app to get `StopwatchFragment` working in a new temporary activity called `TempActivity`. This will enable us to confirm that `StopwatchFragment` works before we add it to `WorkoutDetailFragment` later in the chapter.

Here's how the new version of the app will work:

- 1 **When the app gets launched, it starts `TempActivity`.**
`TempActivity` uses `activity_temp.xml` for its layout, and contains a fragment, `StopwatchFragment`.
- 2 **`StopwatchFragment` displays a stopwatch with `Start`, `Stop`, and `Reset` buttons.**



All of the other activities and fragments we created in Chapters 9 and 10 will still exist in the project, but we're not going to do anything with them until later in the chapter.



Convert stopwatch
Test stopwatch
Add to fragment

Create TempActivity

We'll start by creating TempActivity. Create a new empty activity by switching to the Project view of Android Studio's explorer, highlighting the *com.hfad.workout* package in the *app/src/main/java* folder, going to the File menu and choosing New...→Activity→Empty Activity. Name the activity "TempActivity", name the layout "activity_temp", make sure the package name is *com.hfad.workout*, and **check the Backwards Compatibility (AppCompat) checkbox**.

If prompted for the activity's source language, select the option for Java.

We're going to change our app so that, when it's launched, it starts TempActivity instead of MainActivity. To do this, we need to move MainActivity's launcher intent filter to TempActivity instead. Open the file *AndroidManifest.xml* in the *app/src/main* folder, then make the following changes:

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

    <application
        ...
        <activity android:name=".MainActivity">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
        <activity android:name=".DetailActivity" />
        <activity android:name=".TempActivity">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>

</manifest>
```



This bit specifies that it's the main activity of the app.

↑
This says the activity can be used to launch the app.

We'll update TempActivity on the next page.

TempActivity needs to extend AppCompatActivity

All of the fragments we're using in this app come from the Support Library. As we said back in Chapter 9, all activities that use Support Library fragments must extend the `FragmentActivity` class or one of its subclasses such as `AppCompatActivity`. If they don't, the code will break.

All of the other activities we've created in this app extend `AppCompatActivity`, so we'll make `TempActivity` extend this class too. Here's our code for `TempActivity.java`. Update your version of the code so that it matches ours below:

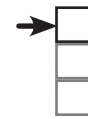
```
package com.hfad.workout;

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

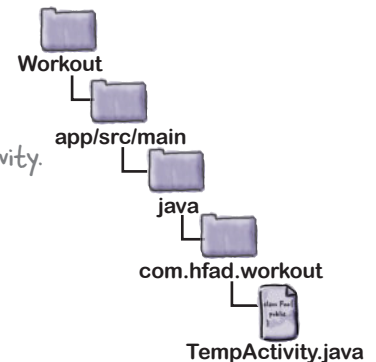
public class TempActivity extends AppCompatActivity {

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

The activity extends AppCompatActivity.



Convert stopwatch
Test stopwatch
Add to fragment



We'll add a new stopwatch fragment

We're going to add a new stopwatch fragment called `StopwatchFragment.java` that uses a layout called `fragment_stopwatch.xml`. We're going to base the fragment on the stopwatch activity we created back in Chapter 4.

We already know that activities and fragments behave in similar ways, but we also know that a fragment is a distinct type of object—a fragment is not a subclass of activity. **Is there some way we could rewrite that stopwatch activity code so that it works like a fragment?**

Fragments and activities have similar lifecycles...

To understand how to rewrite an activity as a fragment, we need to think a little about the similarities and differences between them. If we look at the lifecycles of fragments and activities, we'll see that they're very similar:

Lifecycle method	Activity	Fragment
<code>onAttach()</code>		✓
<code>onCreate()</code>	✓	✓
<code>onCreateView()</code>		✓
<code>onActivityCreated()</code>		✓
<code>onStart()</code>	✓	✓
<code>onPause()</code>	✓	✓
<code>onResume()</code>	✓	✓
<code>onStop()</code>	✓	✓
<code>onDestroyView()</code>		✓
<code>onRestart()</code>	✓	
<code>onDestroy()</code>	✓	✓
<code>onDetach()</code>		✓

...but the methods are slightly different

Fragment lifecycle methods are almost the same as activity lifecycle methods, but there's one major difference: activity lifecycle methods are **protected** and fragment lifecycle methods are **public**. And we've already seen that the ways that activities and fragments create a layout from a layout resource file are different.

Also, in a fragment, we can't call methods like `findViewById()` directly. Instead, we need to find a reference to a `View` object, and then call the view's `findViewById()` method.

With these similarities and differences in mind, it's time you started to write some code...



Sharpen your pencil

This is the code for `StopwatchActivity` we wrote earlier. You're going to convert this code into a fragment called `StopwatchFragment`. With a pencil, make the changes you need. Keep the following things in mind:

- Instead of a layout file called `activity_stopwatch.xml`, it will use a layout called `fragment_stopwatch.xml`.
- Make sure the access restrictions on the methods are correct.
- How will you specify the layout?
- The `runTimer()` method won't be able to call `findViewById()`, so you might want to pass a `View` object into `runTimer()`.

```
public class StopwatchActivity extends Activity {
    //Number of seconds displayed on the stopwatch.
    private int seconds = 0; ← The number of seconds that have passed
    //Is the stopwatch running?
    private boolean running; ← running says whether the stopwatch is running.
    private boolean wasRunning; ← wasRunning says whether the stopwatch was running
                                before the stopwatch was paused.

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_stopwatch);
        if (savedInstanceState != null) {
            seconds = savedInstanceState.getInt("seconds");
            running = savedInstanceState.getBoolean("running");
            wasRunning = savedInstanceState.getBoolean("wasRunning");
        }
        runTimer(); ← Start the runTimer() method.
    }

    @Override
    protected void onPause() { ← Stop the stopwatch if the activity is paused.
        super.onPause();
        wasRunning = running;
        running = false;
    }
}
```

If the activity was destroyed and recreated, restore the state of the variables from the savedInstanceState Bundle.

```

@Override
protected void onResume() {
    super.onResume();
    if (wasRunning) {
        running = true;
    }
}

@Override
public void onSaveInstanceState(Bundle savedInstanceState) {
    savedInstanceState.putInt("seconds", seconds);
    savedInstanceState.putBoolean("running", running);
    savedInstanceState.putBoolean("wasRunning", wasRunning);
}

public void onClickStart(View view) {
    running = true;
}

public void onClickStop(View view) {
    running = false;
}

public void onClickReset(View view) {
    running = false;
    seconds = 0;
}

private void runTimer() {
    final TextView timeView = (TextView) findViewById(R.id.time_view);
    final Handler handler = new Handler();
    handler.post(new Runnable() {
        @Override
        public void run() {
            int hours = seconds/3600;
            int minutes = (seconds%3600)/60;
            int secs = seconds%60;
            String time = String.format(Locale.getDefault(),
                "%d:%02d:%02d", hours, minutes, secs);
            timeView.setText(time);
            if (running) {
                seconds++;
            }
            handler.postDelayed(this, 1000);
        }
    });
}
}

```

Start the stopwatch if the activity is resumed.

Save the activity's state before the activity is destroyed.

Start, stop, or reset the stopwatch depending on which button is clicked.

Use a Handler to post code to increment the number of seconds and update the text view every second.



Sharpen your pencil

Solution

This is the code for `StopwatchActivity` we wrote earlier. You're going to convert this code into a fragment called `StopwatchFragment`. With a pencil, make the changes you need. Keep the following things in mind:

- Instead of a layout file called `activity_stopwatch.xml`, it will use a layout called `fragment_stopwatch.xml`.
- Make sure the access restrictions on the methods are correct.
- How will you specify the layout?
- The `runTimer()` method won't be able to call `findViewById()`, so you might want to pass a `View` object into `runTimer()`.

This is the new name.

```

public class StopwatchActivity StopwatchFragment extends Activity Fragment
    //Number of seconds displayed on the stopwatch.
    private int seconds = 0;
    //Is the stopwatch running?
    private boolean running;
    private boolean wasRunning;

    @Override
    protected public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_stopwatch);
        if (savedInstanceState != null) {
            seconds = savedInstanceState.getInt("seconds");
            running = savedInstanceState.getBoolean("running");
            wasRunning = savedInstanceState.getBoolean("wasRunning");
        }
        runTimer();
    }

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

    @Override
    protected public void onPause() {
        super.onPause();
        wasRunning = running;
        running = false;
    }

```

← This method needs to be public.
 ← You don't set a fragment's layout in its `onCreate()` method.
 ← We're not calling `runTimer()` yet because we've not set the layout—we don't have any views yet.
 ← We set the fragment's layout in the `onCreateView()` method.
 ← We're extending `Fragment`, not `Activity`.
 ← This method needs to be public.

```

@Override
protected public void onResume() {
    super.onResume();
    if (wasRunning) {
        running = true;
    }
}

@Override
public void onSaveInstanceState(Bundle savedInstanceState) {
    savedInstanceState.putInt("seconds", seconds);
    savedInstanceState.putBoolean("running", running);
    savedInstanceState.putBoolean("wasRunning", wasRunning);
}

public void onClickStart(View view) {
    running = true;
}

public void onClickStop(View view) {
    running = false;
}

public void onClickReset(View view) {
    running = false;
    seconds = 0;
}

private void runTimer(View view) {
    final TextView timeView = (TextView) view.findViewById(R.id.time_view);
    final Handler handler = new Handler();
    handler.post(new Runnable() {
        @Override
        public void run() {
            int hours = seconds/3600;
            int minutes = (seconds%3600)/60;
            int secs = seconds%60;
            String time = String.format(Locale.getDefault(),
                "%d:%02d:%02d", hours, minutes, secs);
            timeView.setText(time);
            if (running) {
                seconds++;
            }
            handler.postDelayed(this, 1000);
        }
    });
}
}

```

← This method needs to be public.

← The runTimer() method now takes a View.

← Use the view parameter to call findViewById().

The StopwatchFragment.java code



Convert stopwatch
Test stopwatch
Add to fragment

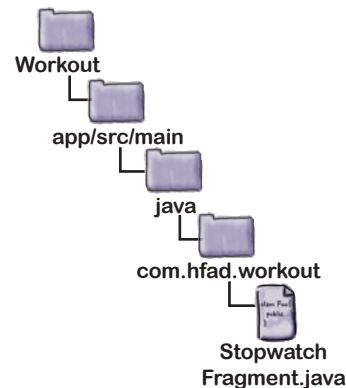
We'll add StopwatchFragment to our Workout project so that we can use it in our app. You do this in the same way you did in Chapter 9. Highlight the `com.hfad.workout` package in the `app/src/main/java` folder, then go to File→New...→Fragment→Fragment (Blank). Give the fragment a name of "StopwatchFragment", give it a layout name of "fragment_stopwatch", and uncheck

If prompted for the fragment's source language, select the option for Java.

the options for including fragment factory methods and interface callbacks. When you click on the Finish button, Android Studio creates a new fragment for you in a file called `StopwatchFragment.java` in the `app/src/main/java` folder. Replace the fragment code Android Studio gives you with the following code (this is the code you updated in the exercise on the previous page):

```
package com.hfad.workout;

import android.os.Bundle;
import android.os.Handler;
import android.support.v4.app.Fragment;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;
import android.widget.TextView;
import java.util.Locale;
```



```
public class StopwatchFragment extends Fragment {
    //Number of seconds displayed on the stopwatch.
    private int seconds = 0;
    //Is the stopwatch running?
    private boolean running;
    private boolean wasRunning;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        if (savedInstanceState != null) {
            seconds = savedInstanceState.getInt("seconds");
            running = savedInstanceState.getBoolean("running");
            wasRunning = savedInstanceState.getBoolean("wasRunning");
        }
    }
}
```

← The number of seconds that have passed

← running says whether the stopwatch is running.

← wasRunning says whether the stopwatch was running before the stopwatch was paused.

← Restore the state of the variables from the savedInstanceState Bundle.

← The code continues on the next page.



StopwatchFragment.java (continued)

```

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

```

Set the fragment's layout and start the runTimer() method, passing in the layout.

```

@Override
public void onPause() {
    super.onPause();
    wasRunning = running;
    running = false;
}

```

If the fragment's paused, record whether the stopwatch was running and stop it.

```

@Override
public void onResume() {
    super.onResume();
    if (wasRunning) {
        running = true;
    }
}

```

If the stopwatch was running before it was paused, set it running again.

```

@Override
public void onSaveInstanceState(Bundle savedInstanceState) {
    savedInstanceState.putInt("seconds", seconds);
    savedInstanceState.putBoolean("running", running);
    savedInstanceState.putBoolean("wasRunning", wasRunning);
}

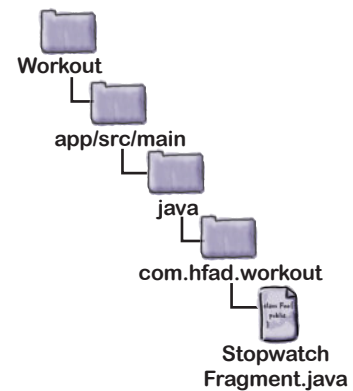
```

```

public void onClickStart(View view) {
    running = true;
}

```

This code needs to run when the user clicks on the Start button.



Put the values of the variables in the Bundle before the activity is destroyed. These are used when the user turns the device.

The code continues on the next page.

StopwatchFragment.java (continued)

```

public void onClickStop(View view) {
    running = false;
}

public void onClickReset(View view) {
    running = false;
    seconds = 0;
}

private void runTimer(View view) {
    final TextView timeView = (TextView) view.findViewById(R.id.time_view);
    final Handler handler = new Handler();
    handler.post(new Runnable() {
        @Override
        public void run() {
            int hours = seconds/3600;
            int minutes = (seconds%3600)/60;
            int secs = seconds%60;
            String time = String.format(Locale.getDefault(),
                "%d:%02d:%02d", hours, minutes, secs);
            timeView.setText(time);
            if (running) {
                seconds++;
            }
            handler.postDelayed(this, 1000);
        }
    });
}

```

This code needs to run when the user clicks on the Stop button.

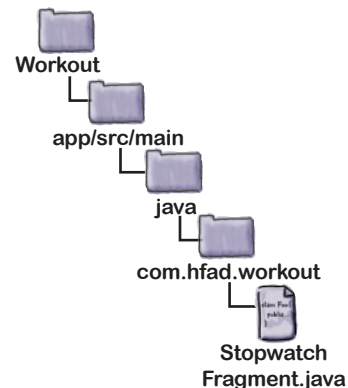
This code needs to run when the user clicks on the Reset button.

Putting the code in a Handler means it can run in the background thread.

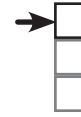
Display the number of seconds that have passed in the stopwatch.

If the stopwatch is running, increment the number of seconds.

Run the Handler code every second.



That's all the Java code we need for our StopwatchFragment. The next thing we need to do is say what the fragment should look like by updating the layout code Android Studio gave us.



The StopwatchFragment layout

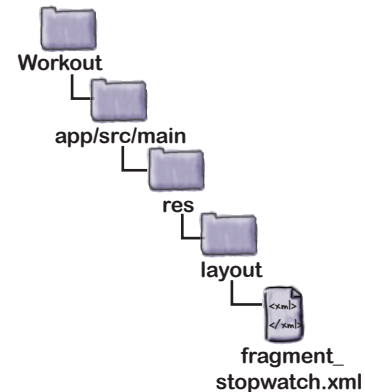
We'll use the same layout for StopwatchFragment as we used in our original Stopwatch app. To do so, replace the contents of `fragment_stopwatch.xml` with the code below:

```
<?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">
```

```
<TextView
    android:id="@+id/time_view"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_gravity="center_horizontal"
    android:textAppearance="@android:style/TextAppearance.Large"
    android:textSize="56sp" />
```

```
<Button
    android:id="@+id/start_button"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_gravity="center_horizontal"
    android:layout_marginTop="20dp"
    android:onClick="onClickStart"
    android:text="@string/start" />
```

```
<Button
    android:id="@+id/stop_button"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_gravity="center_horizontal"
    android:layout_marginTop="8dp"
    android:onClick="onClickStop"
    android:text="@string/stop" />
```

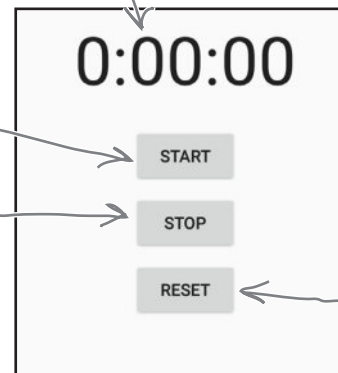


The number of hours, minutes, and seconds that have passed.

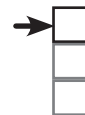
The Start button

The Stop button

The Reset button code is on the next page.



The StopwatchFragment layout (continued)



Convert stopwatch
Test stopwatch
Add to fragment

```

<Button
    android:id="@+id/reset_button" ← The Reset button
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_gravity="center_horizontal"
    android:layout_marginTop="8dp"
    android:onClick="onClickReset"
    android:text="@string/reset" />
</LinearLayout>
    
```

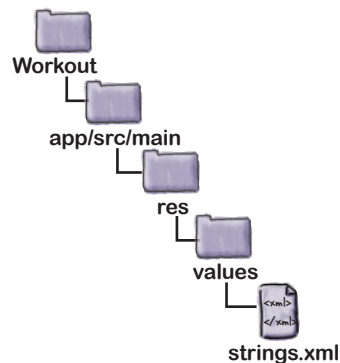
The StopwatchFragment layout uses String values

The XML code in *fragment_stopwatch.xml* uses string values for the text on the Start, Stop, and Reset buttons. We need to add these to *strings.xml*:

```

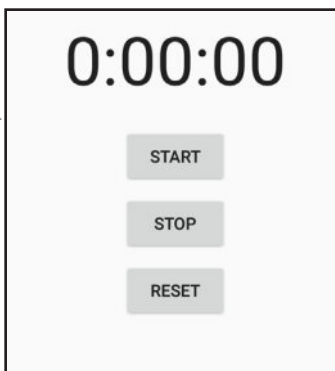
...
<string name="start">Start</string>
<string name="stop">Stop</string>
<string name="reset">Reset</string>
...
    
```

} These are the button labels.



The Stopwatch fragment looks just like it did when it was an activity. The difference is that we can now use it in other activities and fragments.

The stopwatch looks the same as it did when it was an activity. But because it's now a fragment, we can reuse it in different places.



The next thing we need to do is display it in TempActivity's layout.

Convert stopwatch
Test stopwatch
Add to fragment



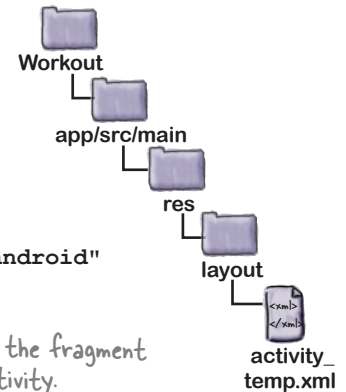
Add StopwatchFragment to TempActivity's layout

The simplest way of adding `StopwatchFragment` to `TempActivity`'s layout is to use the `<fragment>` element. Using the `<fragment>` element means that we can add the fragment directly into the layout instead of writing fragment transaction code.

Here's our code for `activity_temp.xml`. Replace the code that's currently in that file with this updated code:

```
<?xml version="1.0" encoding="utf-8"?>
<fragment xmlns:android="http://schemas.android.com/apk/res/android"
    android:name="com.hfad.workout.StopwatchFragment"
    android:layout_width="match_parent"
    android:layout_height="match_parent"/>
```

This adds the fragment to the activity.



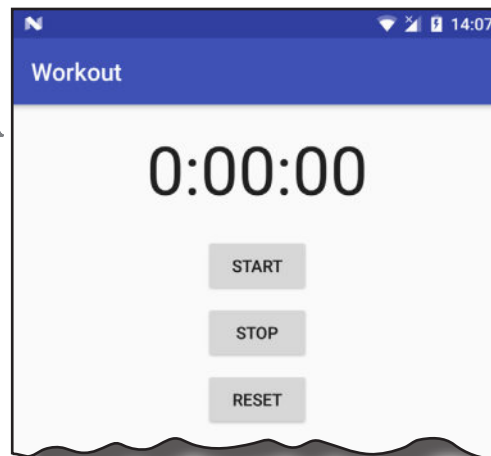
That's everything we need to see `StopwatchFragment` running. Let's take it for a test drive.



Test drive the app

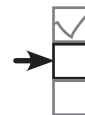
When we run the app, `TempActivity` is displayed. It contains `StopwatchFragment`. The stopwatch is set to 0.

When we run the app, `TempActivity` starts, not `MainActivity`. `TempActivity` displays `StopwatchFragment` as expected.



The next thing we'll do is check that `StopwatchFragment`'s buttons work OK.

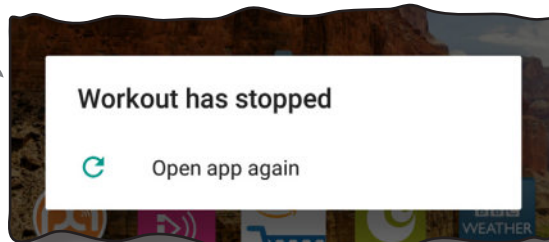
The app crashes if you click on a button



Convert stopwatch
Test stopwatch
Add to fragment

When you click on any one of the buttons in the Workout app's new stopwatch, the app crashes:

This is what happened when we clicked on the Start button in StopwatchFragment.



When we converted the stopwatch activity into a fragment, we didn't change any of the code relating to the buttons. We know this code worked great when the stopwatch was in an activity, so why should it cause the app to crash in a fragment?

Here's the error output from Android Studio. Can you see what may have caused the problem?

Yikes.

```
04-13 11:56:43.623 10583-10583/com.hfad.workout E/AndroidRuntime: FATAL EXCEPTION: main
Process: com.hfad.workout, PID: 10583
java.lang.IllegalStateException: Could not find method onClickStart(View) in a
parent or ancestor Context for android:onClick attribute defined on view class
android.support.v7.widget.AppCompatButton with id 'start_button'
    at android.support.v7.app.AppCompatActivity$DeclaredOnClickListener.
        resolveMethod(AppCompatActivity.java:327)
    at android.support.v7.app.AppCompatActivity$DeclaredOnClickListener.
        onClick(AppCompatActivity.java:284)
    at android.view.View.performClick(View.java:5609)
    at android.view.View$PerformClick.run(View.java:22262)
    at android.os.Handler.handleCallback(Handler.java:751)
    at android.os.Handler.dispatchMessage(Handler.java:95)
    at android.os.Looper.loop(Looper.java:154)
    at android.app.ActivityThread.main(ActivityThread.java:6077)
    at java.lang.reflect.Method.invoke(Native Method)
    at com.android.internal.os.ZygoteInit$MethodAndArgsCaller.
        run(ZygoteInit.java:865)
    at com.android.internal.os.ZygoteInit.main(ZygoteInit.java:755)
```

Let's look at the StopwatchFragment layout code

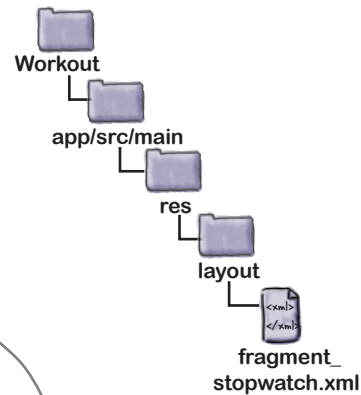
In the layout code for the StopwatchFragment, we're binding the buttons to methods in the same way that we did for an activity, by using the `android:onClick` attribute to say which method should be called when each button is clicked:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    ...
    <Button
        android:id="@+id/start_button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="center_horizontal"
        android:layout_marginTop="20dp"
        android:onClick="onClickStart"
        android:text="@string/start" />

    <Button
        android:id="@+id/stop_button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="center_horizontal"
        android:layout_marginTop="8dp"
        android:onClick="onClickStop"
        android:text="@string/stop" />

    <Button
        android:id="@+id/reset_button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="center_horizontal"
        android:layout_marginTop="8dp"
        android:onClick="onClickReset"
        android:text="@string/reset" />
</LinearLayout>
```

We're using the same layout for the stopwatch now that it's a fragment as we did when it was an activity.



We're using the `android:onClick` attributes in the layout to say which methods should be called when each button is clicked.

This worked OK when we were using an activity, so why should we have a problem now that we're using a fragment?

The `onClick` attribute calls methods in the activity, not the fragment

There's a big problem with using the `android:onClick` attribute to say which method should be called when a view is clicked. The attribute specifies which method should be called in the **current activity**. This is fine when the views are in an *activity's* layout. But when the views are in a *fragment*, this leads to problems. Instead of calling methods in the fragment, Android calls methods in the parent activity. If it can't find the methods in this activity, the app crashes. That's what Android Studio's error message was trying to tell us.

It's not just buttons that have this problem. The `android:onClick` attribute can be used with any views that are subclasses of the `Button` class. This includes checkboxes, radio buttons, switches, and toggle buttons.

Now we *could* move the methods out of the fragment and into the activity, but that approach has a major disadvantage. It would mean that the fragment is no longer self-contained—if we wanted to reuse the fragment in another activity, we'd need to include the code in *that* activity too. Instead, we'll deal with it in the fragment.

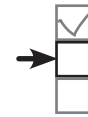
How to make button clicks call methods in the fragment

There are three things you need to do in order to get buttons in a fragment to call methods in the fragment instead of the activity:

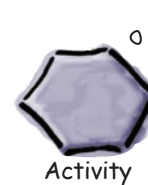
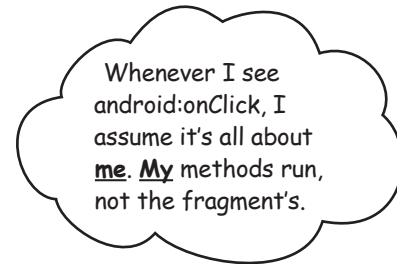
- 1 Remove references to `android:onClick` in the fragment layout.**
Buttons attempt to call methods in the activity when the `onClick` attribute is used, so these need to be removed from the fragment layout.
- 2 Optionally, change the `onClick` method signatures.**
When we created our `onClickStart()`, `onClickStop()`, and `onClickReset()` methods, we made them public and gave them a single `View` parameter. This was so they'd get called when the user clicked on a button. As we're no longer using the `android:onClick` attribute in our layout, we can set our methods to private and remove the `View` parameter.
- 3 Bind the buttons to methods in the fragment by implementing an `OnClickListener`.**
This will ensure that the right methods are called when the buttons are clicked.

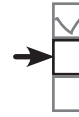
This step's optional, but it's a good opportunity to tidy up our code.

Let's do this now in our `StopwatchFragment`.



Convert stopwatch
Test stopwatch
Add to fragment





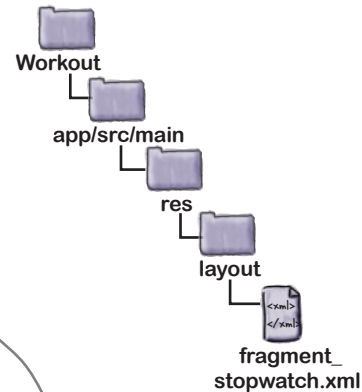
1. Remove the onClick attributes from the fragment's layout

The first thing we'll do is remove the `android:onClick` lines of code from the fragment's layout. This will stop Android from trying to call methods in the activity when the buttons are clicked:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    ...
    <Button
        android:id="@+id/start_button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="center_horizontal"
        android:layout_marginTop="20dp"
        android:onClick="onClickStart"
        android:text="@string/start" />

    <Button
        android:id="@+id/stop_button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="center_horizontal"
        android:layout_marginTop="8dp"
        android:onClick="onClickStop"
        android:text="@string/stop" />

    <Button
        android:id="@+id/reset_button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="center_horizontal"
        android:layout_marginTop="8dp"
        android:onClick="onClickReset"
        android:text="@string/reset" />
</LinearLayout>
```



Remove the `onClick` attributes for each of the buttons in the stopwatch.

The next thing we'll do is tidy up our `onClickStart()`, `onClickStop()`, and `onClickReset()` code.

2. Change the onClick... method signatures



Convert stopwatch
Test stopwatch
Add to fragment

Back in Chapter 4, when we created our `onClickStart()`, `onClickStop()`, and `onClickReset()` methods in `StopwatchActivity`, we had to give them a specific method signature like this:

```

The methods had to be public. → public void onClickStart(View view) {
                                }
                                ↑
                                The methods had to have a void return value.
                                ↑
                                The methods had to have a single parameter of type View.
    
```

The methods had to take this form so that they'd respond when the user clicked on a button. Behind the scenes, when you use the `android:onClick` attribute, Android looks for a public method with a void return value, and with a name that matches the method specified in the layout XML.

Now that our code is in a fragment and we're no longer using the `android:onClick` attribute in our layout code, we can change our method signatures like this:

```

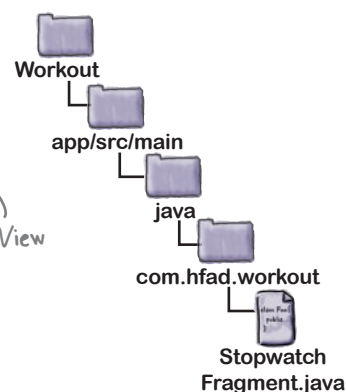
Our methods no longer need to be public, so we can make them private. → private void onClickStart() {
                                                                    }
                                                                    ↑
                                                                    We no longer need the View parameter.
    
```

So let's update our fragment code. Change the `onClickStart()`, `onClickStop()`, and `onClickReset()` methods in `StopwatchFragment.java` to match ours:

```

...
public private void onClickStart(View view) {
    running = true;
}
Change the methods to private. → public private void onClickStop(View view) {
    running = false;
}
public private void onClickReset(View view) {
    running = false;
    seconds = 0;
}
...
    
```

Remove the View parameters.



3. Make the fragment implement OnClickListener

To make the buttons call methods in StopwatchFragment when they are clicked, we'll make the fragment implement the View.OnClickListener interface like this:

```
public class StopwatchFragment extends Fragment implements View.OnClickListener {
    ...
}
```

This turns the fragment into an OnClickListener.

This turns StopwatchFragment into a type of View.OnClickListener so that it can respond when views are clicked.

You tell the fragment how to respond to clicks by implementing the View.OnClickListener onClick() method. This method gets called whenever a view in the fragment is clicked.

```
@Override
public void onClick(View v) {
    ...
}
```

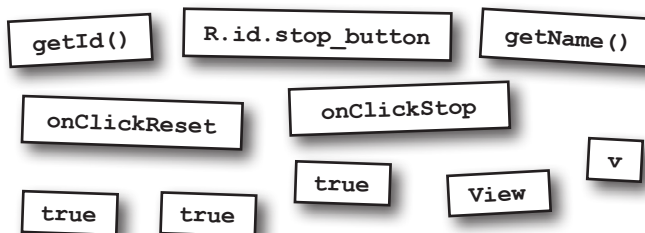
← You must override the onClick() method in your fragment code.

The onClick() method has a single View parameter. This is the view that the user clicks on. You can use the view's getId() method to find out which view the user clicked on, and then decide how to react.



Code Magnets

See if you can complete the StopwatchFragment onClick() method. You need to call the onClickStart() method when the Start button is clicked, the onClickStop() method when the Stop button is clicked, and the onClickReset() method when the Reset button is clicked.



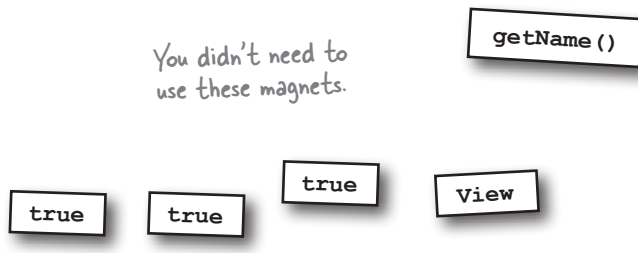
```
@Override
public void onClick(View v) {
    switch (.....:.....) {
        case R.id.start_button:
            onClickStart();
            break;
        case .....:
            .....();
            break;
        case R.id.reset_button:
            .....();
    }
}
```



Code Magnets Solution

See if you can complete the `StopwatchFragment.onClick()` method. You need to call the `onClickStart()` method when the Start button is clicked, the `onClickStop()` method when the Stop button is clicked, and the `onClickReset()` method when the Reset button is clicked.

You didn't need to use these magnets.



```
@Override
public void onClick(View v) {
    switch (v.getId()) {
        case R.id.start_button:
            onClickStart();
            break;
        case R.id.stop_button:
            onClickStop();
            break;
        case R.id.reset_button:
            onClickReset();
    }
}
```

The StopwatchFragment onClick() method

We need to make a few changes to `StopwatchFragment.java`; we'll show you the changes one at a time, then the fully updated code a couple of pages ahead.

Here's the code to implement the `StopwatchFragment.onClick()` method so that the correct method gets called when each button is clicked:

```
@Override
public void onClick(View v) {
    switch (v.getId()) {
        case R.id.start_button:
            onClickStart();
            break;
        case R.id.stop_button:
            onClickStop();
            break;
        case R.id.reset_button:
            onClickReset();
            break;
    }
}
```

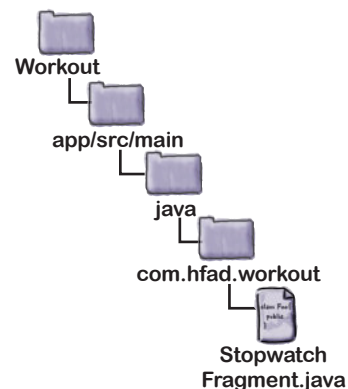
← This is the View the user clicked on.

← Check which View was clicked.

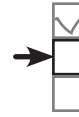
← If the Start button was clicked, call the `onClickStart()` method.

← If the Stop button was clicked, call the `onClickStop()` method.

← If the Reset button was clicked, call the `onClickReset()` method.



There's just one more thing we need to do to get our buttons working: attach the listener to the buttons in the fragment.



Attach the OnClickListener to the buttons

To make views respond to clicks, you need to call each view's `setOnClickListener()` method. The `setOnClickListener()` method takes an `OnClickListener` object as a parameter. Because `StopwatchFragment` implements the `OnClickListener` interface, we can use the keyword `this` to pass the fragment as the `OnClickListener` in the `setOnClickListener()` method.

As an example, here's how you attach the `OnClickListener` to the Start button:

```
Button startButton = (Button) layout.findViewById(R.id.start_button);
startButton.setOnClickListener(this);
```

Get a reference to the button. (arrow pointing to `findViewById`)

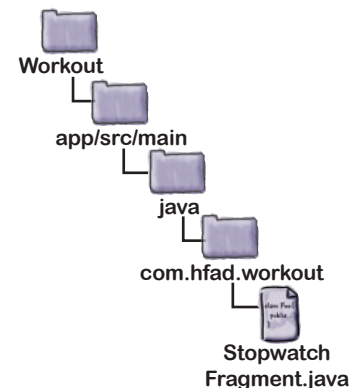
Attach the listener to the button. (arrow pointing to `setOnClickListener`)

The call to each view's `setOnClickListener()` method needs to be made after the fragment's views have been created. This means they need to go in the `StopwatchFragment onCreateView()` method like this:

```
@Override
public View onCreateView(LayoutInflater inflater, ViewGroup container,
    Bundle savedInstanceState) {
    View layout = inflater.inflate(R.layout.fragment_stopwatch, container, false);
    runTimer(layout);
    Button startButton = (Button) layout.findViewById(R.id.start_button);
    startButton.setOnClickListener(this);
    Button stopButton = (Button) layout.findViewById(R.id.stop_button);
    stopButton.setOnClickListener(this);
    Button resetButton = (Button) layout.findViewById(R.id.reset_button);
    resetButton.setOnClickListener(this);
    return layout;
}
```

This attaches the listener to each of the buttons. (arrow pointing to `setOnClickListener` calls)

We'll show you the full `StopwatchFragment` code on the next page.



The StopwatchFragment code

Here's the revised code for *StopwatchFragment.java*; update your version to match ours:

```
package com.hfad.workout;

import java.util.Locale;
import android.os.Bundle;
import android.os.Handler;
import android.support.v4.app.Fragment;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;
import android.widget.TextView;
import android.widget.Button;
```

We're using the Button class, so we'll import it.

```
public class StopwatchFragment extends Fragment implements View.OnClickListener {
    //Number of seconds displayed on the stopwatch.
    private int seconds = 0;
    //Is the stopwatch running?
    private boolean running;
    private boolean wasRunning;
```

The fragment needs to implement the View.OnClickListener interface.

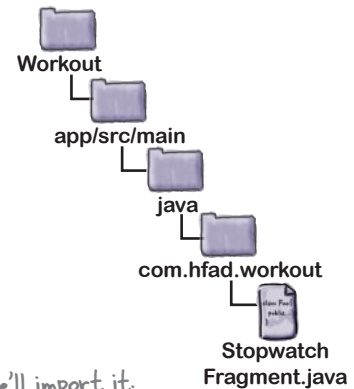
```
@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    if (savedInstanceState != null) {
        seconds = savedInstanceState.getInt("seconds");
        running = savedInstanceState.getBoolean("running");
        wasRunning = savedInstanceState.getBoolean("wasRunning");
    }
}
```

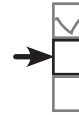
We're not changing the onCreate() method.

```
@Override
public View onCreateView(LayoutInflater inflater, ViewGroup container,
    Bundle savedInstanceState) {
    View layout = inflater.inflate(R.layout.fragment_stopwatch, container, false);
    runTimer(layout);
    Button startButton = (Button)layout.findViewById(R.id.start_button);
    startButton.setOnClickListener(this);
    Button stopButton = (Button)layout.findViewById(R.id.stop_button);
    stopButton.setOnClickListener(this);
    Button resetButton = (Button)layout.findViewById(R.id.reset_button);
    resetButton.setOnClickListener(this);
    return layout;
}
```

Update the onCreateView() method to attach the listener to the buttons.

The code continues on the next page.





The StopwatchFragment code (continued)

```

@Override
public void onClick(View v) {
    switch (v.getId()) {
        case R.id.start_button:
            onClickStart();
            break;
        case R.id.stop_button:
            onClickStop();
            break;
        case R.id.reset_button:
            onClickReset();
            break;
    }
}

@Override
public void onPause() {
    super.onPause();
    wasRunning = running;
    running = false;
}

@Override
public void onResume() {
    super.onResume();
    if (wasRunning) {
        running = true;
    }
}

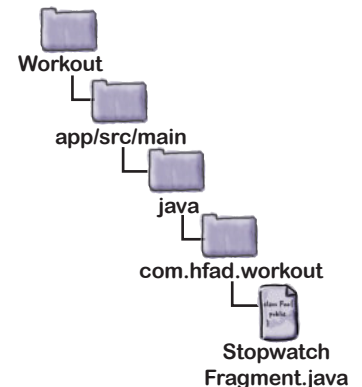
@Override
public void onSaveInstanceState(Bundle savedInstanceState) {
    savedInstanceState.putInt("seconds", seconds);
    savedInstanceState.putBoolean("running", running);
    savedInstanceState.putBoolean("wasRunning", wasRunning);
}

```

As we're implementing the `OnClickListener` interface, we need to override the `onClick()` method.

Call the appropriate method in the fragment for the button that was clicked.

We've not changed these methods.



The code continues on the next page. →

The StopwatchFragment code (continued)

```

private void onClickStart() {
    running = true;
}

private void onClickStop() {
    running = false;
}

private void onClickReset() {
    running = false;
    seconds = 0;
}

private void runTimer(View view) {
    final TextView timeView = (TextView) view.findViewById(R.id.time_view);
    final Handler handler = new Handler();
    handler.post(new Runnable() {
        @Override
        public void run() {
            int hours = seconds/3600;
            int minutes = (seconds%3600)/60;
            int secs = seconds%60;
            String time = String.format(Locale.getDefault(),
                "%d:%02d:%02d", hours, minutes, secs);
            timeView.setText(time);
            if (running) {
                seconds++;
            }
            handler.postDelayed(this, 1000);
        }
    });
}

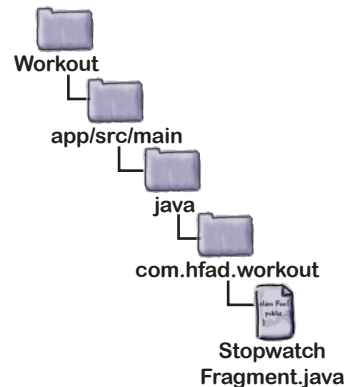
```

We've changed these methods so they're private. We've also removed the View parameter, as we no longer needed it.

We've not changed this method.



**Convert stopwatch
Test stopwatch
Add to fragment**

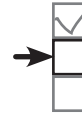


Those are all the code changes needed for *StopwatchFragment.java*.
Let's see what happens when we run the app.

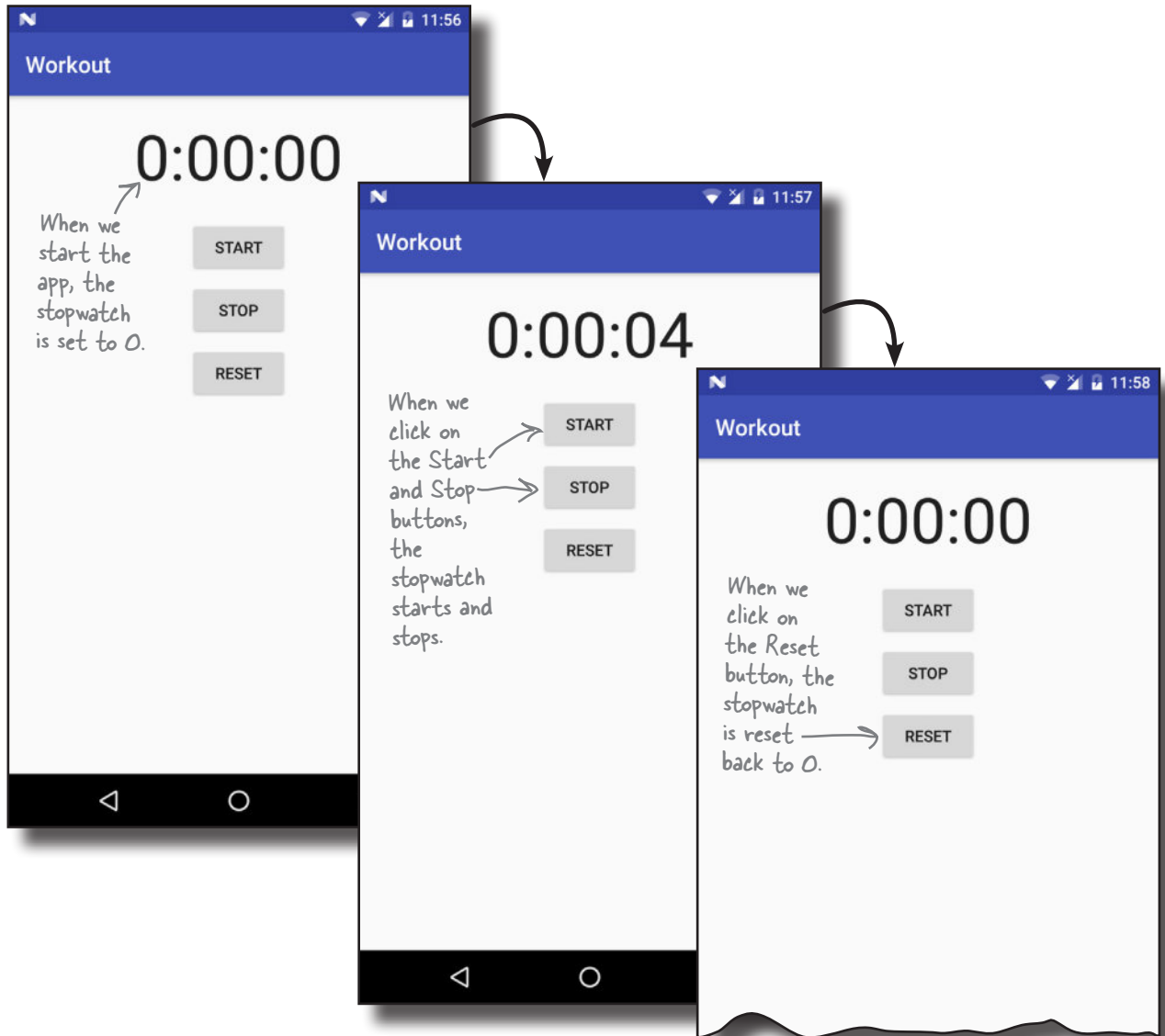


Test drive the app

When we run the app, the stopwatch is displayed as before. This time, however, the Start, Stop, and Reset buttons work.



dynamic fragments
Convert stopwatch
Test stopwatch
Add to fragment



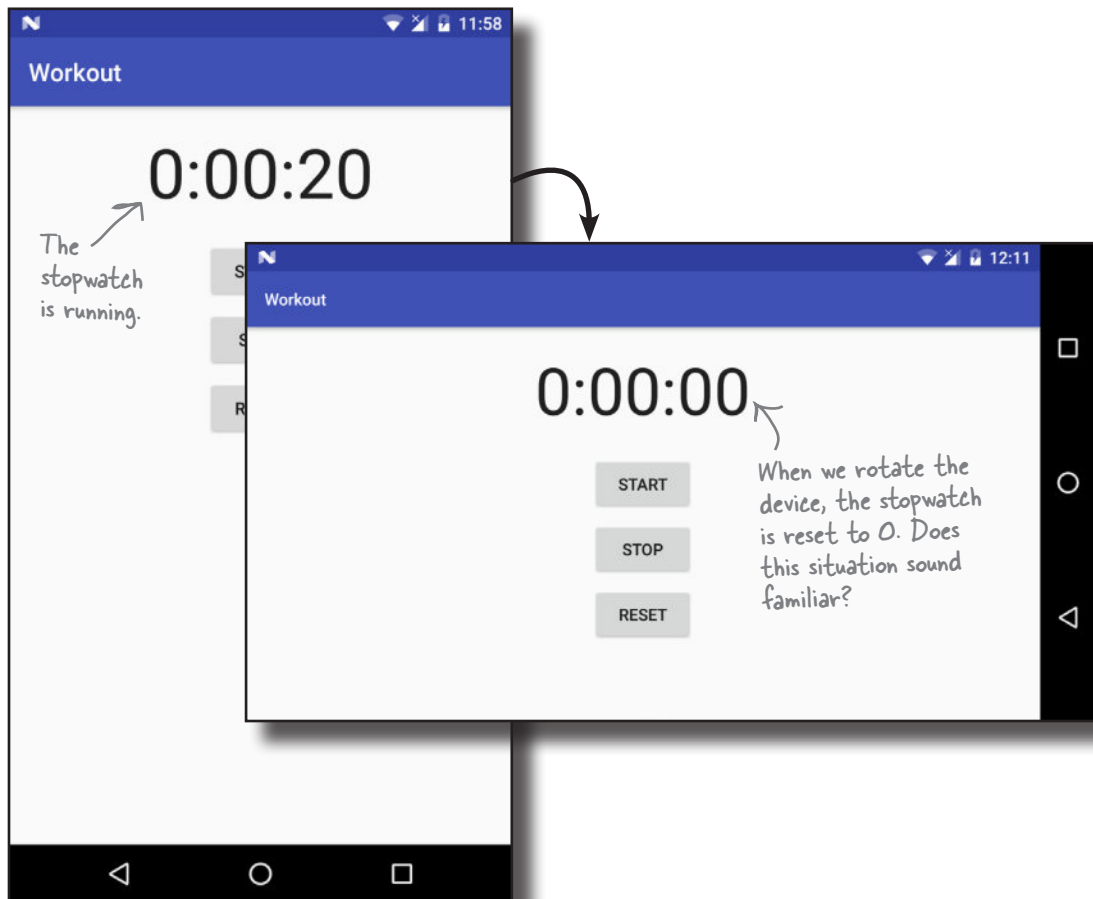
Now that we've got the buttons working, the next thing we need to test is what happens when we rotate the device.

Rotating the device resets the stopwatch

There's still one more problem we need to sort out. When we rotate our device, the stopwatch gets reset back to 0.

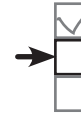


Convert stopwatch
Test stopwatch
Add to fragment



We encountered a similar problem when we first created `StopwatchActivity` back in Chapter 4. `StopwatchActivity` lost the state of any instance variables when it was rotated because activities are destroyed and recreated when the device is rotated. We solved this problem by saving and restoring the state of any instance variables used by the stopwatch.

This time, the problem isn't due to the code in `StopwatchFragment`. Instead, it's because of how we're adding `StopwatchFragment` to `TempActivity`.



Use <fragment> for static fragments...

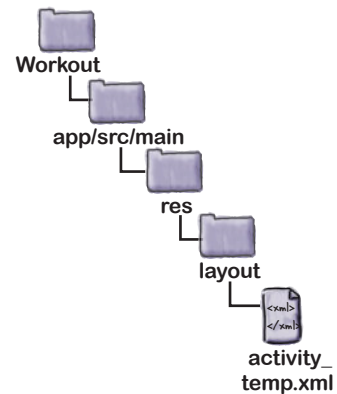
When we added StopwatchFragment to TempActivity, we did it by adding a <fragment> element to its layout like this:

```
<?xml version="1.0" encoding="utf-8"?>
<fragment xmlns:android="http://schemas.android.com/apk/res/android"
    android:name="com.hfad.workout.StopwatchFragment"
    android:layout_width="match_parent"
    android:layout_height="match_parent"/>
```

We did this because it was the simplest way to display our fragment in an activity and see it working.

As we said back in Chapter 9, the <fragment> element is a placeholder for where the fragment's layout should be inserted. When Android creates the activity's layout, it replaces the <fragment> element with the fragment's user interface.

When you rotate the device, Android recreates the activity. If your activity contains a <fragment> element, *it reinserts a new version of the fragment each time the activity is recreated*. The old fragment is discarded, and any instance variables are set back to their original values. In this particular example, this means that the stopwatch is set back to 0.



...but dynamic fragments need a fragment transaction

The <fragment> element works well for fragments that display static data. If you have a fragment that's dynamic, like our stopwatch, you need to add the fragment using a fragment transaction instead.

We're going to change TempActivity so that we no longer display StopwatchFragment using a <fragment>. Instead, we'll use a fragment transaction. To do this, we need to make changes to *activity_temp.xml* and *TempActivity.java*.

Change activity_temp.xml to use a FrameLayout

As you learned back in Chapter 10, when you want to add a fragment to an activity using a fragment transaction, you first need to add a placeholder for the fragment in the activity's layout. We did this in Chapter 10 by adding a frame layout to the layout, and giving it an ID so we could refer to it in our Java code.

We need to do the same thing with *activity_temp.xml*. We'll replace the `<fragment>` element with a frame layout, and give the frame layout an ID of `stopwatch_container`. Update your version of *activity_temp.xml* so that it reflects ours:

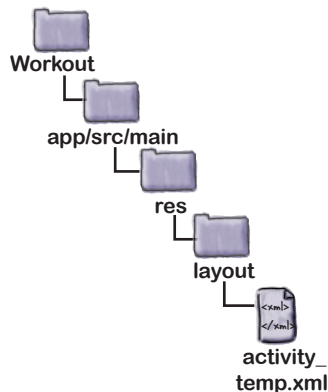
```
<?xml version="1.0" encoding="utf-8"?>
```

```
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:name="com.bfad.workout.StopwatchFragment"
    android:id="@+id/stopwatch_container"
    android:layout_width="match_parent"
    android:layout_height="match_parent"/>
```

Replace the fragment with a FrameLayout.



Convert stopwatch
Test stopwatch
Add to fragment



Delete this line.

Add a fragment transaction to TempActivity.java

Once you've added the frame layout to your activity's layout, you can create the fragment transaction that will add the fragment to the frame layout.

We want to add `StopwatchFragment` to `TempActivity` as soon as `TempActivity` gets created. We only want to add a new fragment, however, if one hasn't previously been added to it. We don't want to override any existing fragment.

To do this, we'll add code to `TempActivity`'s `onCreate()` method that checks whether the `savedInstanceState` `Bundle` parameter is null.

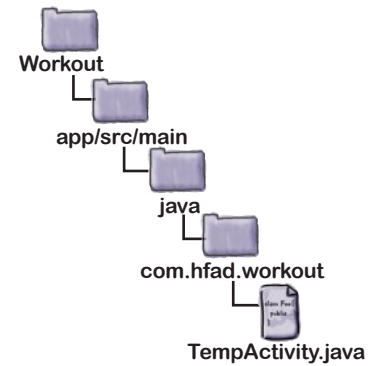
If `savedInstanceState` is null, this means that `TempActivity` is being created for the first time. In that case, we need to add `StopwatchFragment` to the activity.

If `savedInstanceState` is not null, that means that `TempActivity` is being recreated after having been destroyed. In that situation, we don't want to add a new instance of `StopwatchFragment` to the activity, as it would overwrite an existing fragment.

Pool Puzzle



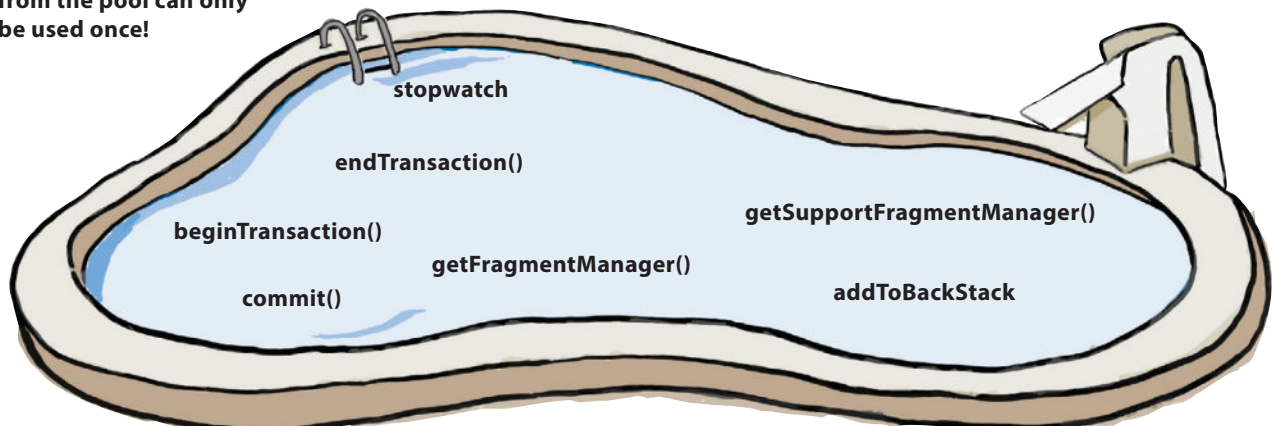
Your **job** is to take code snippets from the pool and place them into the blank lines in `TempActivity.java`. You may **not** use the same code snippet more than once, and you won't need to use all the code snippets. Your **goal** is to create a fragment transaction that will add an instance of `StopwatchFragment` to `TempActivity`.



```

@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_temp);
    if (savedInstanceState == null) {
        StopwatchFragment stopwatch = new StopwatchFragment();
        FragmentTransaction ft = .....;
        ft.add(R.id.stopwatch_container, .....);
        ft.....(null);
        ft.setTransition(FragmentTransaction.TRANSIT_FRAGMENT_FADE);
        ft. ....;
    }
}
  
```

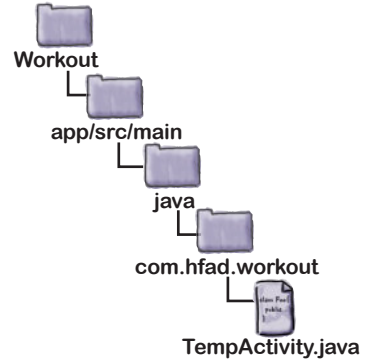
Note: each snippet from the pool can only be used once!



Pool Puzzle Solution



Your **job** is to take code snippets from the pool and place them into the blank lines in `TempActivity.java`. You may **not** use the same code snippet more than once, and you won't need to use all the code snippets. Your **goal** is to create a fragment transaction that will add an instance of `StopwatchFragment` to `TempActivity`.



```

@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_temp);
    if (savedInstanceState == null) {
        StopwatchFragment stopwatch = new StopwatchFragment();
        FragmentTransaction ft = getSupportFragmentManager().beginTransaction();
        ft.add(R.id.stopwatch_container, stopwatch);
        ft.addToBackStack(null);
        ft.setTransition(FragmentTransaction.TRANSIT_FRAGMENT_FADE);
        ft.commit();
    }
}
  
```

This begins the fragment transaction. We need to use `getSupportFragmentManager()`, not `getFragmentManager()`, as we're using fragments from the Support Library.

Add the transaction to the back stack.



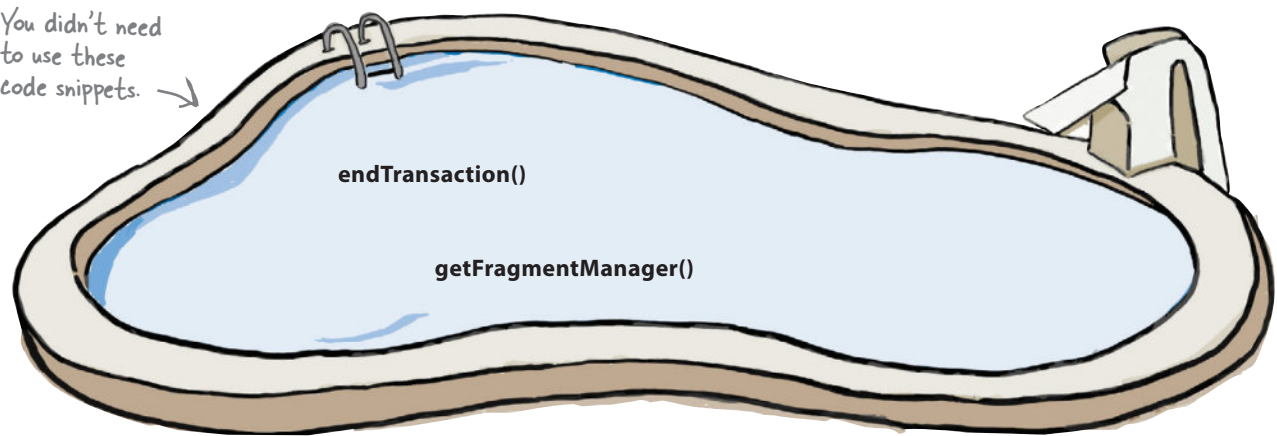
Add an instance of `StopwatchFragment` to `TempActivity`'s layout.

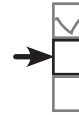


Commit the transaction.



You didn't need to use these code snippets.





The full code for TempActivity.java

We've added a fragment transaction to *TempActivity.java* that adds *StopwatchFragment* to *TempActivity*. Our full code is below. Update your version of *TempActivity.java* so that it matches ours.

```
package com.hfad.workout;
```

```
import android.support.v4.app.FragmentTransaction;
```

```
import android.support.v7.app.AppCompatActivity;
```

```
import android.os.Bundle;
```

```
public class TempActivity extends AppCompatActivity {
```

```
    @Override
```

```
    protected void onCreate(Bundle savedInstanceState) {
```

```
        super.onCreate(savedInstanceState);
```

```
        setContentView(R.layout.activity_temp);
```

```
        if (savedInstanceState == null) {
```

```
            StopwatchFragment stopwatch = new StopwatchFragment();
```

```
            FragmentTransaction ft = getSupportFragmentManager().beginTransaction();
```

```
            ft.add(R.id.stopwatch_container, stopwatch);
```

```
            ft.addToBackStack(null);
```

```
            ft.setTransition(FragmentTransaction.TRANSIT_FRAGMENT_FADE);
```

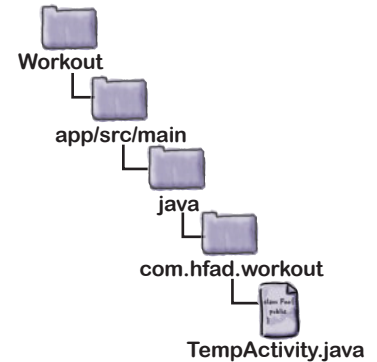
```
            ft.commit();
```

```
        }
```

```
    }
```

```
}
```

You need to import the *FragmentTransaction* class from the Support Library.



We only want to add the fragment if the activity isn't being recreated after having been destroyed.

Begin the fragment transaction.

Add the stopwatch, and add the transaction to the back stack.

Set the fragment transition to fade in and out.

Commit the transaction. This applies the changes.

Those are all the code changes we need to add *StopwatchFragment* to *TempActivity* using a fragment transaction. Let's see what happens when we run the code.

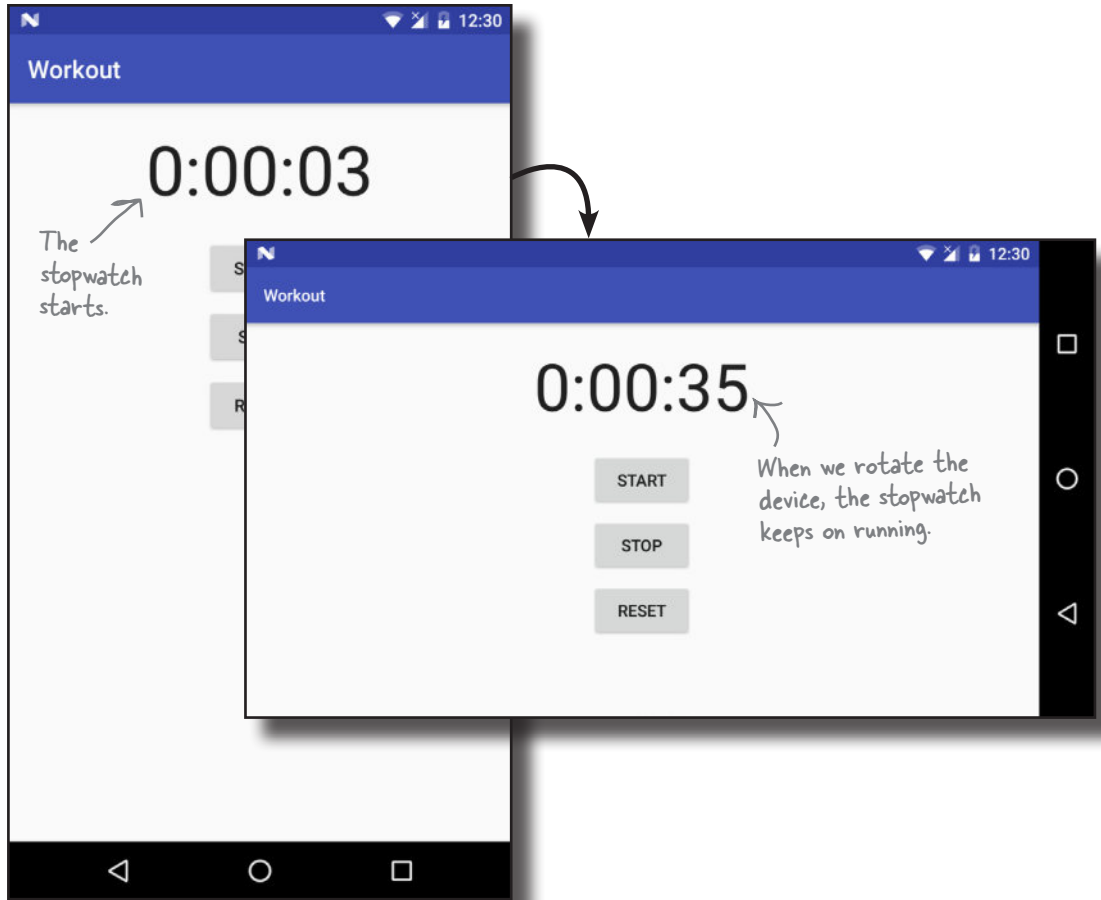


Test drive the app



- Convert stopwatch
- Test stopwatch
- Add to fragment

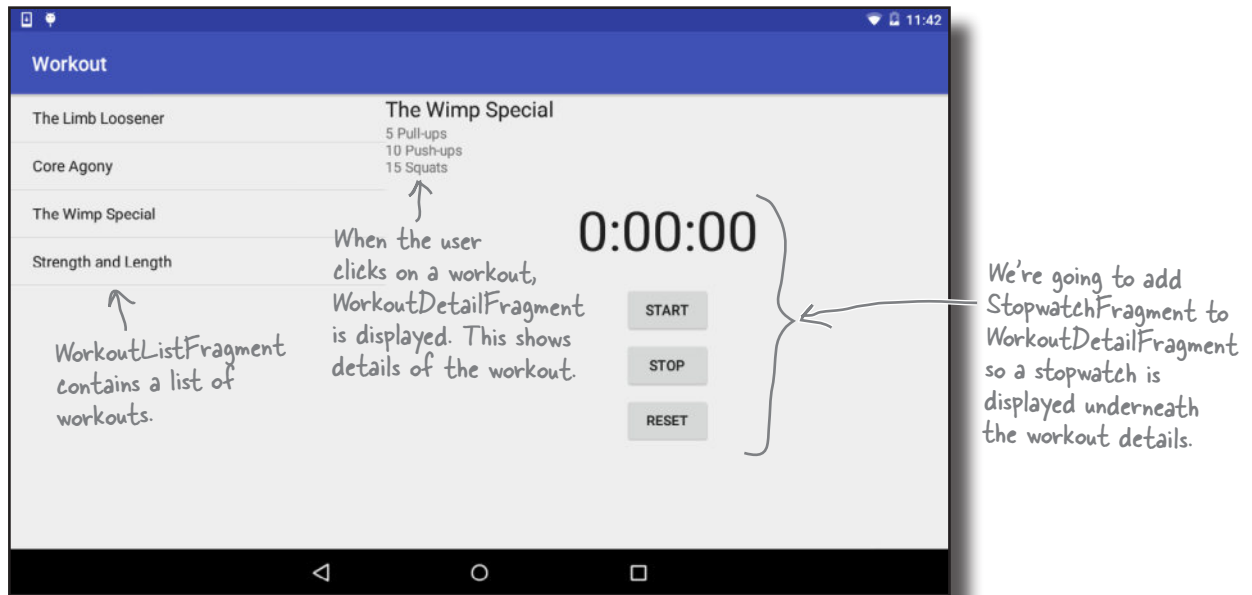
When we run the app, the stopwatch is displayed as before. The Start, Stop, and Reset buttons all work, and when we rotate the app, the stopwatch keeps running.



At the beginning of the chapter, we said we'd first focus on getting `StopwatchFragment` working in a new temporary activity so that we could confirm it works OK. Now that we've achieved that, we can reuse it in `WorkoutDetailFragment`.

Add the stopwatch to WorkoutDetailFragment

We're going to add `StopwatchFragment` to `WorkoutDetailFragment` so that a stopwatch is displayed underneath details of the workout. The stopwatch will appear along with the workout details whenever the user chooses one of the workouts.



Here's how the app will work:

- 1 **When the app gets launched, it starts MainActivity.**
`MainActivity` includes `WorkoutListFragment`, which displays a list of workouts.
- 2 **The user clicks on a workout and `WorkoutDetailFragment` is displayed.**
`WorkoutDetailFragment` displays details of the workout, and contains `StopwatchFragment`.
- 3 **`StopwatchFragment` displays a stopwatch.**



We'll go through the steps on the next page.

What we're going to do



**Convert stopwatch
Test stopwatch
Add to fragment**

There are just a couple of steps we need to go through in order to get the new version of the app up and running.

- 1 **Make the app start MainActivity when it launches.**
Earlier in the chapter, we temporarily changed the app so that it would start `TempActivity`. We need to change the app so that it starts `MainActivity` again.
- 2 **Add StopwatchFragment to WorkoutDetailFragment.**
We'll do this using a fragment transaction.

Let's get started.

Start MainActivity when the app launches


Earlier in the chapter, we updated `AndroidManifest.xml` to make the app start `TempActivity`. This was so that we could get `StopwatchFragment` working before adding it to `WorkoutDetailFragment`.

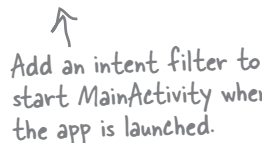
Now that `StopwatchFragment` is working, we need to start `MainActivity` again when the app launches. To do this, update `AndroidManifest.xml` with the following changes:


```

...
<application
    ...
    <activity android:name=".MainActivity">
        <intent-filter>
            <action android:name="android.intent.action.MAIN" />
            <category android:name="android.intent.category.LAUNCHER" />
        </intent-filter>
    </activity>
    <activity android:name=".DetailActivity" />
    <activity android:name=".TempActivity">
        <del>intent-filter>
            <del>action android:name="android.intent.action.MAIN" />
            <del>category android:name="android.intent.category.LAUNCHER" />
        </del>
    </activity>
</application>
...

```









Add a FrameLayout where the fragment should appear

Next we need to add StopwatchFragment to WorkoutDetailFragment. We'll do this by adding a frame layout to *fragment_workout_detail.xml*, just as we did in *activity_temp.xml*. We'll then be able to add StopwatchFragment to WorkoutDetailFragment using a fragment transaction.

Here's our code for *fragment_workout_detail.xml*; update your code so that it matches ours:

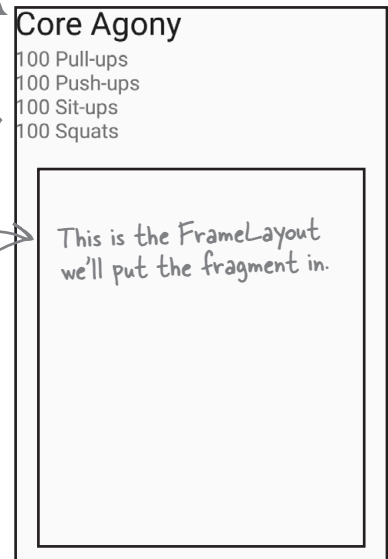
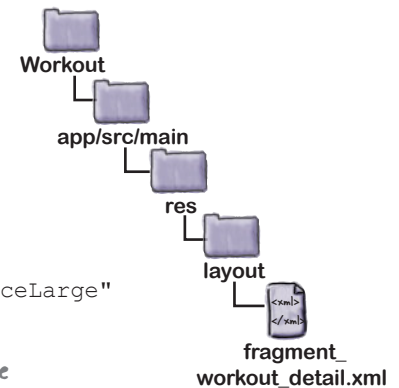
```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_height="match_parent"
    android:layout_width="match_parent"
    android:orientation="vertical">

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:textAppearance="?android:attr/textAppearanceLarge"
        android:id="@+id/textTitle" />

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:id="@+id/textDescription" />

    <FrameLayout
        android:id="@+id/stopwatch_container"
        android:layout_width="match_parent"
        android:layout_height="match_parent" />

</LinearLayout>
```



All that's left to do is to add the fragment transaction to WorkoutDetailFragment.

So far, we've only used fragment transactions in activities



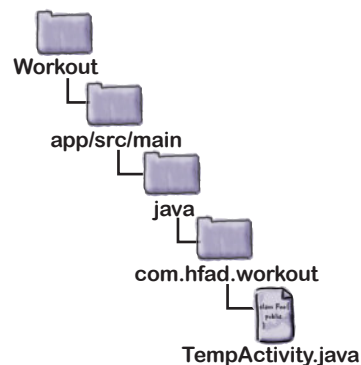
Convert stopwatch
Test stopwatch
Add to fragment

Earlier in the chapter, we added the following code to TempActivity to add StopwatchFragment to its layout:

```
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_temp);
    if (savedInstanceState == null) {
        StopwatchFragment stopwatch = new StopwatchFragment();
        FragmentTransaction ft = getSupportFragmentManager().beginTransaction();
        ft.add(R.id.stopwatch_container, stopwatch);
        ft.addToBackStack(null);
        ft.setTransition(FragmentTransaction.TRANSIT_FRAGMENT_FADE);
        ft.commit();
    }
}
```

This code adds StopwatchFragment to TempActivity when TempActivity is created.

The above code worked well when we wanted to add StopwatchFragment to an activity. How will it need to change now that we want to add StopwatchFragment to a *fragment*?



Using fragment transactions in fragments uses most of the same code

The good news is that you can use nearly all of the same code when you want to use a fragment transaction inside a fragment. There's just one key difference: fragments don't have a method called `getSupportFragmentManager()`, so we need to edit this line of code:

```
FragmentTransaction ft = getSupportFragmentManager().beginTransaction();
```

In order to create the fragment transaction, we need to get a reference to a fragment manager. Fragments have *two* methods you can use for this purpose: `getFragmentManager()` and `getChildFragmentManager()`. So what's the difference between these two methods, and which one should we use in our app?

Convert stopwatch
Test stopwatch
Add to fragment



Using `getFragmentManager()` creates extra transactions on the back stack

The `getFragmentManager()` method gets the fragment manager associated with the fragment's *parent activity*. Any fragment transaction you create using this fragment manager is added to the back stack as a separate transaction.

In our case, when someone clicks on a workout, we want the app to display the details of the workout and the stopwatch. `MainActivity` creates a transaction that displays `WorkoutDetailFragment`. If we use `getFragmentManager()` to create a transaction to display `StopwatchFragment`, this will be added to the back stack as a separate transaction.

The problem with using two transactions to display the workout and stopwatch is what happens when the user presses the Back button.

Suppose the user clicks on a workout. Details of the workout will be displayed, along with the stopwatch. If the user then clicks on the Back button, they will expect the screen to go back to how it looked before they selected a workout. But **the Back button simply pops the last transaction on the back stack**. That means if we create two transactions to add the workout detail and the stopwatch, when the user clicks the Back button, only the stopwatch will be removed. They have to click the Back button again to remove the workout details.

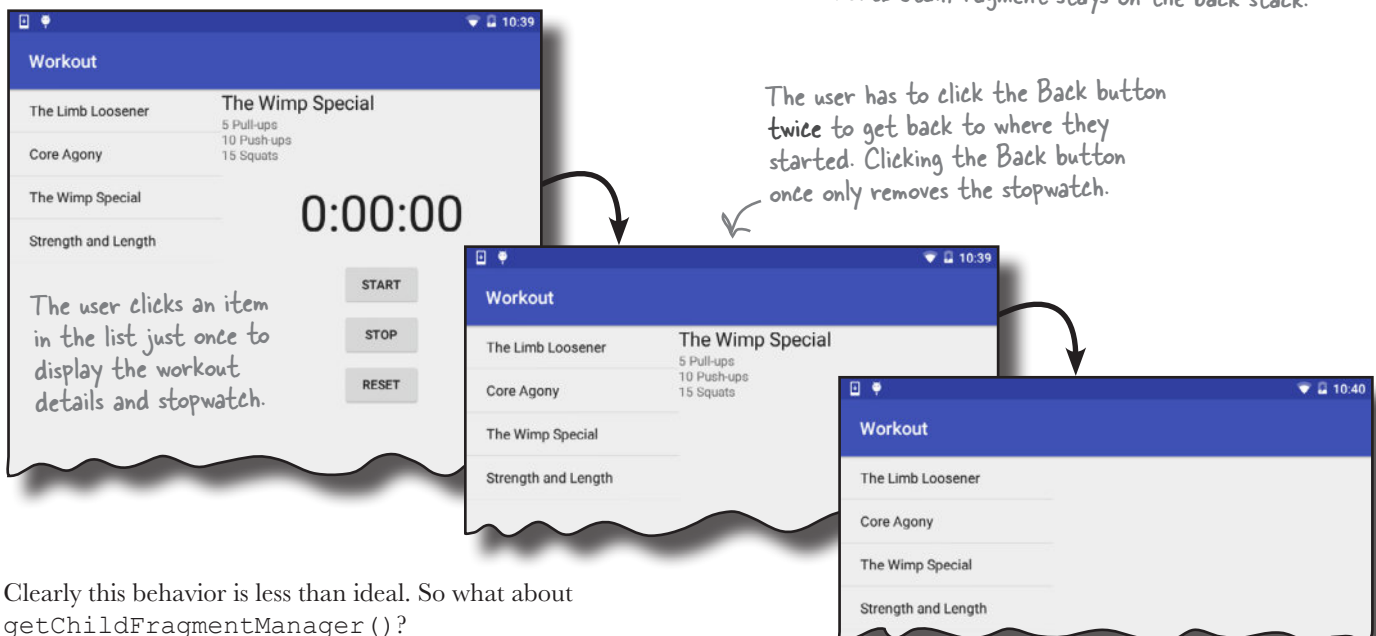


A transaction for `WorkoutDetailFragment` is added to the back stack, followed by a separate transaction for `StopwatchFragment`.



When the user hits the Back button, the `StopwatchFragment` transaction is popped off the back stack. The transaction for `WorkoutDetailFragment` stays on the back stack.

The user has to click the Back button twice to get back to where they started. Clicking the Back button once only removes the stopwatch.



Clearly this behavior is less than ideal. So what about `getChildFragmentManager()`?

Using getChildFragmentManager() creates nested transactions instead



Convert stopwatch
Test stopwatch
Add to fragment

The `getChildFragmentManager()` method gets the fragment manager associated with the fragment's *parent fragment*. Any fragment transaction you create using this fragment manager is added to the back stack inside the parent fragment transaction, not as a separate transaction.

In our particular case, this means that the fragment transaction that displays `WorkoutDetailFragment` contains a second transaction that displays `StopwatchFragment`.

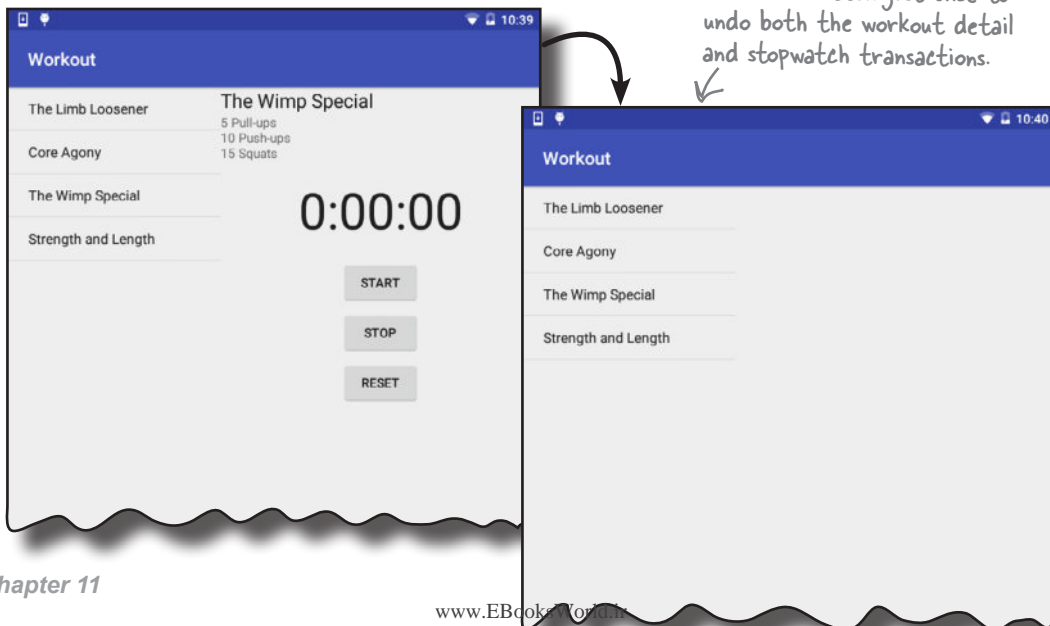


The transaction to add `StopwatchFragment` is nested inside the transaction to add `WorkoutDetailFragment`.



`WorkoutDetailFragment` and `StopwatchFragment` are still displayed when the user clicks on a workout, but the behavior is different when the user clicks on the Back button. As the two transactions are nested, *both* transactions are popped off the back stack when the user presses the Back button. The workout details and the stopwatch are both removed if the user presses the Back button once. That's what we want, so we'll use this method in our app.

This time the user has to press the Back button just once to undo both the workout detail and stopwatch transactions.

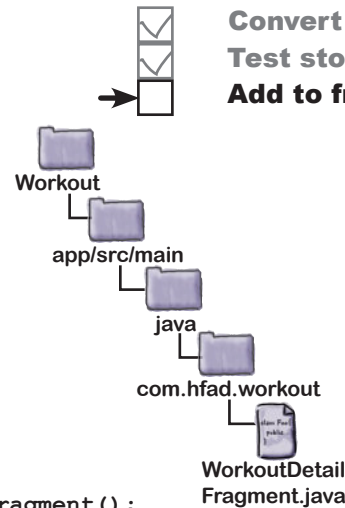


**Convert stopwatch
Test stopwatch
Add to fragment**

What getChildFragmentManager() fragment transaction code looks like

We've written code that will add `StopwatchFragment` to `WorkoutDetailFragment`. It creates a fragment transaction using the fragment manager returned by `getChildFragmentManager()`. Here's the code:

```
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    if (savedInstanceState == null) {
        StopwatchFragment stopwatch = new StopwatchFragment();
        FragmentTransaction ft = getChildFragmentManager().beginTransaction();
        ft.add(R.id.stopwatch_container, stopwatch);
        ft.addToBackStack(null);
        ft.setTransition(FragmentTransaction.TRANSIT_FRAGMENT_FADE);
        ft.commit();
    } else {
        workoutId = savedInstanceState.getLong("workoutId");
    }
}
```



We're using `getChildFragmentManager()` instead of `getSupportFragmentManager()`. Apart from that, the code is the same as we had earlier.

We need to add this code to `WorkoutDetailFragment.java`. We'll show you the full code on the next page.

there are no Dumb Questions

Q: I can see that the child fragment manager handles the case where I put one fragment inside another. But what if I put one fragment inside another, inside another, inside another...?

A: The transactions will all be nested within each other, leaving just a single transaction at the activity level. So the nested set of child transactions will be undone by a single Back button click.

Q: Fragments seem more complicated than activities. Should I use fragments in my apps?

A: That depends on your app and what you want to achieve. One of the major benefits of using fragments is that you can use them to support a wide range of different screen sizes. You can, say, choose to display fragments side by side on tablets and on separate screens on smaller devices. You'll also see in the next chapter that some UI designs require you to use fragments.

The full WorkoutDetailFragment.java code



Convert stopwatch
Test stopwatch
Add to fragment

Here's the full code for *WorkoutDetailFragment.java*. Update your version of the code to include our changes.

```

package com.hfad.workout;

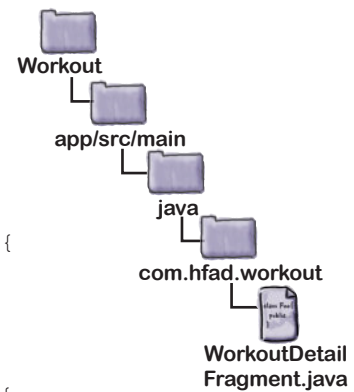
import android.support.v4.app.Fragment;
import android.support.v4.app.FragmentTransaction;
import android.os.Bundle;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;
import android.widget.TextView;

public class WorkoutDetailFragment extends Fragment {
    private long workoutId;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        if (savedInstanceState != null) {
        if (savedInstanceState == null) {
            StopwatchFragment stopwatch = new StopwatchFragment();
            FragmentTransaction ft = getChildFragmentManager().beginTransaction();
            ft.add(R.id.stopwatch_container, stopwatch);
            ft.addToBackStack(null);
            ft.setTransition(FragmentTransaction.TRANSIT_FRAGMENT_FADE);
            ft.commit();
        } else {
            workoutId = savedInstanceState.getLong("workoutId");
        }
    }
}

```

You need to import the *FragmentTransaction* class from the Support Library.



We only want to add the fragment if the activity isn't being recreated after having been destroyed.

Delete this line.

Begin the fragment transaction.

Add the stopwatch, and add the transaction to the back stack.

Commit the transaction.

Set the fragment transition to fade in and out.

The code continues on the next page.



The full code (continued)

```

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

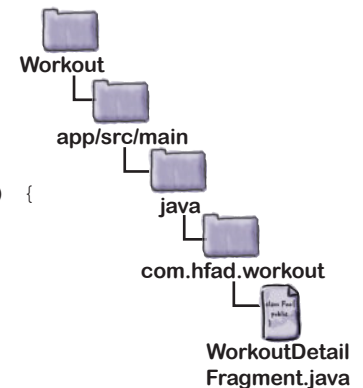
@Override
public void onStart() {
    super.onStart();
    View view = getView();
    if (view != null) {
        TextView title = (TextView) view.findViewById(R.id.textTitle);
        Workout workout = Workout.workouts[(int) workoutId];
        title.setText(workout.getName());
        TextView description = (TextView) view.findViewById(R.id.textDescription);
        description.setText(workout.getDescription());
    }
}

@Override
public void onSaveInstanceState(Bundle savedInstanceState) {
    savedInstanceState.putLong("workoutId", workoutId);
}

public void setWorkout(long id) {
    this.workoutId = id;
}
}

```

*We didn't change any of
the methods on this page.*



That's everything we need for our app. Let's take it for a test drive and check that it works OK.



Test drive the app



Convert stopwatch
Test stopwatch
Add to fragment

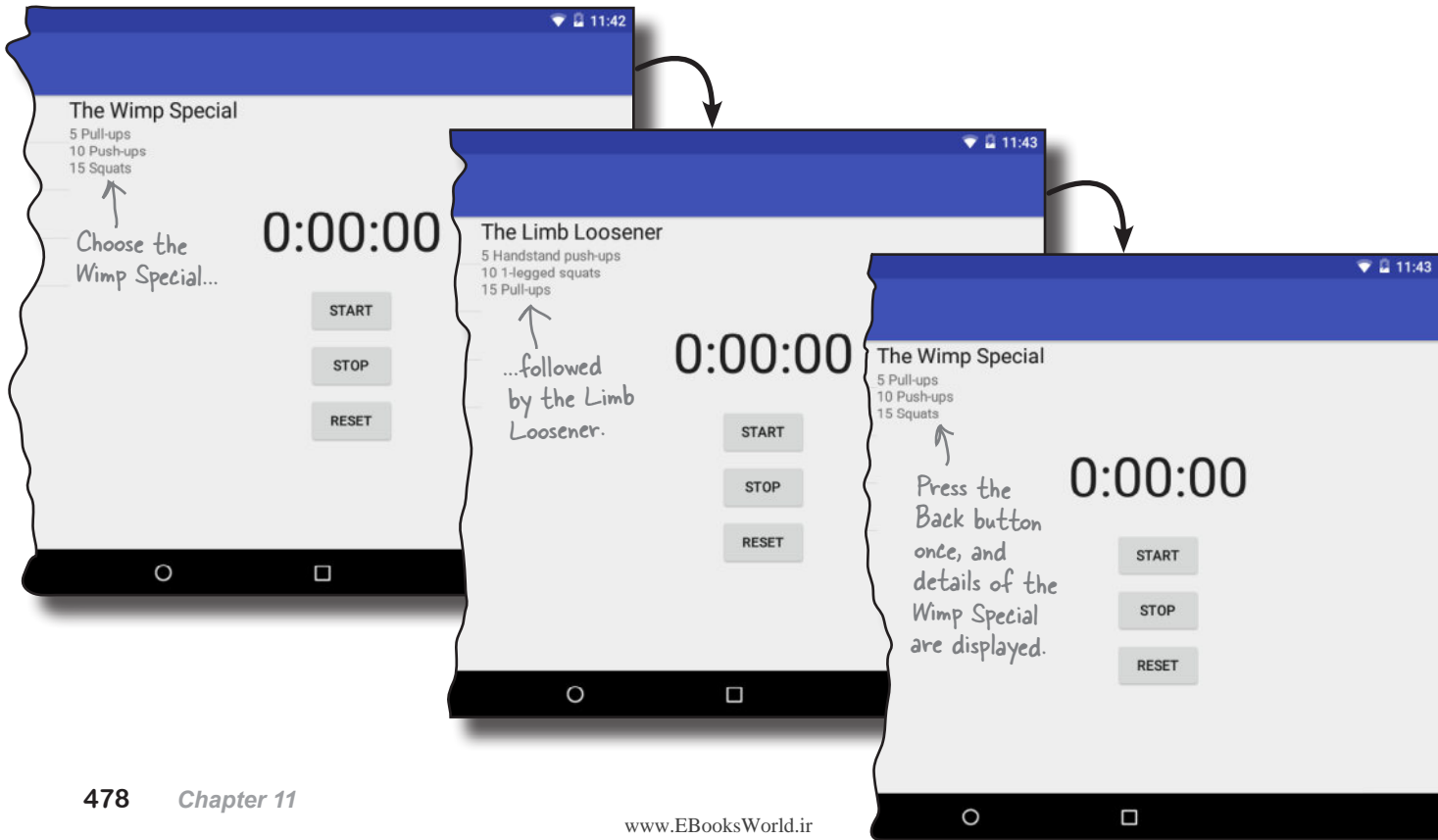
We'll start by testing the app on a tablet.

When we start the app, MainActivity is displayed.

MainActivity starts when we launch the app.



When we click on one of the workouts, details of that workout are displayed along with a stopwatch. If we click on a second workout and then click on the Back button, details of the first workout are displayed.





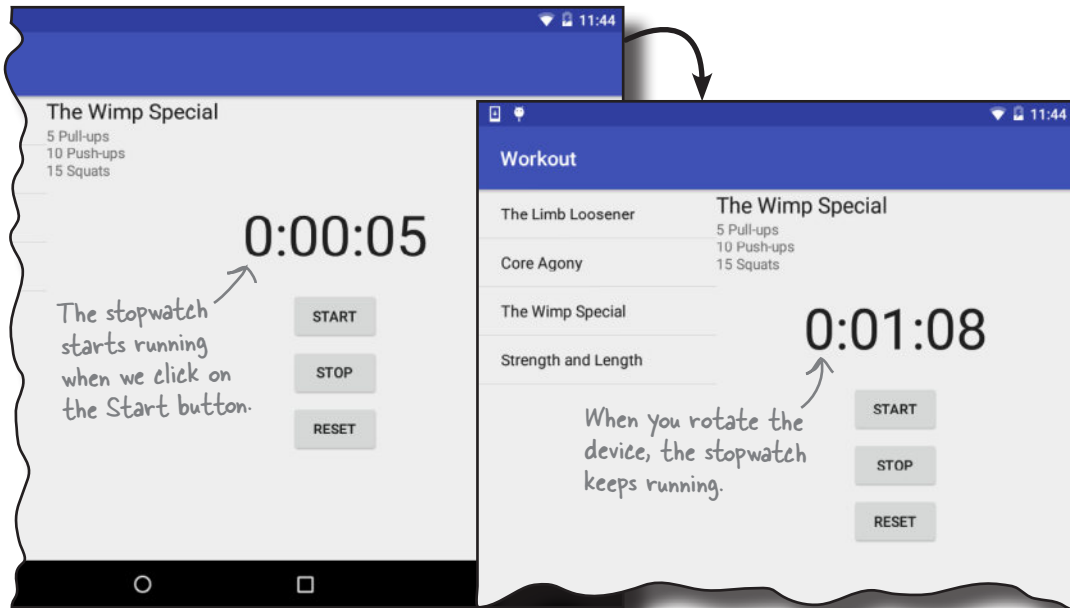
Test drive (continued)

dynamic fragments

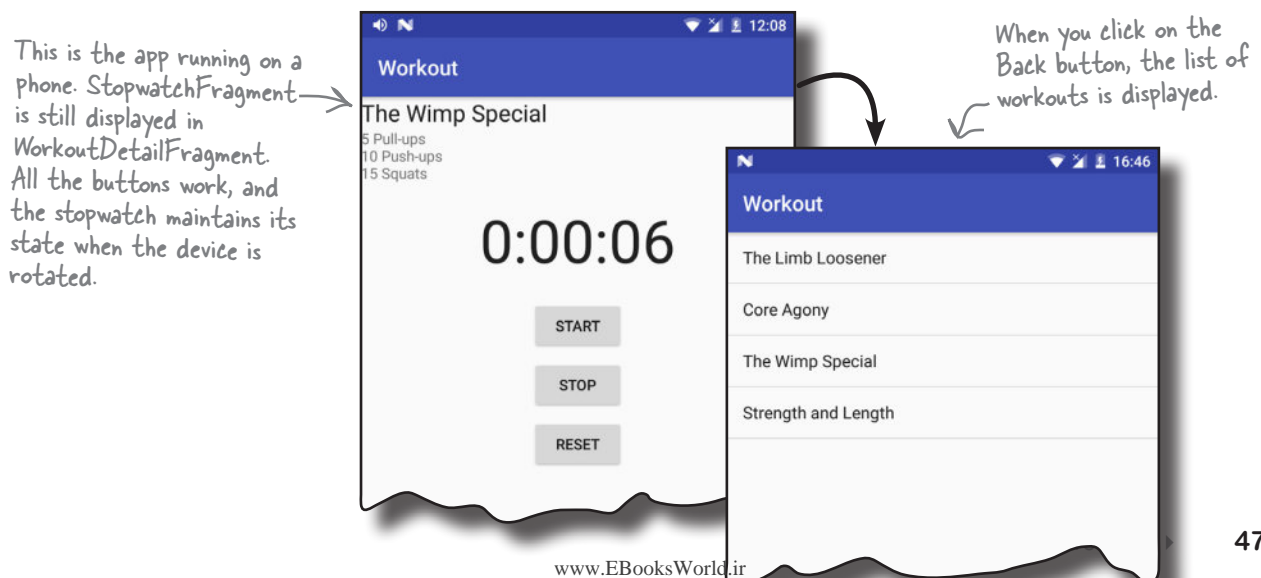
**Convert stopwatch
Test stopwatch
Add to fragment**



When we click on the stopwatch buttons, they all work as expected.
When we rotate the app, the stopwatch maintains its state.



When we run the app on a phone, `WorkoutDetailFragment` is displayed inside a separate activity, `DetailActivity`. The stopwatch is still displayed underneath the workout details, and functions as expected.





Your Android Toolbox

You've got **Chapter 11** under your belt and now you've added **dynamic fragments** to your toolbox.

You can download the full code for the chapter from <https://tinyurl.com/HeadFirstAndroid>.



BULLET POINTS

- Fragments can contain other fragments.
- If you use the `android:onClick` attribute in a fragment, Android will look for a method of that name in the fragment's parent activity.
- Instead of using the `android:onClick` attribute in a fragment, make the fragment implement the `View.OnClickListener` interface and implement its `onClick()` method.
- If you use the `<fragment>` element in your layout, the fragment gets recreated when you rotate the device. If your fragment is dynamic, use a fragment transaction instead.
- Fragments contain two methods for getting a fragment manager, `getFragmentManager()` and `getChildFragmentManager()`.
- `getFragmentManager()` gets a reference to the fragment manager associated with the fragment's parent activity. Any fragment transactions you create using this fragment manager are added to the back stack as extra transactions.
- `getChildFragmentManager()` gets a reference to the fragment manager associated with the fragment's parent fragment. Any fragment transactions you create using this fragment manager are nested inside the parent fragment transaction.

12 design support library

Swipe Right



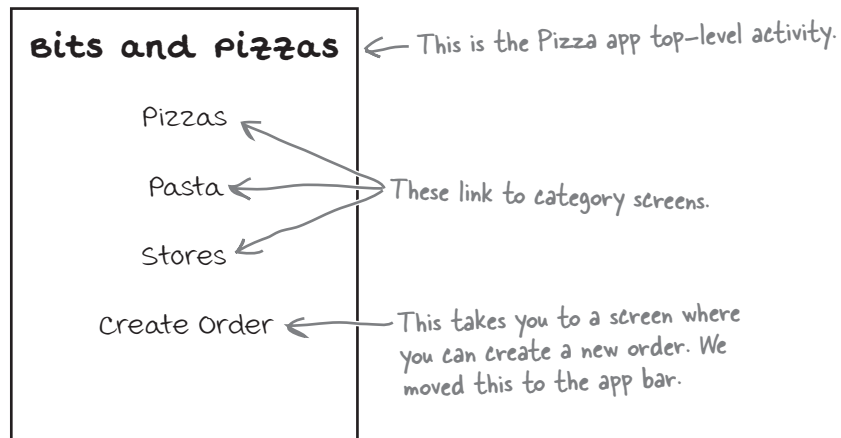
This new snackbar's awesome; it does so much more than toast.

Ever wondered how to develop apps with a rich, slick UI?

With the release of the **Android Design Support Library**, it became much easier to create apps with an intuitive UI. In this chapter, we'll show you around some of the highlights. You'll see how to add **tabs** so that your users can *navigate around your app more easily*. You'll discover how to **animate your toolbars** so that they can *collapse or scroll on a whim*. You'll find out how to add **floating action buttons** for common user actions. Finally, we'll introduce you to **snackbars**, a way of displaying short, informative messages to the user that they can interact with.

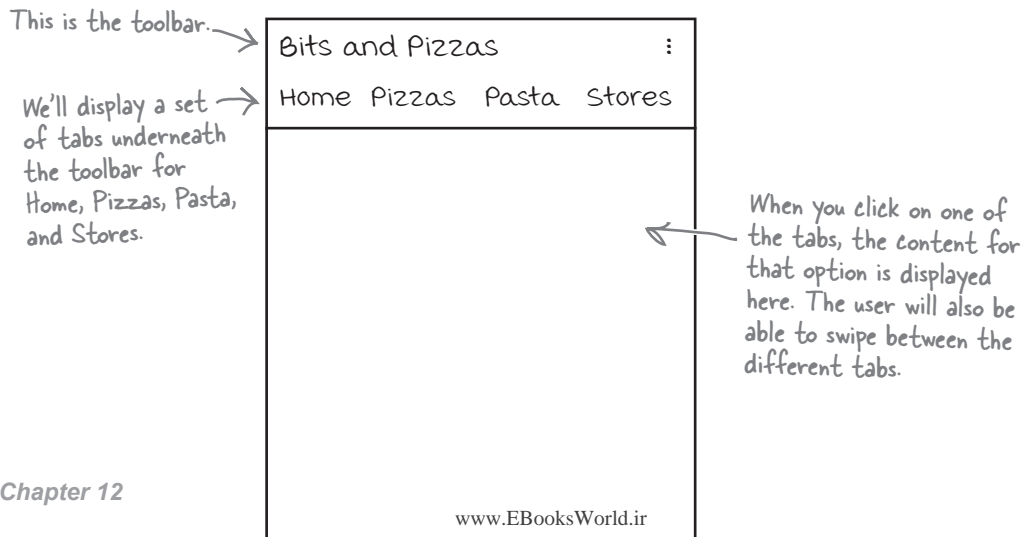
The Bits and Pizzas app revisited

In Chapter 8, we showed you a sketch of the top-level screen of the Bits and Pizzas app. It contained a list of places in the app the user could go to. The first three options linked to category screens for pizzas, pasta, and stores, and the final option linked to a screen where the user could create an order.



So far you've seen how to add actions to the app bar. These are used for simple commands, such as Create Order or Send Feedback. But what about the category screens? As we want to use these for navigating through the app rather than taking an action, we'll take a different approach.

We're going to change the Bits and Pizzas app so that it uses **tab navigation**. We'll display a set of tabs underneath the toolbar, with each option on a different tab. When the user clicks on a tab, the screen for that option will be displayed. We'll also let the user swipe left and right between the different tabs.

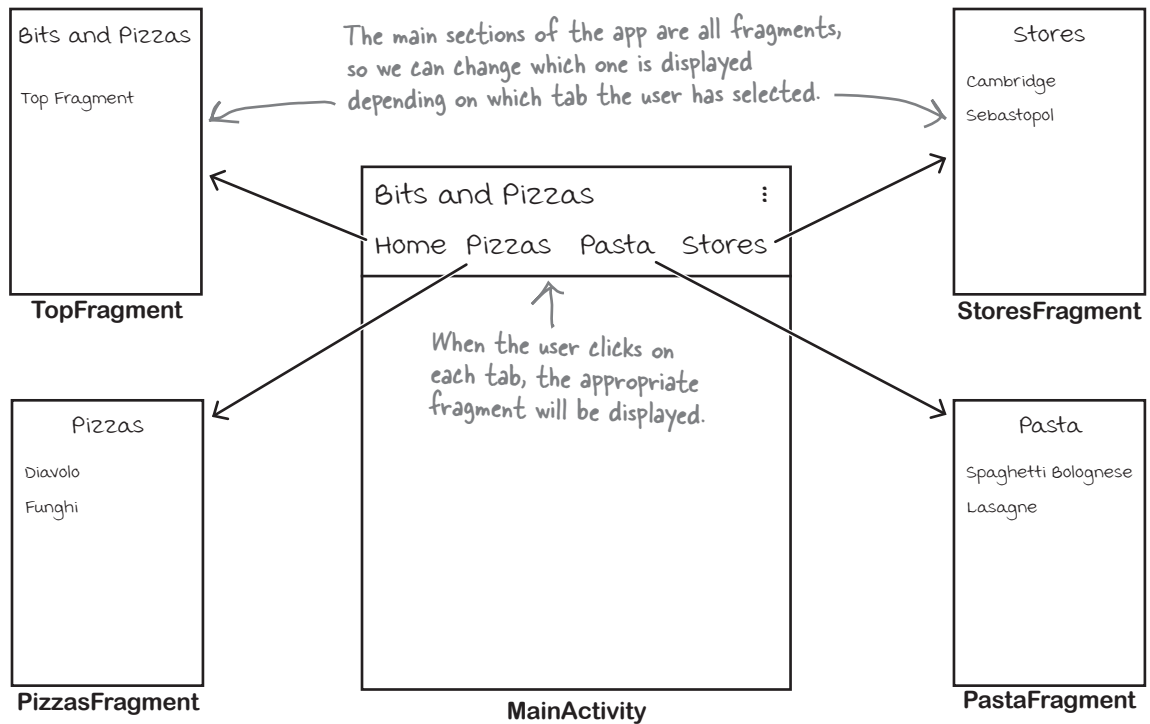
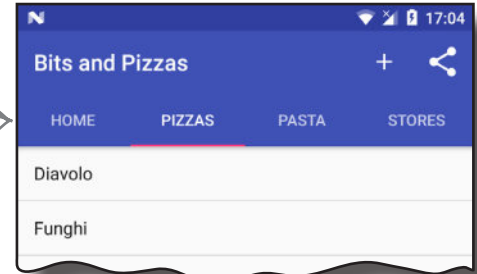


The app structure

We're going to change `MainActivity` so that it uses tabs. The tabs will include options for Home, Pizzas, Pasta, and Stores, so that the user can easily navigate to the main sections of the app.

We'll create fragments for these different options; when the user clicks on one of the tabs, the fragment for that option will be displayed:

This is what the new version of the app will look like.



We'll go through the steps for how to do this on the next page.

Here's what we're going to do

There are three main steps we'll go through to get tabs working:

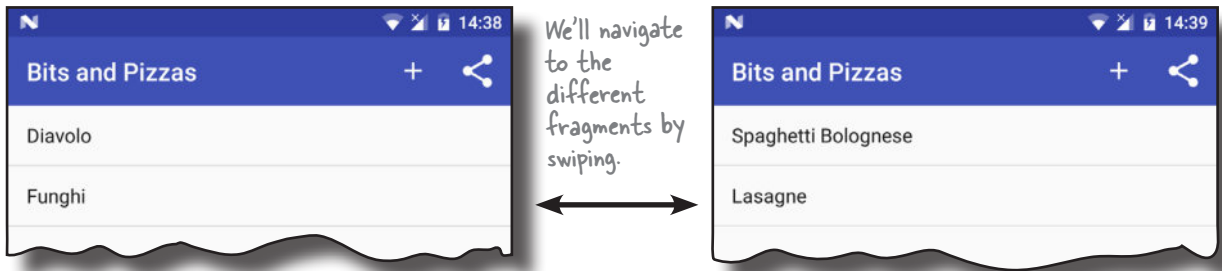
1 Create the fragments.

We'll create basic versions of `TopFragment`, `PizzaFragment`, `PastaFragment`, and `StoresFragment` so that we can easily tell which fragment is displayed on each of the tabs.



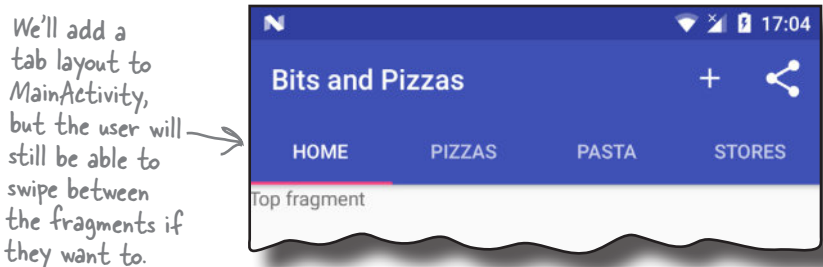
2 Enable swipe navigation between the fragments.

We'll update `MainActivity` so that the user can swipe between the different fragments.



3 Add the tab layout.

Finally, we'll add a tab layout to `MainActivity` that will work in conjunction with the swipe navigation. The user will be able to navigate to each fragment by clicking on a tab, or swiping between them.



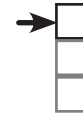
Do this!

We're going to update the Bits and Pizzas app in this chapter, so open your original Bits and Pizzas project from Chapter 8 in Android Studio.

We'll start by creating the fragments.

Create TopFragment

design support library



Add fragments
Add swiping
Add tabs

We'll use TopFragment to display content that will appear on the Home tab. For now, we'll display the text "Top fragment" so that we know which fragment is displayed. Highlight the `com.hfad.bitsandpizzas` package in the `app/src/main/java` folder, then go to File→New...→Fragment→Fragment (Blank). Name the fragment "TopFragment" and name its layout "fragment_top". Then replace the code for `TopFragment.java` with the code below:

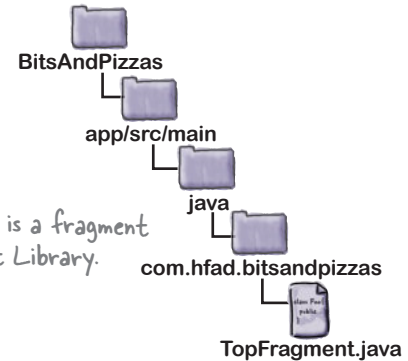
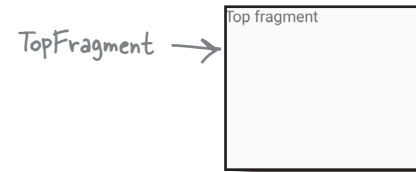
```
package com.hfad.bitsandpizzas;

import android.os.Bundle;
import android.support.v4.app.Fragment;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;

public class TopFragment extends Fragment {

    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup container,
        Bundle savedInstanceState) {
        return inflater.inflate(R.layout.fragment_top, container, false);
    }
}
```

TopFragment.java is a fragment from the Support Library.



Add the following string resource to `strings.xml`; we'll use this in our fragment layout:

```
<string name="title_top">Top fragment</string>
```

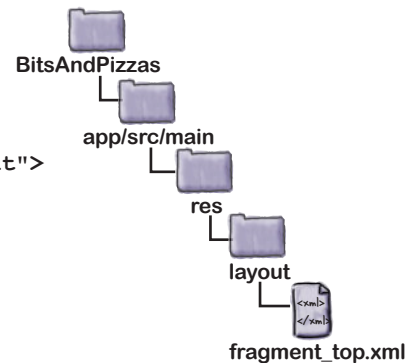
Add this to strings.xml. We'll use it in the layout so we know when TopFragment is being displayed.

Then update the code for `fragment_top.xml` as follows:

```
<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="com.hfad.bitsandpizzas.TopFragment">

    <TextView
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:text="@string/title_top" />

</LinearLayout>
```



Create *PizzaFragment*

We'll use a `ListFragment` called `PizzaFragment` to display the list of pizzas. Highlight the `com.hfad.bitsandpizzas` package in the `app/src/main/java` folder, then go to `File`→`New`...→`Fragment`→`Fragment (Blank)`. Name the fragment "PizzaFragment", and uncheck the option to create a layout. Why? Because list fragments don't need a layout—they use their own.

Next, add a new string array resource called "pizzas" to `strings.xml` (this contains the names of the pizzas):

```
<string-array name="pizzas">
    <item>Diavolo</item>
    <item>Funghi</item>
</string-array>
```

← Add the array of pizzas to strings.xml.

Then change the code for `PizzaFragment.java` so that it's a `ListFragment`. Its list view needs to be populated with the pizza names. Here's the updated code:

```
package com.hfad.bitsandpizzas;

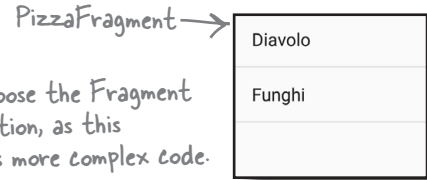
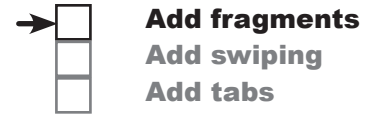
import android.os.Bundle;
import android.support.v4.app.ListFragment;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;
import android.widget.ArrayAdapter;

public class PizzaFragment extends ListFragment {

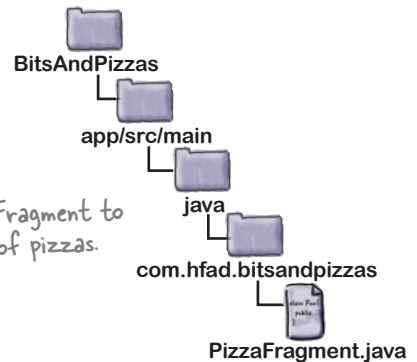
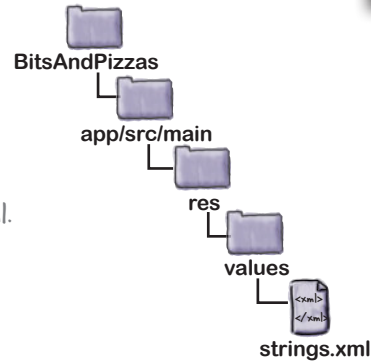
    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup container,
                             Bundle savedInstanceState) {
        ArrayAdapter<String> adapter = new ArrayAdapter<>(
            inflater.getContext(),
            android.R.layout.simple_list_item_1,
            getResources().getStringArray(R.array.pizzas));
        setListAdapter(adapter);
        return super.onCreateView(inflater, container, savedInstanceState);
    }
}
```

← We'll use a `ListFragment` to display the list of pizzas.

← The `ArrayAdapter` populates the `ListFragment`'s `ListView` with the pizza names.



Don't choose the `Fragment (List)` option, as this generates more complex code.



Create PastaFragment

We'll use a `ListFragment` called `PastaFragment` to display the list of pasta. Highlight the `com.hfad.bitsandpizzas` package in the `app/src/main/java` folder and create a new blank fragment named "PastaFragment". You can uncheck the option to create a layout, as list fragments use their own layouts.

Next, add a new string array resource called "pasta" to `strings.xml` (this contains the names of the pasta):

```
<string-array name="pasta">
    <item>Spaghetti Bolognese</item>
    <item>Lasagne</item>
</string-array>
```

← Add the array of pasta to strings.xml.

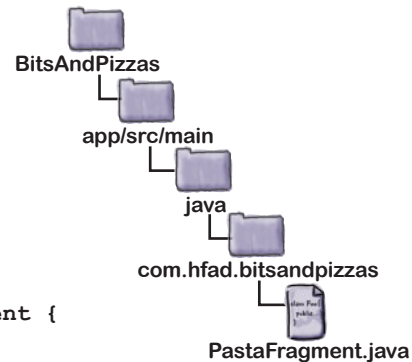
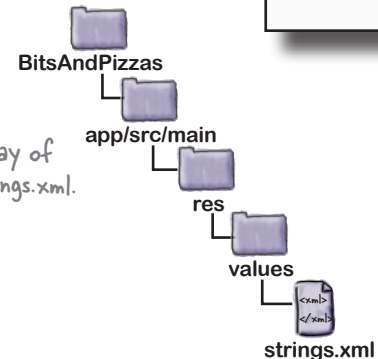
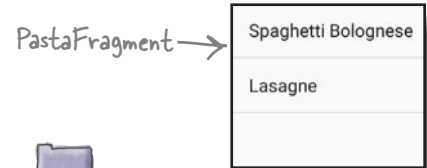
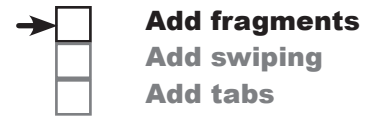
Then change the code for `PastaFragment.java` so that it's a `ListFragment` that displays a list of the pasta names. Here's the updated code:

```
package com.hfad.bitsandpizzas;

import android.os.Bundle;
import android.support.v4.app.ListFragment;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;
import android.widget.AdapterView;

public class PastaFragment extends ListFragment {

    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup container,
        Bundle savedInstanceState) {
        ArrayAdapter<String> adapter = new ArrayAdapter<>(
            inflater.getContext(),
            android.R.layout.simple_list_item_1,
            getResources().getStringArray(R.array.pasta));
        setListAdapter(adapter);
        return super.onCreateView(inflater, container, savedInstanceState);
    }
}
```



Create StoresFragment

We'll use a `ListFragment` called `StoresFragment` to display the list of stores. Highlight the `com.hfad.bitsandpizzas` package in the `app/src/main/java` folder and create a new blank fragment named "StoresFragment." Uncheck the option to create a layout, as list fragments define their own layouts.

Next, add a new string array resource called "stores" to `strings.xml` (this contains the names of the stores):

```
<string-array name="stores">
    <item>Cambridge</item>
    <item>Sebastopol</item>
</string-array>
```

← Add the array of stores to strings.xml.

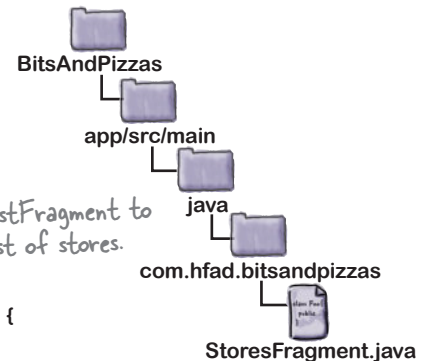
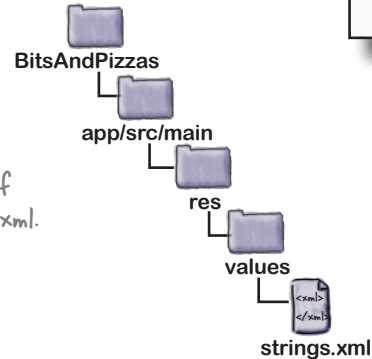
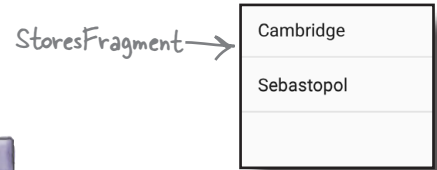
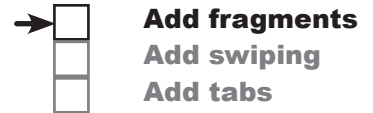
Then change the code for `StoresFragment.java` so that it's a `ListFragment`. Its list view needs to be populated with the store names. Here's the updated code:

```
package com.hfad.bitsandpizzas;

import android.os.Bundle;
import android.support.v4.app.ListFragment;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;
import android.widget.ArrayAdapter;

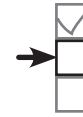
public class StoresFragment extends ListFragment {

    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup container,
        Bundle savedInstanceState) {
        ArrayAdapter<String> adapter = new ArrayAdapter<>(
            inflater.getContext(),
            android.R.layout.simple_list_item_1,
            getResources().getStringArray(R.array.stores));
        setListAdapter(adapter);
        return super.onCreateView(inflater, container, savedInstanceState);
    }
}
```



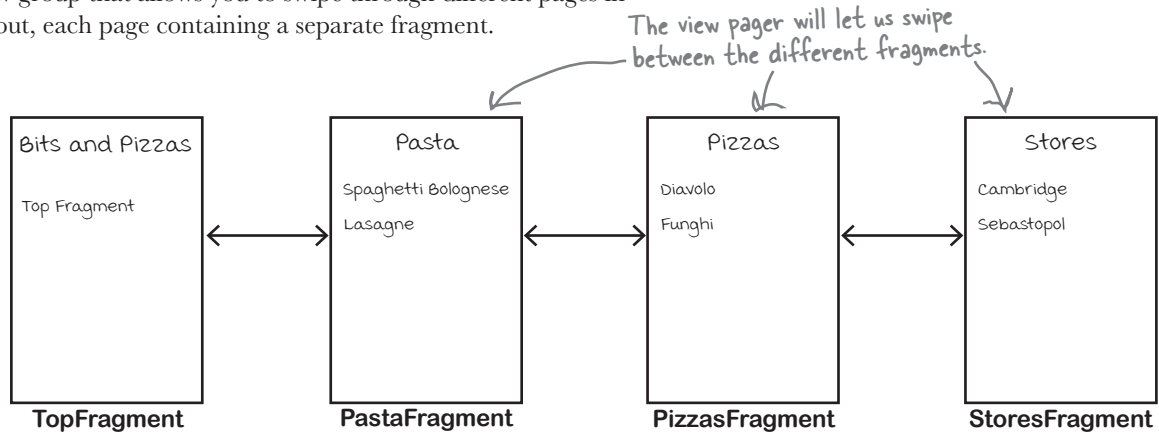
We'll use a `ListFragment` to display the list of stores.

We've now added all the fragments we need, so let's move on to the next step.



Use a view pager to swipe through fragments

We want to be able to swipe through the different fragments we've just created. To do this, we'll use a **view pager**, which is a view group that allows you to swipe through different pages in a layout, each page containing a separate fragment.



You use a view pager by adding it to your layout, then writing activity code to control which fragments should be displayed. The `ViewPager` class comes from the v4 Support Library, which is included in the v7 AppCompat Support Library, so you also need to make sure you add one of these libraries to your project as a dependency. In our particular case, we already added the v7 AppCompat Support Library to our project in Chapter 8.

You can check which Support Libraries are included in your project in Android Studio by choosing Project Structure from the File menu, clicking on the app module, and then choosing Dependencies.

What view pager layout code looks like

You add a view pager to your layout using code like this:

The `ViewPager` class is found in the v4 Support Library (which is included in the v7 AppCompat Support Library).

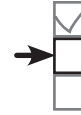
```
<android.support.v4.view.ViewPager
    android:id="@+id/pager"
    android:layout_width="match_parent"
    android:layout_height="match_parent" />
```

You need to give the `ViewPager` an ID so that you can control its behavior in your activity code.

The above code defines the view pager, and gives it an ID of `pager`. Every view pager you create *must* have an ID so that you can get a reference to it in your activity code. Without this ID, you can't specify which fragments should appear on each page of the view pager.

We're going to add a view pager to `MainActivity`. We'll look at the full code for its layout on the next page.

Add a view pager to MainActivity's layout



Add fragments
Add swiping
Add tabs

We're going to add a view pager to MainActivity's layout, and remove the text view that's already there. Open the file `activity_main.xml`, then update your code so that it matches ours below (we've bolded our changes):

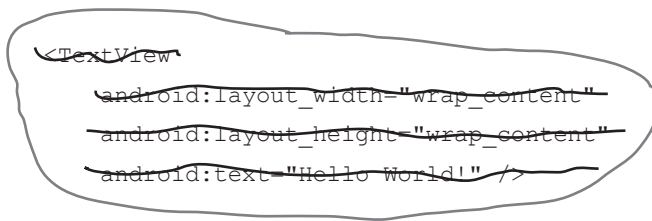
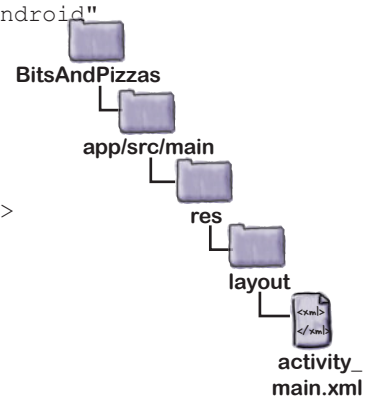
```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    tools:context="com.hfad.bitsandpizzas.MainActivity">

    <include
        layout="@layout/toolbar_main"
        android:id="@+id/toolbar" />

    <android.support.v4.view.ViewPager <-- Add the ViewPager
        android:id="@+id/pager"           below the Toolbar.
        android:layout_width="match_parent"
        android:layout_height="match_parent" />

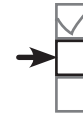
    <del>TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hello World!" />

</LinearLayout>
```



We're no longer displaying a TextView in MainActivity, so delete these lines of code.

That's everything we need to add a view pager to our layout. To get our new view pager to display fragments, we need to write some activity code. We'll do that next.



Add fragments

Add swiping

Add tabs

Tell a view pager about its pages using a fragment pager adapter

To get a view pager to display a fragment on each of its pages, there are two key pieces of information you need to give it: the number of pages it should have, and which fragment should appear on each page. You do this by creating a **fragment pager adapter**, and adding it to your activity code.

A fragment pager adapter is a type of adapter that specializes in adding fragments to pages in a view pager. You generally use one when you want to have a small number of pages that are fairly static, as each fragment the user visits is kept in memory.

If you want your view pager to have a large number of pages, you would use a fragment state pager adapter instead. We're not covering it here, but the code is almost identical.

Fragment pager adapter code looks like this:

```
private class SectionsPagerAdapter extends FragmentPagerAdapter {
    public SectionsPagerAdapter(FragmentManager fm) {
        super(fm);
    }
    @Override
    public int getCount() {
        //The number of pages in the ViewPager
    }
    @Override
    public Fragment getItem(int position) {
        //The fragment to be displayed on each page
    }
}
```

We're setting this to private, as we're going to add it to MainActivity as an inner class.

You must have a constructor that takes a `FragmentManager` parameter.

You need to extend the `FragmentPagerAdapter` class.

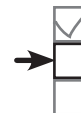
You need to override the `getCount()` method to specify the number of pages in the view pager.

You need to say which fragment should appear on each page. The position gives the page number, starting at 0.

When you create a fragment pager adapter, there are two key methods you **must** override: `getCount()` and `getItem()`. You use `getCount()` to specify how many pages there should be in the view pager, and the `getItem()` to say which fragment should be displayed on each page.

We'll show you the code for the Bits and Pizzas fragment pager adapter on the next page.

The code for our fragment pager adapter



Add fragments
Add swiping
Add tabs

We want our view pager to have four pages. We'll display TopFragment on the first page, PizzaFragment on the second, PastaFragment on the third, and StoresFragment on the fourth.

To accomplish this, we're going to create a fragment pager adapter called SectionsPagerAdapter. Here's the code (we'll add it to MainActivity.java in a couple of pages):

```
private class SectionsPagerAdapter extends FragmentPagerAdapter {

    public SectionsPagerAdapter(FragmentManager fm) {
        super(fm);
    }

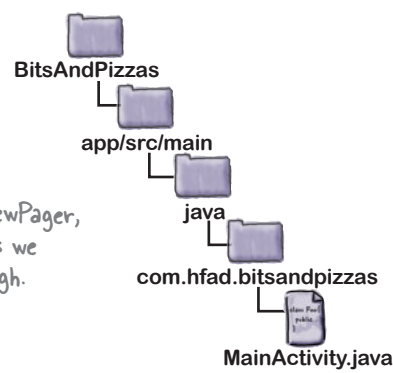
    @Override
    public int getCount() {
        return 4;
    }

    @Override
    public Fragment getItem(int position) {
        switch (position) {
            case 0:
                return new TopFragment();
            case 1:
                return new PizzaFragment();
            case 2:
                return new PastaFragment();
            case 3:
                return new StoresFragment();
        }
        return null;
    }
}
```

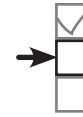
We'll have four pages in our ViewPager, one for each of the fragments we want to be able to swipe through.

The getCount() method specifies 4 pages, so the getItem() method should only request the fragments for these 4 page positions.

We want to display TopFragment first, so we'll return a new instance of it for position 0 of the ViewPager.



That's all the code we need for our SectionsPagerAdapter. Next we need to get our view pager to use it.



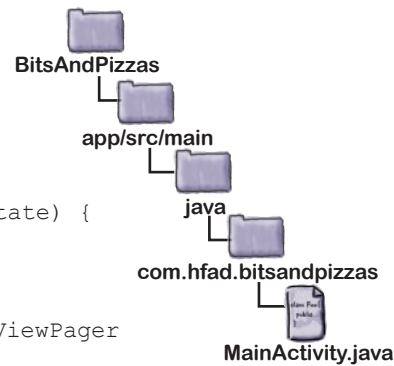
Add fragments
Add swiping
Add tabs

Attach the fragment pager adapter to the view pager

Finally, we need to attach our `SectionsPagerAdapter` to the view pager so that the view pager can use it. You attach a fragment pager adapter to a view pager by calling the `ViewPager.setAdapter()` method, and passing it a reference to an instance of the fragment pager adapter.

Here's the code to attach the fragment pager adapter we created to the view pager:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    ...
    //Attach the SectionsPagerAdapter to the ViewPager
    SectionsPagerAdapter pagerAdapter =
        new SectionsPagerAdapter(getSupportFragmentManager());
    ViewPager pager = (ViewPager) findViewById(R.id.pager);
    pager.setAdapter(pagerAdapter);
}
```



← This attaches the `FragmentPagerAdapter` we created to the `ViewPager`.

↑ We're using support fragments, so we need to pass our adapter a reference to the support fragment manager.

That's everything we need to be able to swipe through our fragments. We'll show you the full code for `MainActivity` on the next page.

there are no Dumb Questions

Q: When should I use tabs in my app?

A: Tabs work well when you want to give the user a quick way of navigating between a small number of sections or categories. You would generally put each one on a separate tab.

Q: What if I have a large number of categories? Can I still use tabs?

A: You can, but you may want to consider other forms of navigation such as navigation drawers. These are panels that slide out from the side of the screen. We'll show you how to create them in Chapter 14.

Q: You mentioned the fragment state pager adapter. What's that?

A: It's very similar to a fragment pager adapter, except that it also handles saving and restoring a fragment's state. It uses less memory than a fragment pager adapter, as when pages aren't visible, the fragment it displays may be destroyed. It's useful if your view pager has a large number of pages.

The full code for MainActivity.java



Add fragments
Add swiping
Add tabs

Here's our full code for *MainActivity.java*. Update your version of the code to match our changes (in bold):

```

package com.hfad.bitsandpizzas;

import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.support.v7.widget.Toolbar;
import android.view.Menu;
import android.view.MenuItem;
import android.content.Intent;
import android.support.v7.widget.ShareActionProvider;
import android.support.v4.view.MenuItemCompat;
import import android.support.v4.view.ViewPager;
import import android.support.v4.app.Fragment;
import import android.support.v4.app.FragmentManager;
import import android.support.v4.app.FragmentPagerAdapter;

public class MainActivity extends AppCompatActivity {

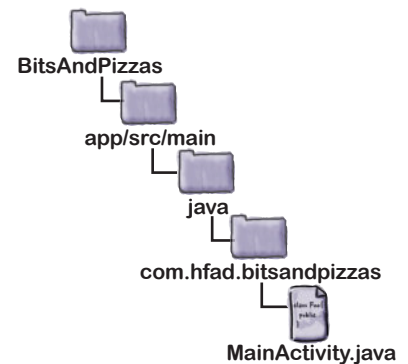
    private ShareActionProvider shareActionProvider;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        Toolbar toolbar = (Toolbar) findViewById(R.id.toolbar);
        setSupportActionBar(toolbar);

        //Attach the SectionsPagerAdapter to the ViewPager
        SectionsPagerAdapter pagerAdapter =
            new SectionsPagerAdapter(getSupportFragmentManager());
        ViewPager pager = (ViewPager) findViewById(R.id.pager);
        pager.setAdapter(pagerAdapter);
    }
}

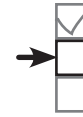
```

We're using these extra classes so we need to import them.



↑
 Attach the **FragmentPagerAdapter** to the **ViewPager**.

The code continues on the next page. →



The MainActivity.java code (continued)

None of the code on
this page has changed.

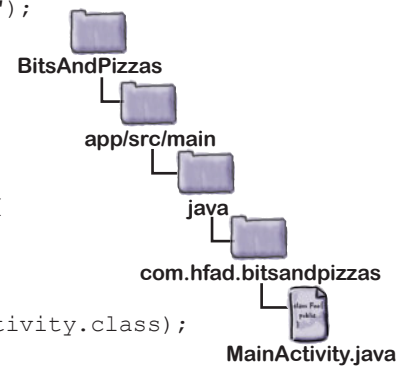
```

@Override
public boolean onCreateOptionsMenu(Menu menu) {
    getMenuInflater().inflate(R.menu.menu_main, menu);
    MenuItem menuItem = menu.findItem(R.id.action_share);
    shareActionProvider =
        (ShareActionProvider) MenuItemCompat.getActionProvider(menuItem);
    setShareActionIntent("Want to join me for pizza?");
    return super.onCreateOptionsMenu(menu);
}

@Override
public boolean onOptionsItemSelected(MenuItem item) {
    switch (item.getItemId()) {
        case R.id.action_create_order:
            Intent intent = new Intent(this, OrderActivity.class);
            startActivity(intent);
            return true;
        default:
            return super.onOptionsItemSelected(item);
    }
}

private void setShareActionIntent(String text) {
    Intent intent = new Intent(Intent.ACTION_SEND);
    intent.setType("text/plain");
    intent.putExtra(Intent.EXTRA_TEXT, text);
    shareActionProvider.setShareIntent(intent);
}

```



The code continues
on the next page. →

The MainActivity.java code (continued)



Add fragments
Add swiping
Add tabs

```
private class SectionsPagerAdapter extends FragmentPagerAdapter {

    public SectionsPagerAdapter(FragmentManager fm) {
        super(fm);
    }

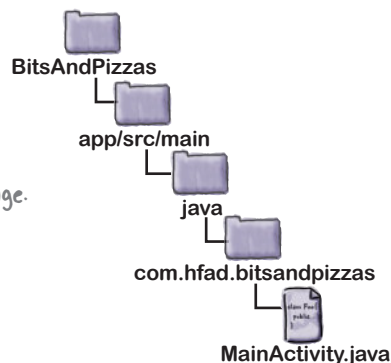
    @Override
    public int getCount() {
        return 4;
    }

    @Override
    public Fragment getItem(int position) {
        switch (position) {
            case 0:
                return new TopFragment();
            case 1:
                return new PizzaFragment();
            case 2:
                return new PastaFragment();
            case 3:
                return new StoresFragment();
        }
        return null;
    }
}
}
```

The FragmentPagerAdapter passes information to the ViewPager.

Say how many pages the ViewPager should contain.

Specify which fragment should appear on each page.



Now that we've updated our MainActivity code, let's take our app for a test drive and see what happens.



Test drive the app

design support library

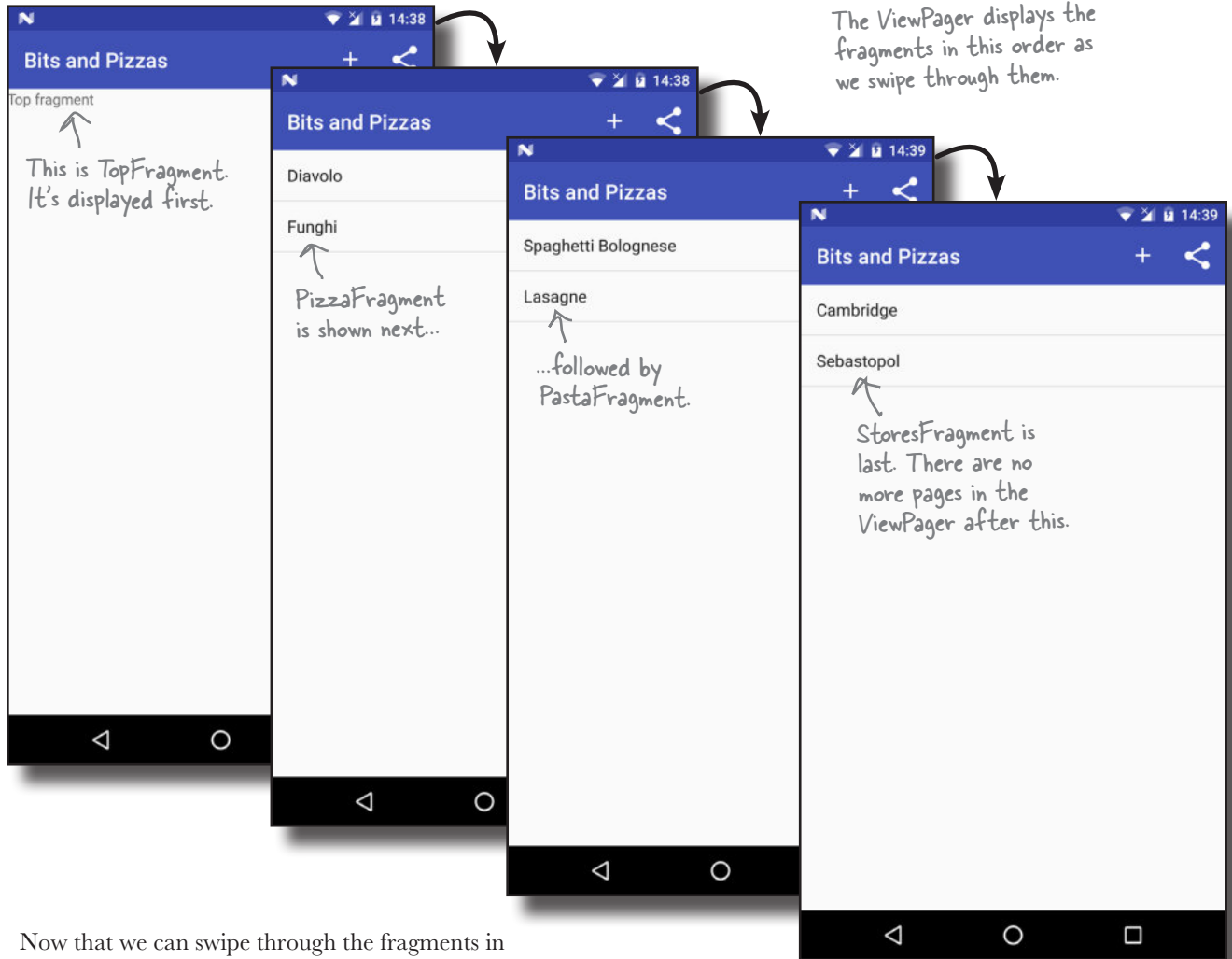


Add fragments

Add swiping

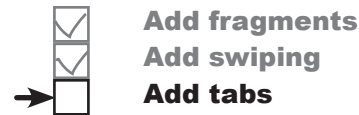
Add tabs

When we run the app, TopFragment is displayed. When we swipe the screen to the left, PizzaFragment is displayed, followed by PastaFragment and StoresFragment. When we swipe the screen in the opposite direction starting from StoresFragment, PastaFragment is displayed, followed by PizzaFragment and TopFragment.

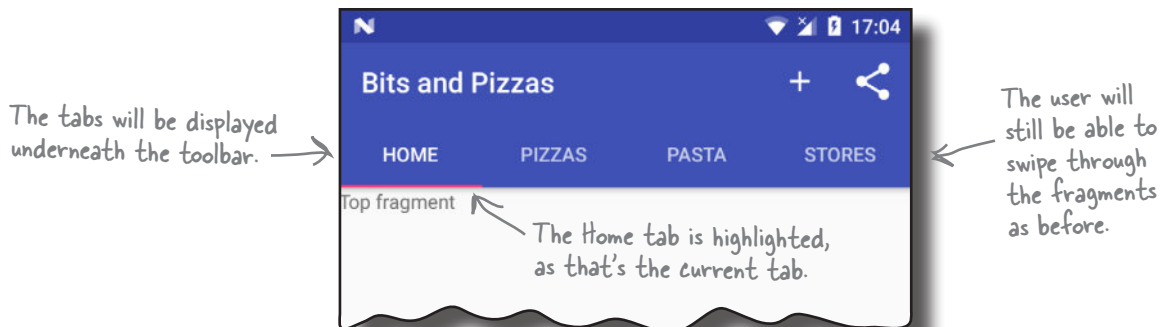


Now that we can swipe through the fragments in MainActivity, let's add tabs.

Add tab navigation to MainActivity

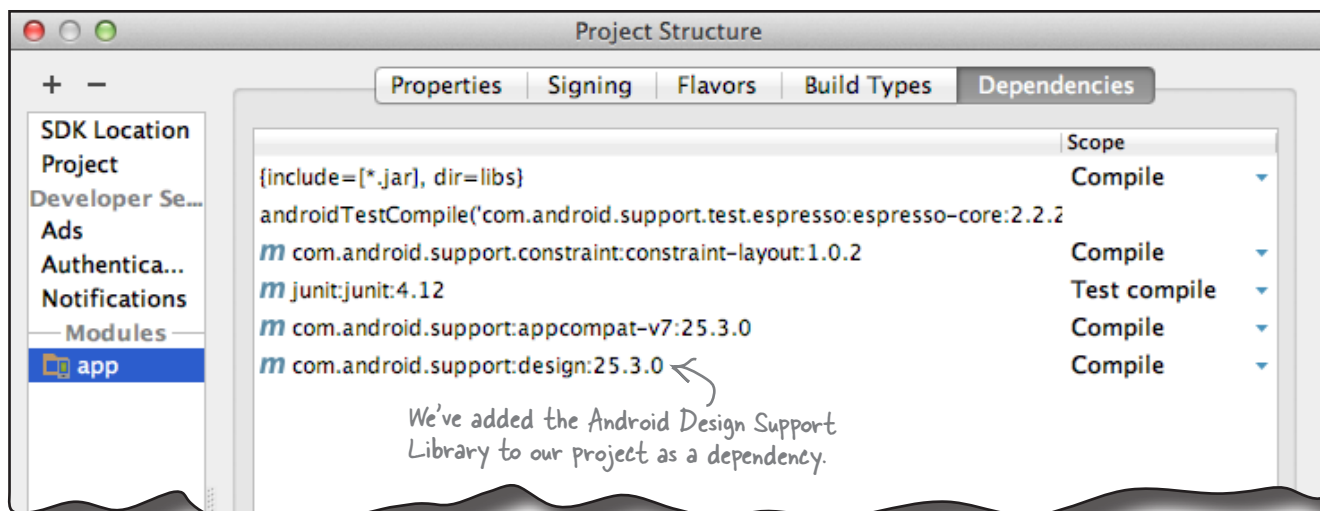


We're going to add tabs to MainActivity as an additional way of navigating through our fragments. Each fragment will be displayed on a separate tab, and clicking on each tab will show that fragment. We'll also be able to swipe through the tabs using the existing view pager.



You use tabs by adding them to your layout, then writing activity code to link the tabs to the view pager. The classes we need to do this come from the **Android Design Support Library**, so you need to add this library to your project as a dependency. To do this, choose File→Project Structure in Android Studio, click on the app module, then choose Dependencies. When you're presented with the project dependencies screen, click on the "+" button at the bottom or right side of the screen. When prompted, choose the Library Dependency option, then select the Design Library from the list of possible libraries. Finally, use the OK buttons to save your changes.

We'll look at the Design Support Library in more detail later in the chapter.





How to add tabs to your layout

You add tabs to your layout using two components from the Design Support Library: a **TabLayout** and an **AppBarLayout**. You use a **TabLayout** to add the tabs, and the **AppBarLayout** to group the tabs and your toolbar together.

The code to add tabs to your layout looks like this:

```
<android.support.design.widget.AppBarLayout
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:theme="@style/ThemeOverlay.AppCompat.Dark.ActionBar" >
```

← The **AppBarLayout** comes from the Design Support Library.

```
<android.support.v7.widget.Toolbar
    android:id="@+id/toolbar"
    android:layout_width="match_parent"
    android:layout_height="?attr/actionBarSize" />
```

← You include the **Toolbar** inside the **AppBarLayout**.

← This line applies a theme to the **Toolbar** and **TabLayout** so that they have a consistent appearance.

```
<android.support.design.widget.TabLayout
    android:id="@+id/tabs"
    android:layout_width="match_parent"
    android:layout_height="wrap_content" />
```

← The **TabLayout** comes from the **DesignSupportLibrary**. You add it to the **AppBarLayout**.

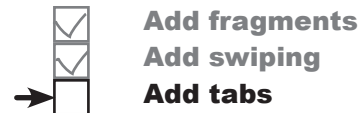
```
</android.support.design.widget.AppBarLayout>
```

The **Toolbar** and **TabLayout** elements both have IDs because you need to be able to reference them in your activity code in order to control their behavior.

The **AppBarLayout** contains both the **Toolbar** and the **TabLayout**. It's a type of vertical linear layout that's designed to work with app bars. The `android:theme` attribute is used to style the **Toolbar** and **TabLayout**. We've given ours a theme of `ThemeOverlay.AppCompat.Dark.ActionBar`.

On the next page we'll show you the code to add tabs to `MainActivity`'s layout.

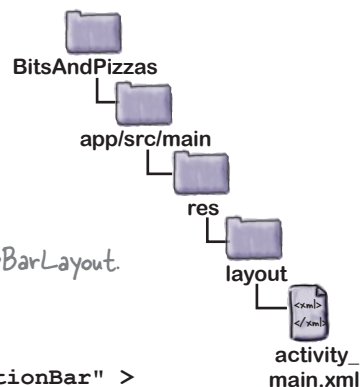
Add tabs to MainActivity's layout



Here's our code for `activity_main.xml`. Update your version of the code to match our 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"
    tools:context="com.hfad.bitsandpizzas.MainActivity">

    <android.support.design.widget.AppBarLayout ← Add an AppBarLayout.
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:theme="@style/ThemeOverlay.AppCompat.Dark.ActionBar" >
```



```
<include
    layout="@layout/toolbar_main"
    android:id="@+id/toolbar" />
```

We've decided to put our Toolbar code in `activity_main.xml` instead of including it from a separate file. This is so that we can show you the full code in one place. In practice, using the `<include>` still works.

The Toolbar goes inside the AppBarLayout.

```
<android.support.v7.widget.Toolbar
    android:id="@+id/toolbar"
    android:layout_width="match_parent"
    android:layout_height="?attr/actionBarSize" />
```

```
<android.support.design.widget.TabLayout ← Add a TabLayout inside the AppBarLayout.
    android:id="@+id/tabs"
    android:layout_width="match_parent"
    android:layout_height="wrap_content" />
</android.support.design.widget.AppBarLayout>
```

```
<android.support.v4.view.ViewPager
    android:id="@+id/pager"
    android:layout_width="match_parent"
    android:layout_height="match_parent" />
</LinearLayout>
```



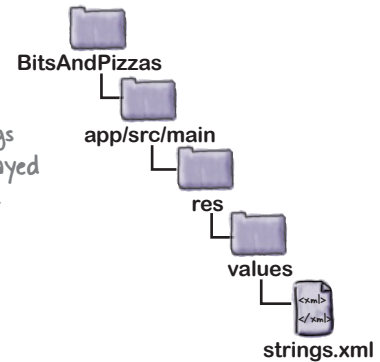
Link the tab layout to the view pager

Once you've added the tab layout, you need to write some activity code to control it. Most of the tab layout's behavior (such as which fragment appears on which tab) comes from the view pager you've already created. All you need to do is implement a method in the view pager's fragment pager adapter to specify the text you want to appear on each tab, then link the view pager to the tab layout.

We're going to add the text we want to appear on each of the tabs as String resources. Open the file *strings.xml*, then add the following Strings:

```
<string name="home_tab">Home</string>
<string name="pizza_tab">Pizzas</string>
<string name="pasta_tab">Pasta</string>
<string name="store_tab">Stores</string>
```

These Strings will be displayed on the tabs.

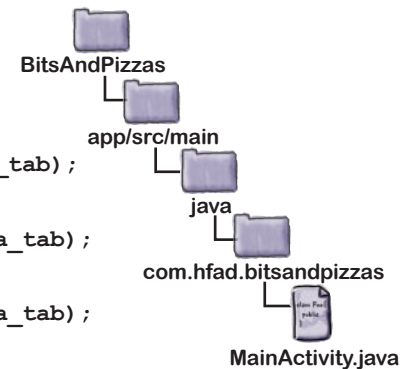


To add the text to each of the tabs, you need to implement the fragment pager adapter's `getPageTitle()` method. This takes one parameter, an `int` for the tab's position, and needs to return the text that should appear on that tab. Here's the code we need to add the above String resources to our four tabs (we'll add it to *MainActivity.java* on the next page):

```
@Override
public CharSequence getPageTitle(int position) {
    switch (position) {
        case 0:
            return getResources().getText(R.string.home_tab);
        case 1:
            return getResources().getText(R.string.pizza_tab);
        case 2:
            return getResources().getText(R.string.pasta_tab);
        case 3:
            return getResources().getText(R.string.store_tab);
    }
    return null;
}
```

This is a new method in the fragment pager adapter we created earlier.

These lines of code add the String resources to the tabs.



Finally, you need to attach the view pager to the tab layout. You do this by calling the `TabLayout` object's `setupWithViewPager()` method, and passing in a reference to the `ViewPager` object as a parameter:

```
TabLayout tabLayout = (TabLayout) findViewById(R.id.tabs);
tabLayout.setupWithViewPager(pager);
```

This line attaches the `ViewPager` to the `TabLayout`. The `TabLayout` uses the `ViewPager` to determine how many tabs there should be, and what should be on each tab.

That's everything we need to get our tabs working. We'll show you the full code for *MainActivity* on the next page.

The full code for MainActivity.java



Add fragments
Add swiping
Add tabs

Here's our full code for *MainActivity.java*. Update your version of the code to match our changes (in bold):

```

package com.hfad.bitsandpizzas;

import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.support.v7.widget.Toolbar;
import android.view.Menu;
import android.view.MenuItem;
import android.content.Intent;
import android.support.v7.widget.ShareActionProvider;
import android.support.v4.view.MenuItemCompat;
import android.support.v4.view.ViewPager;
import android.support.v4.app.Fragment;
import android.support.v4.app.FragmentManager;
import android.support.v4.app.FragmentPagerAdapter;
import android.support.design.widget.TabLayout;

public class MainActivity extends AppCompatActivity {

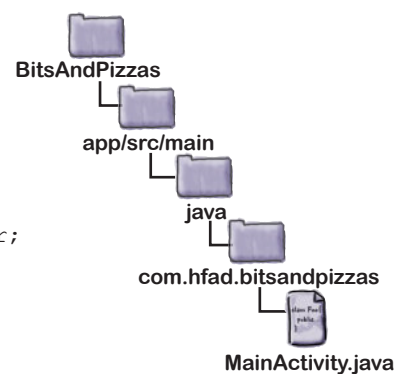
    private ShareActionProvider shareActionProvider;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        Toolbar toolbar = (Toolbar) findViewById(R.id.toolbar);
        setSupportActionBar(toolbar);

        //Attach the SectionsPagerAdapter to the ViewPager
        SectionsPagerAdapter pagerAdapter =
            new SectionsPagerAdapter(getSupportFragmentManager());
        ViewPager pager = (ViewPager) findViewById(R.id.pager);
        pager.setAdapter(pagerAdapter);

        //Attach the ViewPager to the TabLayout
        TabLayout tabLayout = (TabLayout) findViewById(R.id.tabs);
        tabLayout.setupWithViewPager(pager);
    }
}

```



← We're using the *TabLayout* class, so we need to import it.

This links the *ViewPager* to the *TabLayout*.

The code continues on the next page. →



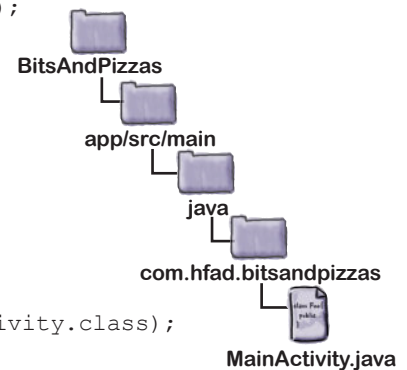
The MainActivity.java code (continued)

None of the code on
this page has changed.

```
@Override
public boolean onCreateOptionsMenu(Menu menu) {
    getMenuInflater().inflate(R.menu.menu_main, menu);
    MenuItem menuItem = menu.findItem(R.id.action_share);
    shareActionProvider =
        (ShareActionProvider) MenuItemCompat.getActionProvider(menuItem);
    setShareActionIntent("Want to join me for pizza?");
    return super.onCreateOptionsMenu(menu);
}
```

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
    switch (item.getItemId()) {
        case R.id.action_create_order:
            Intent intent = new Intent(this, OrderActivity.class);
            startActivity(intent);
            return true;
        default:
            return super.onOptionsItemSelected(item);
    }
}
```

```
private void setShareActionIntent(String text) {
    Intent intent = new Intent(Intent.ACTION_SEND);
    intent.setType("text/plain");
    intent.putExtra(Intent.EXTRA_TEXT, text);
    shareActionProvider.setShareIntent(intent);
}
```



The code continues
on the next page. →

The MainActivity.java code (continued)



Add fragments
Add swiping
Add tabs

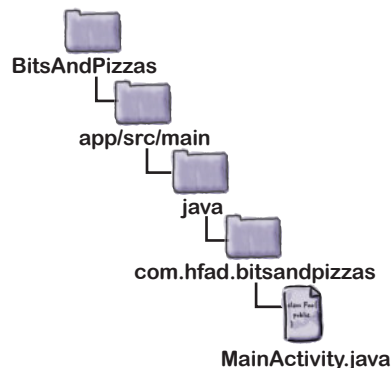
```
private class SectionsPagerAdapter extends FragmentPagerAdapter {

    public SectionsPagerAdapter(FragmentManager fm) {
        super(fm);
    }

    @Override
    public int getCount() {
        return 4;
    }

    @Override
    public Fragment getItem(int position) {
        switch (position) {
            case 0:
                return new TopFragment();
            case 1:
                return new PizzaFragment();
            case 2:
                return new PastaFragment();
            case 3:
                return new StoresFragment();
        }
        return null;
    }

    @Override
    public CharSequence getPageTitle(int position) {
        switch (position) {
            case 0:
                return getResources().getText(R.string.home_tab);
            case 1:
                return getResources().getText(R.string.pizza_tab);
            case 2:
                return getResources().getText(R.string.pasta_tab);
            case 3:
                return getResources().getText(R.string.store_tab);
        }
        return null;
    }
}
}
```



This method adds the text to the tabs.



Test drive the app

design support library

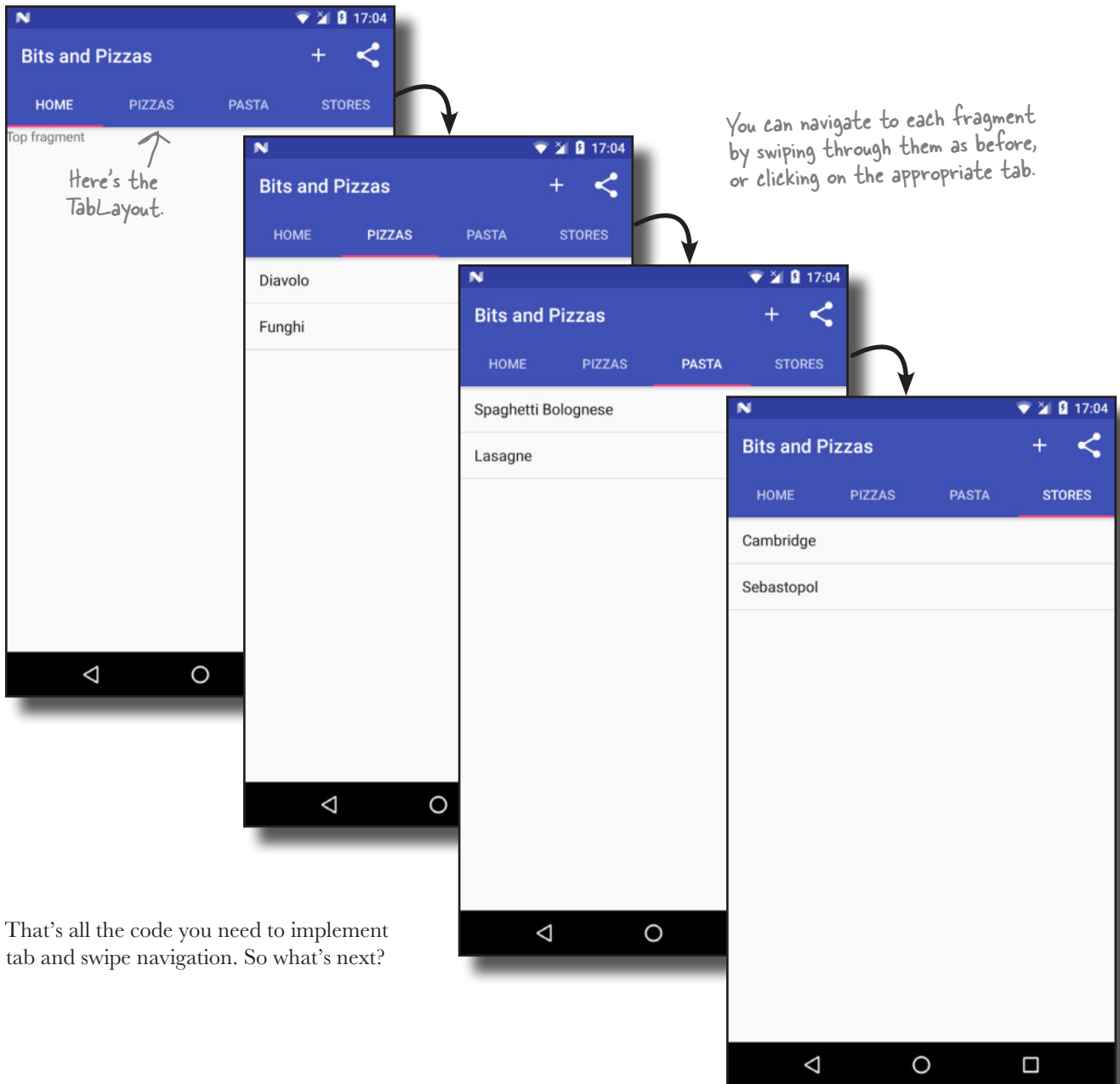
Add fragments

Add swiping

Add tabs



When we run the app, `MainActivity` includes a tab layout. We can swipe through the fragments as before, and we can also navigate to each fragment by clicking on the appropriate tab.



That's all the code you need to implement tab and swipe navigation. So what's next?

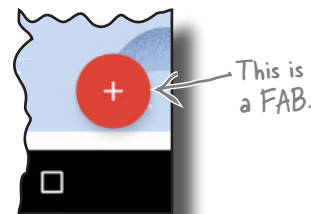
The Design Support Library helps you implement material design

So far, we've added tabs to our app to help the user navigate around the app. To do this, we've used two components from the Design Support Library: the `TabLayout` and `AppBarLayout`.

The Design Support Library was introduced as a way of making it easier for developers to use **material design** components in their apps. Material design was introduced with Lollipop as a way of giving a consistent look and feel to all Android apps. The idea 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. It's inspired by paper and ink, and uses print-based design principles and movement to reflect how real-world objects (such as index cards and pieces of paper) look and behave.

The Design Support Library lets you do more than add tabs to your apps.

- ★ **It lets you add floating action buttons (FABs).**
These are special action buttons that float above the main screen.
- ★ **It includes snackbars, a way of displaying interactive short messages to the user as an alternative to toasts.**
Unlike a toast (which you learned about in Chapter 5), you can add actions to snackbars so that the user can interact with them.



- ★ **You can use it to animate your toolbars.**
You can make your toolbar scroll off the screen, or collapse, if the user scrolls content in another view.
- ★ **It includes a navigation drawer layout.**
This is a slide-out drawer you can use as an alternative to using tabs. We'll look at this feature in Chapter 14.

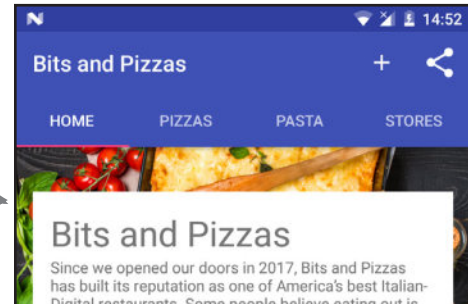
For the rest of this chapter, we're going to show you how to implement some of these features in the Bits and Pizzas app.

Here's what we'll do

We're going to add more goodness from the Design Support Library to the Bits and Pizzas app. Here are the steps we'll go through:

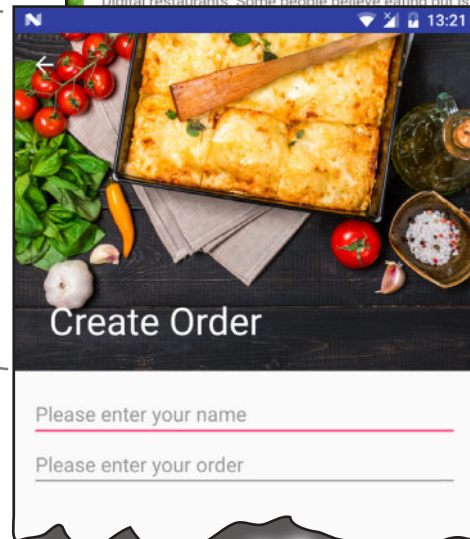
- 1 Enable MainActivity's toolbar to scroll.**
 We'll change MainActivity so that the toolbar scrolls up and down when the user scrolls the contents of the view pager we added earlier. To see this working, we'll add content we can scroll to TopFragment.

When the user scrolls this content, the toolbar will scroll up too.

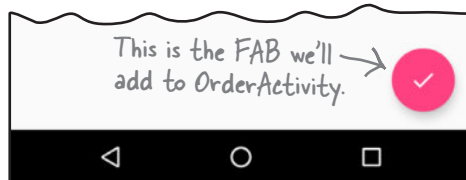


- 2 Add a collapsing toolbar to OrderActivity.**
 We'll start by adding a plain collapsing toolbar to OrderActivity. The toolbar will collapse when the user scrolls OrderActivity's contents. After we've got the plain collapsing toolbar working, we'll add an image to it.

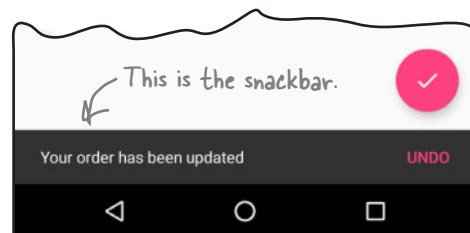
This is a toolbar with an image. When the user scrolls the main content, we'll get it to collapse.



- 3 Add a FAB to OrderActivity.**
 We'll display a floating action button to the bottom-right corner.



- 4 Make the FAB display a snackbar.**
 The snackbar will appear at the bottom of the screen when the user clicks on the FAB. The FAB will move up when the snackbar appears, and move back down when the snackbar is no longer there.



We'll start by getting our toolbar to scroll in response to the user scrolling the content in the view pager.

Make the toolbar respond to scrolls

We're going to change our app so that MainActivity's toolbar scrolls whenever the user scrolls content in TopFragment. To enable this, there are two things we need to do:

- 1 Change MainActivity's layout to enable the toolbar to scroll.
- 2 Change TopFragment to include scrollable content.

We'll start by changing MainActivity's layout.

Use a CoordinatorLayout to coordinate animations between views

To get the toolbar to move when content in the fragment is scrolled, we'll add a **coordinator layout** to MainActivity. A coordinator layout is like a souped-up frame layout that's used to coordinate animations and transitions between different views. In this case, we'll use the coordinator layout to coordinate scrollable content in TopFragment and MainActivity's toolbar.

You add a coordinator layout to an activity's layout using code like this:

```
<android.support.design.widget.CoordinatorLayout
    android:layout_width="match_parent"
    android:layout_height="match_parent">
    ...
</android.support.design.widget.CoordinatorLayout>
```

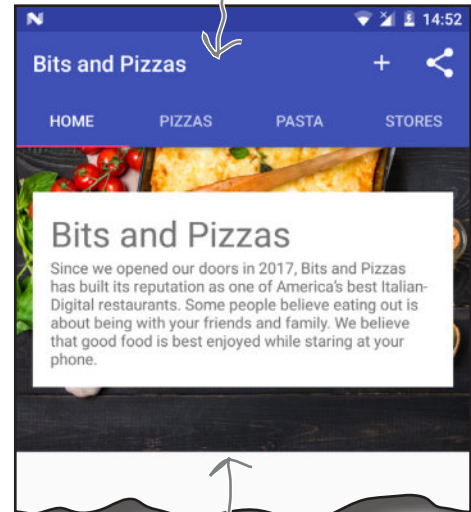
... ← You add any views whose behavior you want to coordinate inside the CoordinatorLayout.

Any views in your layout whose animations you want to coordinate must be included in the `<CoordinatorLayout>` element. In our case, we want to coordinate animations between the toolbar and the contents of the view pager, so these views need to be included in the coordinator layout.



Scrolling toolbar
Collapsing toolbar
FAB
Snackbar

We'll get the toolbar to scroll when the user scrolls the content in TopFragment.



We'll add this scrollable content to TopFragment.

← The CoordinatorLayout comes from the Design Support Library.

A CoordinatorLayout allows the behavior of one view to affect the behavior of another.

Add a coordinator layout to MainActivity's layout

We're going to replace the linear layout in `activity_main.xml` with a coordinator layout. Here's our code; update your version to match our changes (in bold):

```
<?xml version="1.0" encoding="utf-8"?>
```

```
<android.support.design.widget.CoordinatorLayout
```

```
<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="com.hfad.bitsandpizzas.MainActivity">
```

```
<android.support.design.widget.AppBarLayout
```

```
    android:layout_width="match_parent"
```

```
    android:layout_height="wrap_content"
```

```
    android:theme="@style/ThemeOverlay.AppCompat.Dark.ActionBar">
```

```
<android.support.v7.widget.Toolbar
```

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

```
    android:layout_width="match_parent"
```

```
    android:layout_height="?attr/actionBarSize" />
```

```
<android.support.design.widget.TabLayout
```

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

```
    android:layout_width="match_parent"
```

```
    android:layout_height="wrap_content" />
```

```
</android.support.design.widget.AppBarLayout>
```

```
<android.support.v4.view.ViewPager
```

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

```
    android:layout_width="match_parent"
```

```
    android:layout_height="match_parent" />
```

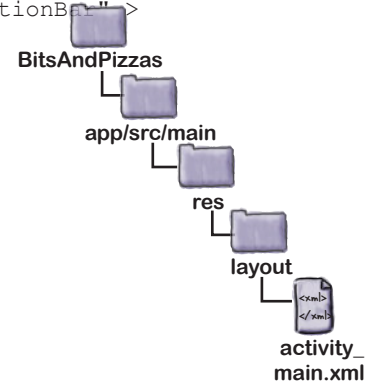
```
</android.support.design.widget.CoordinatorLayout>
```

```
<LinearLayout
```

We're replacing the `LinearLayout` with a `CoordinatorLayout`.

Add the app namespace, as we'll need to use attributes from it over the next few pages.

Delete this line, as we're no longer using a `LinearLayout`.



We're replacing the `LinearLayout` with a `CoordinatorLayout`.

How to coordinate scroll behavior

As well as adding views to the coordinator layout, you need to say how you want them to behave. In our case, we want the toolbar to scroll in response to another view's scroll event. This means that we need to **mark the view the user will scroll**, and **tell the toolbar to respond to it**.

Mark the view the user will scroll

You mark the view the user will scroll by giving it an attribute of **app:layout_behavior** and setting it to the built-in String "`@string/appbar_scrolling_view_behavior`". This tells the coordinator layout that you want views in the app bar layout to be able to respond when the user scrolls this view. In our case, we want the toolbar to scroll in response to the user scrolling the view pager's content, so we need to add the `app:layout_behavior` attribute to the `ViewPager` element:

```
<android.support.v4.view.ViewPager
    ...
    app:layout_behavior="@string/appbar_scrolling_view_behavior" />
```



Scrolling toolbar
Collapsing toolbar
FAB
Snackbar

You add this line to the `ViewPager` to tell the `CoordinatorLayout` you want to react to the user scrolling its content.

Tell the toolbar to respond to scroll events

You tell views in the app bar layout how to respond to scroll events using the **app:layout_scrollFlags** attribute. In our case, we're going to set the toolbar to scroll upward off the screen when the user scrolls the view pager content up, and quickly return to its original position when the user scrolls down. To do this, we need to set the `Toolbar` `app:layout_scrollFlags` attribute to "`scroll|enterAlways`".

The `scroll` value allows the view to scroll off the top of screen. Without this, the toolbar would stay pinned to the top of the screen. The `enterAlways` value means that the toolbar quickly scrolls down to its original position when the user scrolls the corresponding view. The toolbar will still scroll down without this value, but it will be slower.

Here's the code we need to add to the toolbar to enable it to scroll:

```
<android.support.v7.widget.Toolbar
    ...
    app:layout_scrollFlags="scroll|enterAlways" />
```

The toolbar MUST be contained within an app bar layout in order for it to scroll. The app bar layout and coordinator layout work together to enable the toolbar to scroll.

← This line tells the `CoordinatorLayout` (and `AppBarLayout`) how you want the `Toolbar` to react to the user scrolling content.

We'll look at the full code for `MainActivity`'s layout on the next page.



The code to enable to toolbar to scroll

Here's our updated code for *activity_main.xml*. Update your version of the code to match our changes (in bold):

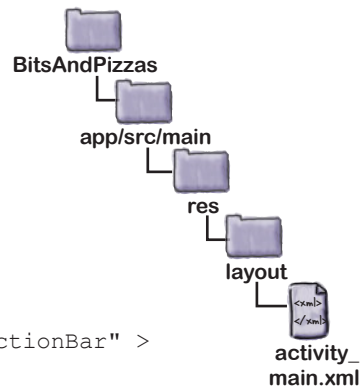
```
<?xml version="1.0" encoding="utf-8"?>
<android.support.design.widget.CoordinatorLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"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

    <android.support.design.widget.AppBarLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:theme="@style/ThemeOverlay.AppCompat.Dark.ActionBar" >

        <android.support.v7.widget.Toolbar
            android:id="@+id/toolbar"
            android:layout_width="match_parent"
            android:layout_height="?attr/actionBarSize"
            app:layout_scrollFlags="scroll|enterAlways" />

        <android.support.design.widget.TabLayout
            android:id="@+id/tabs"
            android:layout_width="match_parent"
            android:layout_height="wrap_content" />
    </android.support.design.widget.AppBarLayout>

    <android.support.v4.view.ViewPager
        android:id="@+id/pager"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        app:layout_behavior="@string/appbar_scrolling_view_behavior" />
</android.support.design.widget.CoordinatorLayout>
```



← Add this line to enable the Toolbar to scroll. If you wanted the TabLayout to scroll too, you'd add the code to that element as well.

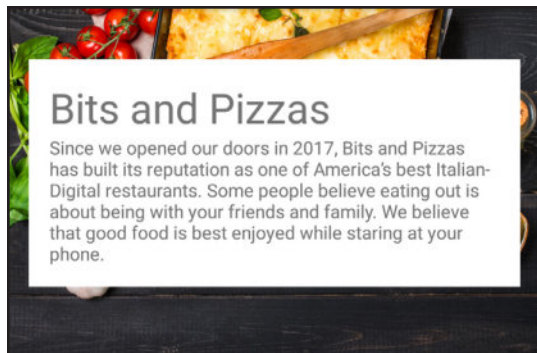
↑ This line marks the view whose content you expect the user to scroll.

Those are all the changes we need to make to MainActivity. Next we'll add scrollable content to TopFragment.

Add scrollable content to TopFragment

We're going to change `TopFragment`'s layout so that it contains scrollable content. We'll add an image of one of the Bits and Pizzas restaurants, along with some text describing the company ethos.

Here's what the new version of `TopFragment` will look like:



Scrolling toolbar
Collapsing toolbar
FAB
Snackbar

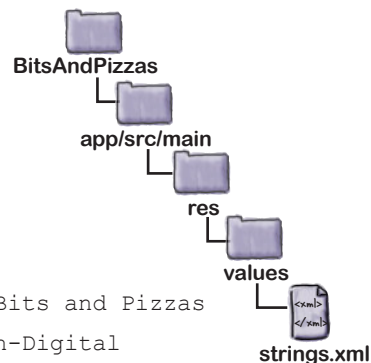
← We'll change `TopFragment` to include an image and some text. We want the user to be able to scroll the entire contents of the fragment.

We'll start by adding the String and image resources to our project.

Add String and image resources

We'll add the String resources first. Open `strings.xml`, then add the following:

```
<string name="company_name">Bits and Pizzas</string>
<string name="restaurant_image">Restaurant image</string>
<string name="home_text">Since we opened our doors in 2017, Bits and Pizzas
    has built its reputation as one of America's best Italian-Digital
    restaurants. Some people believe eating out is about being with your
    friends and family. We believe that good food is best enjoyed while
    staring at your phone.</string>
```



Next we'll add the restaurant image to the `drawable-nodpi` folder. First, switch to the Project view of Android Studio's explorer and check whether the `app/src/main/res/drawable-nodpi` folder exists in your project. If it's not already there, highlight 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 it "drawable-nodpi" and click on OK.

Once you have a `drawable-nodpi` folder, download the file `restaurant.jpg` from <https://git.io/v9oet>, and add it to the `drawable-nodpi` folder.

You need to add this image to the `drawable-nodpi` folder.



Use a nested scroll view to make layout content scrollable

We'll allow the user to scroll the contents of `TopFragment` using a **nested scroll view**. 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 scroll when the user scrolls the content.

← Another view that enables nested scrolling is the recycler view. You'll find out how to use this in the next chapter.

You add a nested scroll view to your layout using code like this:

```
<android.support.v4.widget.NestedScrollView
    android:layout_width="match_parent"
    android:layout_height="match_parent" >
```

← The `NestedScrollView` comes from the Design Support Library.

```
... ← You add any views you want the user to be
    able to scroll to the NestedScrollView.
```

```
</android.support.v4.widget.NestedScrollView >
```

You add any views you want the user to be able to scroll to the nested scroll view. If you just have one view, you can add this to the nested scroll view directly. If you want to scroll multiple views, however, these must be added to a separate layout inside the scroll view. This is because a nested scroll view can only have one direct child. As an example, here's how you'd add two text views to a nested scroll view with the help of a linear layout:

```
<android.support.v4.widget.NestedScrollView
    android:layout_width="match_parent"
    android:layout_height="match_parent" >
```

```
<LinearLayout
    ... >
```

← We're just using a `LinearLayout` as an example—it could be some other sort of layout instead. The key point is that the `NestedScrollView` can only have one direct child. If you want to put more than one view in the `NestedScrollView`, in this case two `TextView`s, you must put them in another layout first.

```
<TextView
    ... />
```

```
<TextView
    ... />
```

```
</LinearLayout>
```

```
</android.support.v4.widget.NestedScrollView >
```

Next we'll update `TopFragment`'s layout so it uses a nested scroll view.

How we'll structure TopFragment's layout

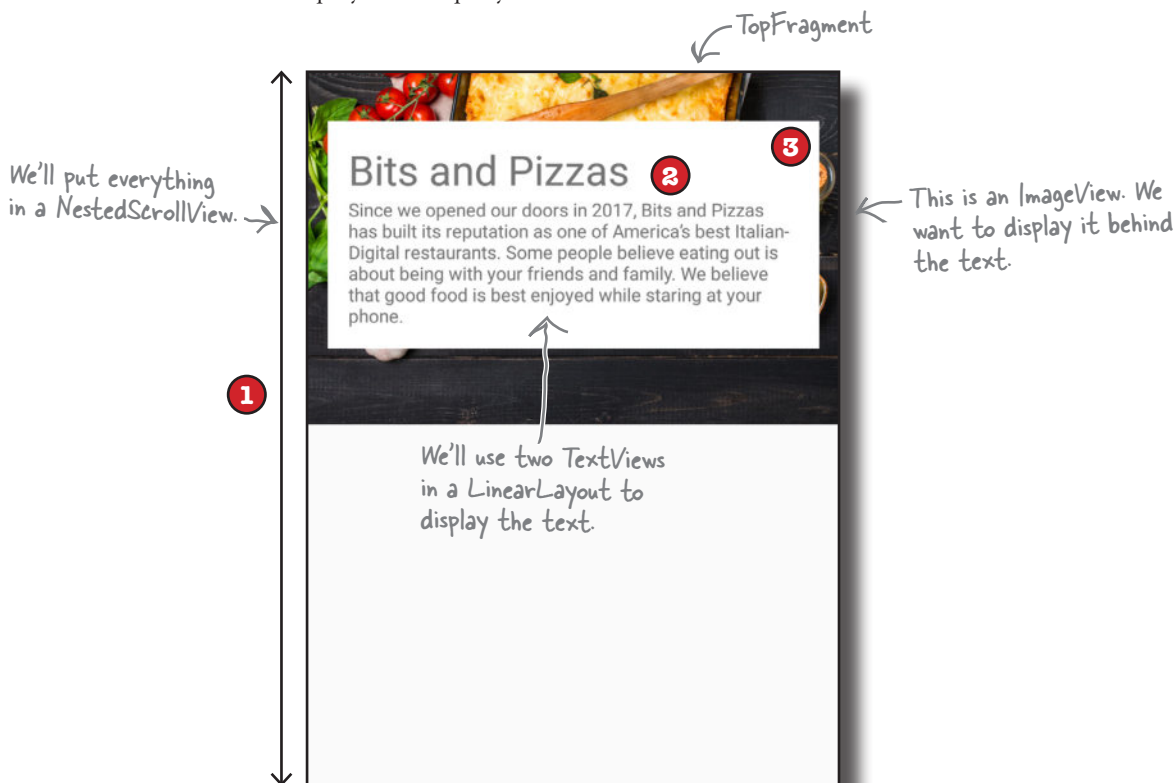


Scrolling toolbar
Collapsing toolbar
FAB
Snackbar

We're going to add a restaurant image and some text to TopFragment's layout. Before we write the code, here's a breakdown of how we'll structure it.

- 1 We want the entire fragment to be scrollable. This means we need to put all the views in a nested scroll view.
- 2 We'll use two text views for the Bits and Pizzas company name and text. We'll put these in a vertical linear layout with a white background.
- 3 We want to display the linear layout containing the two text views on top of the image. We'll do this by putting them both in a frame layout.

Putting this together, we'll use a nested scroll view for our layout, and this will contain a frame layout. The frame layout will include two elements, an image view and a linear layout. The linear layout will contain two text views to display the company name and ethos.



On the next page we'll show you the full code for `fragment_top.xml`. Once you've updated your code, we'll take the app for a test drive.



The full code for `fragment_top.xml`

Here's the full code for `fragment_top.xml`; update your code to match ours:

```

<android.support.v4.widget.NestedScrollView ← We want the whole fragment to be able to scroll.
    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="com.hfad.bitsandpizzas.TopFragment">
    <FrameLayout ← We're using a FrameLayout because we want to
        android:layout_width="match_parent"
        android:layout_height="wrap_content" >

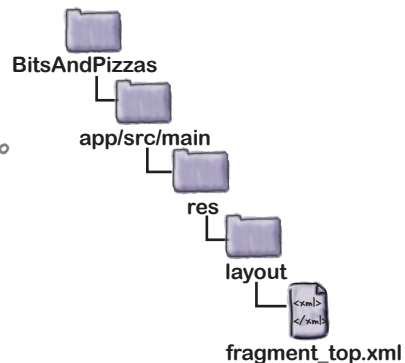
        <ImageView android:id="@+id/info_image"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:scaleType="centerCrop"
            android:src="@drawable/restaurant"
            android:contentDescription="@string/restaurant_image" />

        <LinearLayout ← We're using a LinearLayout to
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:layout_marginTop="40dp"
            android:layout_marginLeft="16dp"
            android:layout_marginRight="16dp"
            android:padding="16dp"
            android:background="#FFFFFF"
            android:orientation="vertical">

            <TextView
                android:textSize="32sp"
                android:layout_width="match_parent"
                android:layout_height="wrap_content"
                android:text="@string/company_name" />

            <TextView
                android:layout_width="match_parent"
                android:layout_height="wrap_content"
                android:text="@string/home_text" />
        </LinearLayout>
    </FrameLayout>
</android.support.v4.widget.NestedScrollView>

```

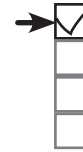


test drive



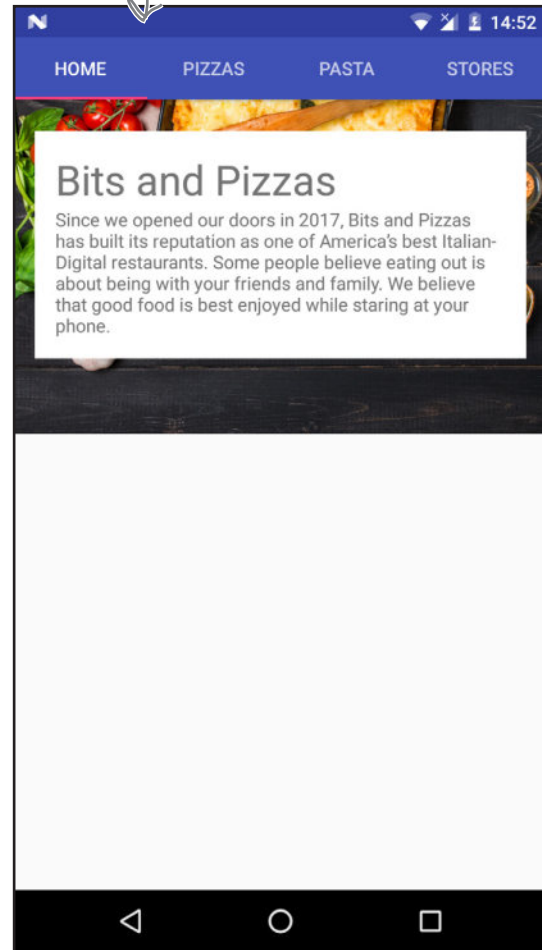
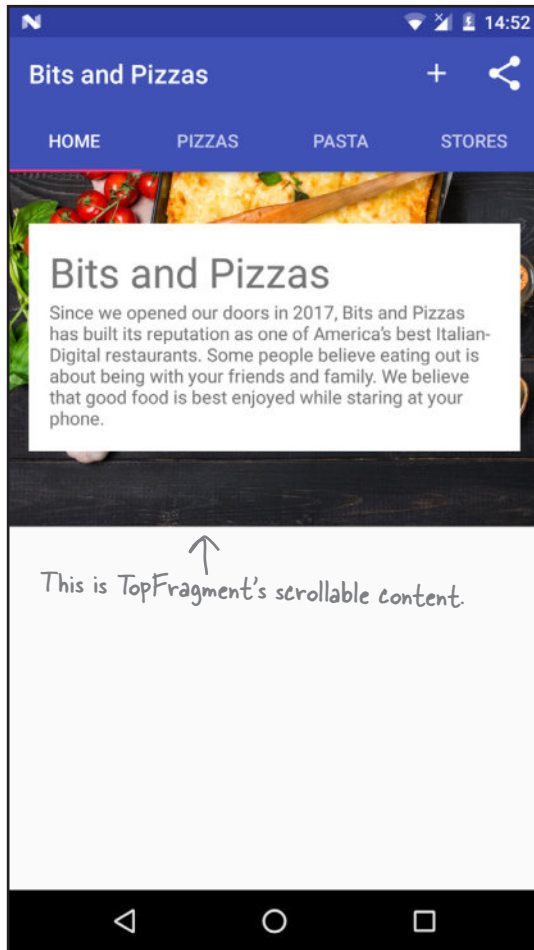
Test drive the app

When we run the app, TopFragment displays the new layout. When we scroll the content, the toolbar scrolls too.

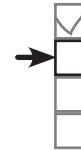


Scrolling toolbar
Collapsing toolbar
FAB
Snackbar

When you scroll the content up,
the toolbar scrolls up too.

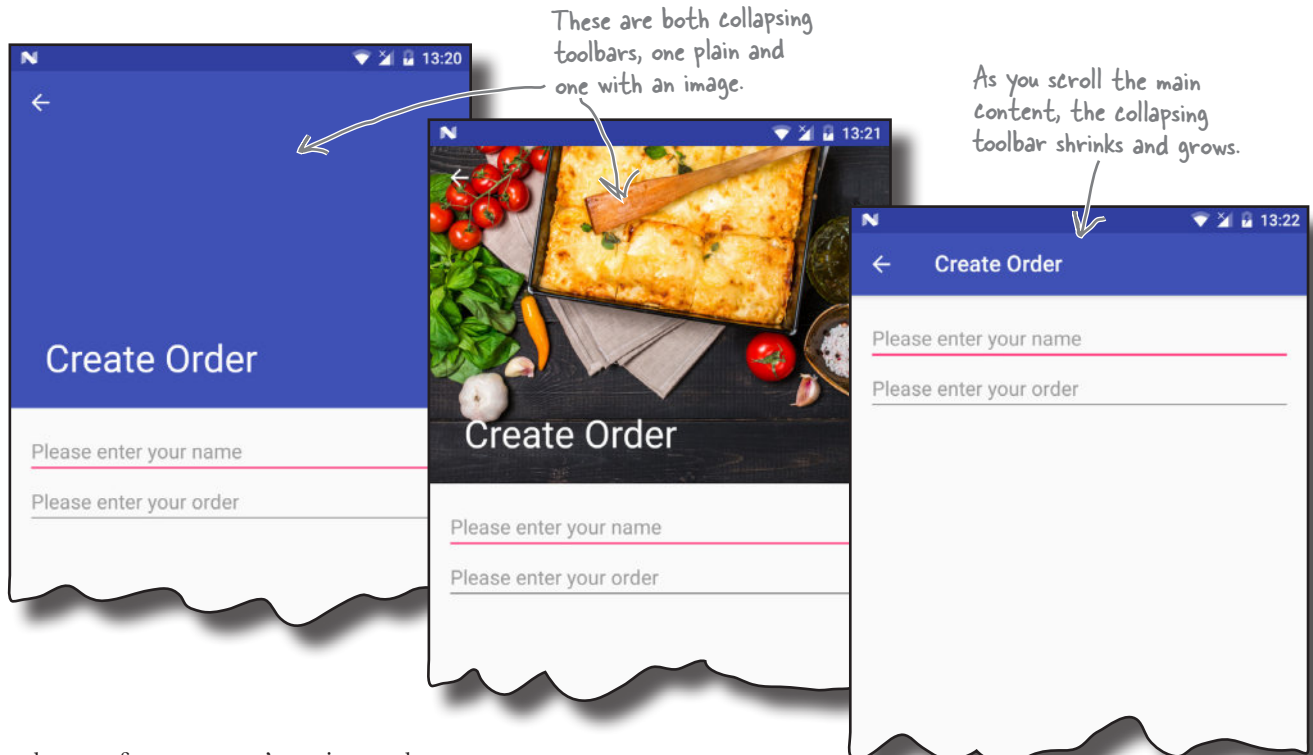


By allowing the toolbar to scroll, you free up more space for content. An added bonus is that you don't have to write any activity or fragment code to control the toolbar's behavior. All of the functionality came from using widgets from the Design Support Library.



Add a collapsing toolbar to OrderActivity

A variant of allowing your toolbar to scroll is the **collapsing toolbar**. A collapsing toolbar is one that starts off large, shrinks when the user scrolls the screen content up, and grows again when the user scrolls the screen content 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:



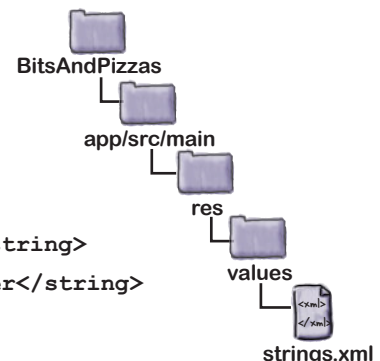
Over the next few pages, we're going to change `OrderActivity` to include a collapsing toolbar.

First add some String resources

Before we get started, we need to add some String resources to `strings.xml` that we'll use in `OrderActivity`'s layout. Open `strings.xml`, then add the following resources:

```
<string name="order_name_hint">Please enter your name</string>
<string name="order_details_hint">Please enter your order</string>
```

We'll start updating the layout on the next page.



How to create a plain collapsing toolbar

You add a collapsing toolbar to your activity's layout using the collapsing toolbar layout from the Design Support Library. In order for it to work, you need to add the collapsing toolbar layout to an app bar layout that's included within a coordinator layout. The collapsing toolbar layout should contain the toolbar you want to collapse.

As the collapsing toolbar needs to respond to scroll events in a separate view, you also need to add scrollable content to the coordinator layout, for example using a nested scroll view.

Here's an overview of how you need to structure your layout file in order to use a collapsing toolbar:

```
<android.support.design.widget.CoordinatorLayout
    ... >
```

```
    <android.support.design.widget.AppBarLayout
        ... >
```

```
        <android.support.design.widget.CollapsingToolbarLayout
            ... >
```

```
            <android.support.v7.widget.Toolbar
                ... />
```

```
        </android.support.design.widget.CollapsingToolbarLayout>
    </android.support.design.widget.AppBarLayout>
```

```
    <android.support.v4.widget.NestedScrollView
        ...>
```

```
        ... ← The scrollable content goes here.
```

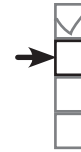
```
    </android.support.v4.widget.NestedScrollView>
</android.support.design.widget.CoordinatorLayout>
```

In addition to structuring your layout in a particular way, there are some key attributes you need to use to get your collapsing toolbar to work properly. We'll look at these next.



Scrolling toolbar
Collapsing toolbar
FAB
Snackbar

You add the CollapsingToolbarLayout to an AppBarLayout, which sits inside a CoordinatorLayout. The CollapsingToolbarLayout contains the Toolbar.



Nested scroll view attributes

As before, you need to tell the coordinator layout which view you expect the user to scroll. You do this by setting the nested scroll view's `layout_behavior` attribute to `"@string/appbar_scrolling_view_behavior"`:

```
<android.support.v4.widget.NestedScrollView
    ...
    app:layout_behavior="@string/appbar_scrolling_view_behavior" >
```

This is the same as when we created a scrolling toolbar.

Collapsing toolbar layout attributes

You want the collapsing toolbar layout to collapse and expand in response to scroll events, so you need to set its `layout_scrollFlags` attribute to control this behavior. In our case, we want the collapsing toolbar layout to collapse until it's the size of a standard toolbar, so we'll set the attribute to a value of `"scroll|exitUntilCollapsed"`:

```
<android.support.design.widget.CollapsingToolbarLayout
    ...
    app:layout_scrollFlags="scroll|exitUntilCollapsed" >
```

This means we want the toolbar to collapse until it's done collapsing.

App bar layout attributes

As before, you apply a theme to your app bar layout to control the appearance of its contents. You also need to specify a height for the contents of the app bar layout. This is the maximum height the collapsing toolbar will be able to expand to. In our case, we'll apply a theme of `"@style/ThemeOverlay.AppCompat.Dark.ActionBar"` as before and set the height to 300dp:

```
<android.support.design.widget.AppBarLayout
    android:layout_width="match_parent"
    android:layout_height="300dp"
    android:theme="@style/ThemeOverlay.AppCompat.Dark.ActionBar" >
```

This is the maximum height of the collapsing toolbar.

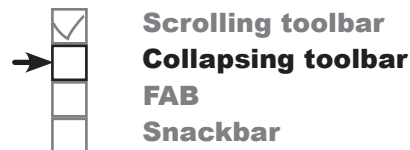
Toolbar attributes

If you have items on your toolbar such as an Up button, these may scroll off the screen as the toolbar collapses. You can prevent this from happening by setting the `layout_collapseMode` attribute to `"pin"`:

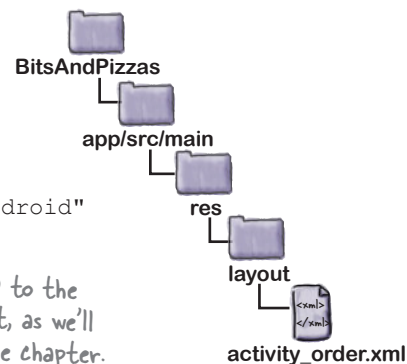
```
<android.support.v7.widget.Toolbar
    android:id="@+id/toolbar"
    android:layout_width="match_parent"
    android:layout_height="?attr/actionBarSize"
    app:layout_collapseMode="pin" />
```

This pins anything that's on the toolbar, such as the Up button, to the top of the screen.

The full code to add a collapsing toolbar to activity_order.xml



Here's how to add a collapsing toolbar to OrderActivity's layout. Replace your existing code for `activity_order.xml` with the code below:



```
<?xml version="1.0" encoding="utf-8"?>
<android.support.design.widget.CoordinatorLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:id="@+id/coordinator" ← We've added an ID to the
    android:layout_width="match_parent"      CoordinatorLayout, as we'll
    android:layout_height="match_parent" >
```

```
<android.support.design.widget.AppBarLayout
    android:layout_width="match_parent"
    android:layout_height="300dp"
    android:theme="@style/ThemeOverlay.AppCompat.Dark.ActionBar" >
```

```
<android.support.design.widget.CollapsingToolbarLayout
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    app:layout_scrollFlags="scroll|exitUntilCollapsed" >
```

← This is the CollapsingToolbarLayout. It needs to be within an AppBarLayout.

```
<android.support.v7.widget.Toolbar ← The CollapsingToolbarLayout
    android:id="@+id/toolbar"          contains a toolbar.
    android:layout_width="match_parent"
    android:layout_height="?attr/actionBarSize"
    app:layout_collapseMode="pin" />
```

```
</android.support.design.widget.CollapsingToolbarLayout>
</android.support.design.widget.AppBarLayout>
```

The code continues on the next page. →

The activity_order.xml code (continued)



```
<android.support.v4.widget.NestedScrollView
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    app:layout_behavior="@string/appbar_scrolling_view_behavior" >
```

← The NestedScrollView contains the content we want the user to be able to scroll.

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

    <EditText
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:hint="@string/order_name_hint" />

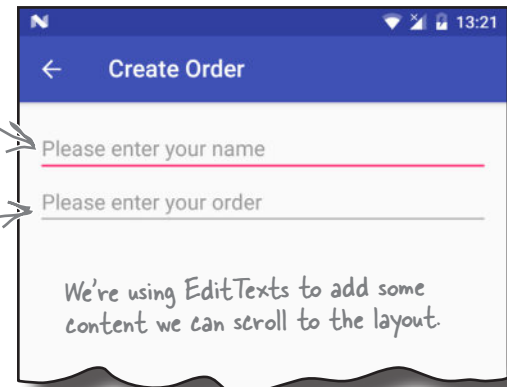
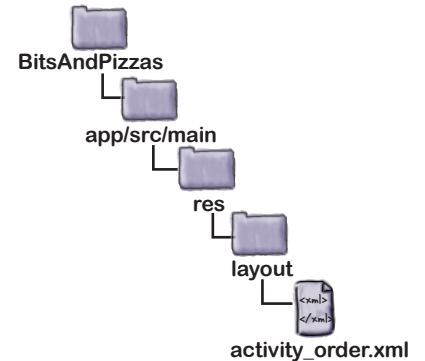
    <EditText
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:hint="@string/order_details_hint" />

</LinearLayout>

</android.support.v4.widget.NestedScrollView>

</android.support.design.widget.CoordinatorLayout>
```

We're using a *LinearLayout* to position the scrollable content.

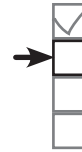


Let's see what happens when we run the app.



Test drive the app

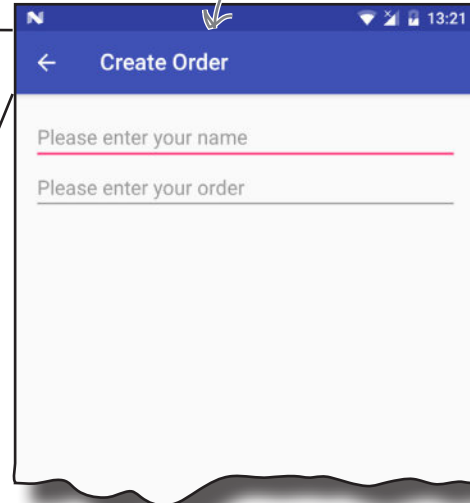
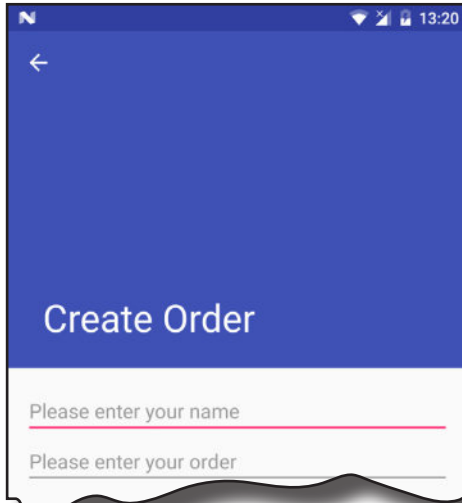
When we run the app, `OrderActivity` displays the new layout, including the collapsing toolbar. The collapsing toolbar starts off large, and collapses as we scroll the content.



Scrolling toolbar
Collapsing toolbar
FAB
Snackbar

This is the toolbar when it's collapsed.

This is the toolbar when it's expanded.

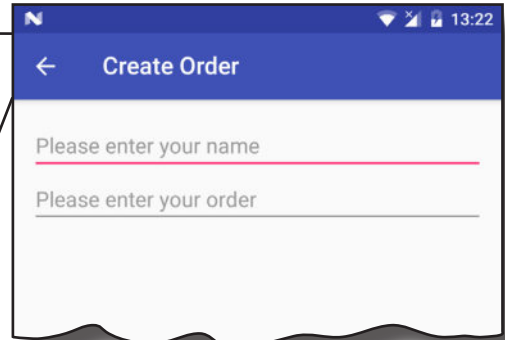
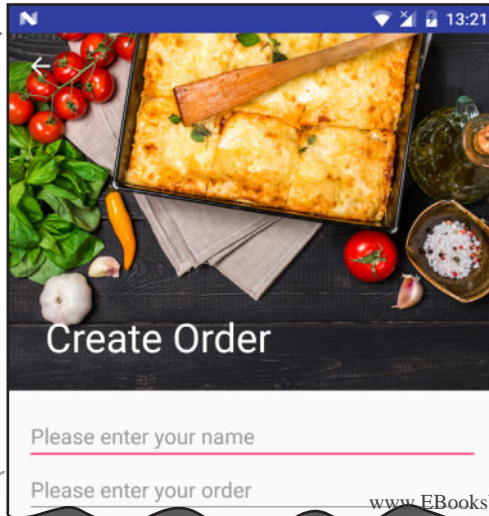


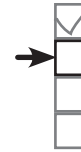
You can add images to collapsing toolbars too

The collapsing toolbar we've created so far is quite plain. It has a plain background, which grows and shrinks as we scroll the content in the activity.

We can improve this by adding an image to the collapsing toolbar. We'll display the image when the collapsing toolbar is large, and when it shrinks we'll display a standard toolbar instead:

This is the same collapsing toolbar, except we're going to add an image to it.





How to add an image to a collapsing toolbar

We're going to update the collapsing toolbar we just created so that it includes an image. For convenience, we'll use the same image that we added to TopFragment.

You add an image to a collapsing toolbar by adding an `ImageView` to the `CollapsingToolBarLayout`, specifying the image you want to use. As an optional extra, you can add a parallax effect to the `ImageView` so that the image scrolls at a different rate than the rest of the toolbar. You do this by adding a `layout_collapseMode` attribute to the `ImageView` with a value of "parallax".

We want to use a drawable named "restaurant" for our image. Here's the code we need:

```
<android.support.design.widget.CollapsingToolBarLayout
... >
```

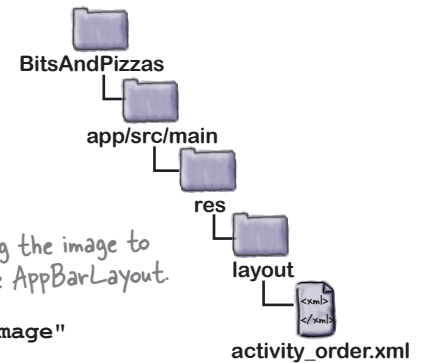
```
<ImageView
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:scaleType="centerCrop"
    android:src="@drawable/restaurant"
    android:contentDescription="@string/restaurant_image"
    app:layout_collapseMode="parallax" />
```

```
<Toolbar
... >
```

```
</android.support.design.widget.CollapsingToolBarLayout>
```

We're cropping the image to fit inside the `AppBarLayout`.

This line is optional. It adds parallax animation so the image moves at a different rate than the scrollable content.



By default, when the toolbar is collapsed, it will continue to display the image as its background. To get the toolbar to revert to a plain background color when it's collapsed, you add a `contentScrim` attribute to the `CollapsingToolBarLayout`, setting it to the value of the color. We want our toolbar to have the same background color as before, so we'll set it to "`?attr/colorPrimary`":

```
<android.support.design.widget.CollapsingToolBarLayout
...
app:layout_scrollFlags="scroll|exitUntilCollapsed"
app:contentScrim="?attr/colorPrimary" >
```

This line changes the toolbar back to its default color when it's collapsed.

Those are all the changes we need, so we'll update the code on the next page, and then take it for a test drive.

The updated code for activity_order.xml

Here's the updated code for `activity_order.xml` to add an image to the collapsing toolbar (update your version to match our changes in bold>):

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.design.widget.CoordinatorLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:id="@+id/coordinator"
    android:layout_width="match_parent"
    android:layout_height="match_parent" >

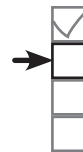
    <android.support.design.widget.AppBarLayout
        android:layout_width="match_parent"
        android:layout_height="300dp"
        android:theme="@style/ThemeOverlay.AppCompat.Dark.ActionBar" >

        <android.support.design.widget.CollapsingToolbarLayout
            android:layout_width="match_parent"
            android:layout_height="match_parent"
            app:layout_scrollFlags="scroll|exitUntilCollapsed"
            app:contentScrim="?attr/colorPrimary" >

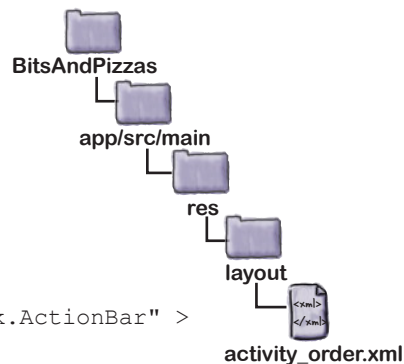
            <ImageView
                android:layout_width="match_parent"
                android:layout_height="match_parent"
                android:scaleType="centerCrop"
                android:src="@drawable/restaurant"
                android:contentDescription="@string/restaurant_image"
                app:layout_collapseMode="parallax" />

            <android.support.v7.widget.Toolbar
                android:id="@+id/toolbar"
                android:layout_width="match_parent"
                android:layout_height="?attr/actionBarSize"
                app:layout_collapseMode="pin" />

        </android.support.design.widget.CollapsingToolbarLayout>
    </android.support.design.widget.AppBarLayout>
```



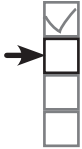
Scrolling toolbar
Collapsing toolbar
FAB
Snackbar



These lines add an image to the collapsing toolbar. It uses snazzy parallax animation.

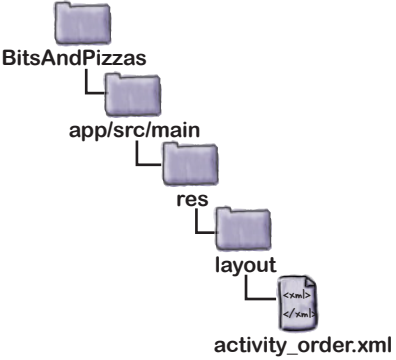
← This line changes the toolbar background when it's collapsed.

The code continues on the next page. →



The activity_order.xml code (continued)

```
<android.support.v4.widget.NestedScrollView
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    app:layout_behavior="@string/appbar_scrolling_view_behavior" >
    ...
</android.support.v4.widget.NestedScrollView>
</android.support.design.widget.CoordinatorLayout>
```



Let's see what the app looks like when we run it.



Test drive the app

When we run the app, `OrderActivity`'s collapsing toolbar includes an image. As the toolbar collapses, the image fades, and the toolbar's background changes to its original color. When the toolbar is expanded again, the image reappears.

The image appears on the toolbar.

The toolbar changes color when it's collapsed.

FABs and snackbars

There are two final additions we're going to make to `OrderActivity` from the Design Support Library: a FAB and a snackbar.

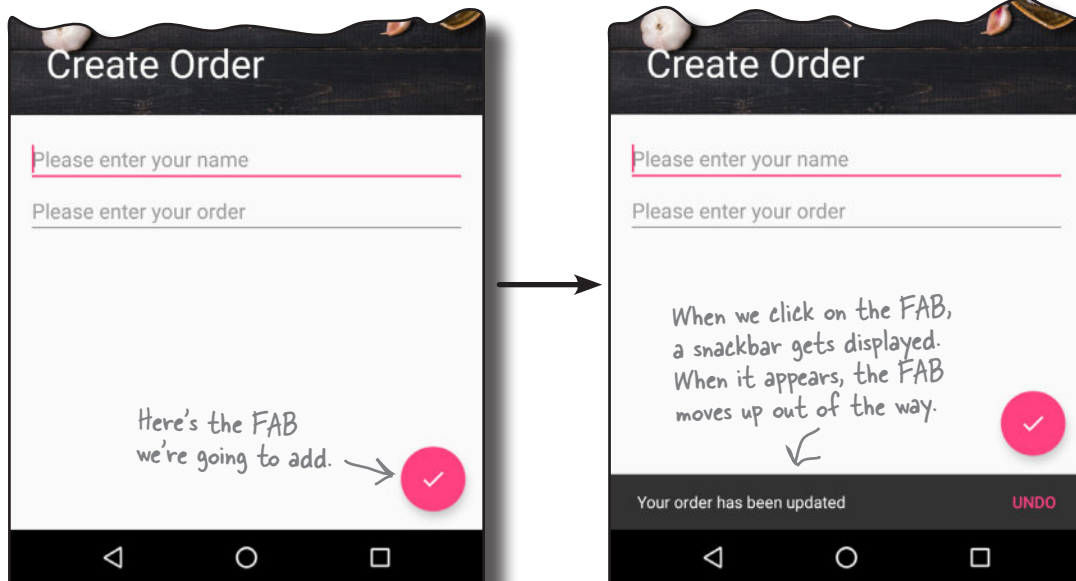
A **FAB** is a **floating action button**. It's a circled icon that floats above the user interface, for example in the bottom-right corner of the screen. It's used to promote actions that are so common or important that you want to make them obvious to the user.

A **snackbar** is like a toast except that you can interact with it. It's a short message that appears at the bottom of the screen that's used to give the user information about an operation. Unlike with a toast, you can add actions to a snackbar, such as an action to undo an operation.

We'll add a FAB and snackbar to `OrderActivity`

We're going to add a FAB to `OrderActivity`. When the user clicks on the FAB, we'll display a snackbar that shows a message to the user. In the real world, you'd want to use the FAB to perform an action such as saving the user's pizza order, but we're just going to focus on showing you how to add the widgets to your app.

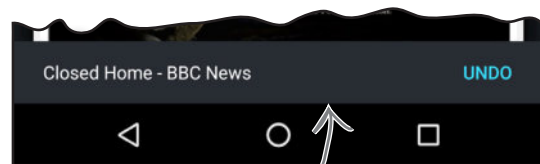
Here's what the new version of `OrderActivity` will look like:



Scrolling toolbar
Collapsing toolbar
FAB
Snackbar



This is a FAB that appears in the Google Calendar app. It floats in the bottom-right corner of the screen, and you use it to add events.



This is the snackbar that appears in the Chrome app when you've just closed a web page. You can reopen the page by clicking on the Undo action in the snackbar.

Add the icon for the FAB

We'll start by adding an icon to our project to display on the FAB. You can either create your own icon from scratch or use one of the icons provided by Google: <https://design.google.com/icons/>.

We're going to use the "done" icon `ic_done_white_24dp`, and we'll add a version of it to our project's `drawable*` folders, one for each screen density. Android will decide at runtime which version of the icon to use depending on the screen density of the device.

First, switch to the Project view of Android Studio's explorer, highlight the `app/src/main/res` folder in your project, then create folders called `drawable-hdpi`, `drawable-mdpi`, `drawable-xhdpi`, `drawable-xxhdpi`, and `drawable-xxxhdpi` if they don't already exist. Then go to <http://tinyurl.com/HeadFirstAndroidDoneIcons>, and download `ic_done_white_24dp.png` Bits and Pizzas images. Add the image in the `drawable-hdpi` folder to the `drawable-hdpi` folder in your project, then repeat this process for the other folders.

How to add a FAB to your layout

You add a FAB to your layout using code like this:

```
<android.support.design.widget.CoordinatorLayout ...>

...

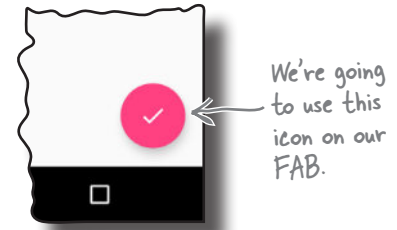
<android.support.design.widget.FloatingActionButton
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_gravity="end|bottom"
    android:layout_margin="16dp"
    android:src="@drawable/ic_done_white_24dp"
    android:onClick="onClickDone" />

</android.support.design.widget.CoordinatorLayout>
```

The above code adds a FAB to the bottom-end corner of the screen, with a margin of 16dp. It uses the `src` attribute to set the FAB's icon to the `ic_done_white_24dp` drawable. We're also using the FAB's `onClick` attribute to specify that the `onClickDone()` method in the layout's activity will get called when the user clicks on the FAB. We'll create this method later.

You usually use a FAB inside a `CoordinatorLayout`, as this means that you can coordinate movement between the different views in your layout. In our case, it means that the FAB will move up when the `Snackbar` appears.

On the next page we'll show you the code for `OrderActivity`'s layout.



The code for adding a FAB is similar to the code for adding an `ImageButton`. That's because `FloatingActionButton` is a subclass of `ImageButton`.

If you're using the FAB in an activity, you can use the `onClick` attribute to specify which method should be called when it's clicked.

The material design guidelines recommend using no more than one FAB per screen.

The updated code for activity_order.xml

Here's the updated code for *activity_order.xml* (update your version to match our changes in bold>):

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.design.widget.CoordinatorLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:id="@+id/coordinator"
    android:layout_width="match_parent"
    android:layout_height="match_parent" >

    <android.support.design.widget.AppBarLayout
        android:layout_width="match_parent"
        android:layout_height="300dp"
        android:theme="@style/ThemeOverlay.AppCompat.Dark.ActionBar" >

        <android.support.design.widget.CollapsingToolbarLayout
            android:layout_width="match_parent"
            android:layout_height="match_parent"
            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"
                android:contentDescription="@string/restaurant_image"
                app:layout_collapseMode="parallax" />

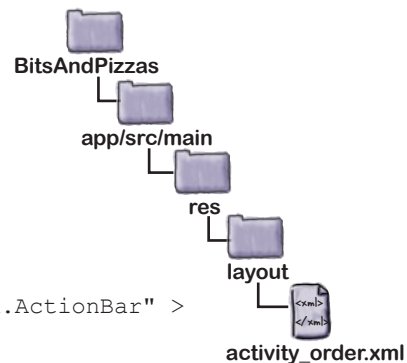
            <android.support.v7.widget.Toolbar
                android:id="@+id/toolbar"
                android:layout_width="match_parent"
                android:layout_height="?attr/actionBarSize"
                app:layout_collapseMode="pin" />

        </android.support.design.widget.CollapsingToolbarLayout>
    </android.support.design.widget.AppBarLayout>
```

We've not changed any of the code on this page.



Scrolling toolbar
Collapsing toolbar
FAB
Snackbar





The activity_order.xml code (continued)

```

<android.support.v4.widget.NestedScrollView
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    app:layout_behavior="@string/appbar_scrolling_view_behavior" >

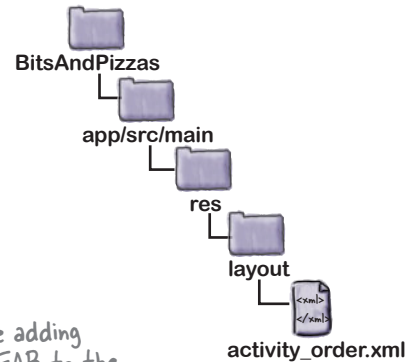
    <LinearLayout
        ...
    </LinearLayout>

</android.support.v4.widget.NestedScrollView>

<android.support.design.widget.FloatingActionButton
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_gravity="end|bottom"
    android:layout_margin="16dp"
    android:src="@drawable/ic_done_white_24dp"
    android:onClick="onClickDone" />

</android.support.design.widget.CoordinatorLayout>

```



We're adding the FAB to the CoordinatorLayout so that it will move out the way when a snackbar gets displayed.

Add the onClickDone() method to OrderActivity

Now that we've added a FAB to OrderActivity's layout, we need to write some activity code to make the FAB do something when it's clicked. You do this in the same way that you would for a button, by adding the method described by the FAB's onClick attribute to your activity code.

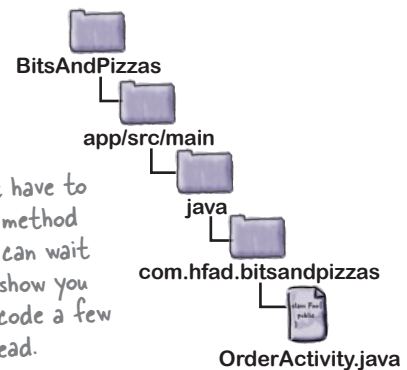
In our case, we've given the onClick attribute a value of "onClickDone", so this means we need to add an onClickDone() method to *OrderActivity.java*:

```

public void onClickDone(View view) {
    //Code that runs when the FAB is clicked
}

```

You don't have to add this method now; you can wait until we show you the full code a few pages ahead.



Now we're going to write some code to display a snackbar when the user clicks on the FAB.

How to create a snackbar

As we said earlier in the chapter, a snackbar is a bar that appears at the bottom of the screen that displays a short message to the user. It's similar to a toast, except that you can interact with it.

To create a snackbar, you call the `Snackbar.make()` method. This method takes three parameters: the `View` you want to hold the snackbar, the text you want to display, and an `int` duration. As an example, here's the code for a snackbar that appears on the screen for a short duration:

```
CharSequence text = "Hello, I'm a Snackbar!";
int duration = Snackbar.LENGTH_SHORT;
Snackbar snackbar = Snackbar.make(findViewById(R.id.coordinator), text, duration);
```

In the above code, we've used a view called `coordinator` to hold the snackbar. This view will usually be your activity's coordinator layout so that it can coordinate the snackbar with other views.

We've set the snackbar's duration to `LENGTH_SHORT`, which shows 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). With any of these options, the user is able to swipe away the snackbar so that it's no longer displayed.

You can add an action to the snackbar by calling its `setAction()` method. This can be useful if, for example, you want the user to be able to undo an operation they've just performed. The `setAction()` method takes two parameters: the text that should appear for the action, and a `View.OnClickListener()`. Any code you want to run when the user clicks on the action should appear in the listener's `onClick()` event:

```
snackbar.setAction("Undo", new View.OnClickListener() {
    @Override
    public void onClick(View view) {
        //Code to run when the user clicks on the Undo action
    }
});
```

Once you've finished creating the snackbar, you display it using its `show()` method:

```
snackbar.show();
```



Scrolling toolbar
Collapsing toolbar
FAB
Snackbar

If you want to display a String resource, you can pass in the resource ID instead of the text.

You pass the `setAction()` method the text that should appear for the action, and a `View.OnClickListener`.

You need to specify what should happen if the user clicks on the Undo action.

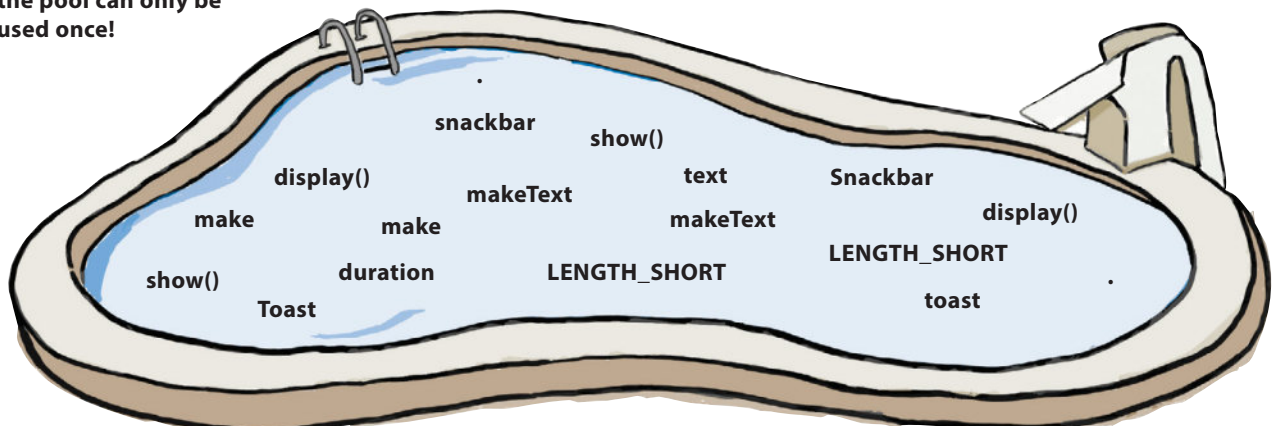
Pool Puzzle



Your **goal** is to make `OrderActivity`'s `onClickDone()` method display 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.

```
public void onClickDone(View view) {
    CharSequence text = "Your order has been updated";
    int duration = .....;
    Snackbar snackbar = Snackbar.....(findViewById(R.id.coordinator), .....);
    snackbar.setAction("Undo", new View.OnClickListener() {
        @Override
        public void onClick(View view) {
            Toast toast = Toast..... OrderActivity.this, "Undone!", .....);
            toast.....;
        }
    });
    snackbar.....;
}
```

Note: each thing from the pool can only be used once!



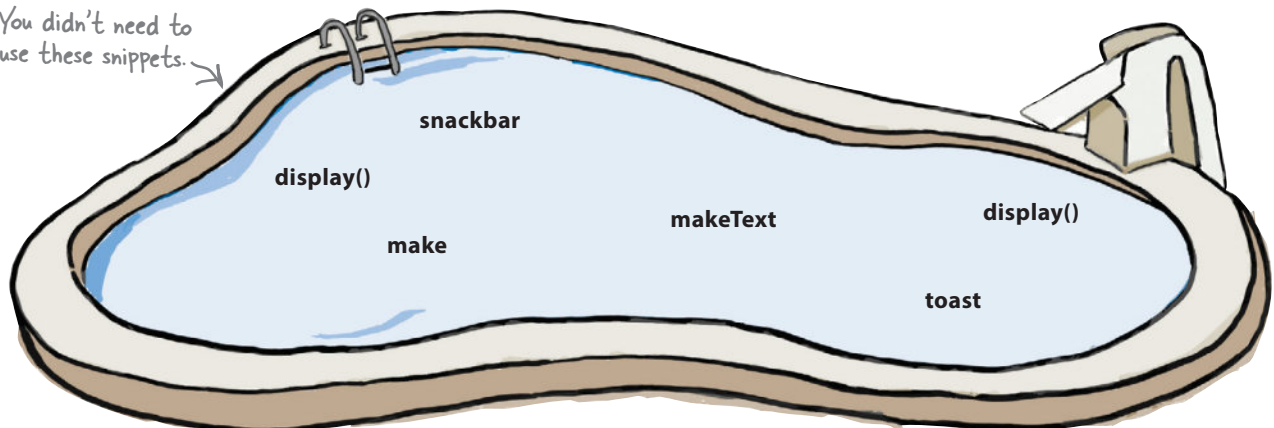
Pool Puzzle Solution



Your **goal** is to make `OrderActivity`'s `onClickDone()` method display 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.

```
public void onClickDone(View view) {
    CharSequence text = "Your order has been updated";
    int duration = Snackbar . LENGTH_SHORT ;
    Snackbar snackbar = Snackbar.make.....(findViewById(R.id.coordinator), text....., duration);
    snackbar.setAction("Undo", new View.OnClickListener() {
        @Override
        public void onClick(View view) {
            Toast toast = Toast.makeText.....OrderActivity.this, "Undone!", Toast . LENGTH_SHORT);
            toast.show().....;
        }
    });
    snackbar.show().....;
}
```

You didn't need to use these snippets.



The full code for OrderActivity.java

design support library
Scrolling toolbar
Collapsing toolbar
FAB
Snackbar



Here's our full code for *OrderActivity.java*, including the code to add a snackbar with an action. Update your version of the code to match our changes (in bold):

```
package com.hfad.bitsandpizzas;

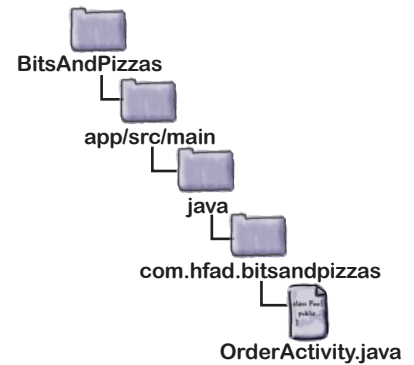
import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.support.v7.widget.Toolbar;
import android.support.v7.app.ActionBar;
import android.view.View;
import android.support.design.widget.Snackbar;
import android.widget.Toast;

public class OrderActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_order);
        Toolbar toolbar = (Toolbar) findViewById(R.id.toolbar);
        setSupportActionBar(toolbar);
        ActionBar actionBar = getSupportActionBar();
        actionBar.setDisplayHomeAsUpEnabled(true);
    }

    public void onClickDone(View view) {
        CharSequence text = "Your order has been updated";
        int duration = Snackbar.LENGTH_SHORT;
        Snackbar snackbar = Snackbar.make(findViewById(R.id.coordinator), text, duration);
        snackbar.setAction("Undo", new View.OnClickListener() {
            @Override
            public void onClick(View view) {
                Toast toast = Toast.makeText(OrderActivity.this, "Undone!", Toast.LENGTH_SHORT);
                toast.show();
            }
        });
        snackbar.show();
    }
}
```

We're using these new classes, so you need to import them.



Create the snackbar.

This method gets called when the user clicks on the FAB.

Add an action to the snackbar.

If the user clicks on the snackbar's action, display a toast.

Display the snackbar.

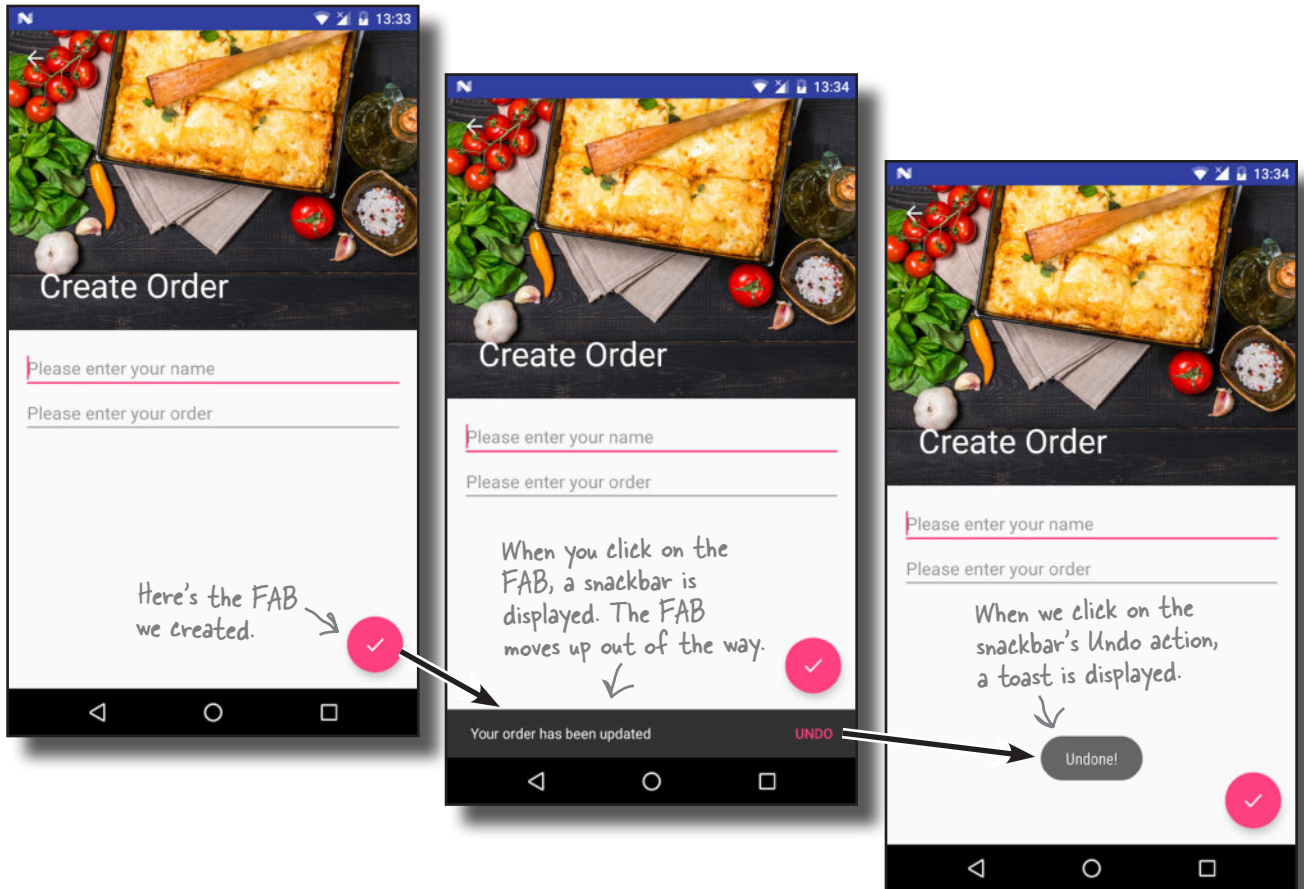


Test drive the app

When we run the app, a FAB is displayed in OrderActivity. When we click on the FAB, a snackbar is displayed and the FAB moves up to accommodate it. When we click on the Undo action on the snackbar, a toast is displayed.



- Scrolling toolbar
- Collapsing toolbar
- FAB
- Snackbar



As you can see, snackbars have a lot in common with toasts, as they're both used to display messages to the user. But if you want the user to be able to *interact* with the information you're showing them, choose a snackbar.



Your Android Toolbox

You've got **Chapter 12** under your belt and now you've added the **Design Support Library** to your toolbox.

You can download the full code for the chapter from <https://tinyurl.com/HeadFirstAndroid>.



BULLET POINTS

- Enable swipe navigation using a **view pager**.
- You tell a view pager about its pages by implementing a **fragment pager adapter**.
- Use the fragment pager adapter's **getCount ()** method to tell the view pager how many pages it should have. Use its **getItem ()** method to tell it which fragment should appear on each page.
- Add tab navigation by implementing a **tab layout**. Put the toolbar and tab layout inside an **app bar layout** in your layout code, then attach the tab layout to the view pager in your activity code.
- The tab layout comes from the **Android Design Support Library**. This library helps you implement the **material design guidelines** in your app.
- Use a **coordinator layout** to coordinate animations between views.
- Add scrollable content the coordinator layout can coordinate using a **nested scroll view**.
- Use a **collapsing toolbar layout** to add a toolbar that collapses and grows in response to user scroll actions.
- Use a **FAB** (floating action button) to promote common or important user actions.
- A **snackbar** lets you display short messages that the user can interact with.

13 recycler views and card views

Get Recycling



You've already seen how the humble list view is a key part of **most apps**. But compared to some of the *material design* components we've seen, it's somewhat plain. In this chapter, we'll introduce you to the **recycler view**, a more advanced type of list that gives you *loads more flexibility* and *fits in with the material design ethos*. You'll see how to create **adapters** tailored to your data, and how to completely change the look of your list with *just two lines of code*. We'll also show you how to use **card views** to give your data a *3D material design* appearance.

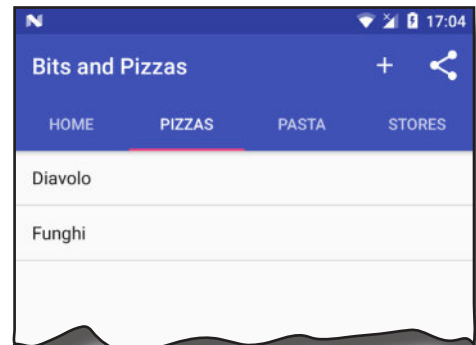
There's still work to do on the Bits and Pizzas app

In Chapter 12, we updated the Bits and Pizzas app to include components from the Design Support Library, including a tab layout, FAB, and collapsing toolbar. We added these to help users navigate to places in the app more easily, and to implement a consistent material design look and feel. If you recall, material design is inspired by paper and ink, and uses print-based design principles and movement to reflect how real-world objects (such as index cards and pieces of paper) look and behave. But there's one key area we didn't look at: lists.

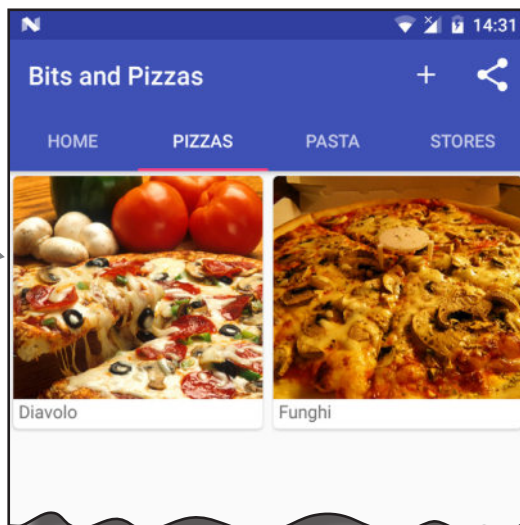
We're currently using list views in `PizzaFragment`, `PastaFragment`, and `StoresFragment` to display the available pizzas, pasta, and stores. These lists are very plain compared with the rest of the app, and could do with some work to give them the same look and feel.

Another disadvantage of list views is that they don't implement nested scrolling. In Chapter 12, we made `MainActivity`'s toolbar scroll in response to the user scrolling content in the activity's fragments. This currently works for `TopFragment`, as it uses a nested scroll view. As none of the other fragments use nested scrolling, however, the toolbar remains fixed when the user tries to scroll their content.

To address these issues, we're going to change `PizzaFragment` to use a **recycler view**. This is a more advanced and flexible version of a list view that implements nested scrolling. Instead of displaying just the names of each pizza in a list view, we'll use a recycler view to display its name and image:



↑
This is the current `PizzaFragment`. It includes a list of pizzas, but it looks quite plain.



→ This is the recycler view we're going to add to `PizzaFragment`.

← When you scroll the recycler view, the toolbar moves up. This matches the behavior of `TopFragment`.

Recycler views from 10,000 feet

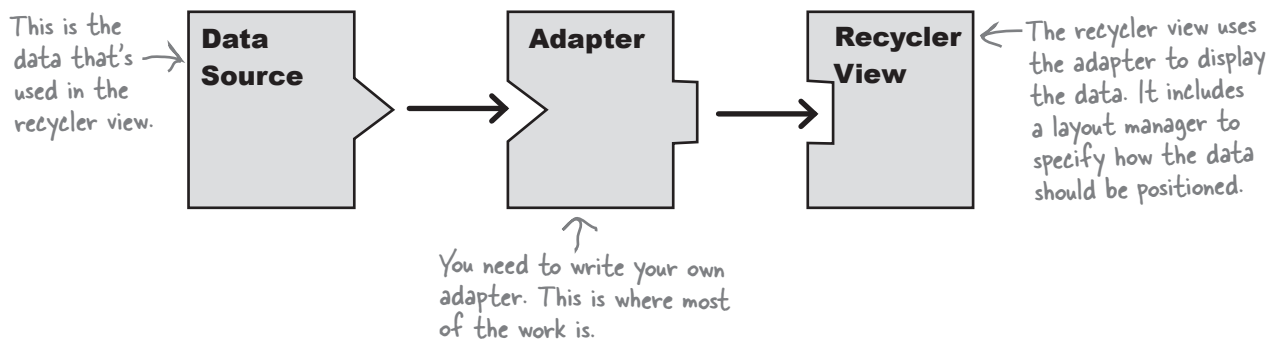
Before we dive into the code, let's take a look at how recycler views work. As a recycler view is more flexible than a list view, it takes a lot more setting up.

Like list views, recycler views efficiently manage a small number of views to give the appearance of a large collection of views that extend beyond the screen. They allow you greater flexibility about how the data is displayed than list views do.

A recycler view accesses its data using an **adapter**. Unlike a list view, however, it doesn't use any of the built-in Android adapters such as array adapters. Instead, ***you have to write an adapter of your own that's tailored to your data.*** This includes specifying the type of data, creating views, and binding the data to the views.

Items are positioned in a recycler view using a **layout manager**. There are a number of built-in layout managers you can use that allow you to position items in a linear list or grid.

Here's a diagram of all those elements put together:



In our particular case, we're going to create a recycler view to display pizza names and images. We'll go through the steps for how to do this on the next page.

Here's what we're going to do

There are five main steps we'll go through to get the recycler view working:

1 Add the pizza data to the project.
We'll add images of the pizzas to the app, along with a new `Pizza` class. This class will be the recycler view's data source.

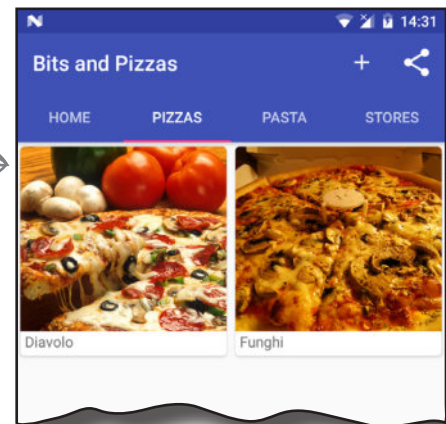
2 Create a card view for the pizza data.
We're going to make each pizza in the recycler view look as though it's displayed on a separate card. To do this, we'll use a new type of view called a *card view*.

3 Create a recycler view adapter.
As we said on the previous page, when you use a recycler view you need to write your own adapter for it. Our adapter needs to take the pizza data and bind each item to a card view. Each card will then be able to be displayed in the recycler view.

4 Add a recycler view to `PizzaFragment`.
After we've created the adapter, we'll add the recycler view to `PizzaFragment`. We'll make it use the adapter, and use a layout manager to display pizza data in a two-column grid.

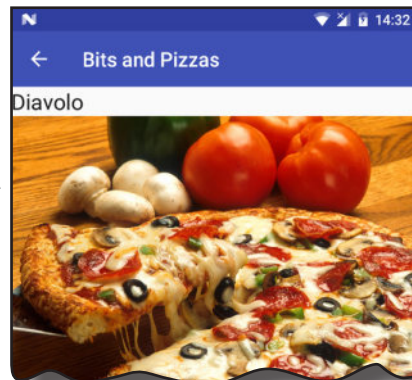
5 Make the recycler view respond to clicks.
We'll create a new activity, `PizzaDetailActivity`, and get it to start when the user clicks on one of the pizzas. We'll display details of the pizza in the activity.

These are card views displaying pizza data.



This is the recycler view.

This is `PizzaDetailActivity`.



Do this!

We're going to update the Bits and Pizzas app in this chapter, so open your Bits and Pizzas project in Android Studio.

The first thing we'll do is add the pizza data.

Add the pizza data

We'll start by adding the pizza images to the Bits and Pizzas project. Download the files *diavolo.jpg* and *funghi.jpg* from <https://git.io/v9oet>, then add them to the folder *app/src/main/res/drawable-nodpi*. This folder should already exist in your project, as we added an image to it in Chapter 12.

Add the Pizza class

We'll get our data from a *Pizza* class, which we need to add. The class defines an array of two pizzas, where each pizza has a name and image resource ID. Switch to the Project view of Android Studio's explorer, highlight the *com.hfad.bitsandpizzas* package in the *app/src/main/java* folder, then go to File→New...→Java class. When prompted, name the class "Pizza" and make sure the package name is *com.hfad.bitsandpizzas*. Finally, replace the code in *Pizza.java* with the following:

```
package com.hfad.bitsandpizzas;
```

```
public class Pizza {  
    private String name;  
    private int imageResourceId;
```

Each Pizza has a name and image resource ID. The image resource ID refers to the pizza images we added to the project above.

```
    public static final Pizza[] pizzas = {  
        new Pizza("Diavolo", R.drawable.diavolo),  
        new Pizza("Funghi", R.drawable.funghi)  
    };
```

```
    private Pizza(String name, int imageResourceId) {  
        this.name = name;  
        this.imageResourceId = imageResourceId;  
    }
```

```
    public String getName() {  
        return name;  
    }
```

These are getters for the private variables.

```
    public int getImageResourceId() {  
        return imageResourceId;  
    }
```

```
}
```

recycler views and card views

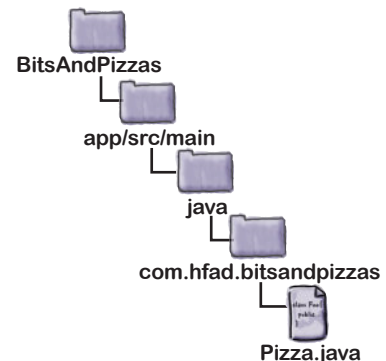


Pizza data
Card view
Adapter
RecyclerView
view
Clicks

These are the pizza images.



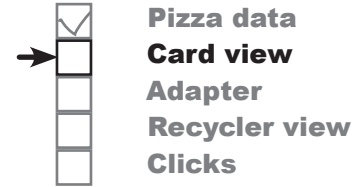
In a real app, we might use a database for this. We're using a Java class here for simplicity.



Display the pizza data in a card

The next thing we need to do is define a layout for the pizza data. This layout will be used by the recycler view's adapter to determine how each item in the recycler view should look. We're going to use a **card view** for this layout.

A card view is a type of frame layout that lets you display information on virtual cards. Card views can have rounded corners and shadows to make it look as though they're positioned above their background. If we use a card view for our pizza data, each pizza will look as though it's displayed on a separate card in the recycler view.



← These are card views. We'll use cards to display the pizza data in the recycler view. →

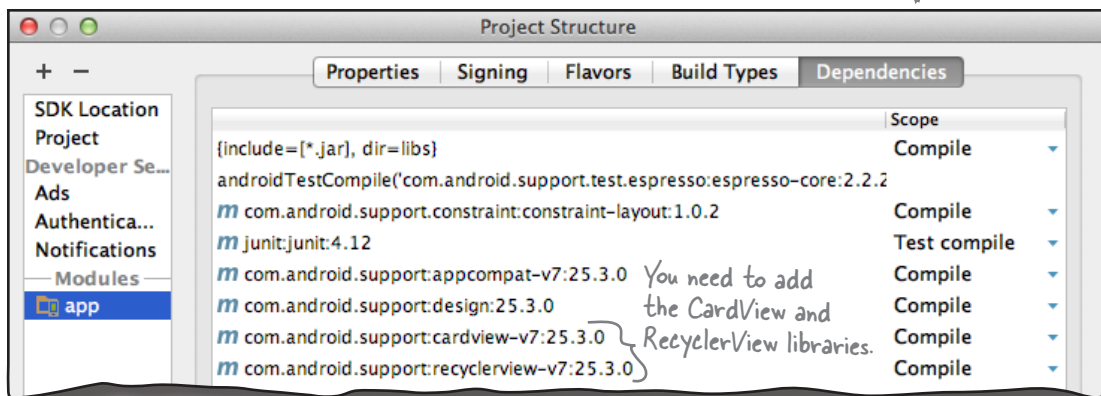


Add the CardView and RecyclerView Support Libraries

Card views and recycler views come from the CardView and RecyclerView v7 Support Libraries, respectively, so before we can go any further, you need to add them to your project as dependencies.

In Android Studio go to File→Project Structure. In the Project Structure window, click on the “app” option and switch to the Dependencies tab. Then click on the “+” button at the bottom or right side of the screen, choose the “Library dependency” option, and add the CardView Library. Repeat these steps to add the RecyclerView-v7 Library, then click on the OK button to save your changes.

Make sure you add both libraries.



Now that you've added the Support Libraries, we'll create a card view that we can use for our pizza data.

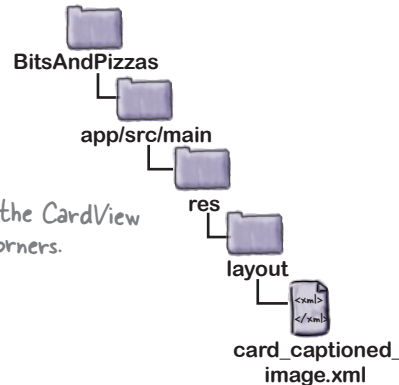
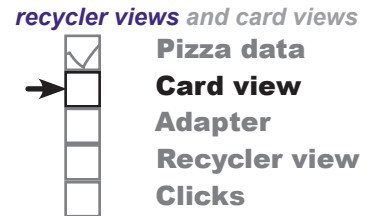
How to create a card view

We're going to create a card view that displays an image with a caption. We'll use it here for the name and image of individual pizzas, but you could also use the same layout for different categories of data such as pasta or stores.

You create a card view by adding a `<CardView>` element to a layout. If you want to use the card view in a recycler view (as we do here), you need to create a new layout file for the card view. Do this by highlighting the `app/src/main/res/layout` folder, and choosing File→New→Layout resource file. When prompted, name the layout "card_captioned_image".

You add a card view to your layout using code like this:

```
<android.support.v7.widget.CardView ← This defines the CardView.
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:card_view="http://schemas.android.com/apk/res-auto"
    android:id="@+id/card_view"
    android:layout_width="match_parent"
    android:layout_height="200dp"
    android:layout_margin="4dp"
    card_view:cardElevation="2dp" ← Setting the card's elevation gives it a drop shadow.
    card_view:cardCornerRadius="4dp" > ← This gives the CardView rounded corners.
    ... ← You add any views you want to be displayed to the CardView.
</android.support.v7.widget.CardView>
```



In the above code, we've added an extra namespace of:

```
xmlns:card_view="http://schemas.android.com/apk/res-auto"
```

so that we can give the card rounded corners and a drop shadow to make it look higher than its background. You add rounded corners using the `card_view:cardCornerRadius` attribute, and the `card_view:cardElevation` attribute sets its elevation and adds drop shadows.

Once you've defined the card view, you need to add any views you want to display to it. In our case, we'll add a text view and image view to display the name and image of the pizza. We'll show you the full code for this on the next page.

The full card_captioned_image.xml code

Here's the full code for `card_captioned_image.xml` (update your version of the file to match ours):

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.v7.widget.CardView
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:card_view="http://schemas.android.com/apk/res-auto"
    android:id="@+id/card_view"
    android:layout_width="match_parent"
    android:layout_height="200dp"
    android:layout_margin="5dp"
    card_view:cardElevation="2dp"
    card_view:cardCornerRadius="4dp">
```

The card view will be as wide as its parent allows, and 200dp high.

```
<LinearLayout
```

```
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical">
```

We've put the `ImageView` and `TextView` in a `LinearLayout`, as the `CardView` can only have one direct child.

```
    <ImageView android:id="@+id/info_image"
        android:layout_height="0dp"
        android:layout_width="match_parent"
        android:layout_weight="1.0"
        android:scaleType="centerCrop"/>
```

The image will be as wide as the `CardView` allows. We're using `centerCrop` to make sure the image scales uniformly.

```
    <TextView
        android:id="@+id/info_text"
        android:layout_marginLeft="4dp"
        android:layout_marginBottom="4dp"
        android:layout_height="wrap_content"
        android:layout_width="match_parent"/>
```

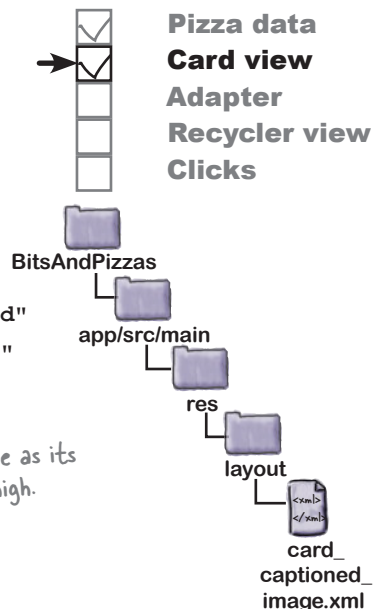
The `CardView` contains an `ImageView` and a `TextView`.



This is what the `CardView` will look like when data's been added to it. We'll do this via the `recycler view's adapter`.

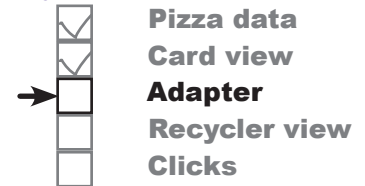
```
</LinearLayout>
```

```
</android.support.v7.widget.CardView>
```



Note that the above layout doesn't explicitly mention pizza data. This means we can use the same layout for any data items that consist of a caption and an image, such as pasta.

Now that we've created a layout for the card views, we'll move on to creating the `recycler view's adapter`.



How our recycler view adapter will work

As we said earlier, when you use a recycler view in your app, you need to create a recycler view adapter. That's because unlike a list view, recycler views don't use any of the built-in adapters that come with Android. While writing your own adapter may seem like hard work, on the plus side it gives you more flexibility than using a built-in one.

The adapter has two main jobs: to create each of the views that are visible within the recycler view, and to bind each view to a piece of data. In our case, the recycler view needs to display a set of cards, each containing a pizza image and caption. This means that the adapter needs to create each card and bind data to it.

We'll create the recycler view adapter over the next few pages. Here are the steps we'll go through to create it:

- 1 Specify what data the adapter should work with.**
We want the adapter to work with the pizza data. Each pizza has a name and image resource ID, so we'll pass the adapter an array of pizza names, and an array of image resource IDs.
- 2 Define the views the adapter should populate.**
We want to use the data to populate a set of pizza cards defined by *card_captioned_image.xml*. We then need to create a set of these cards that will be displayed in the recycler view, one card per pizza.
- 3 Bind the data to the cards.**
Finally, we need to display the pizza data in the cards. To make that happen, we need to populate the `info_text` text view with the name of the pizza, and the `info_image` image view with the pizza's image.

We'll start by adding a `RecyclerView.Adapter` class to our project.

there are no Dumb Questions

Q: Why doesn't Android provide ready-made adapters for recycler views?

A: Because recycler view adapters don't just specify the data that will appear. They also specify the views that will be used for each item in the collection. That means that recycler view adapters are both more powerful, and less general, than list view adapters.

Add a recycler view adapter

You create a recycler view adapter by extending the `RecyclerView.Adapter` class and overriding various methods; we'll cover these over the next few pages. You also need to define a `ViewHolder` as an inner class, which tells the adapter which views to use for the data items.

We're going to create a recycler view adapter called `CaptionedImagesAdapter`. In Android Studio, highlight the `com.hfad.bitsandpizzas` package in the `app/src/main/java` folder, then go to `File→New...→Java class`. When prompted, name the class "CaptionedImagesAdapter" and make sure the package name is `com.hfad.bitsandpizzas`. Then replace the code in `CaptionedImagesAdapter.java` with the following:

```
package com.hfad.bitsandpizzas;
```

```
import android.support.v7.widget.RecyclerView; ← We're extending the
                                                    RecyclerView class, so
                                                    we need to import it.
```

```
class CaptionedImagesAdapter extends
```

```
    RecyclerView.Adapter<CaptionedImagesAdapter.ViewHolder>{
```

```
    public static class ViewHolder extends RecyclerView.ViewHolder {
```

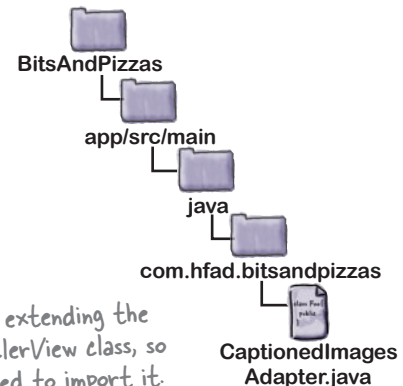
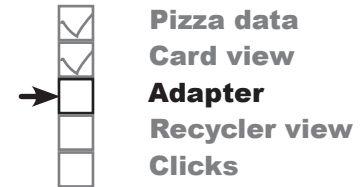
```
        //Define the view to be used for each data item
```

```
    }
```

```
}
```

As you can see, the `ViewHolder` inner class you define is a key part of the adapter. We've left the `ViewHolder` class empty for now, but we'll come back to it later in the chapter.

Before we look in more detail at view holders, we'll tell the adapter what sort of data it should use by adding a constructor.



The ViewHolder is used to specify which views should be used for each data item.

You define the ViewHolder as an inner class. We'll complete this later in the chapter.



Don't worry if Android Studio gives you error messages when you add the above code to your project.

It's just warning you that the code isn't complete yet. We still need to override various methods in our adapter code to tell it how to behave, and we'll do this over the next few pages.

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 it should use. You do this by defining a constructor that includes the data types you want the adapter to use as parameters.

In our case, we want the adapter to take `String` captions and `int` image IDs. We'll therefore add `String []` and `int []` parameters to the constructor, and save the arrays as private variables. Here's the code that does this; you can either update your version of `CaptionedImagesAdapter.java` now, or wait until we show you the full adapter code later in the chapter.

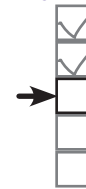
```
class CaptionedImagesAdapter extends
    RecyclerView.Adapter<CaptionedImagesAdapter.ViewHolder>{

    private String[] captions;
    private int[] imageIds;
    ...
    public CaptionedImagesAdapter(String[] captions, int[] imageIds){
        this.captions = captions;
        this.imageIds = imageIds;
    }
}
```

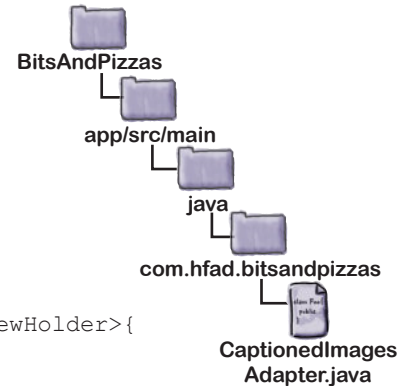
We'll use these variables to hold the pizza data.

We'll pass the data to the adapter using its constructor.

recycler views and card views



Pizza data
Card view
Adapter
RecyclerView
Clicks



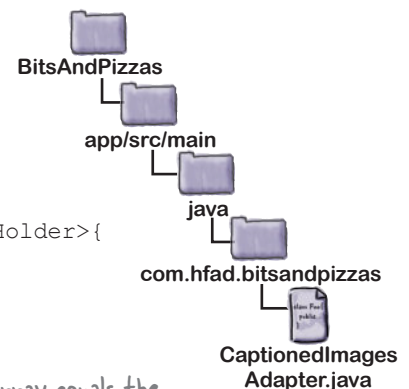
...and implement the getItemCount() method

We also need to tell the adapter how many data items there are. You do this by overriding the `RecyclerView.Adapter` `getItemCount()` method. This returns an `int` value, the number of data items. We can derive this from the number of captions we pass the adapter. Here's the code:

```
class CaptionedImagesAdapter extends
    RecyclerView.Adapter<CaptionedImagesAdapter.ViewHolder>{

    ...
    @Override
    public int getItemCount(){
        return captions.length;
    }
}
```

The length of the captions array equals the number of data items in the recycler view.



Next we'll define the adapter's view holder.

Define the adapter's view holder

The view holder is used to define what view or views the recycler view should use for each data item it's given. You can think of it as a holder for the views you want the recycler view to display. In addition to views, the view holder contains extra information that's useful to the recycler view, such as its position in the layout.

In our case, we want to display each item of pizza data on a card, which means we need to specify that the adapter's view holder uses a card view. Here's the code to do this (we'll show you the full adapter code later in the chapter):

```
...
import android.support.v7.widget.CardView;

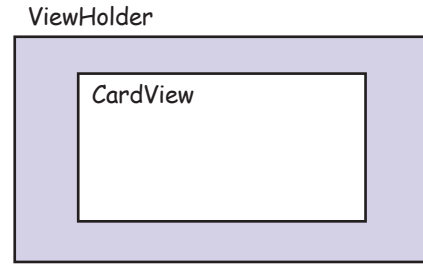
class CaptionedImagesAdapter extends
    RecyclerView.Adapter<CaptionedImagesAdapter.ViewHolder>{
...
    public static class ViewHolder extends RecyclerView.ViewHolder {

        private CardView cardView;

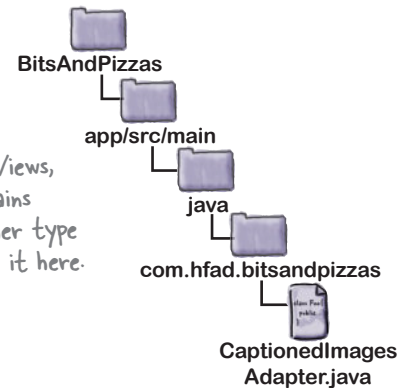
        public ViewHolder(CardView v) {
            super(v);
            cardView = v;
        }
    }
}
```

We're using the CardView class, so we need to import it.

Our recycler view needs to display CardViews, so we specify that our ViewHolder contains CardViews. If you want to display another type of data in the recycler view, you define it here.



Each of our ViewHolders will contain a CardView. We created the layout for this CardView earlier in the chapter.



When you create a view holder, you must call the ViewHolder super constructor using:

```
super(v);
```

This is because the ViewHolder superclass includes metadata such as the item's position in the recycler view, and you need this information for the adapter to work properly.

Now that we've defined our view holders, we need to tell the adapter how to construct one. We'll do this by overriding the adapter's onCreateViewHolder() method.

Override the onCreateViewHolder() method

The `onCreateViewHolder()` method gets called when the recycler view requires a new view holder. The recycler view calls the method repeatedly when the recycler view is first constructed to build the set of view holders that will be displayed on the screen.

The method takes two parameters: a `ViewGroup` parent object (the recycler view itself) and an `int` parameter called `viewType`, which is used if you want to display different kinds of views for different items in the list. It returns a view holder object. Here's what the method looks like:

```
@Override
public CaptionedImagesAdapter.ViewHolder onCreateViewHolder (
    ViewGroup parent, int viewType) {
    //Code to instantiate the ViewHolder
}
```

We need to add code to the method to instantiate the view holder. To do this, we need to call the `ViewHolder`'s constructor, which we defined on the previous page. The constructor takes one parameter, a `CardView`. We'll create the `CardView` from the `card_captioned_image.xml` layout we created earlier in the chapter using this code:

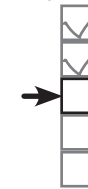
```
CardView cv = (CardView) LayoutInflater.from(parent.getContext())
    .inflate(R.layout.card_captioned_image, parent, false);
```

Here's the full code for the `onCreateViewHolder()` method (we'll add this to the adapter later):

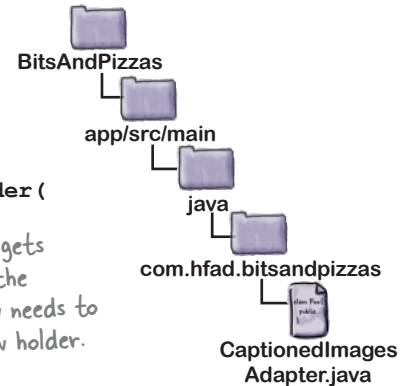
```
@Override
public CaptionedImagesAdapter.ViewHolder onCreateViewHolder (
    ViewGroup parent, int viewType) {
    CardView cv = (CardView) LayoutInflater.from(parent.getContext())
        .inflate(R.layout.card_captioned_image, parent, false);
    return new ViewHolder(cv);
}
```

Now that the adapter can create view holders, we need to get it to populate the card views they contain with data.

recycler views and card views



Pizza data
Card view
Adapter
Recycler view
Clicks



This method gets called when the recycler view needs to create a view holder.

Get a `LayoutInflater` object.

Use the `LayoutInflater` to turn the layout into a `CardView`. This is nearly identical to code you've already seen in the `onCreateView()` of fragments.

Specify what layout to use for the contents of the `ViewHolder`.

Add the data to the card views

You add data to the card views by implementing the adapter's `onBindViewHolder()` method. This gets called whenever the recycler view needs to display data in a view holder. It takes two parameters: the view holder the data needs to be bound to, and the position in the data set of the data that needs to be bound.

Our card view contains two views, an image view with an ID of `info_image`, and a text view with an ID of `info_text`. We'll populate these with data from the captions and imageIds arrays. Here's the code that will do that:

```
...
import android.widget.ImageView;
import android.widget.TextView;
import android.graphics.drawable.Drawable;
import android.support.v4.content.ContextCompat;
```

We're using these extra classes, so we need to import them.

```
class CaptionedImagesAdapter extends
    RecyclerView.Adapter<CaptionedImagesAdapter.ViewHolder>{
```

```
private String[] captions;
private int[] imageIds;
```

We added these variables earlier. They contain the captions and image resource IDs of the pizzas.

...

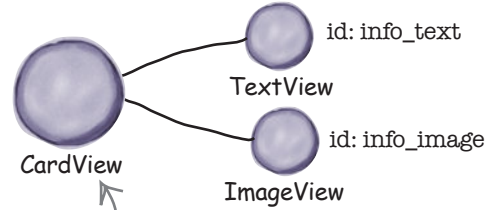
`@Override`

```
public void onBindViewHolder(ViewHolder holder, int position){
    CardView cardView = holder.cardView;
    ImageView imageView = (ImageView) cardView.findViewById(R.id.info_image);
    Drawable drawable =
        ContextCompat.getDrawable(cardView.getContext(), imageIds[position]);
    imageView.setImageDrawable(drawable);
    imageView.setContentDescription(captions[position]);
    TextView textView = (TextView) cardView.findViewById(R.id.info_text);
    textView.setText(captions[position]);
}
}
```

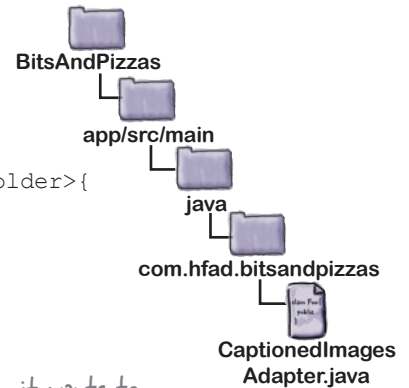
The recycler view calls this method when it wants to use (or reuse) a view holder for a new piece of data.

Display the caption in the TextView.

Pizza data
Card view
Adapter
RecyclerView
Clicks



Each CardView contains a TextView and ImageView. We need to populate these with the caption and image of each pizza.



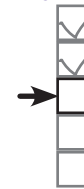
Display the image in the ImageView.

That's all the code we need for our adapter. We'll show you the full code over the next couple of pages.

The full code for CaptionedImagesAdapter.java

Here's our complete code for the adapter. Update your version of *CaptionedImagesAdapter.java* so that it matches ours.

recycler views and card views

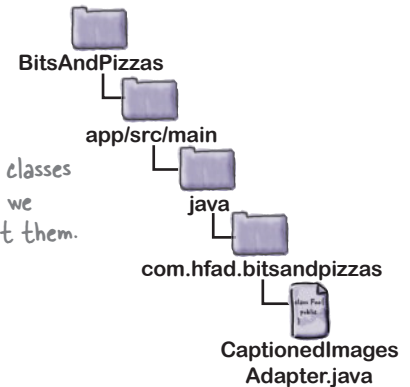


Pizza data
Card view
Adapter
RecyclerView
Clicks

```
package com.hfad.bitsandpizzas;

import android.support.v7.widget.RecyclerView;
import android.support.v7.widget.CardView;
import android.view.ViewGroup;
import android.view.LayoutInflater;
import android.widget.ImageView;
import android.widget.TextView;
import android.graphics.drawable.Drawable;
import android.support.v4.content.ContextCompat;
```

These are the classes we're using, so we need to import them.



```
class CaptionedImagesAdapter extends
    RecyclerView.Adapter<CaptionedImagesAdapter.ViewHolder>{
```

```
private String[] captions;
private int[] imageIds;
```

We're using these variables for the captions and image resource IDs.

```
public static class ViewHolder extends RecyclerView.ViewHolder {
```

```
private CardView cardView;
```

```
public ViewHolder(CardView v) {
    super(v);
    cardView = v;
}
```

Each ViewHolder will display a CardView.

```
public CaptionedImagesAdapter(String[] captions, int[] imageIds){
    this.captions = captions;
    this.imageIds = imageIds;
}
```

Pass data to the adapter in its constructor.

The code continues on the next page. →

The full CaptionedImagesAdapter.java code (continued)

```

@Override
public int getItemCount(){ ← The number of data items
    return captions.length;
}

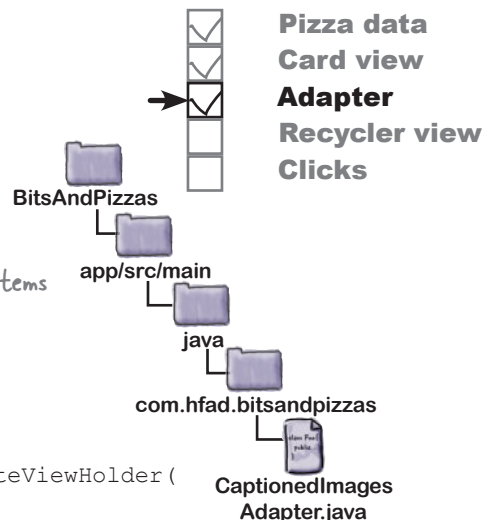
@Override
public CaptionedImagesAdapter.ViewHolder onCreateViewHolder(
    ViewGroup parent, int viewType){
    CardView cv = (CardView) LayoutInflater.from(parent.getContext())
        .inflate(R.layout.card_captioned_image, parent, false);
    return new ViewHolder(cv);
}

@Override
public void onBindViewHolder(ViewHolder holder, int position){
    CardView cardView = holder.cardView;
    ImageView imageView = (ImageView) cardView.findViewById(R.id.info_image);
    Drawable drawable =
        ContextCompat.getDrawable(cardView.getContext(), imageIds[position]);
    imageView.setImageDrawable(drawable);
    imageView.setContentDescription(captions[position]);
    TextView textView = (TextView) cardView.findViewById(R.id.info_text);
    textView.setText(captions[position]); ← Populate the CardView's ImageView
}
}

```

Use the layout we created earlier for the CardViews.

Populate the CardView's ImageView and TextView with data.

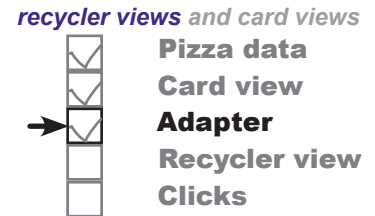


That's all the code we need for our adapter. So what's next?

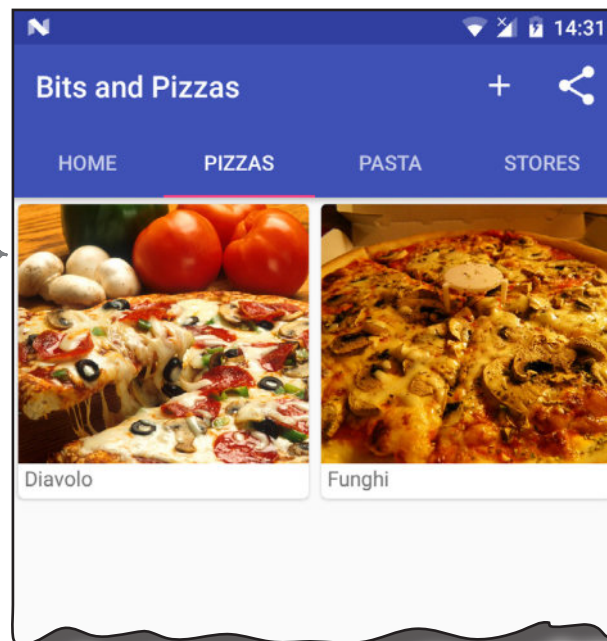
Create the recycler view

So far we've created a card view layout that displays captioned images, and an adapter that creates the cards and populates them with data. The next thing we need to do is create the recycler view, which will pass pizza data to the adapter so that it can populate the cards with the pizza images and captions. The recycler view will then display the cards.

We're going to add the recycler view to our existing `PizzaFragment`. Whenever the user clicks on the Pizzas tab in `MainActivity`, the pizzas will be displayed:



This is what the recycler view in `PizzaFragment` will look like. It will display the pizza cards in a two-column grid.



Add a layout for `PizzaFragment`

Before we can add the recycler view, we need to add a new layout file to our project for `PizzaFragment` to use. This is because we initially created `PizzaFragment` as a `ListFragment`, and these define their own layout.

To add the layout file, highlight the `app/src/main/res/layout` folder in Android Studio, and choose `File`→`New`→`Layout resource file`. When prompted, name the layout “`fragment_pizza`”.

Add the RecyclerView to PizzaFragment's layout

You add a recycler view to a layout using the `<RecyclerView>` element from the RecyclerView Support Library.

Our `PizzaFragment` layout only needs to display a single recycler view, so here's the full code for `fragment_pizza.xml` (update your version of the code to match ours):

```
<android.support.v7.widget.RecyclerView ← This defines the RecyclerView.
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/pizza_recycler" ← We've given the recycler view an ID so
    android:layout_width="match_parent"      that we can refer to it in our Java code.
    android:layout_height="match_parent"
    android:scrollbars="vertical" /> ← This adds a vertical scrollbar.
```

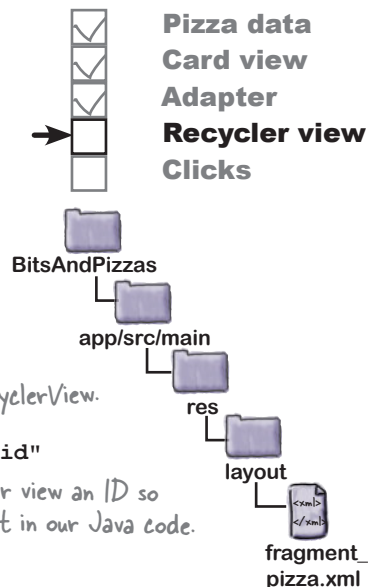
You add scrollbars to the recycler view using the `android:scrollbars` attribute. We've set this to "vertical" because we want our recycler view to be able to scroll vertically. We've also given the recycler view an ID so that we can get a reference to it in our `PizzaFragment` code; we need this in order to control its behavior.

Now that we've added a recycler view to `PizzaFragment`'s layout, we need to update our fragment code to get the recycler view to use the adapter we created.

Get the recycler view to use the adapter

To get the recycler view to use the adapter, there are two things we need to do: tell the adapter what data to use, then attach the adapter to the recycler view. We can tell the adapter what data to use by passing it the pizza names and image resource IDs via its constructor. We'll then use the `RecyclerView.setAdapter()` method to assign the adapter to the recycler view.

The code to do this is all code that you've seen before, so we'll show you the full `PizzaFragment` code on the next page.



The full PizzaFragment.java code

Here's our full code for *PizzaFragment.java* (update your version of the code to match our changes):

```

package com.hfad.bitsandpizzas;

import android.os.Bundle;
import android.support.v4.app.Fragment;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;
import android.widget.AdapterView;
import android.support.v7.widget.RecyclerView;

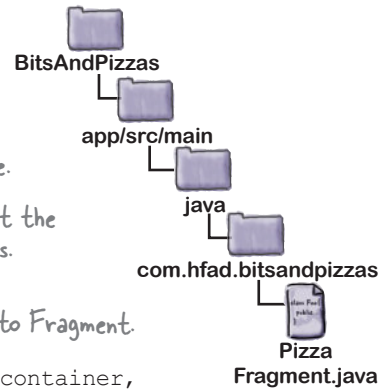
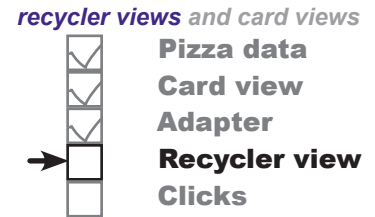
public class PizzaFragment extends ListFragment {
    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup container,
        Bundle savedInstanceState) {
        
            ArrayAdapter<String> adapter = new ArrayAdapter<>(
                inflater.getContext(),
                android.R.layout.simple_list_item_1,
                getResources().getStringArray(R.array.pizzas));
            setListAdapter(adapter);
            return super.onCreateView(inflater, container, savedInstanceState);
        
        RecyclerView pizzaRecycler = (RecyclerView)inflater.inflate(
            R.layout.fragment_pizza, container, false);

        String[] pizzaNames = new String[Pizza.pizzas.length];
        for (int i = 0; i < pizzaNames.length; i++) {
            pizzaNames[i] = Pizza.pizzas[i].getName();
        }

        int[] pizzaImages = new int[Pizza.pizzas.length];
        for (int i = 0; i < pizzaImages.length; i++) {
            pizzaImages[i] = Pizza.pizzas[i].getImageResourceId();
        }

        CaptionedImagesAdapter adapter = new CaptionedImagesAdapter(pizzaNames, pizzaImages);
        pizzaRecycler.setAdapter(adapter);
        return pizzaRecycler;
    }
}

```



We're changing PizzaFragment to be a Fragment, not a ListFragment.

We're no longer using an ArrayAdapter, so delete this line.

We need to import the RecyclerView class.

Change this from ListFragment to Fragment.

Delete these lines, as they're no longer necessary.

Use the layout we updated on the previous page.

Add the pizza names to an array of Strings, and the pizza images to an array of ints.

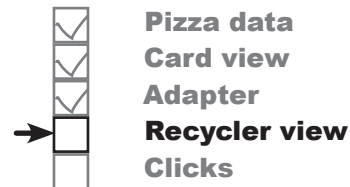
Pass the arrays to the adapter.

There's just one more thing we need to do: specify how the views in the recycler view should be arranged.

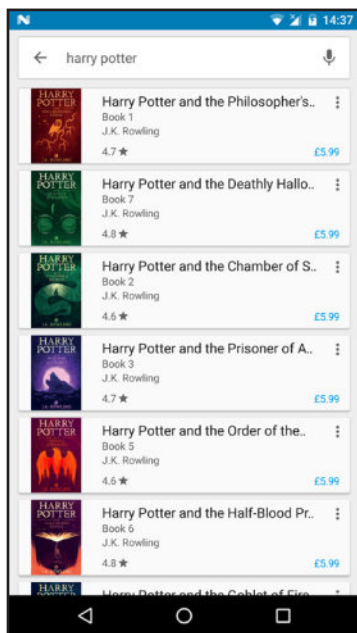
A recycler view uses a layout manager to arrange its views

One of the ways in which a recycler view is more flexible than a list view is in how it arranges its views. A list view displays its views in a single vertical list, but a recycler view gives you more options. You can choose to display views in a linear list, a grid, or a staggered grid.

You specify how to arrange the views using a **layout manager**. The layout manager positions views inside a recycler view: the type of layout manager you use determines how items are positioned. Here are some examples:

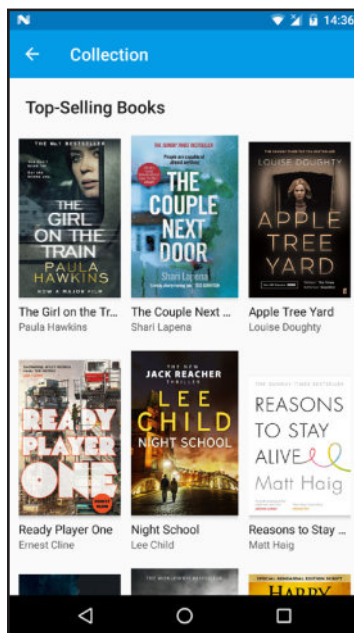


We don't cover how to do it, but you can also write your own layout managers. If you search for "android recyclerview layoutmanager" you'll find many third-party ones you can use in your code, from carousels to circles.



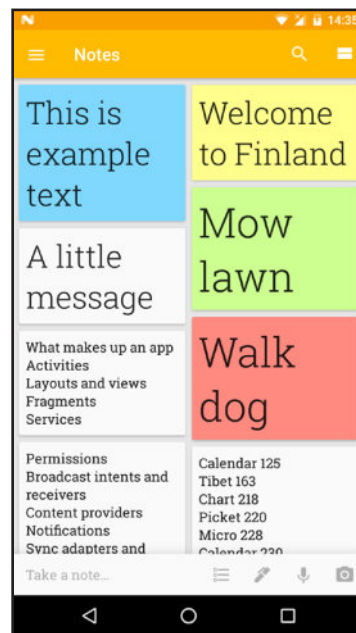
Linear layout manager

This arranges items in a vertical or horizontal list.



Grid layout manager

This arranges items in a grid.



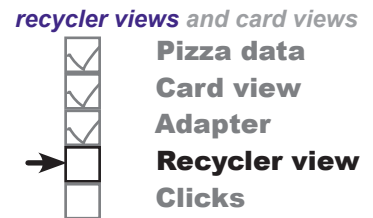
Staggered grid layout manager

This arranges unevenly sized items in a staggered grid.

On the next page, we'll show you how to specify which layout manager to use in your recycler view.

Specify the layout manager

You tell the recycler view which layout manager it should use by creating a new instance of the type of layout manager you want to use, then attaching it to the recycler view.



Linear layout manager

To tell the recycler view that you want it to display its views in a linear list, you'd use the following code:

```
LinearLayoutManager layoutManager = new LinearLayoutManager(getActivity());  
pizzaRecycler.setLayoutManager(layoutManager);
```

The `LinearLayoutManager` constructor takes one parameter, a `Context`. If you're using the code in an activity, you'd normally use `this` to pass it the current activity (a context). The above code uses `getActivity()` instead, as our recycler view is in a fragment.

This needs to be a Context. If you use this code in an activity, you use "this" instead of `getActivity()`.

Grid layout manager

You use similar code to specify a grid layout manager, except that you need to create a new `GridLayoutManager` object instead. The `GridLayoutManager` takes two parameters in its constructor: a `Context`, and an `int` value specifying the number of columns the grid should have.

This says that the `GridLayoutManager` should be two columns wide.

```
GridLayoutManager layoutManager = new GridLayoutManager(getActivity(), 2);
```

You can also change the orientation of the grid. To do this, you add two more parameters to the constructor: the orientation, and whether you want the views to appear in reverse order.

Gives the `GridLayoutManager` a horizontal orientation.

```
GridLayoutManager layoutManager =  
    new GridLayoutManager(getActivity(), 1, GridLayoutManager.HORIZONTAL, false);
```

Staggered grid layout manager

You tell the recycler view to use a staggered grid layout manager by creating a new `StaggeredGridLayoutManager` object. Its constructor takes two parameters: an `int` value for the number of columns or rows, and an `int` value for its orientation. As an example, here's how you'd specify a staggered grid layout oriented vertically with two rows:

If you wanted to display the list in reverse order, you'd set this to true.

```
StaggeredGridLayoutManager layoutManager =  
    new StaggeredGridLayoutManager(2, StaggeredGridLayoutManager.VERTICAL);
```

This gives the staggered grid layout a vertical orientation.

Let's add a layout manager to our recycler view.

The full PizzaFragment.java code

We're going to use a `GridLayoutManager` to display the pizza data in a grid. Here's the full code for `PizzaFragment.java`, update your version of the code to match our changes (in bold):

```
package com.hfad.bitsandpizzas;

import android.os.Bundle;
import android.support.v4.app.Fragment;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;
import android.support.v7.widget.RecyclerView;
import android.support.v7.widget.GridLayoutManager;

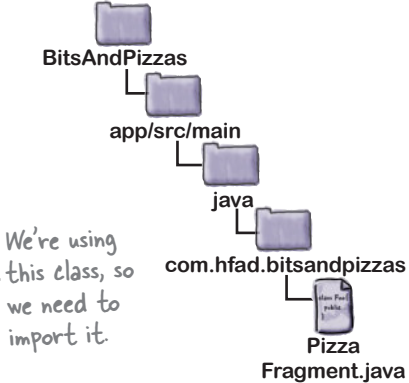
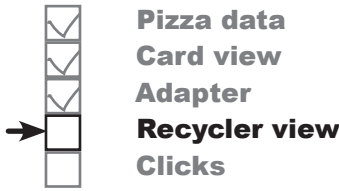
public class PizzaFragment extends Fragment {

    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup container,
        Bundle savedInstanceState) {
        RecyclerView pizzaRecycler = (RecyclerView)inflater.inflate(
            R.layout.fragment_pizza, container, false);

        String[] pizzaNames = new String[Pizza.pizzas.length];
        for (int i = 0; i < pizzaNames.length; i++) {
            pizzaNames[i] = Pizza.pizzas[i].getName();
        }

        int[] pizzaImages = new int[Pizza.pizzas.length];
        for (int i = 0; i < pizzaImages.length; i++) {
            pizzaImages[i] = Pizza.pizzas[i].getImageResourceId();
        }

        CaptionedImagesAdapter adapter = new CaptionedImagesAdapter(pizzaNames, pizzaImages);
        pizzaRecycler.setAdapter(adapter);
GridLayoutManager layoutManager = new GridLayoutManager(getActivity(), 2);
pizzaRecycler.setLayoutManager(layoutManager);
        return pizzaRecycler;
    }
}
```

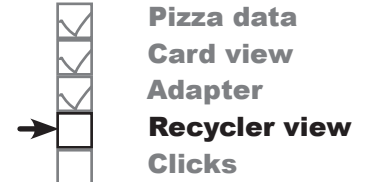


We're going to display the CardViews in a grid with two columns, so we're using a `GridLayoutManager`.

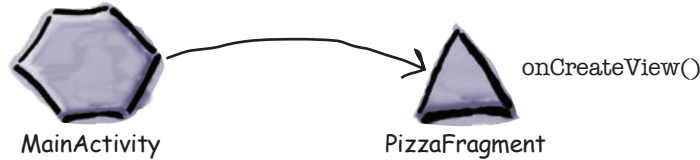
Next we'll examine what happens when the code runs, then take our app for a test drive.

What happens when the code runs

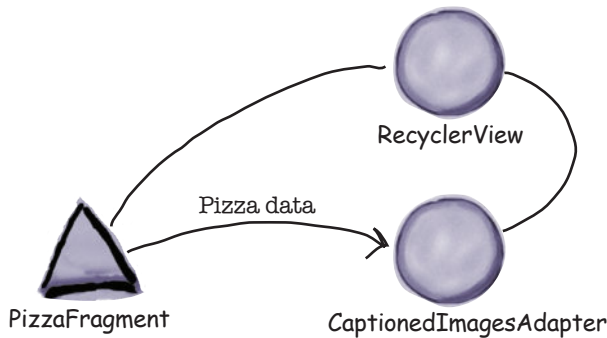
recycler views and card views



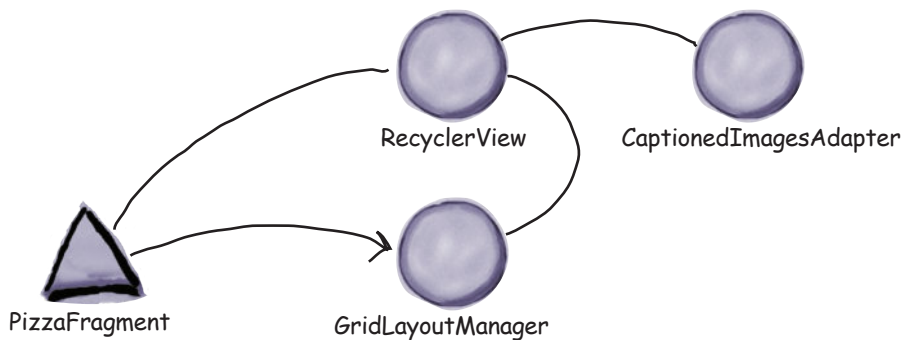
- 1 The user clicks on the Pizzas tab in MainActivity. PizzaFragment is displayed, and its onCreateView() method runs.



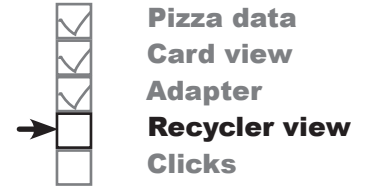
- 2 The PizzaFragment onCreateView() method creates a new CaptionedImagesAdapter. The method passes the names and images of the pizzas to the adapter using the adapter's constructor, and sets the adapter to the recycler view.



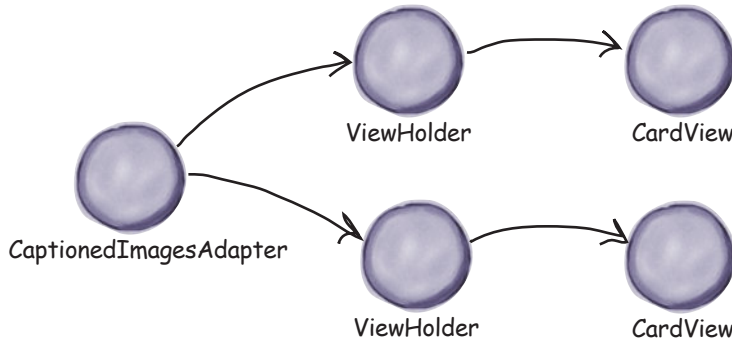
- 3 The PizzaFragment onCreateView() method creates a GridLayoutManager and assigns it to the recycler view. The GridLayoutManager means that the views will be displayed in a grid. As the recycler view has a vertical scrollbar, the list will be displayed vertically.



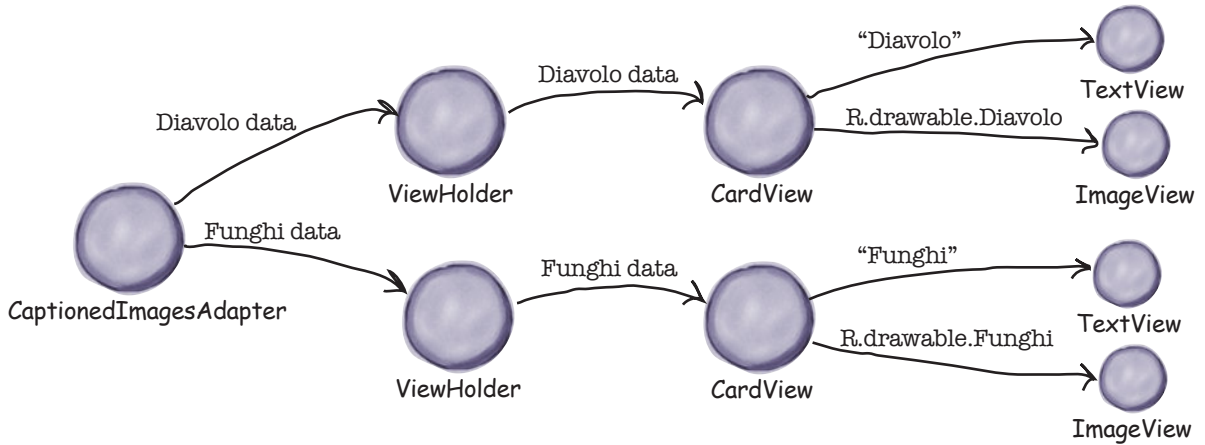
The story continues



4 The adapter creates a view holder for each of the CardViews the recycler view needs to display.



5 The adapter then binds the pizza names and images to the text view and image view in each card view.



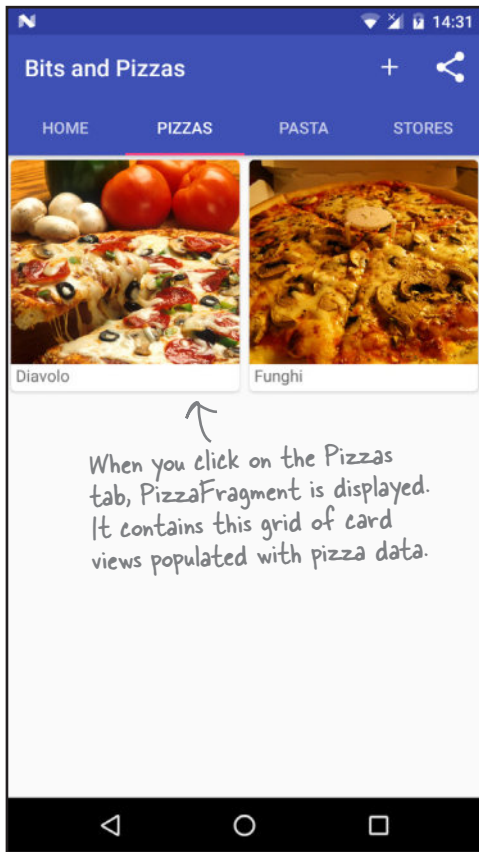
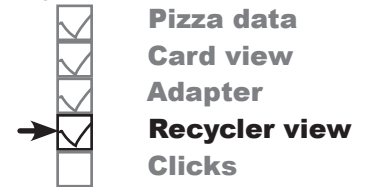
Let's run the app and see how it looks.



Test drive the app

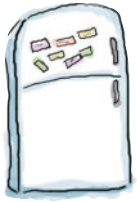
When we run the app, MainActivity is displayed. When we click on or swipe to the Pizzas tab, the pizzas are displayed in a grid. When we scroll the pizza data, MainActivity's toolbar responds.

recycler views and card views



As you can see, adding a recycler view is more involved than adding a list view, but it gives you a lot more flexibility. Most of the work comes from having to write a bespoke recycler view adapter, but you can reuse it elsewhere in your app. As an example, suppose you wanted to display pasta cards in a recycler view. You would use the same adapter we created earlier, but pass it pasta data instead of pizzas.

Before we move on, have a go at the following exercise.



RecyclerView Magnets

Use the magnets on this page and the next to create a new recycler view for the pasta dishes. The recycler view should contain a grid of card views, each one displaying the name and image of a pasta dish.

← This is the code for the Pasta class.

```
package com.hfad.bitsandpizzas;

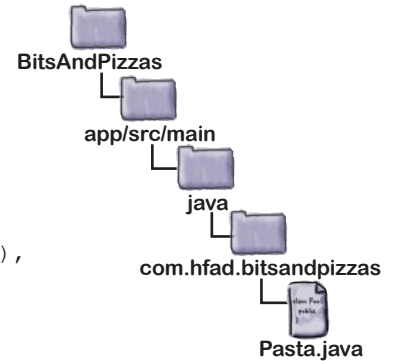
public class Pasta {
    private String name;
    private int imageResourceId;

    public static final.....[] pastas = {
        new Pasta("Spaghetti Bolognese", R.drawable.spag_bol),
        new Pasta("Lasagne", R.drawable.lasagne)
    };

    private Pasta(String name, int imageResourceId) {
        this.name = name;
        this.imageResourceId = imageResourceId;
    }

    public String .....{
        return name;
    }

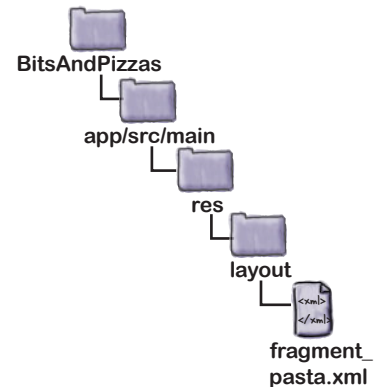
    public int .....{
        return imageResourceId;
    }
}
```



This is the code for the layout. ↘

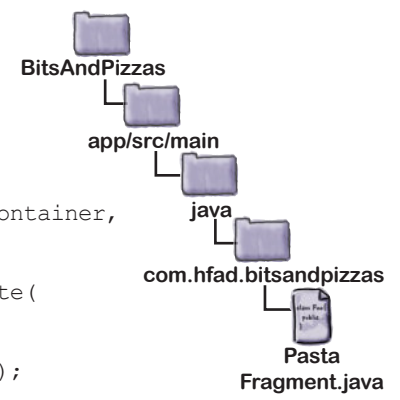
```
< .....
xmlns:android="http://schemas.android.com/apk/res/android"
android:id="@+id/pasta_recycler"

.....
android:layout_width="match_parent"
android:layout_height="match_parent"/>
```



...

← This is the code for PastaFragment.java.



```

public class PastaFragment extends Fragment {

    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup container,
        Bundle savedInstanceState) {
        RecyclerView pastaRecycler = (RecyclerView)inflater.inflate(
            ..... , container, false);

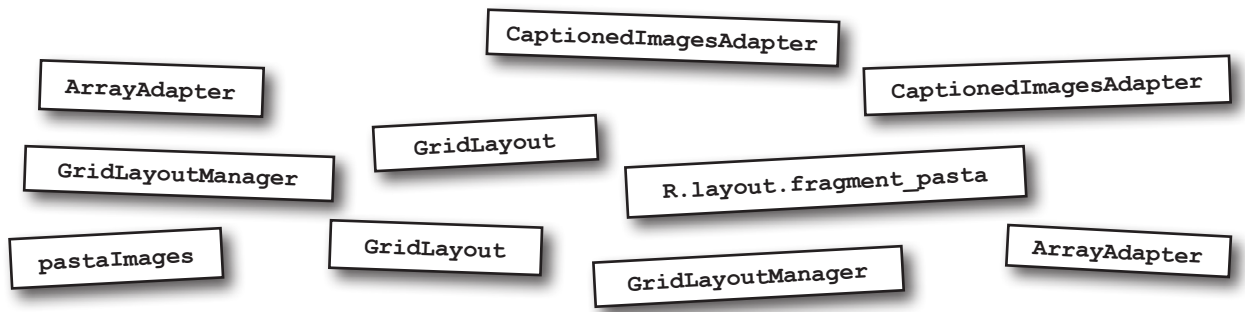
        String[] pastaNames = new String[Pasta.pastas.length];
        for (int i = 0; i < pastaNames.length; i++) {
            pastaNames[i] = Pasta.pastas[i].getName();
        }

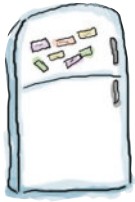
        int[] pastaImages = new int[Pasta.pastas.length];
        for (int i = 0; i < pastaImages.length; i++) {
            pastaImages[i] = Pasta.pastas[i].getImageResourceId();
        }

        .....adapter =

            new ..... (pastaNames, .....);
        pastaRecycler.setAdapter(adapter);

        ..... layoutManager = new ..... (getActivity(), 2);
        pastaRecycler.setLayoutManager(layoutManager);
        return pastaRecycler;
    }
}
  
```





RecyclerView Magnets Solution

Use the magnets on this page and the next to create a new recycler view for the pasta dishes. The recycler view should contain a grid of card views, each one displaying the name and image of a pasta dish.

```
package com.hfad.bitsandpizzas;

public class Pasta {
    private String name;
    private int imageResourceId;

    public static final Pasta[] pastas = {
        new Pasta("Spaghetti Bolognese", R.drawable.spag_bol),
        new Pasta("Lasagne", R.drawable.lasagne)
    };

    private Pasta(String name, int imageResourceId) {
        this.name = name;
        this.imageResourceId = imageResourceId;
    }

    public String getName() {
        return name;
    }

    public int getImageResourceId() {
        return imageResourceId;
    }
}
```

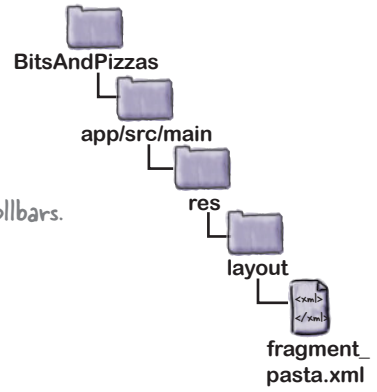
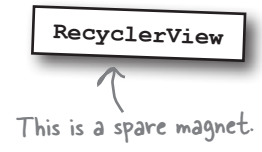
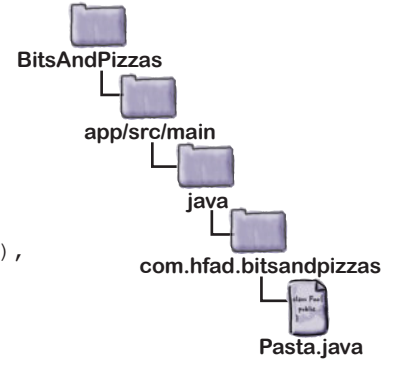
It's an array of Pasta objects.

These methods are used by PastaFragment.java.

Add the recycler view to the layout.

```
< android.support.v7.widget.RecyclerView
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/pasta_recycler"
    android:scrollbars = "vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent"/>
```

Add vertical scrollbars.



...

```
public class PastaFragment extends Fragment {

    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup container,
        Bundle savedInstanceState) {
        RecyclerView pastaRecycler = (RecyclerView)inflater.inflate(
```

Use this layout. →

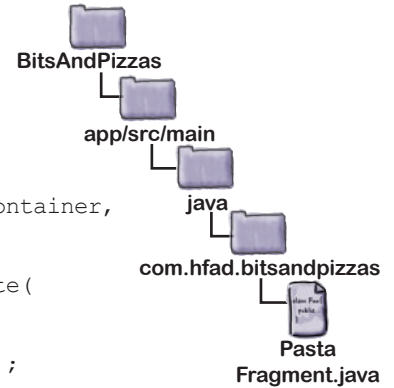
```
... R.layout.fragment_pasta ...container, false);
```

```
String[] pastaNames = new String[Pasta.pastas.length];
for (int i = 0; i < pastaNames.length; i++) {
    pastaNames[i] = Pasta.pastas[i].getName();
}
```

```
int[] pastaImages = new int[Pasta.pastas.length];
for (int i = 0; i < pastaImages.length; i++) {
    pastaImages[i] = Pasta.pastas[i].getImageResourceId();
}
```

```
new CaptionedImagesAdapter (pastaNames, pastaImages);
pastaRecycler.setAdapter(adapter);
```

```
LayoutManager layoutManager = new GridLayoutManager (getActivity(), 2);
pastaRecycler.setLayoutManager(layoutManager);
return pastaRecycler;
```



CaptionedImagesAdapter

CaptionedImagesAdapter

pastaImages

GridLayoutManager

GridLayoutManager

GridLayout

GridLayout

ArrayAdapter

ArrayAdapter

We're using the CaptionedImagesAdapter we wrote earlier.

Pass the pasta names and images to the adapter.

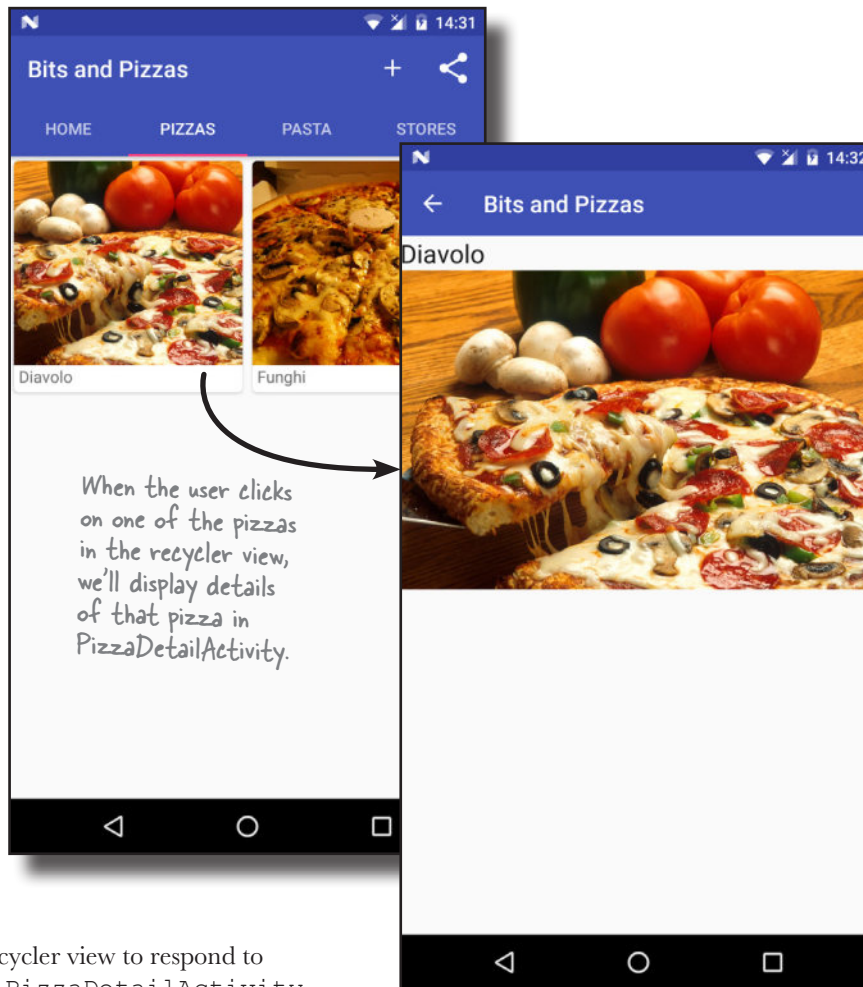
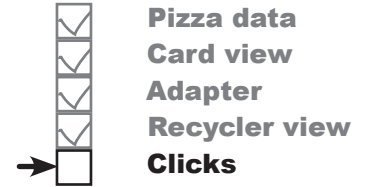
Use the GridLayoutManager to display the card views in a grid.

You didn't need to use these magnets.

Make the recycler view respond to clicks

So far, we've added a recycler view to `PizzaFragment`, and created an adapter to populate it with pizza data.

The next thing we need to do is get the recycler view to respond to clicks. We'll create a new activity, `PizzaDetailActivity`, which will start when the user clicks on one of the pizzas. The name and image of the pizza the user selects will be displayed in the activity:



Before we can get the recycler view to respond to clicks, we need to create `PizzaDetailActivity`.

Create PizzaDetailActivity

To create `PizzaDetailActivity`, click on the `com.hfad.bitsandpizzas` package in the Bits and Pizzas folder structure, then go to `File→New...→Activity→Empty Activity`. Name the activity “`PizzaDetailActivity`”, name the layout “`activity_pizza_detail`”, make sure the package name is `com.hfad.bitsandpizzas`, and **check the Backwards Compatibility (AppCompat) option**.

Now let’s update `PizzaDetailActivity`’s layout. Open `activity_pizza_detail.xml`, and update it with the code below, which adds a text view and image view to the layout that we’ll use to display details of the pizza:

```
<?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="com.hfad.bitsandpizzas.PizzaDetailActivity">

    <include
        layout="@layout/toolbar_main"
        android:id="@+id/toolbar" />

    <TextView
        android:id="@+id/pizza_text"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:textAppearance="?android:attr/textAppearanceLarge" />

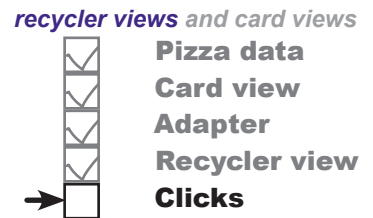
    <ImageView
        android:id="@+id/pizza_image"
        android:layout_height="wrap_content"
        android:layout_width="match_parent"
        android:adjustViewBounds="true" />

</LinearLayout>
```

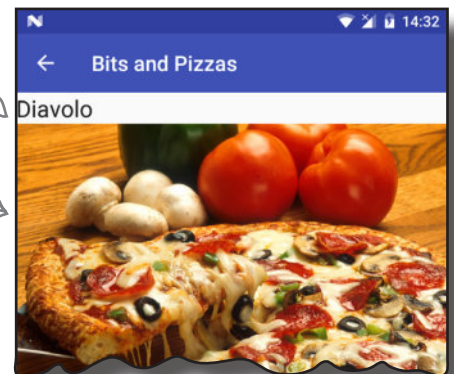
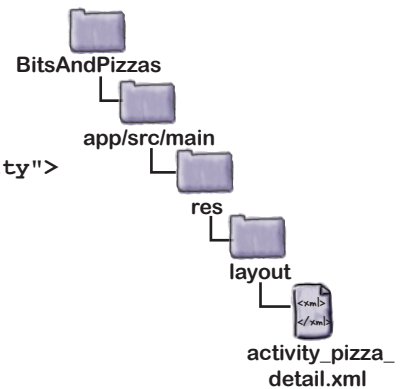
We'll add a toolbar to the activity.

We'll put the name of the pizza in the TextView.

We'll put the pizza image in the ImageView.



If prompted for the activity's source language, select the option for Java.



We’ll look at what we need the code for `PizzaDetailActivity.java` to do on the next page.

What `PizzaDetailActivity` needs to do

There are a couple of things that we need `PizzaDetailActivity` to do:



Pizza data
Card view
Adapter
RecyclerView view
Clicks

- ★ `PizzaDetailActivity`'s main purpose is to display the name and image of the pizza the user has selected. To do this, we'll get the selected pizza's ID from the intent that starts the activity. We'll pass this to `PizzaDetailActivity` from `PizzaFragment` when the user clicks on one of the pizzas in the recycler view.
- ★ We'll enable the `PizzaDetailActivity`'s Up button so that when the user clicks on it, they'll get returned to `MainActivity`.

Update `AndroidManifest.xml` to give `PizzaDetailActivity` a parent

We'll start by updating `AndroidManifest.xml` to specify that `MainActivity` is the parent of `PizzaDetailActivity`. This means that when the user clicks on the Up button in `PizzaDetailActivity`'s app bar, `MainActivity` will be displayed. Here's our version of `AndroidManifest.xml` (update your version to match our changes in bold):

```
<manifest ...>
  <application
    ...>
    <activity
      android:name=".MainActivity">
      ...
    </activity>
    <activity
      android:name=".OrderActivity"
      ...
    </activity>
    <activity
      android:name=".PizzaDetailActivity"
      android:parentActivityName=".MainActivity">
    </activity>
  </application>
</manifest>
```



↑
 This sets `MainActivity` as `PizzaDetailActivity`'s parent.

Next, we'll update `PizzaDetailActivity.java`. You've already seen how to do everything we need, so we're just going to show you the full code.

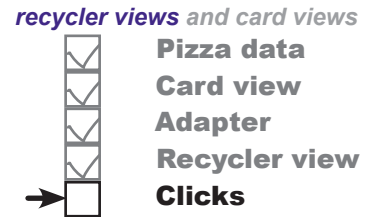
The code for PizzaDetailActivity.java

Here's the full code for *PizzaDetailActivity.java*; update your version of the code to match ours:

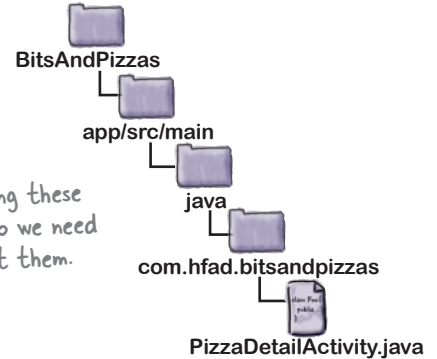
```
package com.hfad.bitsandpizzas;

import android.support.v7.app.ActionBar;
import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.support.v7.widget.Toolbar;
import android.widget.ImageView;
import android.widget.TextView;
import android.support.v4.content.ContextCompat;

public class PizzaDetailActivity extends AppCompatActivity {
```



We're using these classes, so we need to import them.



```
    public static final String EXTRA_PIZZA_ID = "pizzaId";

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

        //Set the toolbar as the activity's app bar
        Toolbar toolbar = (Toolbar) findViewById(R.id.toolbar);
        setSupportActionBar(toolbar);
        ActionBar actionBar = getSupportActionBar();
        actionBar.setDisplayHomeAsUpEnabled(true);

        //Display details of the pizza
        int pizzaId = (Integer) getIntent().getExtras().get(EXTRA_PIZZA_ID);
        String pizzaName = Pizza.pizzas[pizzaId].getName();
        TextView textView = (TextView) findViewById(R.id.pizza_text);
        textView.setText(pizzaName);
        int pizzaImage = Pizza.pizzas[pizzaId].getImageResourceId();
        ImageView imageView = (ImageView) findViewById(R.id.pizza_image);
        imageView.setImageDrawable(ContextCompat.getDrawable(this, pizzaImage));
        imageView.setContentDescription(pizzaName);
    }
}
```

We'll use this constant to pass the ID of the pizza as extra information in the intent.

Enable the Up button.

Use the pizza ID to populate the TextView and ImageView.

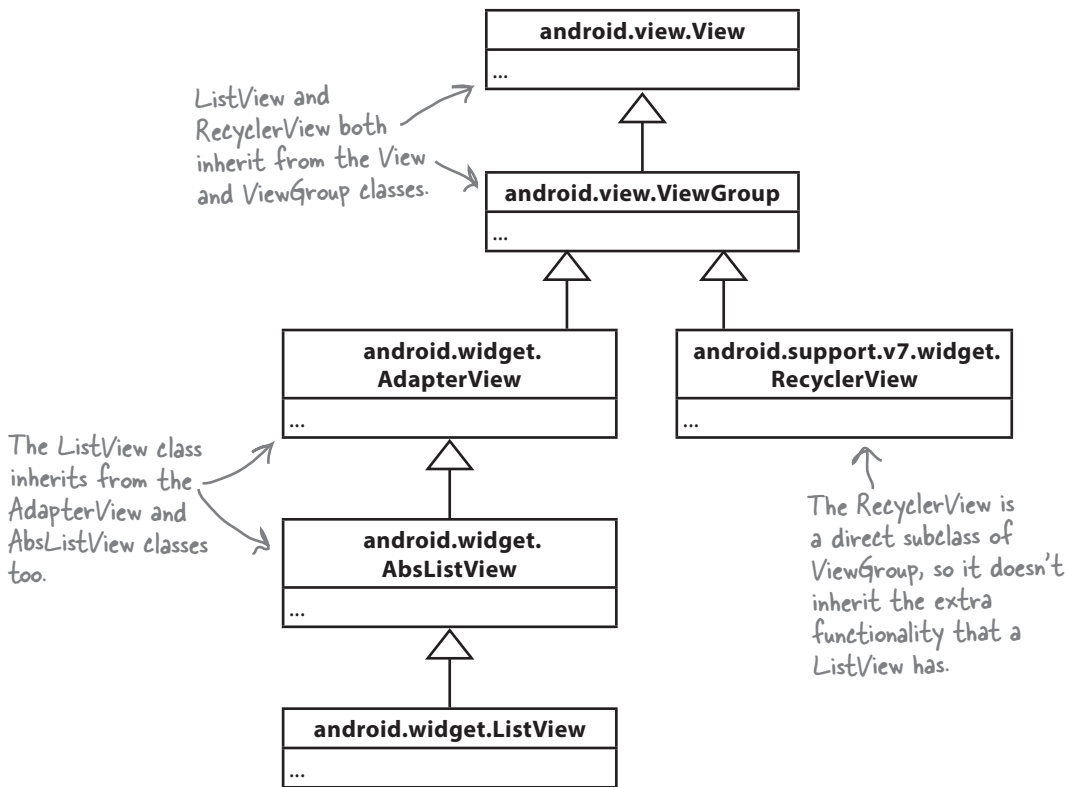
From the intent, get the pizza the user chose.

Get a recycler view to respond to clicks

Next, we need to get items in the recycler view to respond to clicks so that we can start `PizzaDetailActivity` when the user clicks on a particular pizza.

When you create a navigation list with a list view, you can respond to click events within the list by giving the list view an `OnItemClickListener`. The list view then listens to each of the views that it contains, and if any of them are clicked, the list view calls its `OnItemClickListener`. That means that you can respond to list item clicks with very little code.

List views are able to do this because they inherit a bunch of functionality from a deep hierarchy of superclasses. RecyclerViews, however, don't have such a rich set of built-in methods, as they don't inherit from the same superclasses. Here's a class hierarchy diagram for the `ListView` and `RecyclerView` classes:



While this gives recycler views more flexibility, it also means that with a recycler view you have to do a lot more of the work yourself. So how do we get our recycler view to respond to clicks?

You can listen for view events from the adapter

To get your recycler view to respond to click events, you need access to the views that appear inside it. These views are all created inside the recycler view's adapter. When a view appears onscreen, the recycler view calls the `CaptionedImagesAdapter`'s `onBindViewHolder()` method to make the card view match the details of the list item.

When the user clicks on one of the pizza cards in the recycler view, we want to start `PizzaDetailActivity`, passing it the position of the pizza that was clicked. That means you *could* put some code inside the adapter to start an activity like this:

```
class CaptionedImagesAdapter extends
    RecyclerView.Adapter<CaptionedImagesAdapter.ViewHolder>{
...
    @Override
    public void onBindViewHolder(ViewHolder holder, final int position){
        final CardView cardView = holder.cardView;
        ImageView imageView = (ImageView)cardView.findViewById(R.id.info_image);
        Drawable drawable =
            ContextCompat.getDrawable(cardView.getContext(), imageIds[position]);
        imageView.setImageDrawable(drawable);
        imageView.setContentDescription(captions[position]);
        TextView textView = (TextView)cardView.findViewById(R.id.info_text);
        textView.setText(captions[position]);
        cardView.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                Intent intent = new Intent(cardView.getContext(), PizzaDetailActivity.class);
                intent.putExtra(PizzaDetailActivity.EXTRA_PIZZA_ID, position);
                cardView.getContext().startActivity(intent);
            }
        });
    }
}
```

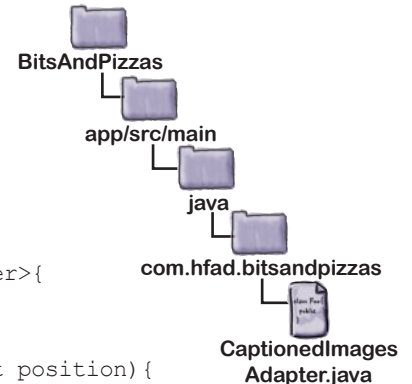
Don't update the adapter code just yet. This is just an example.

Adding this code to the adapter would start `PizzaDetailActivity` when a `CardView` is clicked.

recycler views and card views



Pizza data
Card view
Adapter
Recycler view
Clicks



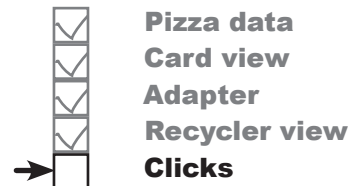
But just because you *could* write this code, it doesn't necessarily mean that you should.



You *could* respond to a click event by adding code to your adapter class. But can you think of a reason why you *wouldn't* want to do that?

Keep your adapters reusable

If you deal with click events in the `CaptionedImagesAdapter` class, you'll limit how that adapter can be used. Think about the app we're building. We want to display lists of pizzas, pasta, and stores. In each case, we'll probably want to display a list of captioned images. If we modify the `CaptionedImagesAdapter` class so that clicks always send the user to an activity that displays details of a single pizza, we won't be able to use the `CaptionedImagesAdapter` for the pasta and stores lists. We'll have to create a separate adapter for each one.



Decouple your adapter with an interface

Instead of that approach, we'll keep the code that starts the activity outside of the adapter. When someone clicks on an item in the list, we want the adapter to call the fragment that contains the list, and then the fragment code can fire off an intent to the next activity. That way we can reuse `CaptionedImagesAdapter` for the pizzas, pasta, and stores lists, and in each case leave it to the fragments to decide what happens in response to a click.

We're going to use a similar pattern to the one that allowed us to decouple a fragment from an activity in Chapter 9. We'll create a `Listener` interface inside `CaptionedImagesAdapter` like this:

```
interface Listener {
    void onClick(int position);
}
```

We'll call the listener's `onClick()` method whenever one of the card views in the recycler view is clicked. We'll then add code to `PizzaFragment` so that it implements the interface; this will allow the fragment to respond to clicks and start an activity.

This is what will happen at runtime:

- 1 A user will click on a card view in the recycler view.
- 2 The `Listener`'s `onClick()` method will be called.
- 3 The `onClick()` method will be implemented in `PizzaFragment`. Code in this fragment will start `PizzaDetailActivity`.

Let's start by adding code to `CaptionedImagesAdapter.java`.

Add the interface to the adapter

We've updated our *CaptionedImagesAdapter.java* code to add the `Listener` interface and call its `onClick()` method whenever one of the card views is clicked. Apply the changes below (in bold) to your code, then save your work:

```
package com.hfad.bitsandpizzas;

import android.support.v7.widget.RecyclerView;
import android.support.v7.widget.CardView;
import android.view.ViewGroup;
import android.view.LayoutInflater;
import android.widget.ImageView;
import android.widget.TextView;
import android.graphics.drawable.Drawable;
import android.support.v4.content.ContextCompat;
import android.view.View; ← We're using this extra class, so we need to import it.
```

```
class CaptionedImagesAdapter extends
    RecyclerView.Adapter<CaptionedImagesAdapter.ViewHolder>{

    private String[] captions;
    private int[] imageIds;
    private Listener listener; ← Add the Listener as a private variable.

    interface Listener { ← Add the interface.
        void onClick(int position);
    }

    public static class ViewHolder extends RecyclerView.ViewHolder {

        private CardView cardView;

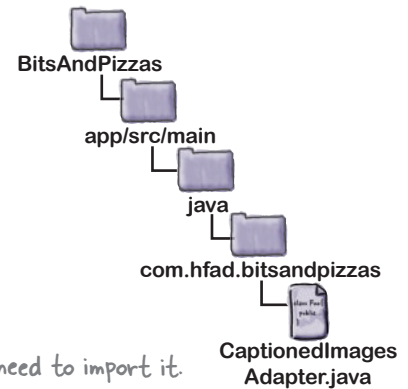
        public ViewHolder(CardView v) {
            super(v);
            cardView = v;
        }
    }

    public CaptionedImagesAdapter(String[] captions, int[] imageIds){
        this.captions = captions;
        this.imageIds = imageIds;
    }
}
```

recycler views and card views



Pizza data
Card view
Adapter
RecyclerView
Clicks



The code continues →
on the next page.

The CaptionedImagesAdapter.java code (continued)

```

@Override
public int getItemCount() {
    return captions.length;
}

public void setListener(Listener listener) {
    this.listener = listener;
}

@Override
public CaptionedImagesAdapter.ViewHolder onCreateViewHolder(
    ViewGroup parent, int viewType) {
    CardView cv = (CardView) LayoutInflater.from(parent.getContext())
        .inflate(R.layout.card_captioned_image, parent, false);
    return new ViewHolder(cv);
}

@Override
public void onBindViewHolder(ViewHolder holder, final int position) {
    CardView cardView = holder.cardView;
    ImageView imageView = (ImageView) cardView.findViewById(R.id.info_image);
    Drawable drawable =
        ContextCompat.getDrawable(cardView.getContext(), imageIds[position]);
    imageView.setImageDrawable(drawable);
    imageView.setContentDescription(captions[position]);
    TextView textView = (TextView) cardView.findViewById(R.id.info_text);
    textView.setText(captions[position]);
    cardView.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View v) {
            if (listener != null) {
                listener.onClick(position);
            }
        }
    });
}
}

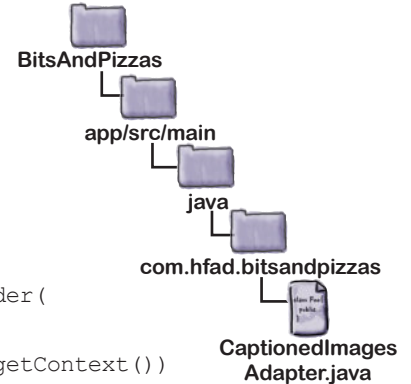
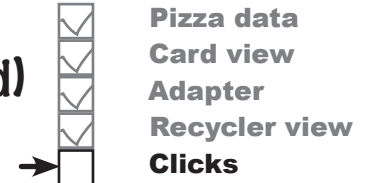
```

Activities and fragments will use this method to register as a listener.

You need to change the position variable to final, as it's used in an inner class.

Add the listener to the CardView.

When the CardView is clicked, call the Listener onClick() method.



Now that we've added a listener to the adapter, we need to implement it in PizzaFragment.

Implement the listener in PizzaFragment.java

We'll implement CaptionedImagesAdapter's Listener interface in PizzaFragment so that when a card view in the recycler view is clicked, PizzaDetailActivity will be started. Here's the updated code; update your version of the code to match ours (our changes are in bold):

```
package com.hfad.bitsandpizzas;

import android.os.Bundle;
import android.support.v4.app.Fragment;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;
import android.support.v7.widget.RecyclerView;
import android.support.v7.widget.GridLayoutManager;
import import android.content.Intent; ← We're using an Intent to start
                                the activity, so import this class.

public class PizzaFragment extends Fragment {

    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup container,
                            Bundle savedInstanceState) {
        RecyclerView pizzaRecycler = (RecyclerView)inflater.inflate(
            R.layout.fragment_pizza, container, false);

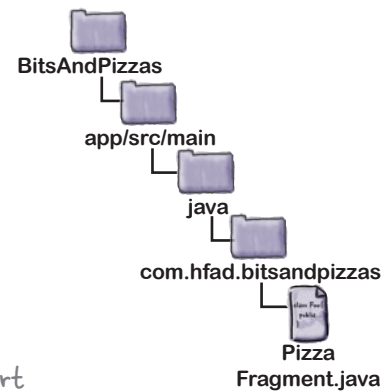
        String[] pizzaNames = new String[Pizza.pizzas.length];
        for (int i = 0; i < pizzaNames.length; i++) {
            pizzaNames[i] = Pizza.pizzas[i].getName();
        }

        int[] pizzaImages = new int[Pizza.pizzas.length];
        for (int i = 0; i < pizzaImages.length; i++) {
            pizzaImages[i] = Pizza.pizzas[i].getImageResourceId();
        }
    }
}
```

recycler views and card views

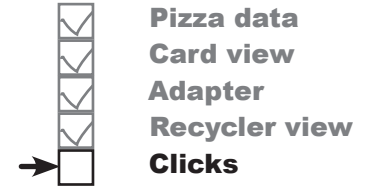


Pizza data
Card view
Adapter
RecyclerView
Clicks



The code continues →
on the next page.

The PizzaFragment.java code (continued)



```

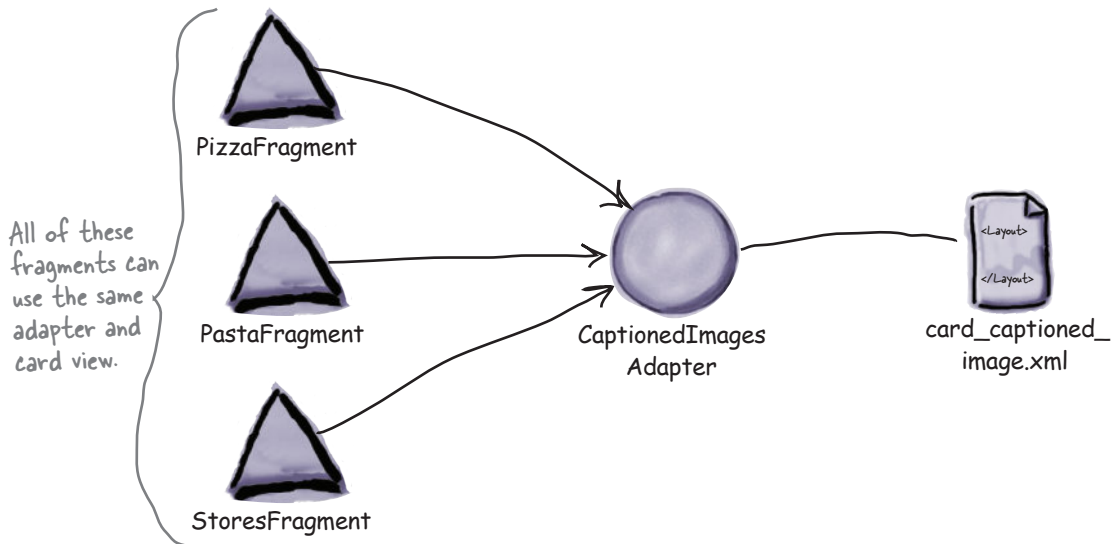
CaptionedImagesAdapter adapter =
    new CaptionedImagesAdapter(pizzaNames, pizzaImages);
pizzaRecycler.setAdapter(adapter);
LayoutManager layoutManager = new GridLayoutManager(getActivity(), 2);
pizzaRecycler.setLayoutManager(layoutManager);

adapter.setOnClickListener(new CaptionedImagesAdapter.Listener() {
    public void onClick(int position) {
        Intent intent = new Intent(getActivity(), PizzaDetailActivity.class);
        intent.putExtra(PizzaDetailActivity.EXTRA_PIZZA_ID, position);
        getActivity().startActivity(intent);
    }
});
return pizzaRecycler;
}

```

This implements the Listener onClick() method. It starts PizzaDetailActivity, passing it the ID of the pizza the user chose.

That's all the code we need to make views in the recycler view respond to clicks. By taking this approach, we can use the same adapter and card view for different types of data that is composed of an image view and text view.



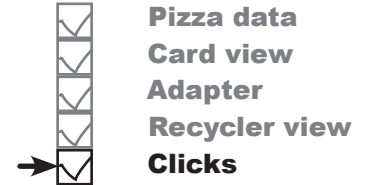
Let's see what happens when we run the code.



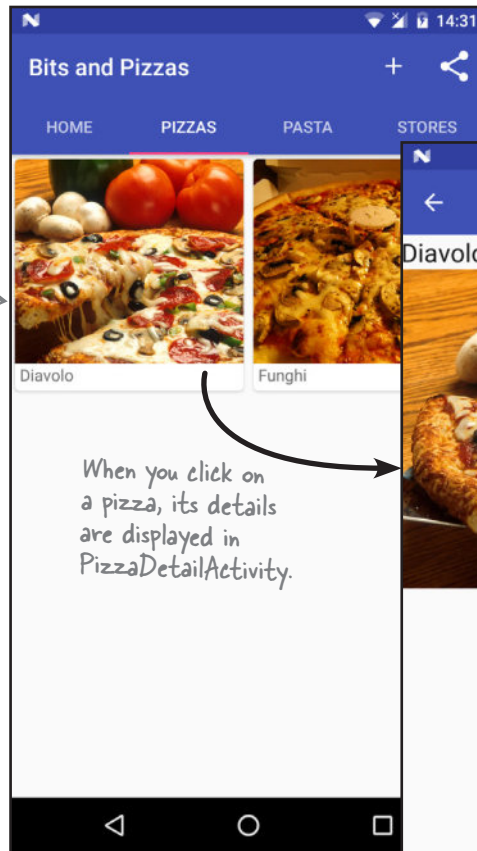
Test drive the app

When we run the app and click on the Pizzas tab, `PizzaFragment` is displayed. When we click on one of the pizzas, `PizzaDetailActivity` starts, and details of that pizza are displayed.

recycler views and *card views*



When you click on the Pizzas tab, `PizzaFragment` is displayed.



When you click on a pizza, its details are displayed in `PizzaDetailActivity`.



`PizzaDetailActivity` displays the pizza's name and image.

The card view responds to clicks, and displays `PizzaDetailActivity`.



Your Android Toolbox

You've got Chapter 13 under your belt and now you've added recycler views and card views to your toolbox.

You can download the full code for the chapter from <https://tinyurl.com/HeadFirstAndroid>.



BULLET POINTS

- Card views and recycler views have their own Support Libraries.
- Add a card view to a layout using the `<android.support.v7.widget.CardView>` element.
- Give the card view rounded corners using the `cardCornerRadius` attribute. This requires a namespace of `"http://schemas.android.com/apk/res-auto"`.
- Give the card view a drop shadow using the `cardElevation` attribute. This requires a namespace of `"http://schemas.android.com/apk/res-auto"`.
- Recycler views work with adapters that are subclasses of `RecyclerView.Adapter`.
- When you create your own `RecyclerView.Adapter`, you must define the view holder and implement the `onCreateViewHolder()`, `onBindViewHolder()`, and `getItemCount()` methods.
- You add a recycler view to a layout using the `<android.support.v7.widget.RecyclerView>` element. You give it a scrollbar using the `android:scrollbars` attribute.
- Use a layout manager to specify how items in a recycler view should be arranged. A `LinearLayoutManager` arranges items in a linear list, a `GridLayoutManager` arranges items in a grid, and a `StaggeredGridLayoutManager` arranges items in a staggered grid.

14 navigation drawers

Going Places



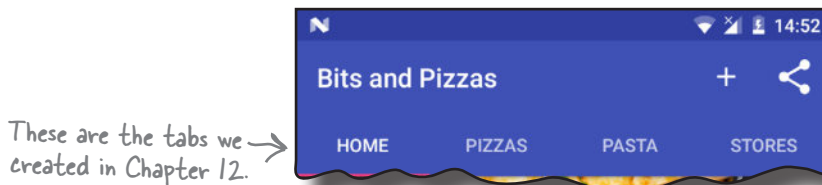
I know I'll never get lost so long as I have my lucky navigation drawers.

You've already seen how tabs help users navigate your apps.

But if you need a *large number* of them, or want to *split them into sections*, the **navigation drawer** is your new BFF. In this chapter, we'll show you how to create a navigation drawer that *slides out from the side of your activity at a single touch*. You'll learn how to give it a header using a **navigation view**, and provide it with a **structured set of menu items** to take the user to all the major hubs of your app. Finally, you'll discover how to set up a **navigation view listener** so that the drawer *responds to the slightest touch and swipe*.

Tab layouts allow easy navigation...

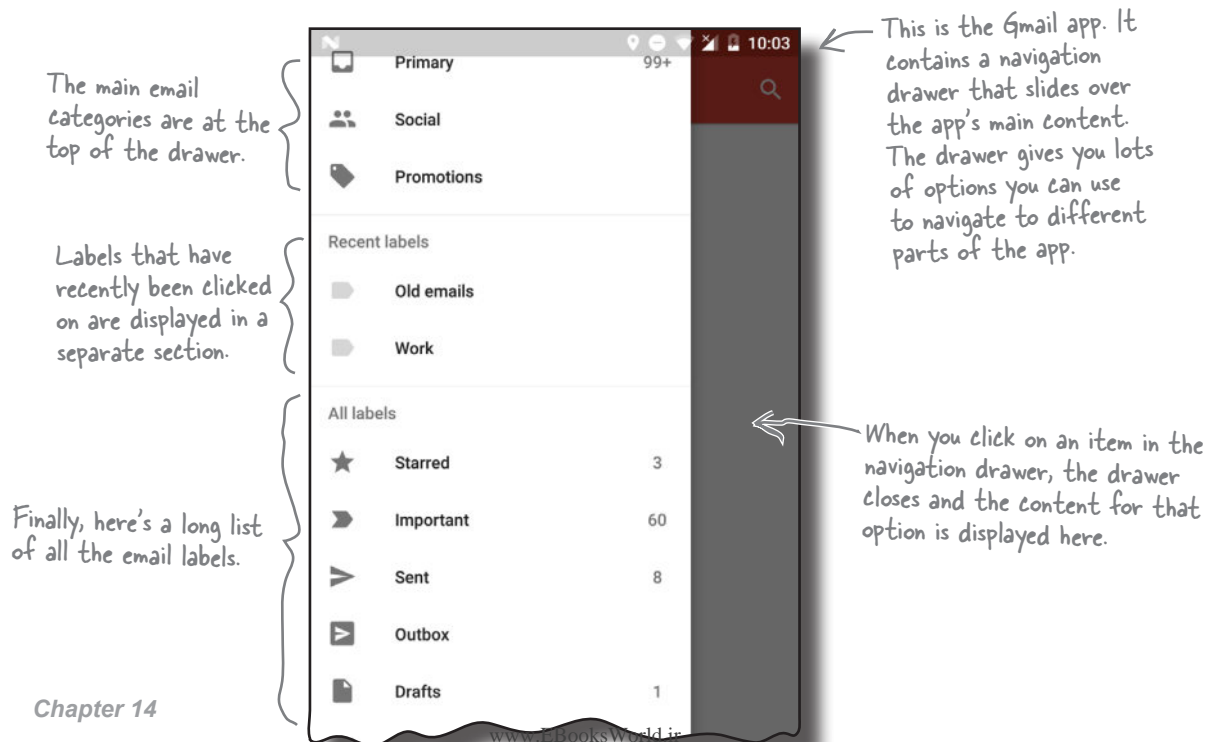
In Chapter 12, we introduced you to the tab layout as a way of making it easy for users to navigate around your app. In that chapter we added a Home screen tab to the Bits and Pizzas app, along with tabs for the Pizzas, Pasta, and Stores categories:



Tab layouts work well if you have a small number of category screens that are all at the same level in the app hierarchy. But what if you want to use a large number of tabs, or group the tabs into sections?

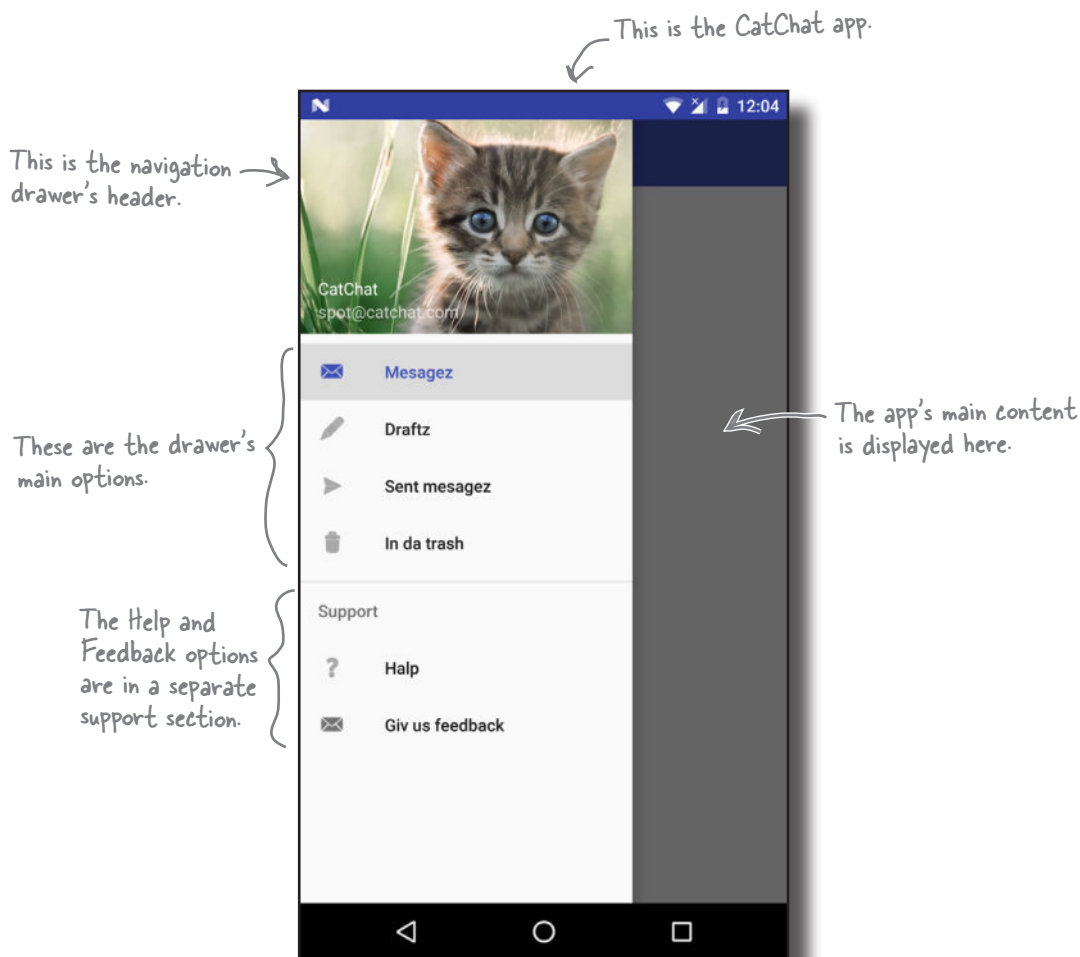
...but navigation drawers let you show more options

If you want users to be able to navigate through a large number of options, or group them into sections, you might prefer to use a **navigation drawer**. This is a slide-out panel that contains links to other parts of the app that you can group into different sections. As an example, the Gmail app uses a navigation drawer that contains sections such as email categories, recent labels, and all labels:



We're going to create a navigation drawer for a new email app

We're going to create a navigation drawer for a new email app called CatChat. The navigation drawer will contain a header (including an image and some text) and a set of options. The main options will be for the user's inbox, draft messages, sent items, and trash. We'll also include a separate support section for help and feedback options:



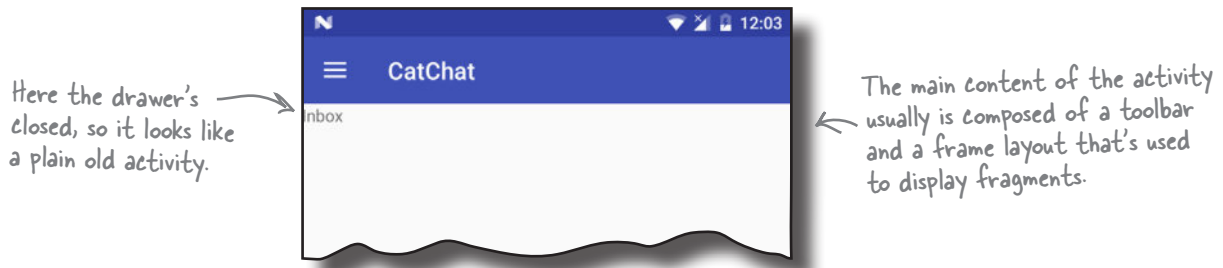
The navigation drawer is composed of several different components. We'll go through these on the next page.

Navigation drawers deconstructed

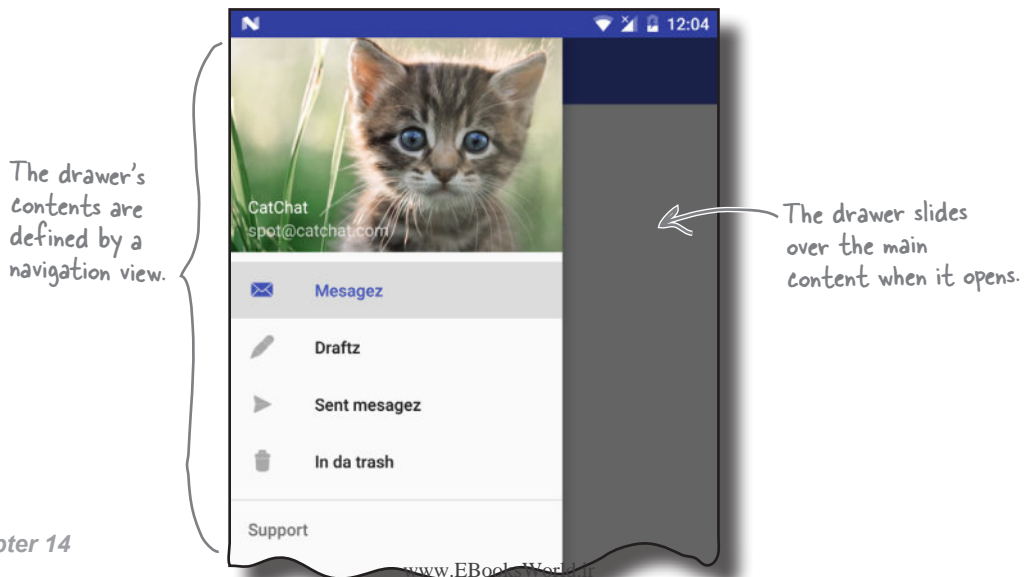
You implement a navigation drawer by adding a **drawer layout** to your activity's layout. This defines a drawer you can open and close, and it needs to contain two views:

- 1 A view for the main content.**
This is usually a layout containing a toolbar and a frame layout, which you use to display fragments.
- 2 A view for the drawer contents.**
This is usually a navigation view, which controls most of the drawer's behavior.

When the drawer's closed, the drawer layout looks just like a normal activity. It displays the layout for its main content:



When you open the navigation drawer, it slides over the activity's main content to display the drawer's contents. This is usually a navigation view, which displays a drawer header image and a list of options. When you click on one of these options, it either starts a new activity or displays a fragment in the activity's frame layout:

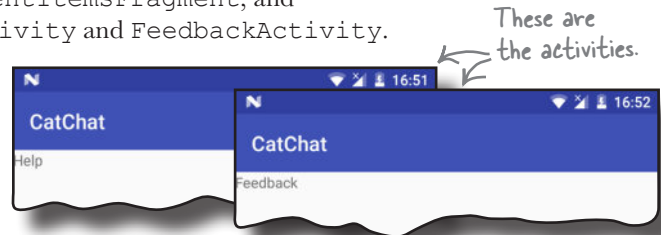
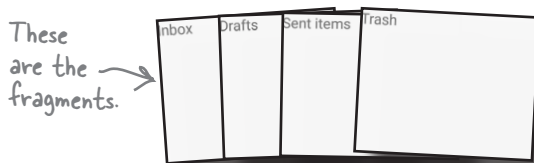


Here's what we're going to do

We're going to create a navigation drawer for the CatChat app. There are four main steps we'll go through to do this:

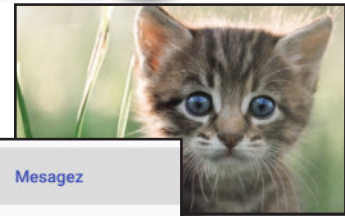
- 1 Create basic fragments and activities for the app's contents.**

When the user clicks on one of the options in the navigation drawer, we want to display the fragment or activity for that option. We'll create the fragments `InboxFragment`, `DraftsFragment`, `SentItemsFragment`, and `TrashFragment`, and activities `HelpActivity` and `FeedbackActivity`.



- 2 Create the drawer's header.**

We'll build a layout, `nav_header.xml`, for the drawer's header. It will contain an image and text.

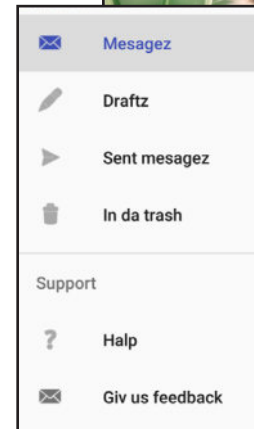


- 3 Create the drawer's options.**

We'll build a menu, `menu_nav.xml`, for the options the drawer will display.

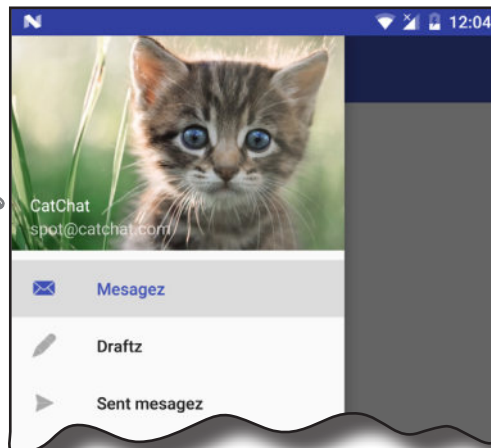
- 4 Create the navigation drawer.**

We'll add the navigation drawer to the app's main activity, and get it to display the header and options. We'll then write activity code to control the drawer's behavior.



↑ The drawer's header and
← options

We'll create this →
navigation drawer.



Let's get started.



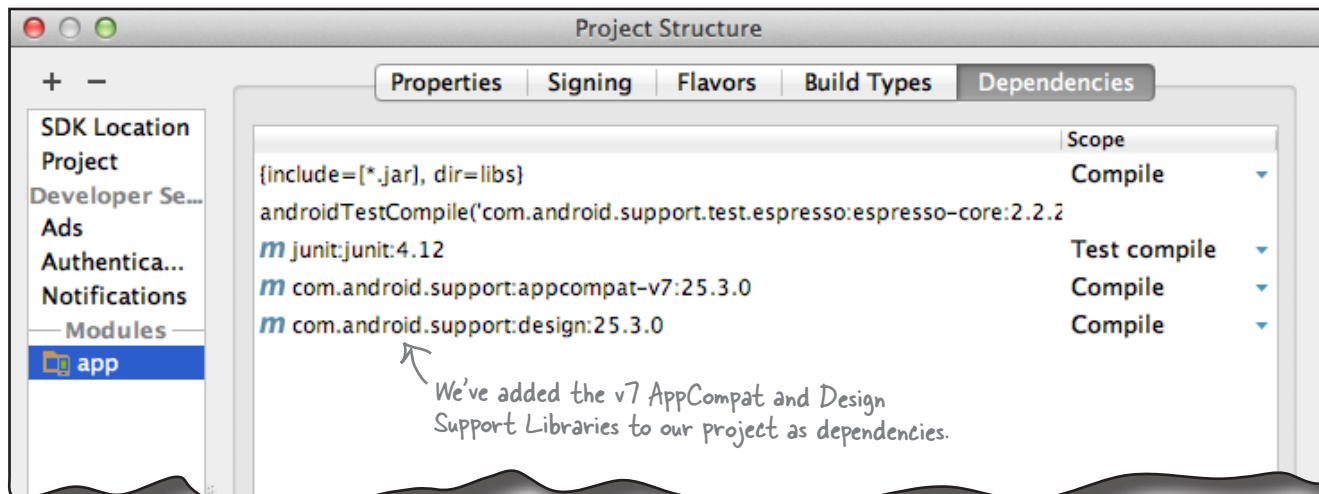
Fragments/activities
Header
Options
Drawer

Create the CatChat project

Before we begin, we need a new project for the CatChat app. Create a new Android project with an empty activity for an application named “CatChat” with a company domain of “hfad.com”, making the package name `com.hfad.catchat`. The minimum SDK should be API level 19 so that it works with most devices. Specify an activity called “MainActivity” and a layout called “activity_main”, and **make sure that you check the Backwards Compatibility (AppCompat) option**.

Add the v7 AppCompat and Design Support Libraries

We’re going to use components and themes from the v7 AppCompat and Design Support Libraries in this chapter, so we need to add them to our project as dependencies. To do this, choose File→Project Structure in Android Studio, click on the app module, then choose Dependencies. When you’re presented with the project dependencies screen, click on the “+” button at the bottom or right side of the screen. When prompted, choose the Library Dependency option, then select the Design Library from the list of possible libraries. Repeat these steps for the v7 AppCompat Support Library if Android Studio hasn’t already added it for you. Finally, use the OK buttons to save your changes.



Next, we’ll create four basic fragments for the app’s inbox, drafts, sent messages, and trash. We’ll use these fragments later in the chapter when we write the code for the navigation drawer.

Create InboxFragment

We'll display `InboxFragment` when the user clicks on the `inbox` option in the navigation drawer. 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`" and name its layout "`fragment_inbox`". Then update the code for `InboxFragment.java` to match our code below:

```
package com.hfad.catchat;

import android.os.Bundle;
import android.support.v4.app.Fragment;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;

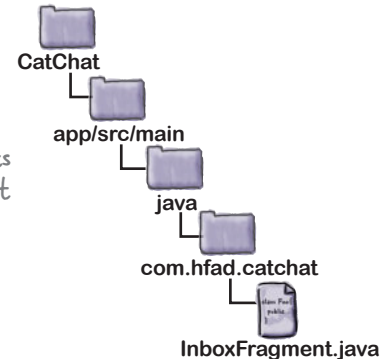
public class InboxFragment extends Fragment {

    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup container,
                             Bundle savedInstanceState) {
        return inflater.inflate(R.layout.fragment_inbox, container, false);
    }
}
```

All the fragments use the `Fragment` class from the Support Library.



This is what `InboxFragment` looks like.

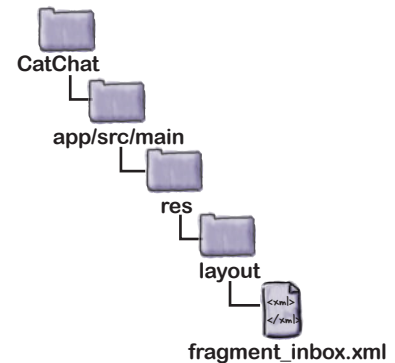


And here's the code for `fragment_inbox.xml` (update your version of this code too):

```
<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="com.hfad.catchat.InboxFragment">

    <TextView
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:text="Inbox" />
</LinearLayout>
```

`InboxFragment's layout just contains a TextView. We're adding this text so we can easily tell when it's displayed.`



Create DraftsFragment

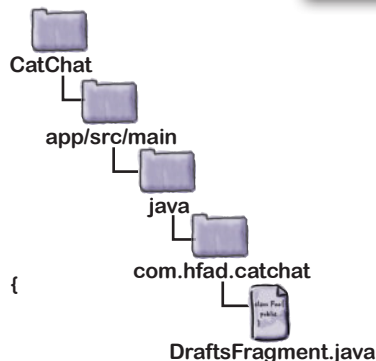
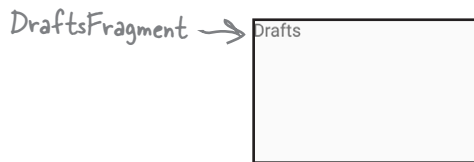
When the user clicks on the drafts option in the navigation drawer, we'll show DraftsFragment. Select the `com.hfad.catchat` package in the `app/src/main/java` folder, and create a new blank fragment named "DraftsFragment" with a layout called "fragment_drafts". Then replace the code for `DraftsFragment.java` with ours below:

```
package com.hfad.catchat;

import android.os.Bundle;
import android.support.v4.app.Fragment;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;

public class DraftsFragment extends Fragment {

    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup container,
                             Bundle savedInstanceState) {
        return inflater.inflate(R.layout.fragment_drafts, container, false);
    }
}
```

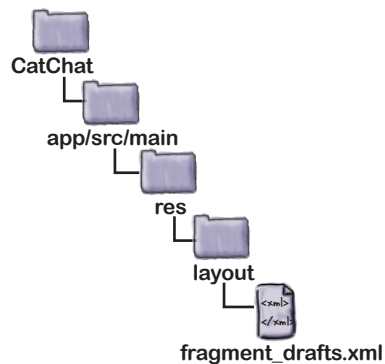


Next replace the code for `fragment_drafts.xml` too:

```
<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="com.hfad.catchat.DraftsFragment">

    <TextView
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:text="Drafts" />

</LinearLayout>
```



Create SentItemsFragment

We'll show `SentItemsFragment` when the user clicks on the sent items option in the navigation drawer. Highlight the `com.hfad.catchat` package in the `app/src/main/java` folder, and create a new blank fragment named "SentItemsFragment" with a layout called "fragment_sent_items". Then update the code for `SentItemsFragment.java` to match our code below:

```
package com.hfad.catchat;

import android.os.Bundle;
import android.support.v4.app.Fragment;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;

public class SentItemsFragment extends Fragment {

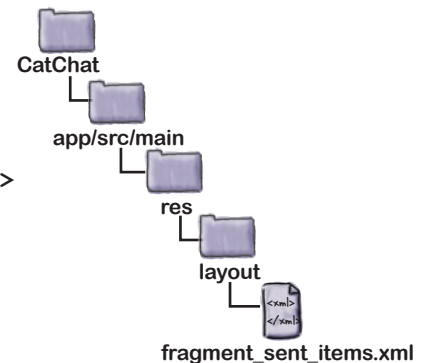
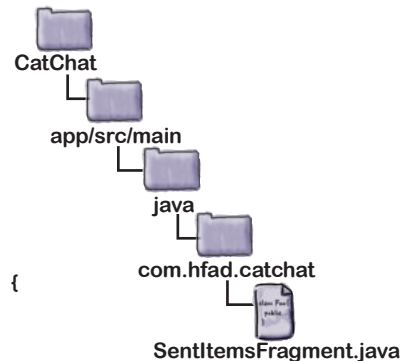
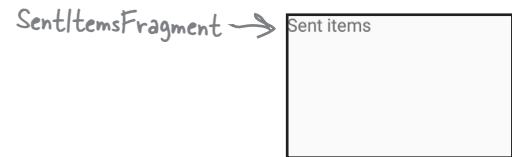
    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup container,
                             Bundle savedInstanceState) {
        return inflater.inflate(R.layout.fragment_sent_items, container, false);
    }
}
```

And here's the code for `fragment_sent_items.xml` (update your version):

```
<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="com.hfad.catchat.SentItemsFragment">

    <TextView
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:text="Sent items" />

</LinearLayout>
```



Create TrashFragment

When the user clicks on the trash option in the navigation drawer, we'll show TrashFragment. Highlight the `com.hfad.catchat` package in the `app/src/main/java` folder, and create a new blank fragment named "TrashFragment" with a layout called "fragment_trash". Then replace the code for `TrashFragment.java` with ours below:

```
package com.hfad.catchat;

import android.os.Bundle;
import android.support.v4.app.Fragment;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;

public class TrashFragment extends Fragment {

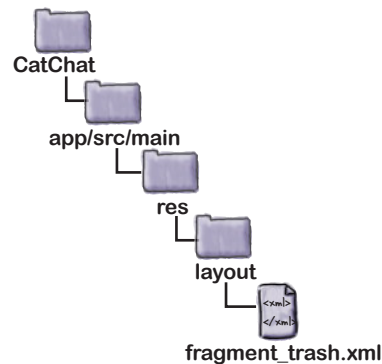
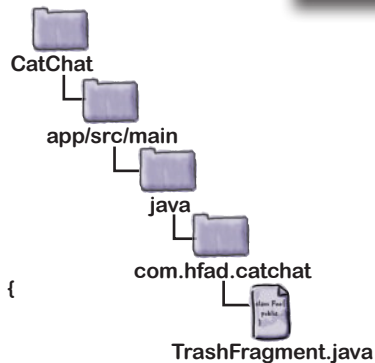
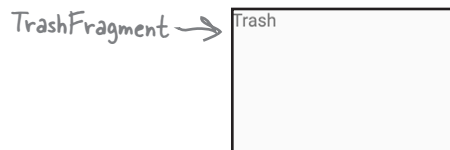
    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup container,
        Bundle savedInstanceState) {
        return inflater.inflate(R.layout.fragment_trash, container, false);
    }
}
```

Next replace the code for `fragment_trash.xml` too:

```
<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="com.hfad.catchat.TrashFragment">

    <TextView
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:text="Trash" />

</LinearLayout>
```



We've now created all the fragments we need. Next, we'll create a toolbar we can include in our activities.



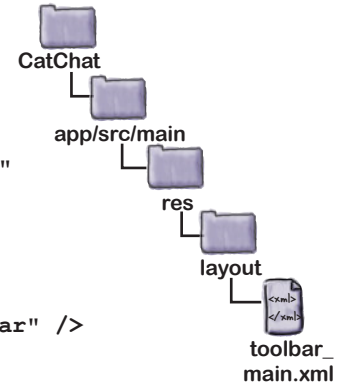
Create a toolbar layout

We're going to add a toolbar in a separate layout so that we can include it in each activity's layout (we'll create our activities soon). Switch to the Project view of Android Studio's explorer, select the `app/src/res/main/layout` folder, then go to the File menu and choose New → Layout resource file. When prompted, name the layout file "toolbar_main", then click on OK.

Next, open `toolbar_main.xml`, and replace the code Android Studio has created for you with the following:

```
<android.support.v7.widget.Toolbar
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="?attr/actionBarSize"
    android:background="?attr/colorPrimary"
    android:theme="@style/ThemeOverlay.AppCompat.Dark.ActionBar" />
```

This is the same toolbar code we've used in previous chapters.



Before we can use the toolbar in any of our activities, we need to change the theme used by your activity. We'll do this in app's style resource.

First, open `AndroidManifest.xml`, and make sure that the value of the theme attribute is set to `@style/AppTheme`. Android Studio may have set this value for you; if not, you'll need to update it to match ours below:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest ...>
  <application
    android:allowBackup="true"
    android:icon="@mipmap/ic_launcher"
    android:label="@string/app_name"
    android:roundIcon="@mipmap/ic_launcher_round"
    android:supportRtl="true"
    android:theme="@style/AppTheme">
    <activity android:name=".MainActivity">
      ...
    </activity>
  </application>
</manifest>
```

Android Studio may have already added this value for you.



We'll update the `AppTheme` style on the next page.

Update the app's theme

Next, we'll update the `AppTheme` style so that it uses a theme of `Theme.AppCompat.Light.NoActionBar`. We'll also override some of the colors that are used in the original theme.

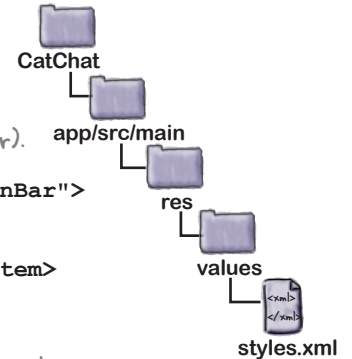
First, open the `app/src/main/res/values` folder and check that Android Studio has created a file for you called `styles.xml`. If this file doesn't exist, you'll need to create it. To do this, select the `values` folder, then go to the File menu and choose `New` → "Values resource file". When prompted, name the file "styles", then click on OK.

Next, update `styles.xml` so that it matches ours:

```
<resources>
  <style name="AppTheme" parent="Theme.AppCompat.Light.NoActionBar">
    <item name="colorPrimary">@color/colorPrimary</item>
    <item name="colorPrimaryDark">@color/colorPrimaryDark</item>
    <item name="colorAccent">@color/colorAccent</item>
  </style>
</resources>
```

This theme removes the default app bar (we're replacing it with a toolbar).

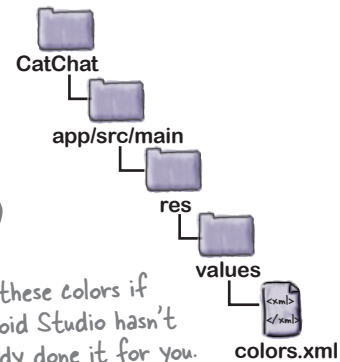
Android Studio may have added these colors for you.



The `AppTheme` style uses color resources, and these need to be included in `colors.xml`. First, make sure that Android Studio has created this file for you in the `app/src/main/res/values` folder (if it hasn't, you'll need to create it yourself). Then update `colors.xml` so that it matches our code below:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <color name="colorPrimary">#3F51B5</color>
  <color name="colorPrimaryDark">#303F9F</color>
  <color name="colorAccent">#FF4081</color>
</resources>
```

Add these colors if Android Studio hasn't already done it for you.



Now that we've set up a style so that we can use a toolbar, we'll create two activities for the help and feedback options in the navigation drawer. We'll display these activities when the user selects the appropriate option.

Create HelpActivity

We'll start by creating `HelpActivity`. Select the `com.hfad.catchat` package in Android Studio, then go to the File menu and choose New. Select the option to create a new empty activity, and give it a name of "HelpActivity", with a layout name of "activity_help". Make sure the package name is `com.hfad.catchat`, and **check the Backwards Compatibility (AppCompat) checkbox**. Then update `activity_help.xml` so that it matches ours 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="com.hfad.catchat.HelpActivity">

    <include
        layout="@layout/toolbar_main"
        android:id="@+id/toolbar" />

    <TextView
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:text="Help" />

</LinearLayout>
```

We're adding a toolbar and "Help" text to HelpActivity.

Next update `HelpActivity.java` to match our version:

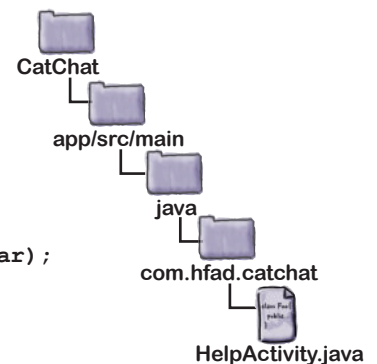
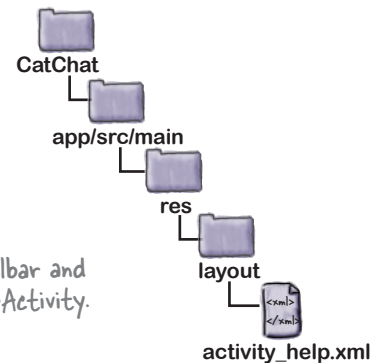
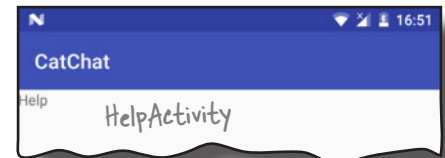
```
package com.hfad.catchat;

import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.support.v7.widget.Toolbar;

public class HelpActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_help);
        Toolbar toolbar = (Toolbar) findViewById(R.id.toolbar);
        setSupportActionBar(toolbar);
    }
}
```

The activity needs to extend AppCompatActivity because we're using an AppCompatActivity theme.



Create FeedbackActivity

Finally, select the `com.hfad.catchat` package again and create an empty activity called “FeedbackActivity”, with a layout name of “activity_feedback”. Make sure the package name is `com.hfad.catchat`, and **check the Backwards Compatibility (AppCompat) checkbox**. Then update `activity_feedback.xml` so that it matches ours 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="com.hfad.catchat.FeedbackActivity">

    <include
        layout="@layout/toolbar_main"
        android:id="@+id/toolbar" />

    <TextView
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:text="Feedback" />

</LinearLayout>
```

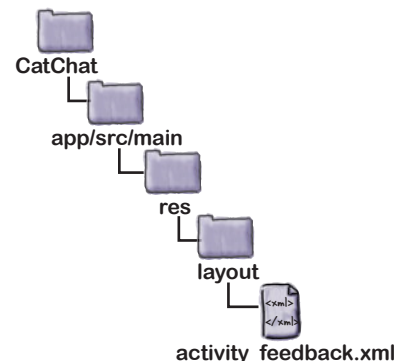
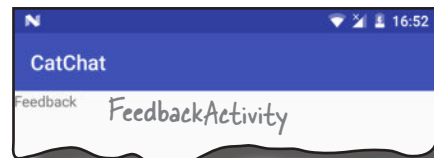
Then update `FeedbackActivity.java` to match this version:

```
package com.hfad.catchat;

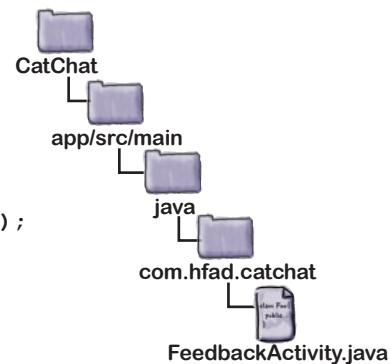
import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.support.v7.widget.Toolbar;

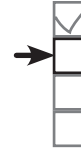
public class FeedbackActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_feedback);
        Toolbar toolbar = (Toolbar) findViewById(R.id.toolbar);
        setSupportActionBar(toolbar);
    }
}
```



This activity needs to extend AppCompatActivity as well.





We need to build a navigation drawer

We've now added all the fragments and activities to our project that the options in the navigation drawer will link to. Next, we'll create the navigation drawer itself.

The navigation drawer comprises two separate components:



A navigation drawer header.

This is a layout that appears at the top of the navigation drawer. It usually consists of an image with some text, for example a photo of the user and their email account.

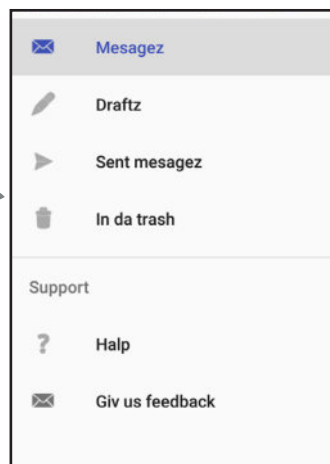
This is the header we'll create. It consists of an image and two pieces of text.



A set of options.

You define a set of options to be displayed in the navigation drawer underneath the header. When the user clicks on one of these options, the screen for that option is displayed as a fragment within the navigation drawer's activity, or as a new activity.

The navigation drawer will contain these options.



We're going to build these components, then use them in `MainActivity` to build a navigation drawer. We'll start with the navigation drawer header.

Create the navigation drawer's header

The navigation drawer's header comprises a simple layout that you add to a new layout file. We're going to use a new file called *nav_header.xml*. Create this file 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".

Our layout is composed of an image and two text views. This means we need to add an image file to our project as a drawable, and two String resources. We'll start with the image file.

Add the image file

To add the image file, first switch to the Project view of Android Studio's explorer if you haven't already done so, and check whether the *app/src/main/res/drawable* folder exists in your project. If it's not already there, select the *app/src/main/res* folder in your project, go to the File menu, choose the New... option, and then click 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.

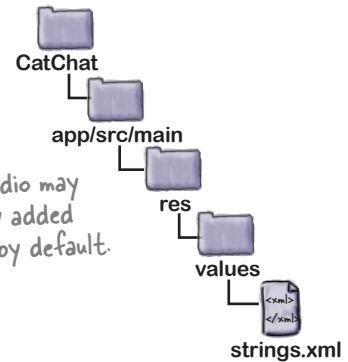
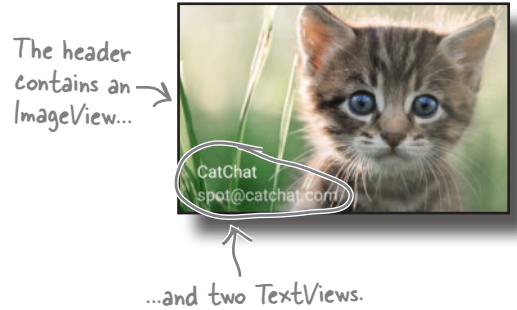
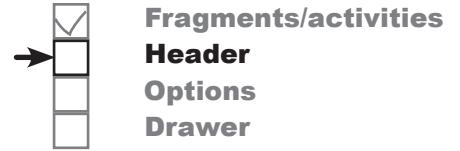
Once you've created the *drawable* folder, download the file *kitten_small.jpg* from <https://git.io/v9oet>, and add it to the *drawable* folder.

Add the String resources

Next, we'll add two String resources, which we'll use for the text views. Open the file *app/src/main/res/values/strings.xml*, then add the following resource:

```
<resources>
...
<string name="app_name">CatChat</string>
<string name="user_name">spot@catchat.com</string>
</resources>
```

Now that you've added the resources, we can write the layout code. You're already familiar with the code we need to do this, so we're going to give you the full code on the next page.





The full nav_header.xml code

Here's our full code for `nav_header.xml`; update your version of the file to match ours:

```

<?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"
    android:theme="@style/ThemeOverlay.AppCompat.Dark" >

    <ImageView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:scaleType="centerCrop"
        android:src="@drawable/kitten_small" />

    <LinearLayout
        android:layout_width="wrap_content"
        android:layout_height="match_parent"
        android:orientation="vertical"
        android:gravity="bottom|start"
        android:layout_margin="16dp" >

        <TextView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="@string/app_name"
            android:textAppearance="@style/TextAppearance.AppCompat.Body1" />

        <TextView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="@string/user_name" />

    </LinearLayout>
</FrameLayout>

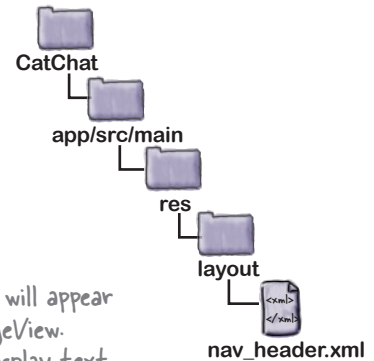
```

We're explicitly setting the height of the layout to 180dp so that it doesn't take up too much space in the drawer.

The image background is quite dark, so we're using this line to make the text light.

This LinearLayout will appear on top of the ImageView. We're using it to display text at the bottom of the image.

This is a built-in style that makes the text look slightly bolder. It comes from the AppCompatActivity Support Library.



Now that we've created the drawer's header, we'll create its list of options.

The drawer gets its options from a menu

The navigation drawer gets its list of options from a menu resource file. The code to do this is similar to that needed to add a set of options to an app bar.

Before we look at the code to add the options to the navigation drawer, we need to add a menu resource file to our project. To do this, select the `app/src/main/res` folder in Android Studio, go to the File menu, and choose New. Then select the option to create a new Android resource file. You'll be prompted for the name of the resource file and the type of resource. Give it a name of "menu_nav", give it a resource type of "Menu", and make sure that the Directory name is "menu". When you click on OK, Android Studio will create the file for you.

Next we'll add String resources for the titles of our menu items so that we can use them later in the chapter. Open `strings.xml` and add the following resources:

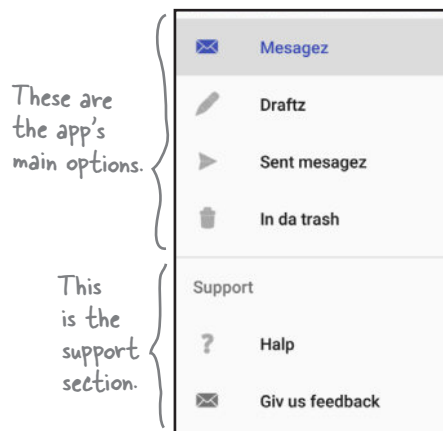
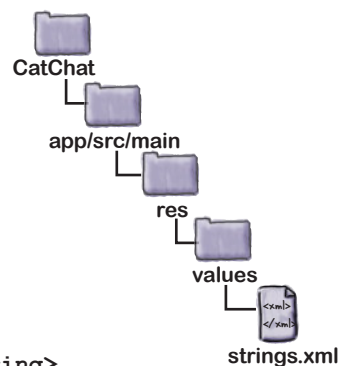
```
<resources>
    ...
    <string name="nav_inbox">Mesagez</string>
    <string name="nav_drafts">Draftz</string>
    <string name="nav_sent">Sent mesagez</string>
    <string name="nav_trash">In da trash</string>
    <string name="nav_support">Support</string>
    <string name="nav_help">Halp</string>
    <string name="nav_feedback">Giv us feedback</string>
</resources>
```

Next we can start constructing our menu code.

We need to create a menu with two sections

As we said earlier, we want to split the items in our navigation drawer into two sections. The first section will contain options for the main places in the app the user will want to visit: her inbox, draft messages, sent items, and trash. We'll then add a separate support section for help and feedback options.

Let's start by adding the main options.





Add items in the order you want them to appear in the drawer

When you design a set of options for a navigation drawer, you generally put the items the user is most likely to want to click on at the top of the list. In our case, these options are for the inbox, draft messages, sent items, and trash.

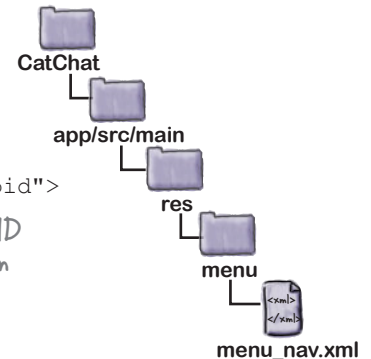
You add items to the menu resource file in the order in which you want them to appear in the drawer. For each item, you specify an ID so you can refer to it in your Java code, and a title for the text you want to appear. You can also specify an icon that will appear alongside the item's text. As an example, here's the code to add an "inbox" item:

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">
  <item
    android:id="@+id/nav_inbox"
    android:icon="@android:drawable/sym_action_email"
    android:title="@string/nav_inbox" />
  ...
</menu>
```

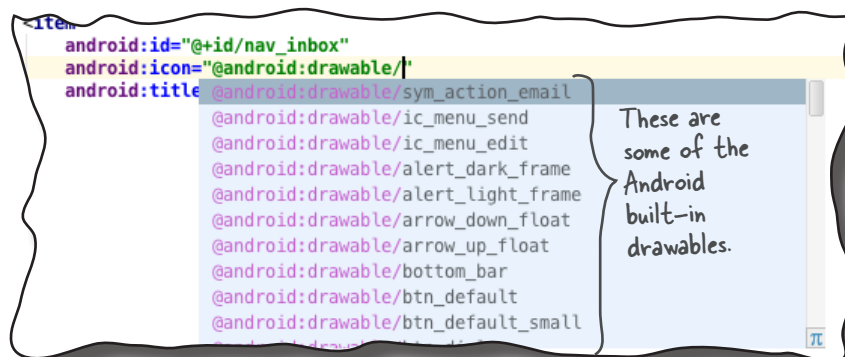
You need to give the item an ID so that your activity code can respond to it being clicked.

This is the text that appears in the navigation drawer.

This is a built-in drawable you can use to display an email icon.



In the above code, we're using one of Android's built-in icons: "@android:drawable/sym_action_email". Android comes with a set of built-in icons that you can use in your apps. The "@android:drawable" part tells Android you want to use one of these icons. You can see the full list of available icons when you start typing the icon name in Android Studio:



How to group items together



Fragments/activities
Header
Options
Drawer

As well as adding menu items individually, you can add them as part of a group. You define a group using the `<group>` element like this:

```
<menu xmlns:android="http://schemas.android.com/apk/res/android">
    <group>
        ... ← Any items you want to
            include in the group go here.
    </group>
</menu>
```

This is useful if you want to apply an attribute to an entire group of items. As an example, you can highlight which item in the drawer the user has selected by setting the group's `android:checkableBehavior` attribute to "single". This behavior is helpful when you intend to display screens for the items as fragments inside the navigation drawer's activity (in our case `MainActivity`), as it makes it easy to tell which option is currently selected:

```
<menu xmlns:android="http://schemas.android.com/apk/res/android">
    <group android:checkableBehavior="single">
        ...
    </group>
</menu>
```

This means that a single item in the group will be highlighted (the option the user selects).

You can highlight an item in the navigation drawer by default by setting its `android:checked` attribute to "true". As an example, here's how you highlight the inbox item:

```
<menu xmlns:android="http://schemas.android.com/apk/res/android">
    <group android:checkableBehavior="single">
        <item
            android:id="@+id/nav_inbox"
            android:icon="@android:drawable/sym_action_email"
            android:title="@string/nav_inbox"
            android:checked="true" />
        ...
    </group>
</menu>
```

This highlights the item in the navigation drawer by default.

We'll show you the full code for the first four menu items on the next page.



We'll use a group for the first section

We're going to add the inbox, drafts, sent messages, and trash options to our menu resource file as a group, and highlight the first item by default. We're using a group for these items because the screen for each option is a fragment, which we'll display in MainActivity.

Here's our code; update your version of `menu_nav.xml` to match ours.

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

```
<group android:checkableBehavior="single"> ← Add this group and the four items it
  contains to your menu resource file so
  they'll appear in the navigation drawer.
```

```
    <item
      android:id="@+id/nav_inbox"
      android:icon="@android:drawable/sym_action_email"
      android:title="@string/nav_inbox"
      android:checked="true" />

    <item
      android:id="@+id/nav_drafts"
      android:icon="@android:drawable/ic_menu_edit"
      android:title="@string/nav_drafts" />

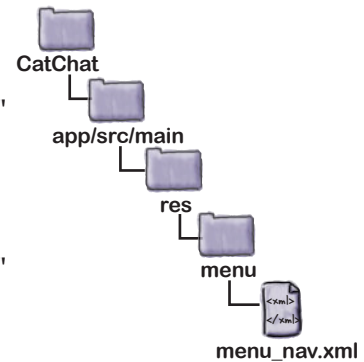
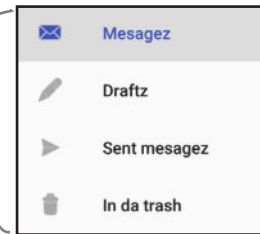
    <item
      android:id="@+id/nav_sent"
      android:icon="@android:drawable/ic_menu_send"
      android:title="@string/nav_sent" />

    <item
      android:id="@+id/nav_trash"
      android:icon="@android:drawable/ic_menu_delete"
      android:title="@string/nav_trash" />
```

```
</group>
```

```
</menu>
```

The code on this page adds these four items.



That's the first group of items sorted. We'll deal with the remaining items next.

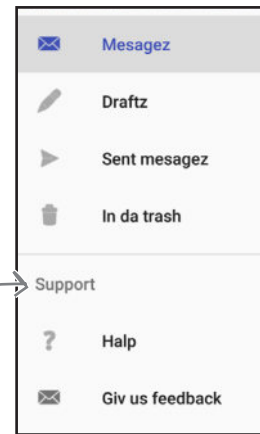
Add the support section as a submenu

The second set of items in the navigation drawer forms a separate section. There's a heading of "Support," along with help and feedback options for the user to click on.

To create this section, we'll start by adding the Support heading as a separate item. As it's a heading, we only need to give it a title; it doesn't need an icon, and we're not assigning it an ID as we don't need it to respond to clicks:

```
...
<item android:title="@string/nav_support">
</item>
...
```

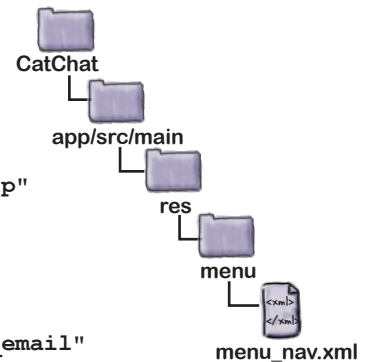
This adds a Support heading to the navigation drawer.



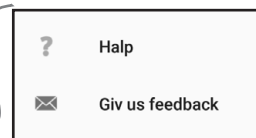
We want the help and feedback options to appear within the Support section, so we'll add them as separate items in a submenu inside the support item:

```
...
<item android:title="@string/nav_support">
  <menu>
    <item
      android:id="@+id/nav_help"
      android:icon="@android:drawable/ic_menu_help"
      android:title="@string/nav_help" />
    <item
      android:id="@+id/nav_feedback"
      android:icon="@android:drawable/sym_action_email"
      android:title="@string/nav_feedback" />
  </menu>
</item>
...
```

This defines a submenu inside the Support item.



The above code adds these two items.



Note that we haven't put these items inside a group, so if the user clicks one of them, it won't be highlighted in the navigation drawer. This is because the help and feedback options will be displayed in new activities, not as fragments in the navigation drawer's activity.

We'll show you the full menu code on the next page.



The full menu_nav.xml code

Here's the full code for `menu_nav.xml`; update your version of the code to match ours:

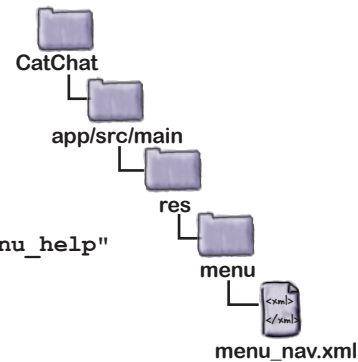
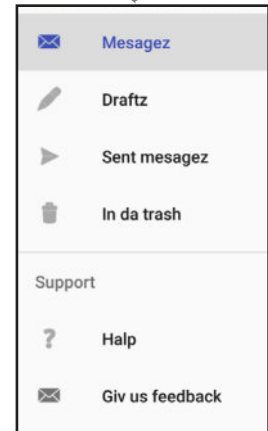
```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">
  <group android:checkableBehavior="single">
    <item
      android:id="@+id/nav_inbox"
      android:icon="@android:drawable/sym_action_email"
      android:title="@string/nav_inbox" />
    <item
      android:id="@+id/nav_drafts"
      android:icon="@android:drawable/ic_menu_edit"
      android:title="@string/nav_drafts" />
    <item
      android:id="@+id/nav_sent"
      android:icon="@android:drawable/ic_menu_send"
      android:title="@string/nav_sent" />
    <item
      android:id="@+id/nav_trash"
      android:icon="@android:drawable/ic_menu_delete"
      android:title="@string/nav_trash" />
  </group>

  <item android:title="@string/nav_support">
    <menu>
      <item
        android:id="@+id/nav_help"
        android:icon="@android:drawable/ic_menu_help"
        android:title="@string/nav_help"/>
      <item
        android:id="@+id/nav_feedback"
        android:icon="@android:drawable/sym_action_email"
        android:title="@string/nav_feedback" />
    </menu>
  </item>
</menu>
```

These are the main options.

This is the support section.

The code on this page creates the full menu.



Now that we've added a menu and navigation drawer header layout, we can create the actual drawer.

How to create a navigation drawer



Fragments/activities
Header
Options
Drawer

You create a navigation drawer by adding a drawer layout to your activity's layout as its root element. The drawer layout 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 as its second:

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.v4.widget.DrawerLayout <— The DrawerLayout defines the drawer.
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:id="@+id/drawer_layout" <— You give it an ID so you can
    android:layout_width="match_parent" refer to it in your activity code.
    android:layout_height="match_parent" >

    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:orientation="vertical" >
        ...
    </LinearLayout>

    <android.support.design.widget.NavigationView <— The NavigationView defines
        android:id="@+id/nav_view" the drawer's contents.
        android:layout_width="wrap_content"
        android:layout_height="match_parent"
        android:layout_gravity="start" <— This attaches the
        app:headerLayout="@layout/nav_header" <— This is the layout for
        app:menu="@menu/menu_nav" /> <— This is the menu resource file
    </android.support.v4.widget.DrawerLayout> <— This is the menu resource file
    containing the drawer's options.
```

The DrawerLayout's first view is a layout for the activity's main content. You see it when the drawer is closed.

This attaches the drawer to the start edge of the activity (the left for left-to-right languages).

There are two key `<NavigationView>` attributes that you use to control the drawer's appearance: `headerLayout` and `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.

You use the `app:menu` attribute to say which menu resource file contains the drawer's options (in this case `menu_drawer.xml`). If you don't include this attribute, your navigation drawer won't include any items.



The full code for activity_main.xml

We're going to add a navigation drawer to MainActivity's layout that uses the header layout and menu we created earlier in the chapter. The layout's main content will comprise a toolbar and frame layout. We'll use the frame layout later in the chapter to display fragments.

Here's our full code for *activity_main.xml*; update your code to match ours:

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.v4.widget.DrawerLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:id="@+id/drawer_layout"
    android:layout_width="match_parent"
    android:layout_height="match_parent" >

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

        <include
            layout="@layout/toolbar_main"
            android:id="@+id/toolbar" />

        <FrameLayout
            android:id="@+id/content_frame"
            android:layout_width="match_parent"
            android:layout_height="match_parent" />

    </LinearLayout>

    <android.support.design.widget.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_nav" />
</android.support.v4.widget.DrawerLayout>
```

The layout's root element is a *DrawerLayout*.

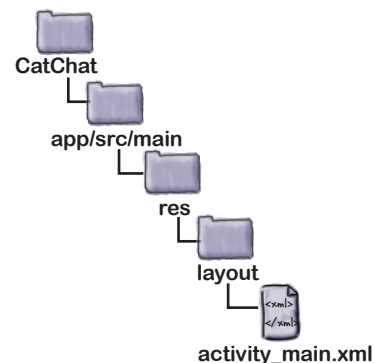
It has an ID so we can refer to it in our activity code later.

This is for the drawer's main content.

The activity's main content is composed of a *Toolbar*, and a *FrameLayout* in which we'll display fragments.

The *NavigationView* defines the drawer's appearance and much of its behavior. We're giving it an ID, as we'll need to refer to it in our activity code.

We're using the layout we created earlier as the drawer's header, and the menu resource file for the list of options.



Before we run the app to see how the navigation drawer's looking, we'll update MainActivity to display *InboxFragment* in the frame layout when the activity gets created.

Add InboxFragment to MainActivity's frame layout

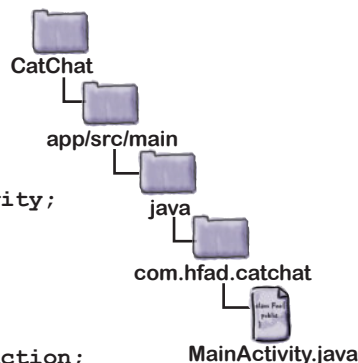


When we created our menu resource file, we set the inbox option to be highlighted by default. We'll therefore display `InboxFragment` in `MainActivity`'s frame layout when the activity is created so that it matches the drawer's contents. We'll also set the toolbar as the activity's app bar so that it displays the app's title.

Here's our code for `MainActivity.java`; replace your version of the code to match ours:

```
package com.hfad.catchat;

import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.support.v7.widget.Toolbar;
import android.support.v4.app.Fragment;
import android.support.v4.app.FragmentTransaction;
```



```
public class MainActivity extends AppCompatActivity {
```

```
    @Override
```

```
    protected void onCreate(Bundle savedInstanceState) {
```

```
        super.onCreate(savedInstanceState);
```

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

```
        Toolbar toolbar = (Toolbar) findViewById(R.id.toolbar);
```

```
        setSupportActionBar(toolbar);
```

Make sure the activity extends the `AppCompatActivity` class, as we're using an `AppCompatActivity` theme and support fragments.

```
        Fragment fragment = new InboxFragment();
```

```
        FragmentTransaction ft = getSupportFragmentManager().beginTransaction();
```

```
        ft.add(R.id.content_frame, fragment);
```

```
        ft.commit();
```

```
    }
```

```
}
```

Use a fragment transaction to display an instance of `InboxFragment`.

{

```
        Fragment fragment = new InboxFragment();
```

```
        FragmentTransaction ft = getSupportFragmentManager().beginTransaction();
```

```
        ft.add(R.id.content_frame, fragment);
```

```
        ft.commit();
```

```
    }
```

```
}
```

Set the `Toolbar` as the activity's app bar.

Let's see what happens when we run the app.



Test drive the app

When we run the app, `InboxFragment` is displayed in `MainActivity`. When you swipe the app from the left side of the screen (in left-to-right languages like English) the navigation drawer is displayed. The navigation drawer contains a header layout, and the list of options we defined in our menu resource file. The first option is automatically highlighted:



navigation drawers

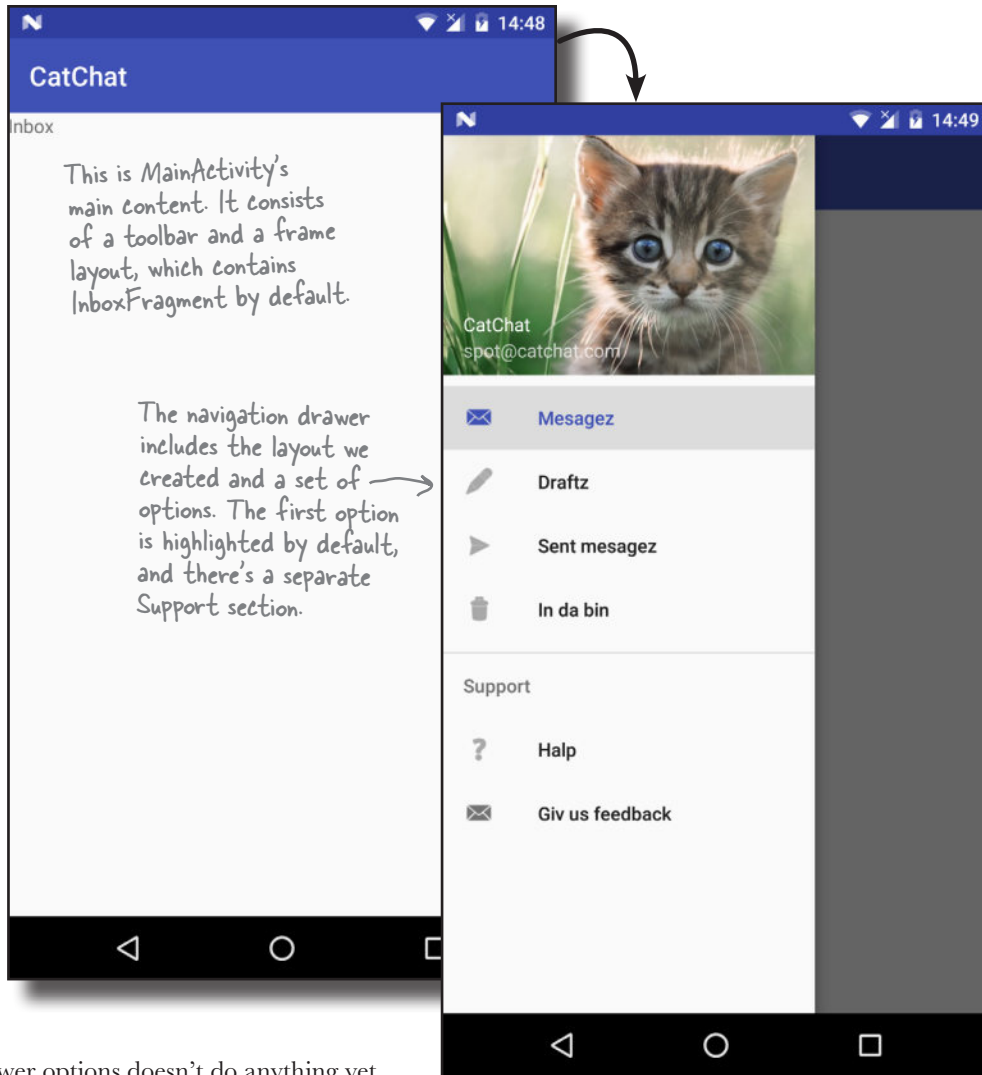
Fragments/activities

Header

Options

Drawer

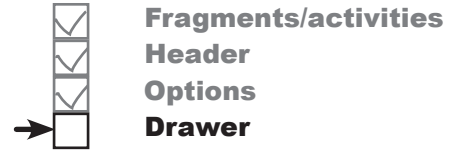
← In right-to-left languages, the drawer will appear on the right side of the screen instead.



Clicking on the drawer options doesn't do anything yet, as we haven't written any code in `MainActivity` to control how the drawer operates. We'll do that next.

What the activity code needs to do

There are three things we need our activity code to do:



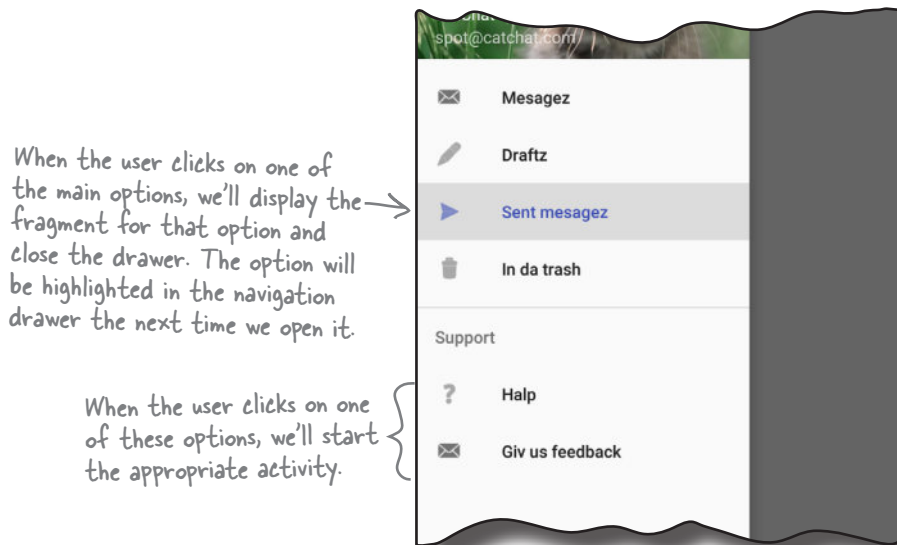
1 Add a drawer toggle.

This provides a visual sign to the user that the activity contains a navigation drawer. It adds a “burger” icon to the toolbar, and you can click on this icon to open the drawer.



2 Make the drawer respond to clicks.

When the user clicks on one of the options in the navigation drawer, we’ll display the appropriate fragment or activity and close the drawer.



3 Close the drawer when the user presses the Back button.

If the drawer’s open, we’ll close it when the user clicks on the Back button. If the drawer’s already closed, we’ll get the Back button to function as normal.

We’ll start by adding the drawer toggle.

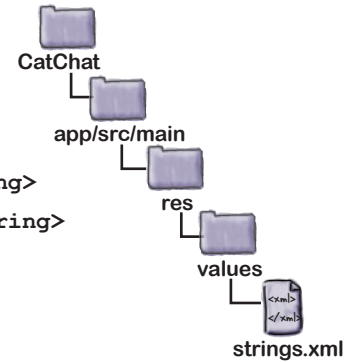


Add a drawer toggle

The first thing we'll do is add a drawer toggle so that we can open the navigation drawer by clicking on an icon in the toolbar.

We'll start by creating two String resources to describe the "open drawer" and "close drawer" actions; these are required for accessibility purposes. Add the two Strings below to *strings.xml*:

```
<string name="nav_open_drawer">Open navigation drawer</string>
<string name="nav_close_drawer">Close navigation drawer</string>
```



You create the drawer toggle in the activity's *onCreate()* method by creating a new instance of the **ActionBarDrawerToggle** class and adding it to the drawer layout. We'll show you the code for this first, then add it to *MainActivity* later in the chapter.

The *ActionBarDrawerToggle* constructor takes five parameters: the current activity, the drawer layout, the toolbar, and the IDs of two String resources for opening and closing the drawer (the String resources we added above):

```
Toolbar toolbar = (Toolbar) findViewById(R.id.toolbar);
...
```

```
DrawerLayout drawer = (DrawerLayout) findViewById(R.id.drawer_layout);
```

```
ActionBarDrawerToggle toggle = new ActionBarDrawerToggle(this, ← The current activity
```

```
drawer, ← The activity's DrawerLayout
```

```
toolbar, ← The activity's toolbar
```

```
R.string.nav_open_drawer,
R.string.nav_close_drawer);
```

↑
This adds the burger icon to your toolbar.

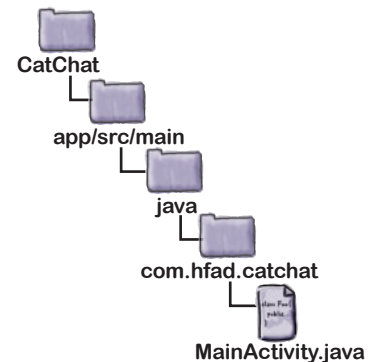
These Strings are needed for accessibility.

Once you've created the drawer toggle, you add it to the drawer layout by calling the *DrawerLayout addDrawerListener()* method, passing the toggle as a parameter:

```
drawer.addDrawerListener(toggle);
```

Finally, you call the toggle's *syncState()* method to synchronize the icon on the toolbar with the state of the drawer. This is because the icon changes when you click on it to open the drawer:

```
toggle.syncState();
```



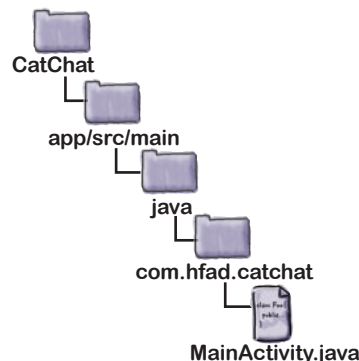
We'll add the drawer toggle to *MainActivity*'s *onCreate()* method in a few pages.

Respond to the user clicking items in the drawer



Next, we'll get `MainActivity` to respond to items in the navigation drawer being clicked by getting the activity to implement a `NavigationView.OnNavigationItemSelectedListener` interface. Doing this means that whenever an item is clicked, a new method we'll create in `MainActivity`, `onNavigationItemSelectedListener()`, will get called. We'll use this method to display the screen for the appropriate option.

First, we'll get `MainActivity` to implement the interface using the code below. This code turns `MainActivity` into a listener for the navigation view:



```
...
import android.support.design.widget.NavigationView;

public class MainActivity extends AppCompatActivity
    implements NavigationView.OnNavigationItemSelectedListener {
    ...
}
```

↑
Implementing this interface means that your activity can respond to the user clicking options in the navigation drawer.

Next we need to register the listener, `MainActivity`, with the navigation view so that it will be notified when the user clicks on one of the options in the drawer. We'll do this by getting a reference to the navigation view in the activity's `onCreate()` method, and calling its `setNavigationItemSelectedListener()` method:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    ...
    NavigationView navigationView = (NavigationView) findViewById(R.id.nav_view);
    navigationView.setNavigationItemSelectedListener(this);
}
```

↑
This registers the activity as a listener on the navigation view so it will be notified if the user clicks on an item.

Finally, we need to implement the `onNavigationItemSelectedListener()` method.



Implement the `onNavigationItemSelectedListener()` method

The `onNavigationItemSelectedListener()` method gets called when the user clicks on one of the items in the navigation drawer. It takes one parameter, the `MenuItem` that was clicked, and returns a boolean to indicate whether the item in the drawer should be highlighted:

```
@Override
public boolean onNavigationItemSelectedListener(MenuItem item) {
    //Code to handle navigation clicks
}
```

This method gets called whenever an item in the drawer is clicked. Its parameter is the clicked item.

The code in this method needs to display the appropriate screen for the clicked item. If the item is an activity, the code needs to start it with an intent. If the item is a fragment, it needs to be displayed in `MainActivity`'s frame layout using a fragment transaction.

When you display fragments by clicking on an item in a navigation drawer, you don't generally add the transaction to the back stack as we did previously. This is because when the user clicks on the Back button, they don't expect to revisit every option they clicked on in the drawer. Instead, you use code like this:

```
FragmentTransaction ft = getSupportFragmentManager().beginTransaction();
ft.replace(R.id.content_frame, fragment);
ft.commit();
```

This is the same fragment transaction code you've seen before except that we're not adding the transaction to the activity's back stack.

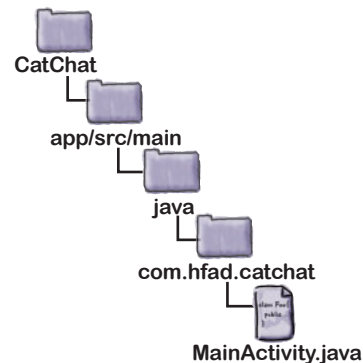
Finally, you need to close the drawer. To do this, you get a reference to the drawer layout, and call its `closeDrawer()` method:

```
DrawerLayout drawer = (DrawerLayout) findViewById(R.id.drawer_layout);
drawer.closeDrawer(GravityCompat.START);
```

We're using `GravityCompat.START` because we've attached the drawer to the activity's start edge. If we'd attached it to the end edge, we'd use `GravityCompat.END` instead.

This closes the drawer so that it slides back to the activity's start edge.

You now know everything you need in order to write the code for the `onNavigationItemSelectedListener()` method, so have a go at the following exercise.





Code Magnets

When the user clicks on an item in the navigation drawer, we need to display the appropriate screen for that item. If it's a fragment, we need to display it in the `content_frame` frame layout. If it's an activity, we need to start it. Finally, we need to close the navigation drawer.

See if you can complete the code below and on the next page. You won't need to use all of the magnets.

```
@Override
public boolean onNavigationItemSelected(MenuItem item) {

    int id = item. ....;
    Fragment fragment = null;
    Intent intent = null;

    switch( ..... ){
        case R.id.nav_drafts:

            fragment = .....;

            .....;
        case R.id.nav_sent:

            fragment = .....;

            .....;
        case R.id.nav_trash:

            fragment = .....;

            .....;
        case R.id.nav_help:

            intent = new Intent(.....,.....);

            .....;
    }
}
```

```

case R.id.nav_feedback:

    intent = new Intent(.....,.....);

    .....;
default:

    fragment = .....;
}

if (.....!= null) {

    FragmentTransaction ft = getSupportFragmentManager()......;

    ft.replace(R.id.content_frame, .....);

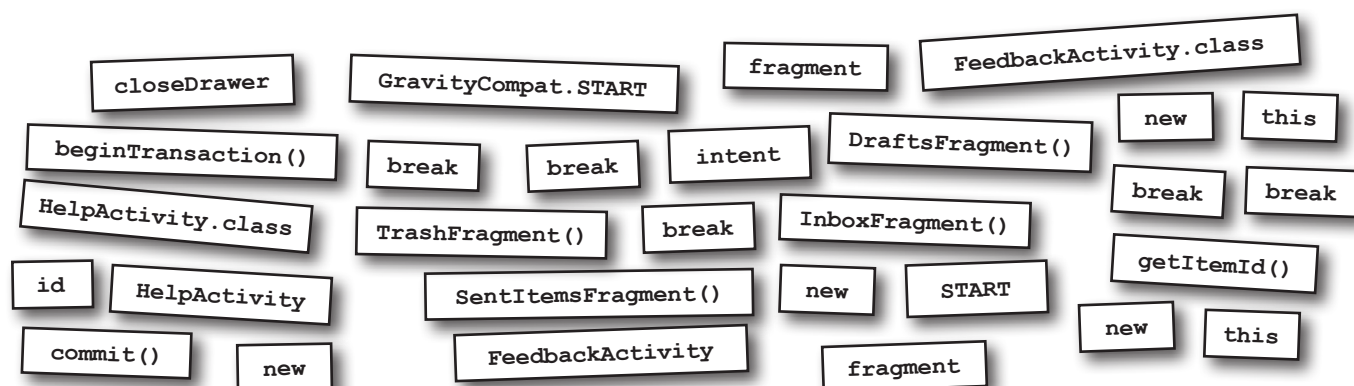
    ft.....;
} else {

    startActivity(.....);
}

DrawerLayout drawer = (DrawerLayout) findViewById(R.id.drawer_layout);

drawer.....(.....);
return true;
}

```





Code Magnets Solution

When the user clicks on an item in the navigation drawer, we need to display the appropriate screen for that item. If it's a fragment, we need to display it in the `content_frame` frame layout. If it's an activity, we need to start it. Finally, we need to close the navigation drawer.

See if you can complete the code below and on the next page. You won't need to use all of the magnets.

```

@Override
public boolean onNavigationItemSelected(MenuItem item) {
    int id = item.getItemId();
    Fragment fragment = null;
    Intent intent = null;

    switch(id) {
        case R.id.nav_drafts:
            fragment = new DraftsFragment();
            break;
        case R.id.nav_sent:
            fragment = new SentItemsFragment();
            break;
        case R.id.nav_trash:
            fragment = new TrashFragment();
            break;
        case R.id.nav_help:
            intent = new Intent(this, HelpActivity.class);
            break;
    }
}

```

Get the ID of the item that was selected.

Save an instance of the fragment we want to display in the fragment variable.

Construct an intent to start HelpActivity if the Help option's clicked.

```

case R.id.nav_feedback:
    intent = new Intent(this, FeedbackActivity.class);
    break;
default:
    fragment = new InboxFragment();
}

if (fragment != null) {
    FragmentTransaction ft = getSupportFragmentManager().beginTransaction();
    ft.replace(R.id.content_frame, fragment);
    ft.commit();
} else {
    startActivity(intent);
}

DrawerLayout drawer = (DrawerLayout) findViewById(R.id.drawer_layout);
drawer.closeDrawer(GravityCompat.START);
return true;
}
    
```

If the feedback option is clicked, we need to start FeedbackActivity.

Display InboxFragment by default, as it's the first option in the drawer.

If we need to display a fragment, use a fragment transaction.

If we need to display an activity, use the intent we constructed to start it.

Finally, close the drawer.

We'll add this code to MainActivity.java in a couple of pages.



Close the drawer when the user presses the Back button

Finally, we'll override what happens when the Back button's pressed. If the user presses the Back button when the navigation drawer's open, we'll close the drawer. If the drawer's already closed we'll get the Back button to function as normal.

To do this, we'll implement the activity's `onBackPressed()` method, which gets called whenever the user clicks on the Back button. Here's the code:

```

@Override
public void onBackPressed() {
    DrawerLayout drawer = (DrawerLayout) findViewById(R.id.drawer_layout);
    if (drawer.isDrawerOpen(GravityCompat.START)) {
        drawer.closeDrawer(GravityCompat.START);
    } else {
        super.onBackPressed();
    }
}

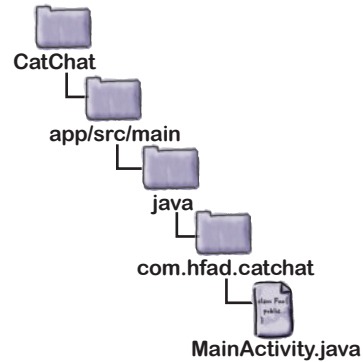
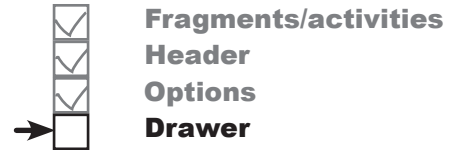
```

This gets called when the Back button gets pressed.

If the drawer is currently open, close it.

Otherwise, call up to the superclass `onBackPressed()` method.

That's everything we need for `MainActivity`. We'll show you the full code over the next couple of pages, and then take it for a test drive.



there are no Dumb Questions

Q: Do you have to use a navigation view for your drawer contents?

A: No, but it's *much* easier if you do. Before the Android Design Library came out, it was common practice to use a list view instead. This approach is still possible, but it requires a lot more code.

Q: Can your activity contain more than one navigation drawer?

A: Your activity can have one navigation drawer per vertical edge of its layout. To add a second navigation drawer, add an extra navigation view to your drawer layout underneath the first.

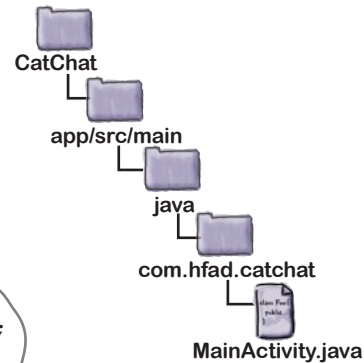


The full MainActivity.java code

Here's the full code for *MainActivity.java*; update your version of the code to match ours:

```
package com.hfad.catchat;

import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.support.v7.widget.Toolbar;
import android.support.v4.app.Fragment;
import android.support.v4.app.FragmentTransaction;
import android.support.v4.widget.DrawerLayout;
import android.support.v7.app.ActionBarDrawerToggle;
import android.support.design.widget.NavigationView;
import android.view.MenuItem;
import android.content.Intent;
import android.support.v4.view.GravityCompat;
```



We're using these extra classes so we need to import them.

```
public class MainActivity extends AppCompatActivity
    implements NavigationView.OnNavigationItemSelectedListener {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        Toolbar toolbar = (Toolbar) findViewById(R.id.toolbar);
        setSupportActionBar(toolbar);

        DrawerLayout drawer = (DrawerLayout) findViewById(R.id.drawer_layout);
        ActionBarDrawerToggle toggle = new ActionBarDrawerToggle(this,
            drawer,
            toolbar,
            R.string.nav_open_drawer,
            R.string.nav_close_drawer);
        drawer.addDrawerListener(toggle);
        toggle.syncState();
```

Implementing this interface means the activity can listen for clicks.

Add a drawer toggle.

The code continues on the next page. →

MainActivity.java (continued)



Fragments/activities
Header
Options
Drawer

```
NavigationView navigationView = (NavigationView) findViewById(R.id.nav_view);
navigationView.setNavigationItemSelectedListener(this);
```

Register the activity with the navigation view as a listener.

```
Fragment fragment = new InboxFragment();
FragmentManager ft = getSupportFragmentManager().beginTransaction();
ft.add(R.id.content_frame, fragment);
ft.commit();
}
```

@Override

```
public boolean onNavigationItemSelectedListener(MenuItem item) {
```

```
    int id = item.getItemId();
```

```
    Fragment fragment = null;
```

```
    Intent intent = null;
```

```
    switch(id){
```

```
        case R.id.nav_drafts:
```

```
            fragment = new DraftsFragment();
```

```
            break;
```

```
        case R.id.nav_sent:
```

```
            fragment = new SentItemsFragment();
```

```
            break;
```

```
        case R.id.nav_trash:
```

```
            fragment = new TrashFragment();
```

```
            break;
```

```
        case R.id.nav_help:
```

```
            intent = new Intent(this, HelpActivity.class);
```

```
            break;
```

```
        case R.id.nav_feedback:
```

```
            intent = new Intent(this, FeedbackActivity.class);
```

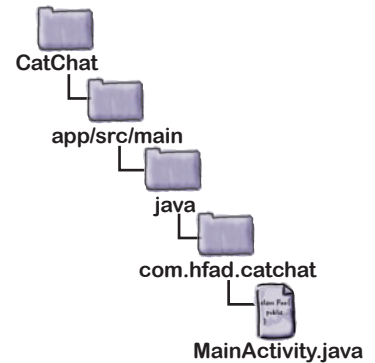
```
            break;
```

```
        default:
```

```
            fragment = new InboxFragment();
```

```
    }
```

This method gets called when the user clicks on one of the items in the drawer.



The code continues on the next page. →



MainActivity.java (continued)

```

if (fragment != null) {
    FragmentTransaction ft = getSupportFragmentManager().beginTransaction();
    ft.replace(R.id.content_frame, fragment);
    ft.commit();
} else {
    startActivity(intent);
}

DrawerLayout drawer = (DrawerLayout) findViewById(R.id.drawer_layout);
drawer.closeDrawer(GravityCompat.START);
return true;
}

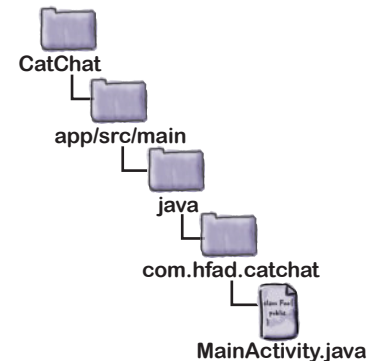
@Override
public void onBackPressed() {
    DrawerLayout drawer = (DrawerLayout) findViewById(R.id.drawer_layout);
    if (drawer.isDrawerOpen(GravityCompat.START)) {
        drawer.closeDrawer(GravityCompat.START);
    } else {
        super.onBackPressed();
    }
}
}

```

Display the appropriate fragment or activity, depending on which option in the drawer the user selects.

Close the drawer when the user selects one of the options.

When the user presses the Back button, close the drawer if it's open.



Let's see what happens when we run the code.



Test drive the app

When we run the app, a drawer toggle icon is displayed in the toolbar. Clicking on this icon opens the navigation drawer. When we click on one of the first four options, the fragment for that option is displayed in `MainActivity` and the drawer closes; the option for that item is highlighted the next time we open the drawer. When we click on one of the last two options, the activity for that option is started.



Fragments/activities
Header
Options
Drawer

MainActivity includes a drawer toggle. Clicking on it opens the drawer.

When we click on the Sent mesagez option, SentItemsFragment gets displayed and the drawer closes. The option is highlighted in the drawer the next time we open it.

When we click on the Halp option, HelpActivity gets displayed and the drawer closes.

We've created a fully operational navigation drawer.



Your Android Toolbox

You've got **Chapter 14** under your belt and now you've added navigation drawers to your toolbox.

You can download the full code for the chapter from <https://tinyurl.com/HeadFirstAndroid>.



BULLET POINTS

- Use a navigation drawer if you want to provide the user with a large number of shortcuts, or group them 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, usually a layout containing a `ToolBar` and `FrameLayout`. Its second element defines the contents of the drawer, usually a `NavigationView`.
- The `NavigationView` comes from the Design Support Library. It controls most of the drawer's behavior.
- You add a header to your drawer by creating a layout for it, and adding the header's resource ID to the navigation view's `headerLayout` attribute.
- You add items to the drawer by creating a menu resource, and adding the menu's resource ID to the navigation view's `menu` attribute.
- Add items to the menu resource in the order in which you want them to appear in the drawer.
- If you want to highlight which item in the drawer the user selects, add the menu items to a group and set the group's `checkableBehavior` attribute to `"single"`.
- Use an `ActionBarDrawerToggle` to display a "burger" icon in the activity's toolbar. This provides a visual sign that the activity has a navigation drawer. Clicking on it opens the drawer.
- Respond to the user clicking on items in the drawer by making your activity implement the `NavigationView.OnNavigationItemSelectedListener` interface. Register the activity with the navigation view as a listener, then implement the `onNavigationItemSelectedListener()` method.
- Close the navigation drawer using the `DrawerLayout.closeDrawer()` method.

15 SQLite databases

★ *Fire Up the Database* ★



When I said I wanted persistence, I was talking about the database.

If you're recording high scores or saving tweets, your app will need to store data. And on Android you usually keep your data safe inside a **SQLite database**. In this chapter, we'll show you how to **create a database, add tables to it, and populate it with data**, all with the help of the friendly **SQLite helper**. You'll then see how you can cleanly roll out **upgrades** to your database structure, and how to **downgrade** it if you need to undo any changes.

Back to Starbuzz

Back in Chapter 7, we created an app for Starbuzz Coffee. The app allows the user to navigate through a series of screens so that she can see the drinks available at Starbuzz.



The Starbuzz database gets its drink data from a `Drink` class containing a selection of drinks available at Starbuzz. While this made building the first version of the app easier, there's a better way of storing and persisting data.

Over the next two chapters, we're going to change the Starbuzz app so that it gets its data from a SQLite database. In this chapter, we'll see how to create the database, and in the next chapter, we'll show you how to connect activities to it.

Android uses SQLite databases to persist data

All apps need to store data, and the main way you do that in Androidville is with a **SQLite database**. Why SQLite?



It's lightweight.

Most database systems need a special database server process in order to work. SQLite doesn't; a SQLite database is just a file. When you're not using the database, it doesn't use up any processor time. That's important on a mobile device, because we don't want to drain the battery.



It's optimized for a single user.

Our app is the only thing that will talk to the database, so we shouldn't have to identify ourselves with a username and password.



It's stable and fast.

SQLite databases are amazingly stable. They can handle database transactions, which means if you're updating several pieces of data and mess up, SQLite can roll the data back. Also, the code that reads and writes the data is written in optimized C code. Not only is it fast, but it also reduces the amount of processor power it needs.

We're going to go through the basics of SQLite in this chapter.

If you plan on doing a lot of database heavy lifting in your apps, we suggest you do more background reading on SQLite and SQL.

Where's the database stored?

Android automatically creates a folder for each app where the app's database can be stored. When we create a database for the Starbuzz app, it will be stored in the following folder on the device:

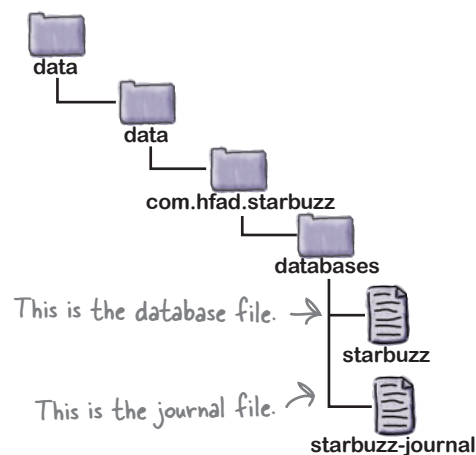
`/data/data/com.hfad.starbuzz/databases`

com.hfad.starbuzz is the app's unique identifier.

An app can store several databases in this folder. Each database consists of two files.

The first file is the **database file** and has the same name as your database—for example, “starbuzz”. This is the main SQLite database file. All of your data is stored in this file.

The second file is the **journal file**. It has the same name as your database, with a suffix of “-journal”—for example, “starbuzz-journal”. The journal file contains all of the changes made to your database. If there's a problem, Android will use the journal to undo your latest changes.



Android comes with SQLite classes

Android uses a set of classes that allows you to manage a SQLite database. There are three types of object that do the bulk of this work:



The SQLite Helper

A SQLite helper enables you to create and manage databases. You create one by extending the `SQLiteOpenHelper` class.

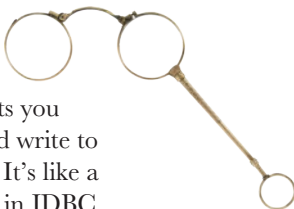


The SQLite Database

The `SQLiteDatabase` class gives you access to the database. It's like a `SqlConnection` in JDBC.

Cursors

A `Cursor` lets you read from and write to the database. It's like a `ResultSet` in JDBC.



We're going to use these objects to show you how to create a SQLite database your app can use to persist data by replacing the `Drink` class with a SQLite database.

there are no Dumb Questions

Q: If there's no username and password on the database, how is it kept secure?

A: The directory where an app's databases are stored is only readable by the app itself. The database is secured down at the operating system level.

Q: Can I write an Android app that talks to some other kind of external database, such as Oracle?

A: There's no reason why you can't access other databases over a network connection, but be careful to conserve the resources used by Android. For example, you might use less battery power if you access your database via a web service. That way, if you're not talking to the database, you're not using up any resources.

Q: Why doesn't Android use JDBC to access SQLite databases?

A: We know we're going to be using a SQLite database, so using JDBC would be overkill. Those layers of database drivers that make JDBC so flexible would just drain the battery on an Android device.

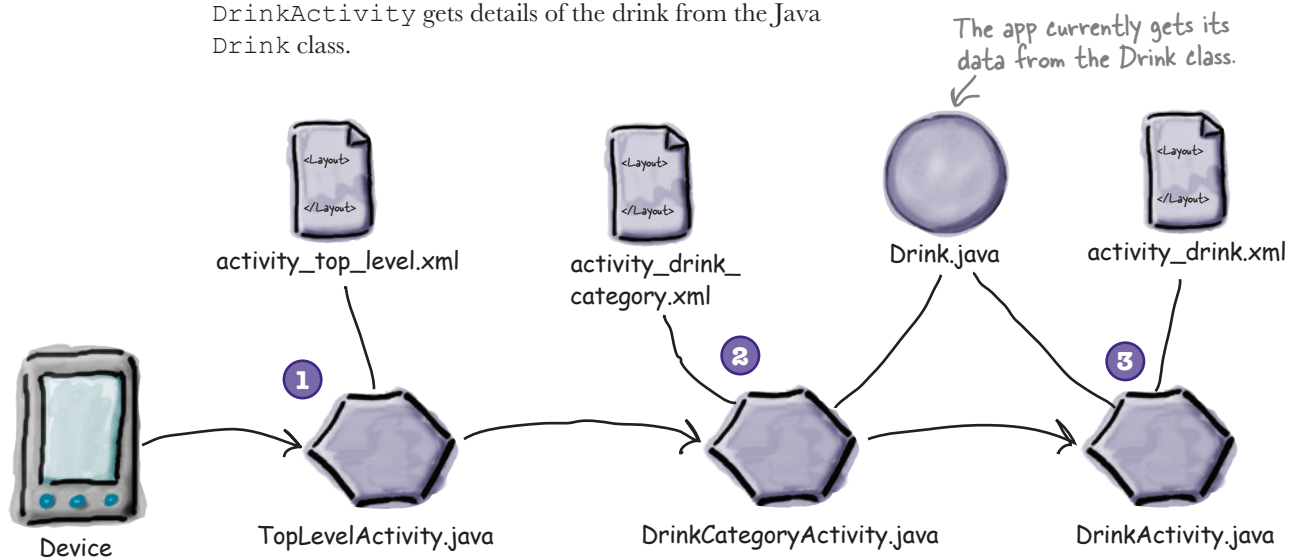
Q: Is the database directory inside the app's directory?

A: No. It's kept in a separate directory from the app's code. That way, the app can be overwritten with a newer version, but the data in the database will be kept safe.

The current Starbuzz app structure

Here's a reminder of the current structure of the Starbuzz app:

- 1 **TopLevelActivity displays a list of options: Drinks, Food, and Stores.**
- 2 **When the user clicks on the Drinks option, it launches DrinkCategoryActivity.**
This activity displays a list of drinks that it gets from the Java Drink class.
- 3 **When the user clicks on a drink, its details get displayed in DrinkActivity.**
DrinkActivity gets details of the drink from the Java Drink class.



How does the app structure need to change if we're to use a SQLite database?

Do this!

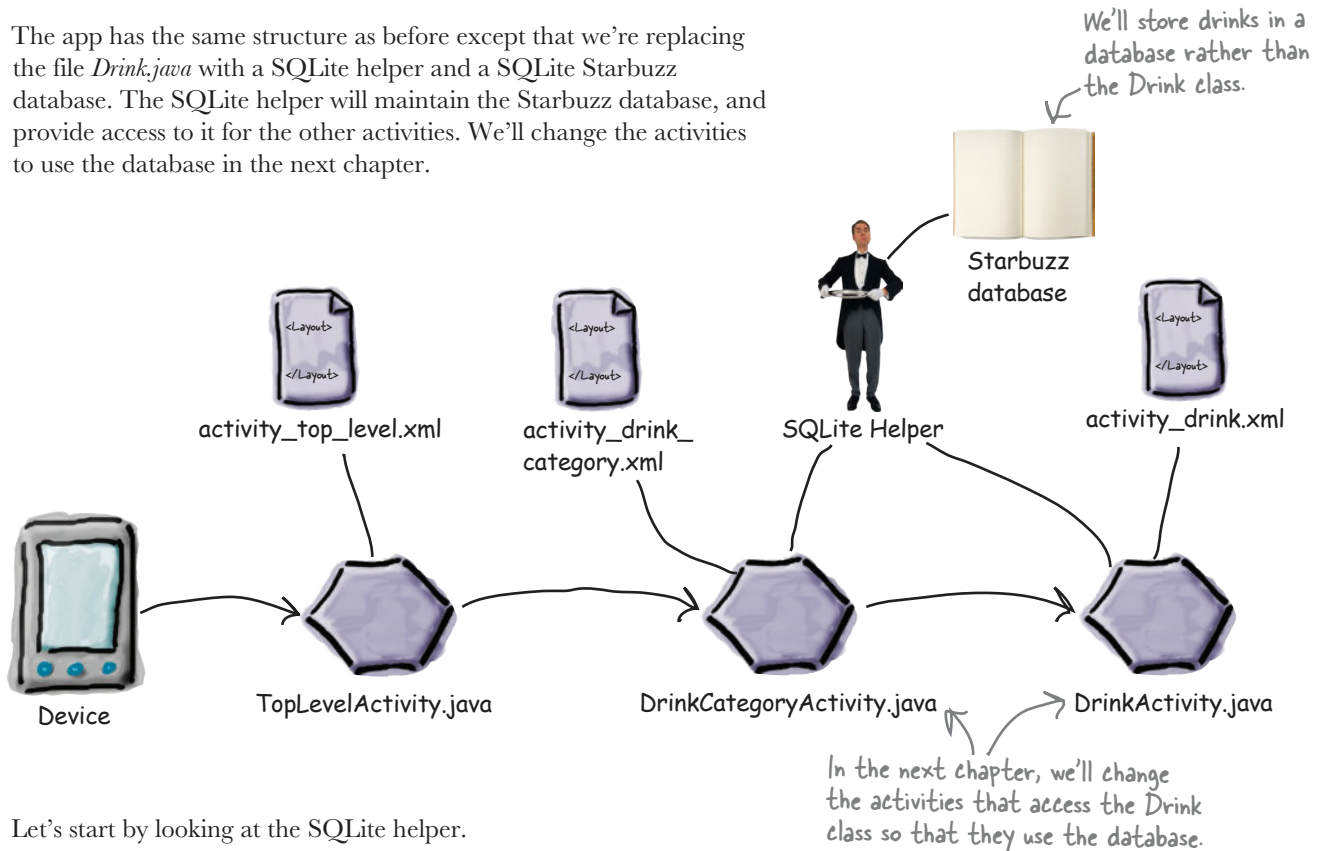
We're going to update the Starbuzz app in this chapter, so open your original Starbuzz project in Android Studio.

Let's change the app to use a database

We'll use a SQLite helper to create a SQLite database we can use with our Starbuzz app. We're going to replace our `Drink` Java class with a database, so we need our SQLite helper to do the following:

- 1 **Create the database.**
Before we can do anything else, we need to get the SQLite helper to create version 1 (the first version) of our Starbuzz database.
- 2 **Create the `Drink` table and populate it with drinks.**
Once we have a database, we can create a table in it. The table's structure needs to reflect the attributes in the current `Drink` class, so it needs to be able to store the name, description, and image resource ID of each drink. We'll then add three drinks to it.

The app has the same structure as before except that we're replacing the file `Drink.java` with a SQLite helper and a SQLite Starbuzz database. The SQLite helper will maintain the Starbuzz database, and provide access to it for the other activities. We'll change the activities to use the database in the next chapter.



Let's start by looking at the SQLite helper.



The SQLite helper manages your database

The `SQLiteOpenHelper` class is there to help you create and maintain your SQLite databases. Think of it as a personal assistant who takes care of the general database housekeeping.

Let's look at some typical tasks that the SQLite helper can assist you with:

Creating the database

When you first install an app, the database file won't exist. The SQLite helper will make sure the database file is created with the correct name and with the correct table structures installed.

Getting access to the database

Our app shouldn't need to know all of the details about where the database file is, so the SQLite helper can serve us with an easy-to-use database object whenever we need it. At all hours, day or night.



The SQLite helper



Keeping the database shipshape

The structure of the database will probably change over time, and the SQLite helper can be relied upon to convert an old version of a database into a shiny, spiffy new version, with all the latest database structures it needs.



Create the SQLite helper

You create a SQLite helper by writing a class that extends the **SQLiteOpenHelper** class. When you do this, you **must** override the `onCreate()` and `onUpgrade()` methods. These methods are mandatory.

The `onCreate()` method gets called when the database first gets created on the device. The method should include all the code needed to create the tables you need for your app.

The `onUpgrade()` method gets called when the database needs to be upgraded. As an example, if you need to modify the structure of the database after it's been released, this is the method to do it in.

In our app, we're going to use a SQLite helper called `StarbuzzDatabaseHelper`. Create this class in your `Starbuzz` project by switching to the Project view of Android Studio's explorer, selecting the `com.hfad.starbuzz` package in the `app/src/main/java` folder, and navigating to `File`→`New`...→`Java Class`. Name the class "`StarbuzzDatabaseHelper`", make sure the package name is `com.hfad.starbuzz` and then replace its contents with the code below:

```
package com.hfad.starbuzz;

import android.database.sqlite.SQLiteOpenHelper;
import android.content.Context;
import android.database.sqlite.SQLiteDatabase;

class StarbuzzDatabaseHelper extends SQLiteOpenHelper {

    StarbuzzDatabaseHelper(Context context) {
    }

    @Override
    public void onCreate(SQLiteDatabase db) {
    }

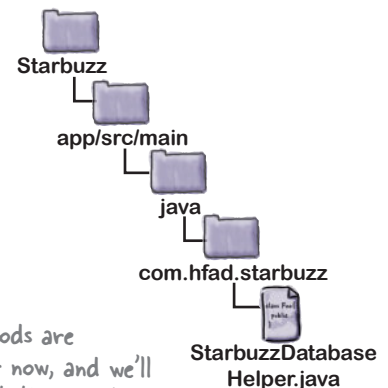
    @Override
    public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
    }
}
```

This is the full path of the SQLiteOpenHelper class.

SQLite helpers must extend the SQLiteOpenHelper class.

We'll write the code for the constructor on the next page.

The onCreate() and onUpgrade() methods are mandatory. We've left them empty for now, and we'll look at them in more detail throughout the chapter.



To get the SQLite helper to do something, we need to add code to its methods. The first thing to do is tell the SQLite helper what database it needs to create.



Specify the database

There are two pieces of information the SQLite helper needs in order to create the database.

First, we need to give the database a name. By giving the database a name, we make sure that the database remains on the device when it's closed. If we don't, the database will only be created in memory, so once the database is closed, it will disappear.

Creating databases that are only held in memory can be useful when you're testing your app.

The second piece of information we need to provide is the version of the database. The database version needs to be an integer value, starting at 1. The SQLite helper uses this version number to determine whether the database needs to be upgraded.

You specify the database name and version by passing them to the constructor of the `SQLiteOpenHelper` superclass. We're going to give our database a name of "starbuzz", and as it's the first version of the database, we'll give it a version number of 1. Here's the code we need (update your version of `StarbuzzDatabaseHelper.java` to match the code below):



Name: "starbuzz"
Version: 1

SQLite database

```

...
class StarbuzzDatabaseHelper extends SQLiteOpenHelper {

    private static final String DB_NAME = "starbuzz"; // the name of our database
    private static final int DB_VERSION = 1; // the version of the database

    StarbuzzDatabaseHelper(Context context) {
        super(context, DB_NAME, null, DB_VERSION);
    }
}
...

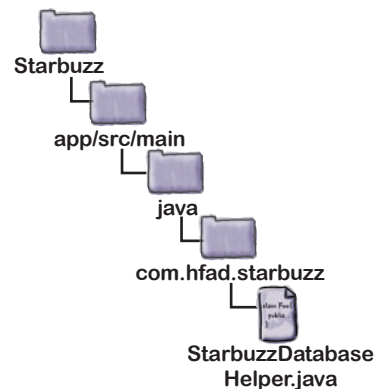
```

We're calling the constructor of the `SQLiteOpenHelper` superclass, and passing it the database name and version.

This parameter is an advanced feature relating to cursors. We'll cover cursors in the next chapter.

The constructor specifies details of the database, but the database doesn't get created at that point. The SQLite helper waits until the app needs to access the database, and then creates the database.

We've now done everything we need to tell the SQLite helper what database to create. The next step is to tell it what tables to create.





Inside a SQLite database

The data inside a SQLite database is stored in tables. A table contains several rows, and each row is split into columns. A column contains a single piece of data, like a number or a piece of text.

You need to create a table for each distinct piece of data that you want to record. In the Starbuzz app, for example, we'll need to create a table for the drink data. It will look something like this:

The columns in the table are `_id`, `NAME`, `DESCRIPTION`, and `IMAGE_RESOURCE_ID`. The Drink class contained similarly named attributes.

_id	NAME	DESCRIPTION	IMAGE_RESOURCE_ID
1	"Latte"	"Espresso and steamed milk"	54543543
2	"Cappuccino"	"Espresso, hot milk and steamed-milk foam"	654334453
3	"Filter"	"Our best drip coffee"	44324234

Some columns can be specified as **primary keys**. A primary key uniquely identifies a single row. If you say that a column is a primary key, then the database won't allow you to store rows with duplicate keys.

We recommend that your tables have a single column called `_id` to hold the primary key that contains integer values. This is because Android code is hardwired to expect a numeric `_id` column, so not having one can cause you problems later on.

Storage classes and data types

Each column in a table is designed to store a particular type of data. For example, in our DRINK table, the DESCRIPTION column will only ever store text data. Here are the main data types you can use in SQLite, and what they can store:

INTEGER	Any integer type
TEXT	Any character type
REAL	Any floating-point number
NUMERIC	Booleans, dates, and date-times
BLOB	Binary Large Object

Unlike most database systems, you don't need to specify the column size in SQLite. Under the hood, the data type is translated into a much broader storage class. This means you can say very generally what kind of data you're going to store, but you're not forced to be specific about the size of data.

It's an Android convention to call your primary key columns `_id`. Android code expects there to be an `_id` column on your data. Ignoring this convention will make it harder to get the data out of your database and into your user interface.

You create tables using Structured Query Language (SQL)

Every application that talks to SQLite needs to use a standard database language called Structured Query Language (SQL). SQL is used by almost every type of database. If you want to create the DRINK table, you will need to do it in SQL.

This is the SQL command to create the table: *The `_id` column is the primary key.*

```
CREATE TABLE DRINK ( The table name _id INTEGER PRIMARY KEY AUTOINCREMENT,
                    These are table columns. NAME TEXT,
                    DESCRIPTION TEXT,
                    IMAGE_RESOURCE_ID INTEGER)
```

The `CREATE TABLE` command says what columns you want in the table, and what the data type is of each column. The `_id` column is the primary key of the table, and the special keyword `AUTOINCREMENT` means that when we store a new row in the table, SQLite will automatically generate a unique integer for it.

The `onCreate()` method is called when the database is created

The SQLite helper is in charge of creating the SQLite database. An empty database is created on the device the first time it needs to be used, and then the SQLite helper's `onCreate()` method is called. The `onCreate()` method has one parameter, a `SQLiteDatabase` object that represents the database that's been created.

You can use the `SQLiteDatabase.execSQL()` method to execute SQL on the database. This method has one parameter, the SQL you want to execute.

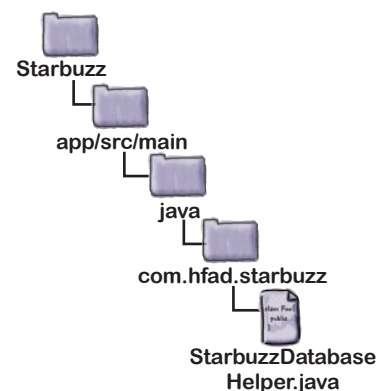
```
execSQL(String sql); ← Execute the SQL in the String on the database.
```

We'll use the `onCreate()` method to create the DRINK table. Here's the code (we'll add the code in a few pages):

```
public void onCreate(SQLiteDatabase db){
    db.execSQL("CREATE TABLE DRINK ("
        + "_id INTEGER PRIMARY KEY AUTOINCREMENT, "
        + "NAME TEXT, "
        + "DESCRIPTION TEXT, "
        + "IMAGE_RESOURCE_ID INTEGER);");
}
```

This gives us an empty DRINK table. We want to prepopulate it with drink data, so let's look at how you do that.

The `SQLiteDatabase` class gives you access to the database.





Insert data using the insert() method

To insert data into a table in a SQLite database, you start by specifying what values you want to insert into the table. To do this, you first create a **ContentValues** object:

```
ContentValues drinkValues = new ContentValues();
```

A **ContentValues** object describes a set of data. You usually create a new **ContentValues** object for each row of data you want to create.

You add data to the **ContentValues** object using its **put()** method. This method adds name/value pairs of data: **NAME** is the column you want to add data to, and **value** is the data:

```
contentValues.put("NAME", "value");
```

← *NAME is the column you want to add data to. value is the value you want it to have.*

As an example, here's how you'd use the **put()** method to add the name, description, and image resource ID of a latte to a **ContentValues** object called **drinkValues**:

```
This is a single row of data. {
  drinkValues.put("NAME", "Latte");
  drinkValues.put("DESCRIPTION", "Espresso and steamed milk");
  drinkValues.put("IMAGE_RESOURCE_ID", R.drawable.latte);
}
```

← Put the value "Latte" in the NAME column.
 ← Put "Espresso and steamed milk" in the DESCRIPTION column.
 ← You need a separate call to the put() method for each value you want to enter.

Once you've added a row of data to the **ContentValues** object, you insert it into the table using the **SQLiteDatabase insert()** method. This method inserts data into a table, and returns the ID of the record once it's been inserted. If the method is unable to insert the record, it returns a value of -1. As an example, here's how you'd insert the data from **drinkValues** into the **DRINK** table:

```
db.insert("DRINK", null, drinkValues);
```

← This inserts a single row into the table.

The middle parameter is usually set to **null**, as in the above code. It's there in case the **ContentValues** object is empty, and you want to insert an empty row into your table. It's unlikely you'd want to do this, but if you did you'd replace the **null** value with the name of one of the columns in your table.

Running the lines of code above will insert a Latte row into the **DRINK** table:

_id	NAME	DESCRIPTION	IMAGE_RESOURCE_ID
1	"Latte"	"Espresso and steamed milk"	54543543

← A shiny new record gets inserted into the table.

The **insert()** methods inserts one row of data at a time. But what if you want to insert multiple records?



Insert multiple records

To insert multiple rows into a table, you need to make repeat calls to the `insert()` method. Each call to the method inserts a separate row.

To insert multiple rows, you usually create a new method that inserts a single row of data, and call it each time you want to add a new row. As an example, here's an `insertDrink()` method we wrote to insert drinks into the `DRINK` table:

```
private static void insertDrink(SQLiteDatabase db,
                                String name,
                                String description,
                                int resourceId) {
    ContentValues drinkValues = new ContentValues();
    drinkValues.put("NAME", name);
    drinkValues.put("DESCRIPTION", description);
    drinkValues.put("IMAGE_RESOURCE_ID", resourceId);
    db.insert("DRINK", null, drinkValues);
}
```

This is the database we want to add records to.

We're passing the data to the method as parameters.

Construct a ContentValues object with the data.

Then insert the data.

To add three drinks to the `DRINK` table, each one a separate row of data, you'd call the method three times:

```
insertDrink(db, "Latte", "Espresso and steamed milk", R.drawable.latte);
insertDrink(db, "Cappuccino", "Espresso, hot milk and steamed-milk foam",
            R.drawable.cappuccino);
insertDrink(db, "Filter", "Our best drip coffee", R.drawable.filter);
```

That's everything you need to know to insert data into tables. On the next page we'll show you the revised code for *StarbuzzDatabaseHelper.java*.

The StarbuzzDatabaseHelper code

Here's the complete code for *StarbuzzDatabaseHelper.java* (update your code to reflect our changes):

```

package com.hfad.starbuzz;

import android.content.ContentValues;
import android.content.Context;
import android.database.sqlite.SQLiteDatabase;
import android.database.sqlite.SQLiteOpenHelper;

class StarbuzzDatabaseHelper extends SQLiteOpenHelper{

    private static final String DB_NAME = "starbuzz"; // the name of our database
    private static final int DB_VERSION = 1; // the version of the database

    StarbuzzDatabaseHelper(Context context){
        super(context, DB_NAME, null, DB_VERSION);
    }

    @Override
    public void onCreate(SQLiteDatabase db){
        db.execSQL("CREATE TABLE DRINK (_id INTEGER PRIMARY KEY AUTOINCREMENT, "
            + "NAME TEXT, "
            + "DESCRIPTION TEXT, "
            + "IMAGE_RESOURCE_ID INTEGER);");

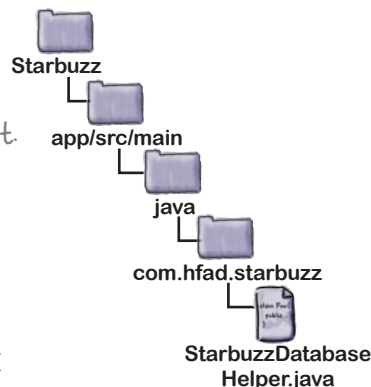
        insertDrink(db, "Latte", "Espresso and steamed milk", R.drawable.latte);
        insertDrink(db, "Cappuccino", "Espresso, hot milk and steamed-milk foam",
            R.drawable.cappuccino);
        insertDrink(db, "Filter", "Our best drip coffee", R.drawable.filter);
    }

    @Override
    public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {

    }

    private static void insertDrink(SQLiteDatabase db, String name,
        String description, int resourceId) {
        ContentValues drinkValues = new ContentValues();
        drinkValues.put("NAME", name);
        drinkValues.put("DESCRIPTION", description);
        drinkValues.put("IMAGE_RESOURCE_ID", resourceId);
        db.insert("DRINK", null, drinkValues);
    }
}

```



You need to add this extra import statement.

Say what the database name and version is. It's the first version of the database, so the version should be 1.

onCreate() gets called when the database first gets created, so we're using it to create the table and insert data.

Create the DRINK table.

Insert each drink in a separate row.

onUpgrade() gets called when the database needs to be upgraded. We'll look at this next.

We need to insert several drinks, so we created a separate method to do this.

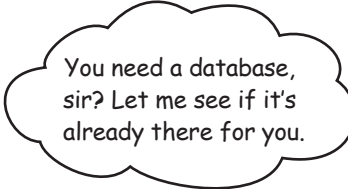


What the SQLite helper code does

- 1 The user installs the app and launches it.**
When the app needs to access the database, the SQLite helper checks to see if the database already exists.



SQLite helper



- 2 If the database doesn't exist, it gets created.**
It's given the name and version number specified in the SQLite helper.



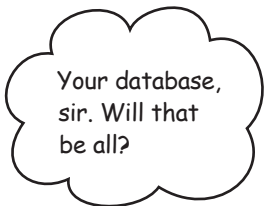
SQLite helper



SQLite database

Name: "starbuzz"
Version: 1

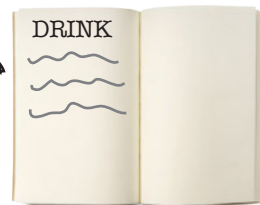
- 3 When the database is created, the onCreate() method in the SQLite helper is called.**
It adds a DRINK table to the database, and populates it with records.



onCreate()



SQLite helper



SQLite database

Name: "starbuzz"
Version: 1

What if you need to make changes to the database?

So far, you've seen how to create a SQLite database that your app will be able to use to persist data. But what if you need to make changes to the database at some future stage?

As an example, suppose lots of users have already installed your Starbuzz app on their devices, and you want to add a new FAVORITE column to the DRINK table. How would you distribute this change to new and existing users?



Well, we could change the CREATE TABLE statement in the onCreate() method, but that doesn't feel entirely right to me. I mean, what if a device already has the old version of the database installed?

When you need to change an app's database, there are two key scenarios you have to deal with.

The first scenario is that the user has never installed your app before, and doesn't have the database installed on their device. In this case, the SQLite helper creates the database the first time the database needs to be accessed, and runs its onCreate() method.

The second scenario is where the user installs a new version of your app that includes a different version of the database. If the SQLite helper spots that the database that's installed is out of date, it will call either the onUpgrade() or onDowngrade() method.

So how can the SQLite helper tell if the database is out of date?



SQLite databases have a version number

The SQLite helper can tell whether the SQLite database needs updating by looking at its version number. You specify the version of the database in the SQLite helper by passing it to the `SQLiteOpenHelper` superclass in its constructor.

Earlier on, we specified the version number of the database like this:

```

...
    private static final String DB_NAME = "starbuzz";
    private static final int DB_VERSION = 1;

    StarbuzzDatabaseHelper(Context context) {
        super(context, DB_NAME, null, DB_VERSION);
    }
...

```

When the database gets created, its version number gets set to the version number in the SQLite helper, and the SQLite helper `onCreate()` method gets called.

When you want to update the database, you change the version number in the SQLite helper code. To *upgrade* the database, specify a number that's larger than you had before, and to *downgrade* your database, specify a number that's lower:

```

...
    private static final int DB_VERSION = 2;
...

```

← Here we're increasing the version number, so the database will get upgraded.

Most of the time, you'll want to upgrade the database, so specify a number that's larger. You usually only downgrade your database when you want to undo changes you made in a previous upgrade.

When the user installs the latest version of the app on their device, the first time the app needs to use the database, the SQLite helper checks its version number against that of the database on the device.

If the version number in the SQLite helper code is **higher** than that of the database, it calls the SQLite helper `onUpgrade()` method. If the version number in the SQLite helper code is **lower** than that of the database, it calls the `onDowngrade()` method instead.

Once it's called either of these methods, it changes the version number of the database to match the version number in the SQLite helper.



Geek Bits

SQLite databases support a version number that's used by the SQLite helper, and an internal schema version. Whenever a change is made to the database schema, such as the table structure, the database increments the schema version by 1. You have no control over this value, it's just used internally by SQLite.



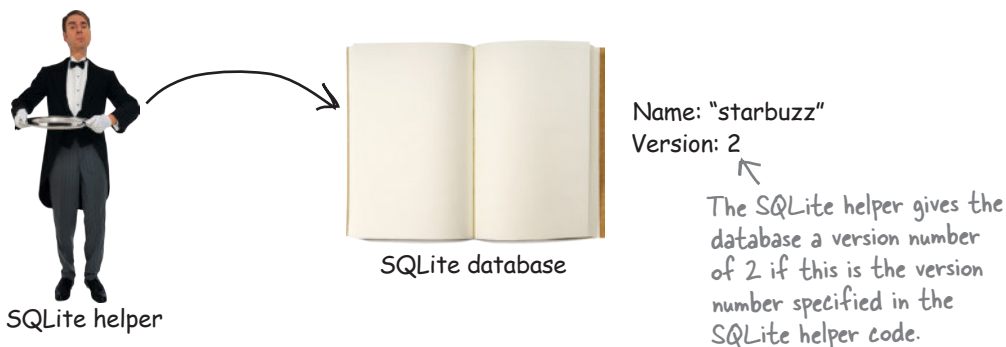
What happens when you change the version number

Let's look at what happens when you release a new version of the app where you've changed the SQLite helper version number from 1 to 2. We'll consider two scenarios: where a first-time user installs the app, and when an existing user installs it.

Scenario 1: A first-time user installs the app

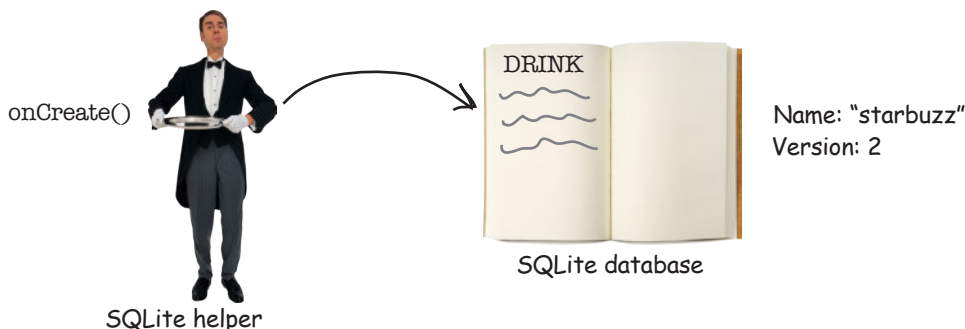
- 1 **The first time the user runs the app, the database doesn't exist, so the SQLite helper creates it.**

The SQLite helper gives the database the name and version number specified in the SQLite helper code.



- 2 **When the database is created, the onCreate() method in the SQLite helper is called.**

The onCreate () method includes code to populate the database.



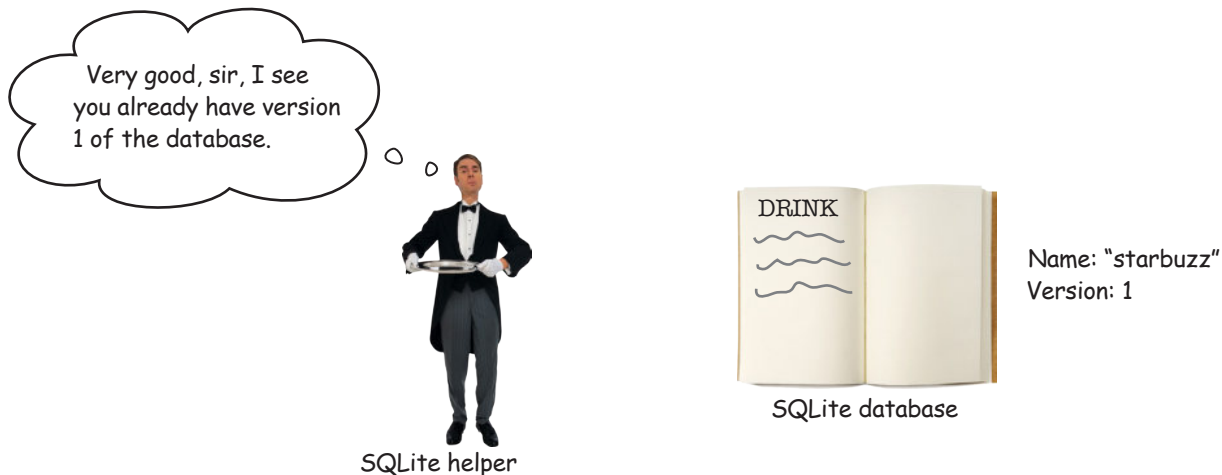
That's what happens when a first-time user installs the app. What about when an existing user installs the new version?



Scenario 2: an existing user installs the new version

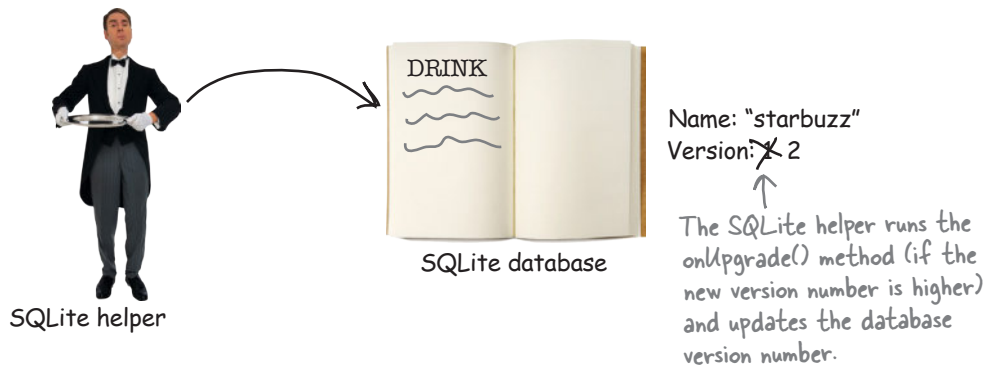
- 1 When the user runs the new version of the app, the database helper checks whether the database exists.

If the database already exists, the SQLite helper doesn't recreate it.



- 2 The SQLite helper checks the version number of the existing database against the version number in the SQLite helper code.

If the SQLite helper version number is higher than the database version, it calls the `onUpgrade()` method. If the SQLite helper version number is lower than the database version, it calls the `onDowngrade()` method. It then changes the database version number to reflect the version number in the SQLite helper code.



Now that you've seen under what circumstances the `onUpgrade()` and `onDowngrade()` methods get called, let's find out more about how you use them.



Upgrade your database with onUpgrade()

The `onUpgrade()` method has three parameters—the SQLite database, the user's version number of the database, and the new version of the database that's passed to the `SQLiteOpenHelper` superclass:

```
@Override
public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
    //Your code goes here
}
```

The user's version of the database, which is out of date

The new version described in the SQLite helper code

Remember, to upgrade the database, the new version must be higher than the user's existing version.

The version numbers are important, as you can use them to say what database changes should be made depending on which version of the database the user already has. As an example, suppose you needed to run code if the user has version 1 of the database, and the SQLite helper version number is higher. Your code would look like this:

```
@Override
public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
    if (oldVersion == 1) {
        //Code to run if the database version is 1
    }
}
```

This code will only run if the user has version 1 of the database, and the SQLite helper version number is higher.

You can also use the version numbers to apply successive updates like this:

```
@Override
public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
    if (oldVersion == 1) {
        //Code to run if the database version is 1
    }
    if (oldVersion < 3) {
        //Code to run if the database version is 1 or 2
    }
}
```

This code will only run if the user's database is at version 1.

This code will run if the user's database is at version 1 or 2.

Using this approach means that you can make sure that the user gets all the database changes applied that they need, irrespective of which version they have installed.

The `onDowngrade()` method works in a similar way to the `onUpgrade()` method. Let's take a look on the next page.



Downgrade your database with onDowngrade()

The `onDowngrade()` method isn't used as often as the `onUpgrade()` method, as it's used to revert your database to a previous version. This can be useful if you release a version of your app that includes database changes, but you then discover that there are bugs. The `onDowngrade()` method allows you to undo the changes and set the database back to its previous version.

Just like the `onUpgrade()` method, the `onDowngrade()` method has three parameters—the SQLite database you want to downgrade, the version number of the database itself, and the new version of the database that's passed to the `SQLiteOpenHelper` superclass:

```
@Override
public void onDowngrade(SQLiteDatabase db, int oldVersion, int newVersion) {
    //Your code goes here
}
```

↑ ↑
To downgrade the database, the new version must be lower than the user's existing version.

Just as with the `onUpgrade()` method, you can use the version numbers to revert changes specific to a particular version. As an example, if you needed to make changes to the database when the user's database version number is 3, you'd use code like following:

```
@Override
public void onDowngrade(SQLiteDatabase db, int oldVersion, int newVersion) {
    if (oldVersion == 3) {
        //Code to run if the database version is 3
    }
}
```

← This code will run if the user has version 3 of the database, but you want to downgrade it to a lower version.

Now that you know how to upgrade and downgrade a database, let's walk through the more common scenario: upgrading.



Let's upgrade the database

Suppose we need to upgrade our database to add a new column to the DRINK table. As we want all new and existing users to get this change, we need to make sure that it's included in both the `onCreate()` and `onUpgrade()` methods. The `onCreate()` method will make sure that all new users get the new column, and the `onUpgrade()` method will make sure that all existing users get it too.

Rather than put similar code in both the `onCreate()` and `onUpgrade()` methods, we're going to create a separate `updateMyDatabase()` method, called by both the `onCreate()` and `onUpgrade()` methods. We'll move the code that's currently in the `onCreate()` method to this new `updateMyDatabase()` method, and we'll add extra code to create the extra column. Using this approach means that you can put all of your database code in one place, and more easily keep track of what changes you've made each time you've updated the database.

Here's the full code for `StarbuzzDatabaseHelper.java` (update your code to reflect our changes):

```
package com.hfad.starbuzz;

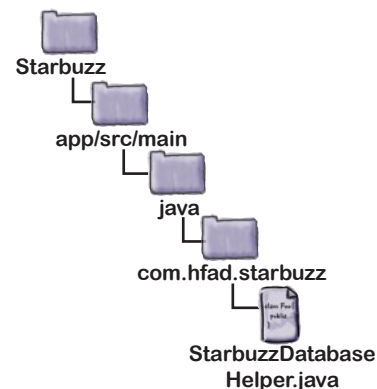
import android.content.ContentValues;
import android.content.Context;
import android.database.sqlite.SQLiteDatabase;
import android.database.sqlite.SQLiteOpenHelper;

class StarbuzzDatabaseHelper extends SQLiteOpenHelper{

    private static final String DB_NAME = "starbuzz"; // the name of our database
    private static final int DB_VERSION = 2; // the version of the database

    StarbuzzDatabaseHelper(Context context){
        super(context, DB_NAME, null, DB_VERSION);
    }

    @Override
    public void onCreate(SQLiteDatabase db){
        updateMyDatabase(db, 0, DB_VERSION);
    }
}
```



Changing the version number to a larger integer means the SQLite helper will know that you want to upgrade the database.

Replace the existing `onCreate()` code with this call to `updateMyDatabase()`.

The code continues on the next page.



The SQLite helper code (continued)

```

@Override
public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
    updateMyDatabase(db, oldVersion, newVersion); ← Call the updateMyDatabase()
                                                    method from onUpgrade(),
                                                    passing along the parameters.
}

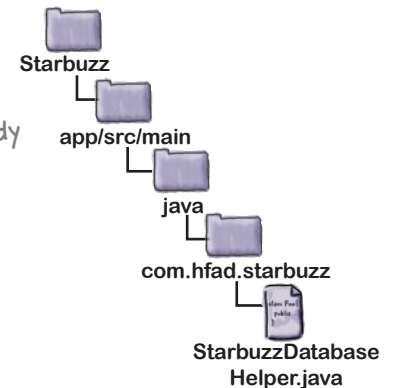
private static void insertDrink(SQLiteDatabase db, String name,
                                String description, int resourceId) {
    ContentValues drinkValues = new ContentValues();
    drinkValues.put("NAME", name);
    drinkValues.put("DESCRIPTION", description);
    drinkValues.put("IMAGE_RESOURCE_ID", resourceId);
    db.insert("DRINK", null, drinkValues);
}

private void updateMyDatabase(SQLiteDatabase db, int oldVersion, int newVersion) {
    if (oldVersion < 1) {
        db.execSQL("CREATE TABLE DRINK (_id INTEGER PRIMARY KEY AUTOINCREMENT, "
            + "NAME TEXT, "
            + "DESCRIPTION TEXT, "
            + "IMAGE_RESOURCE_ID INTEGER);");
        insertDrink(db, "Latte", "Espresso and steamed milk", R.drawable.latte);
        insertDrink(db, "Cappuccino", "Espresso, hot milk and steamed-milk foam",
            R.drawable.cappuccino);
        insertDrink(db, "Filter", "Our best drip coffee", R.drawable.filter);
    }
    if (oldVersion < 2) {
        //Code to add the extra column
    }
}
}

```

This is the code we previously had in our onCreate() method.

This code will run if the user already has version 1 of the database. We need to write this code next.



The next thing we need to do is write the code to upgrade the database. Before we do that, try the exercise on the next page.



BE the SQLite Helper
 On the right, you'll see some SQLite helper code. Your job is to play like you're the SQLite helper and say which code will run for each of the users below. We've labeled the code we want you to consider. We've done the first one to start you off.

User 1 runs the app for the first time.

Code segment A. The user doesn't have the database, so the `onCreate()` method runs.

User 2 has database version 1.

User 3 has database version 2.

User 4 has database version 3.

User 5 has database version 4.

User 6 has database version 5.

...

```
class MyHelper extends SQLiteOpenHelper{

    StarbuzzDatabaseHelper(Context context){
        super(context, "fred", null, 4);
    }

    @Override
    public void onCreate(SQLiteDatabase db){
        A //Run code A
        ...
    }

    @Override
    public void onUpgrade(SQLiteDatabase db,
                          int oldVersion,
                          int newVersion){

        if (oldVersion < 2) {
            B //Run code B
            ...
        }
        if (oldVersion == 3) {
            C //Run code C
            ...
        }
        D //Run code D
        ...
    }

    @Override
    public void onDowngrade(SQLiteDatabase db,
                             int oldVersion,
                             int newVersion){

        if (oldVersion == 3) {
            E //Run code E
            ...
        }
        if (oldVersion < 6) {
            F //Run code F
            ...
        }
    }
}
```

—————> **Answers on page 654.**



Upgrade an existing database

When you need to upgrade your database, there are two types of actions you might want to perform:



Change the database records.

Earlier in the chapter, you saw how to insert records in your database using the `SQLiteDatabase.insert()` method. You may want to add more records when you upgrade the database, or update or delete the records that are already there.



Change the database structure.

You've already seen how you can create tables in the database. You may also want to add columns to existing tables, rename tables, or remove tables completely.

We'll start by looking at how you change database records.

How to update records

You update records in a table in a similar way to how you insert them.

You start by creating a new `ContentValues` object that specifies what you want to update values to. As an example, suppose you wanted to update the Latte data in the `DRINK` table so that the value of the `DESCRIPTION` field is "Tasty":

<code>_id</code>	<code>NAME</code>	<code>DESCRIPTION</code>	<code>IMAGE_RESOURCE_ID</code>
1	"Latte"	"Espresso and steamed milk" "Tasty"	54543543

To do this, you'd create a new `ContentValues` object that describes the data that needs to be updated:

```
ContentValues drinkValues = new ContentValues();
drinkValues.put("DESCRIPTION", "Tasty");
```

Notice that when you're updating records, you only need to specify the data you want to change in the `ContentValues` object, not the entire row of data.

Once you've added the data you want to change to the `ContentValues` object, you use the `SQLiteDatabase.update()` method to update the data. We'll look at this on the next page.

We want to change the value of the DESCRIPTION column to Tasty, so we give the name "DESCRIPTION" a value of "Tasty".



Update records with the update() method

The update () method lets you update records in the database, and returns the number of records it's updated. To use the update () method, you specify the table you want to update records in, the ContentValues object that contains the values you want to update, and the conditions for updating them.

As an example, here's how you'd change the value of the DESCRIPTION column to "Tasty" where the name of the drink is "Latte":

```
ContentValues drinkValues = new ContentValues();
drinkValues.put("DESCRIPTION", "Tasty");
db.update("DRINK",
```

← This is the name of the table whose records you want to update.

```
    drinkValues,
```

← This is the ContentValues object that contains the new values.

```
    "NAME = ?",
    new String[] {"Latte"});
```

← The value "Latte" is substituted for the ? in the "NAME = ?" statement above.

The conditions for updating the data, in this case where NAME = "Latte".

The first parameter of the update () method is the name of the table you want to update (in this case, the DRINK table).

The second parameter is the ContentValues object that describes the values you want to update. In the example above, we've added "DESCRIPTION" and "Tasty" to the ContentValues object, so it updates the value of the DESCRIPTION column to "Tasty".

The last two parameters specify which records you want to update by describing conditions for the update. Together, they form the WHERE clause of a SQL statement.

The third parameter specifies the name of the column in the condition. In the above example, we want to update records where the value of the NAME column is "Latte", so we use "NAME = ?"; it means that we want the value in the NAME column to be equal to some value. The ? symbol is a placeholder for this value.

The last parameter is an array of Strings that says what the value of the condition should be. In our particular case we want to update records where the value of the NAME column is "Latte", so we use:

```
new String[] {"Latte"});
```

We'll look at more complex conditions on the next page.



Watch it!

If you set the last two parameters of the update () method to null,

ALL records in the table will be updated.

As an example, the code:

```
db.update("DRINK",
        drinkValues,
        null, null);
```

will update all records in the DRINK table.



Apply conditions to multiple columns

You can also specify conditions that apply to multiple columns. As an example, here's how you'd update all records where the name of the drink is "Latte", or the drink description is "Our best drip coffee":

```
db.update("DRINK",
    drinkValues,
    "NAME = ? OR DESCRIPTION = ?",
    new String[] {"Latte", "Our best drip coffee"});
```

This means: Where NAME = "Latte" or DESCRIPTION = "Our best drip coffee".

If you want to apply conditions that cover multiple columns, you specify the column names in the update () method's third parameter. As before, you add a placeholder ? for each value you want to add as part of each condition. You then specify the condition values in the update () method's fourth parameter.

Each ? placeholder is replaced with a value from this array. The number of values in the array must match the number of ?s.

The condition values must be Strings, even if the column you're applying the condition to contains some other type of data. As an example, here's how you'd update DRINK records where the _id (numeric) is 1:

```
db.update("DRINK",
    drinkValues,
    "_id = ?",
    new String[] {Integer.toString(1)});
```

Convert the int 1 to a String value.

Delete records with the delete() method

You delete records using the SQLiteDatabase delete () method. It works in a similar way to the update () method you've just seen. You specify the table you want to delete records from, and conditions for deleting the data. As an example, here's how you'd delete all records from the DRINK table where the name of the drink is "Latte":

```
db.delete("DRINK",
    "NAME = ?",
    new String[] {"Latte"});
```

See how similar this is to the update() method.

The entire row is deleted.

_id	NAME	DESCRIPTION	IMAGE_RESOURCE_ID
1	"Latte"	"Espresso and steamed milk"	54543543

The first parameter is the name of the table you want to delete records from (in this case, DRINK). The second and third arguments allow you to describe conditions to specify exactly which records you wish to delete (in this case, where NAME = "Latte").



Here's the onCreate() method of a SQLiteOpenHelper class. Your job is to say what values have been inserted into the NAME and DESCRIPTION columns of the DRINK table when the onCreate() method has finished running.

```

@Override
public void onCreate(SQLiteDatabase db) {
    ContentValues espresso = new ContentValues();
    espresso.put("NAME", "Espresso");
    ContentValues americano = new ContentValues();
    americano.put("NAME", "Americano");
    ContentValues latte = new ContentValues();
    latte.put("NAME", "Latte");
    ContentValues filter = new ContentValues();
    filter.put("DESCRIPTION", "Filter");
    ContentValues mochachino = new ContentValues();
    mochachino.put("NAME", "Mochachino");

    db.execSQL("CREATE TABLE DRINK ("
        + "_id INTEGER PRIMARY KEY AUTOINCREMENT, "
        + "NAME TEXT, "
        + "DESCRIPTION TEXT);");

    db.insert("DRINK", null, espresso);
    db.insert("DRINK", null, americano);
    db.delete("DRINK", null, null);
    db.insert("DRINK", null, latte);
    db.update("DRINK", mochachino, "NAME = ?", new String[] {"Espresso"});
    db.insert("DRINK", null, filter);
}
    
```

You don't need to enter the value of the _id column.

_id	NAME	DESCRIPTION

—————> Answers on page 655.



Change the database structure

In addition to creating, updating, or deleting database records, you may want to make changes to the database structure. As an example, in our particular case we want to add a new FAVORITE column to the DRINK table.

Add new columns to tables using SQL

Earlier in the chapter, you saw how you could create tables using the SQL `CREATE TABLE` command like this:

```
CREATE TABLE DRINK (_id INTEGER PRIMARY KEY AUTOINCREMENT,
                    NAME TEXT,
                    DESCRIPTION TEXT,
                    IMAGE_RESOURCE_ID INTEGER)
```

The table name

The table columns

The `_id` column is the primary key.

You can also use SQL to change an existing table using the `ALTER TABLE` command. As an example, here's what the command looks like to add a column to a table:

```
ALTER TABLE DRINK
ADD COLUMN FAVORITE NUMERIC
```

The table name

The column you want to add

In the example above, we're adding a column to the DRINK table called FAVORITE that holds numeric values.

Renaming tables

You can also use the `ALTER TABLE` command to rename a table. As an example, here's how you'd rename the DRINK table to FOO:

```
ALTER TABLE DRINK
RENAME TO FOO
```

The current table name

The new name of the table

On the next page, we'll show you how to remove a table from the database.



Delete tables by dropping them

If you want to delete a table from the database, you use the `DROP TABLE` command. As an example, here's how you'd delete the `DRINK` table:

```
DROP TABLE DRINK ← The name of the table you want to remove
```

This command is useful if you have a table in your database schema that you know you don't need anymore, and want to remove it in order to save space. Make sure you only use the `DROP TABLE` command if you're absolutely sure you want to get rid of the table and all of its data.

Execute the SQL using `execSQL()`

As you saw earlier in the chapter, you execute SQL commands using the `SQLiteDatabase.execSQL()` method:

```
SQLiteDatabase.execSQL(String sql);
```

You use the `execSQL()` method any time you need to execute SQL on the database. As an example, here's how you'd execute SQL to add a new `FAVORITE` column to the `DRINK` table:

```
db.execSQL("ALTER TABLE DRINK ADD COLUMN FAVORITE NUMERIC;");
```

Now that you've seen the sorts of actions you might want to perform when upgrading your database, let's apply this to `StarbuzzDatabaseHelper.java`.



The full SQLite helper code

Here's the full code for *StarbuzzDatabaseHelper.java* that will add a new FAVORITE column to the DRINK table. Update your code to match ours (the changes are in bold):

```

package com.hfad.starbuzz;

import android.content.ContentValues;
import android.content.Context;
import android.database.sqlite.SQLiteDatabase;
import android.database.sqlite.SQLiteOpenHelper;

class StarbuzzDatabaseHelper extends SQLiteOpenHelper{

    private static final String DB_NAME = "starbuzz"; // the name of our database
    private static final int DB_VERSION = 2; // the version of the database

    StarbuzzDatabaseHelper(Context context){
        super(context, DB_NAME, null, DB_VERSION);
    }

    @Override
    public void onCreate(SQLiteDatabase db){
        updateMyDatabase(db, 0, DB_VERSION);
    }

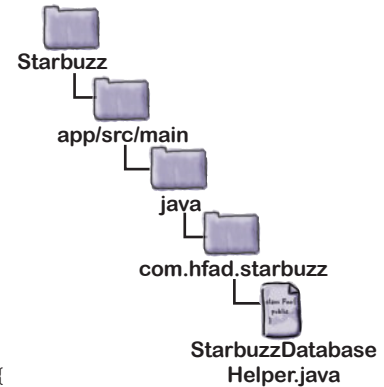
    @Override
    public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
        updateMyDatabase(db, oldVersion, newVersion);
    }
}

```

Changing the version number to a larger integer enables the SQLite helper to spot that you want to upgrade the database.

The code to create any database tables is in the *updateMyDatabase()* method.

The code to upgrade the database is in our *updateMyDatabase()* method.



The code continues over the page.



The SQLite helper code (continued)

```

private static void insertDrink(SQLiteDatabase db, String name,
                                String description, int resourceId) {
    ContentValues drinkValues = new ContentValues();
    drinkValues.put("NAME", name);
    drinkValues.put("DESCRIPTION", description);
    drinkValues.put("IMAGE_RESOURCE_ID", resourceId);
    db.insert("DRINK", null, drinkValues);
}

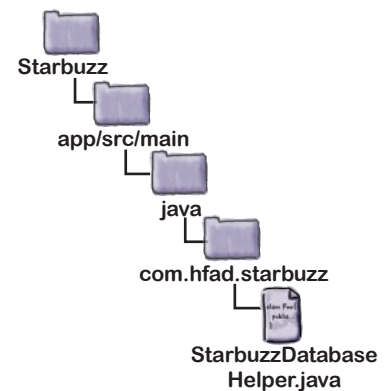
private void updateMyDatabase(SQLiteDatabase db, int oldVersion, int newVersion) {
    if (oldVersion < 1) {
        db.execSQL("CREATE TABLE DRINK (_id INTEGER PRIMARY KEY AUTOINCREMENT, "
            + "NAME TEXT, "
            + "DESCRIPTION TEXT, "
            + "IMAGE_RESOURCE_ID INTEGER);");
        insertDrink(db, "Latte", "Espresso and steamed milk", R.drawable.latte);
        insertDrink(db, "Cappuccino", "Espresso, hot milk and steamed-milk foam",
            R.drawable.cappuccino);
        insertDrink(db, "Filter", "Our best drip coffee", R.drawable.filter);
    }
    if (oldVersion < 2) {
        db.execSQL("ALTER TABLE DRINK ADD COLUMN FAVORITE NUMERIC;");
    }
}
}

```

↑
Add a numeric FAVORITE column to the DRINK table.

The new code in the SQLite helper means that existing users will get the new FAVORITE column added to the DRINK table the next time they access the database. It also means that any new users will get the complete database created for them, including the new column.

We'll go through what happens when the code runs on the next page. Then in the next chapter, you'll see how to use the database data in your activities.



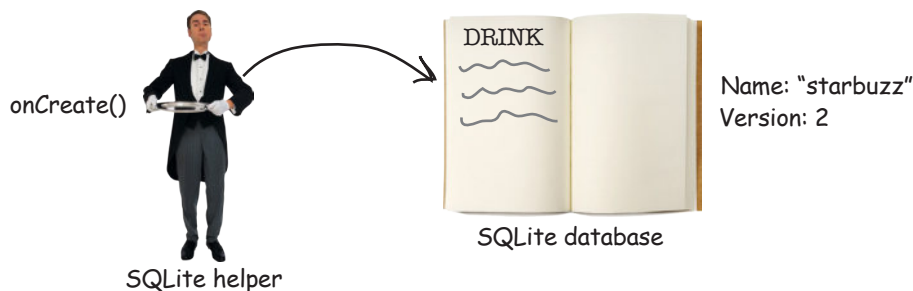
What happens when the code runs

- 1 When the database first needs to be accessed, the SQLite helper checks whether the database already exists.



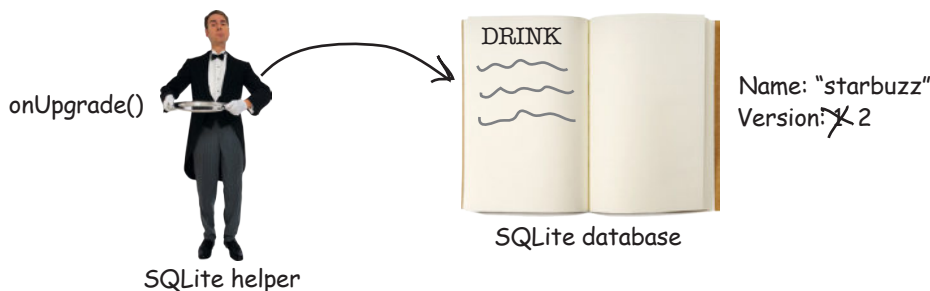
- 2a If the database doesn't exist, the SQLite helper creates it and runs its onCreate() method.

Our onCreate() method code calls the updateMyDatabase() method. This creates the DRINK table (including the new FAVORITE column) and populates the table with records.



- 2b If the database already exists, the SQLite helper checks the version number of the database against the version number in the SQLite helper code.

If the SQLite helper version number is higher than the database version, it calls the onUpgrade() method. If the SQLite helper version number is lower than the database version, it calls the onDowngrade() method. Our SQLite helper version number is higher than that of the existing database, so the onUpgrade() method is called. It calls the updateMyDatabase() method, and this adds the new FAVORITE column to the DRINK table.



BE the SQLite Helper Solution



On the right you'll see some SQLite helper code. Your job is to play like you're the SQLite helper and say which code will run for each of the users below. We've labeled the code we want you to consider. We've done the first one to start you off.

User 1 runs the app for the first time.

Code segment A. The user doesn't have the database, so the onCreate() method runs.

User 2 has database version 1.

Code segment B, then D. The database needs to be upgraded with oldVersion == 1.

User 3 has database version 2.

Code segment D. The database needs to be upgraded with oldVersion == 2.

User 4 has database version 3.

Code segment C then D. The database needs to be upgraded with oldVersion == 3.

User 5 has database version 4.

None. The user has the correct version of the database.

User 6 has database version 5.

Code segment F. The database needs to be downgraded with oldVersion == 5.

...

```
class MyHelper extends SQLiteOpenHelper{

    StarbuzzDatabaseHelper(Context context){
        super(context, "fred", null, 4);
    }

    @Override
    public void onCreate(SQLiteDatabase db){
        A //Run code A
        ...
    }

    @Override
    public void onUpgrade(SQLiteDatabase db,
        int oldVersion,
        int newVersion){
        if (oldVersion < 2) {
            B //Run code B
            ...
        }
        if (oldVersion == 3) {
            C //Run code C
            ...
        }
        D //Run code D
        ...
    }

    @Override
    public void onDowngrade(SQLiteDatabase db,
        int oldVersion,
        int newVersion){
        if (oldVersion == 3) {
            E //Run code E
            ...
        }
        if (oldVersion < 6) {
            F //Run code F
            ...
        }
    }
}
```

The new version of the database is 4.

The onCreate() method will only run if the user doesn't have the database.

This will run if the user has version 1.

This will run if the user has version 3.

This will run if the user has version 1, 2, or 3.

This will never run. If the user has version 3, their database needs to be upgraded, not downgraded.

This will run if the user has version 5. For onDowngrade() to run, the user must have a version greater than 4, as that's the current version number of the helper.

Sharpen your pencil Solution

Here's the onCreate() method of a SQLiteOpenHelper class. Your job is to say what values have been inserted into the NAME and DESCRIPTION columns of the DRINK table when the onCreate() method has finished running.

```
@Override
public void onCreate(SQLiteDatabase db) {
    ContentValues espresso = new ContentValues();
    espresso.put("NAME", "Espresso");
    ContentValues americano = new ContentValues();
    americano.put("NAME", "Americano");
    ContentValues latte = new ContentValues();
    latte.put("NAME", "Latte");
    ContentValues filter = new ContentValues();
    filter.put("DESCRIPTION", "Filter");
    ContentValues mochachino = new ContentValues();
    mochachino.put("NAME", "Mochachino");

    db.execSQL("CREATE TABLE DRINK ("
        + "_id INTEGER PRIMARY KEY AUTOINCREMENT, "
        + "NAME TEXT, "
        + "DESCRIPTION TEXT);");
    db.insert("DRINK", null, espresso);
    db.insert("DRINK", null, americano);
    db.delete("DRINK", null, null);
    db.insert("DRINK", null, latte);
    db.update("DRINK", mochachino, "NAME = ?", new String[] {"Espresso"});
    db.insert("DRINK", null, filter);
}
```

Create the table, adding _id, NAME, and DESCRIPTION columns.

← Insert Espresso in the NAME column.

← Insert Americano in the NAME column.

← Delete all the drinks.

← Insert Latte in the NAME column.

↖ Set NAME to Mochachino where NAME is Espresso. No records get updated.

↑ Insert Filter in the DESCRIPTION column.

Here's the final result of running the above code. →

<u>_id</u>	<u>NAME</u>	<u>DESCRIPTION</u>
	Latte	
		Filter



Your Android Toolbox

You've got Chapter 15 under your belt and now you've added creating, updating, and upgrading databases to your toolbox.

You can download the full code for the chapter from <https://tinyurl.com/HeadFirstAndroid>.



BULLET POINTS

- Android uses SQLite as its backend database to persist data.
- The `SQLiteDatabase` class gives you access to the SQLite database.
- A SQLite helper lets you create and manage SQLite databases. You create a SQLite helper by extending the `SQLiteOpenHelper` class.
- You must implement the `SQLiteOpenHelper` `onCreate()` and `onUpgrade()` methods.
- The database gets created the first time it needs to be accessed. You need to give the database a name and version number, starting at 1. If you don't give the database a name, it will just get created in memory.
- The `onCreate()` method gets called when the database first gets created.
- The `onUpgrade()` method gets called when the database needs to be upgraded.
- Execute SQL using the `SQLiteDatabase` `execSQL(String)` method.
- Use the SQL `ALTER TABLE` command to change an existing table. Use `RENAME TO` to rename the table, and `ADD COLUMN` to add a column.
- Use the SQL `DROP TABLE` command to delete a table.
- Add records to tables using the `insert()` method.
- Update records using the `update()` method.
- Remove records from tables using the `delete()` method.

16 basic cursors

Getting Data Out



Charles gave me a cursor that returned everything from the EXPENSIVE_GIFT table.

So how do you connect your app to a SQLite database?

So far you've seen how to create a SQLite database using a SQLite helper. The next step is to get your activities to access it. In this chapter, we'll focus on how you read data from a database. You'll find out **how to use cursors to get data from the database**. You'll see **how to navigate cursors**, and **how to get access to their data**. Finally, you'll discover how to use **cursor adapters** to bind cursors to list views.

The story so far...

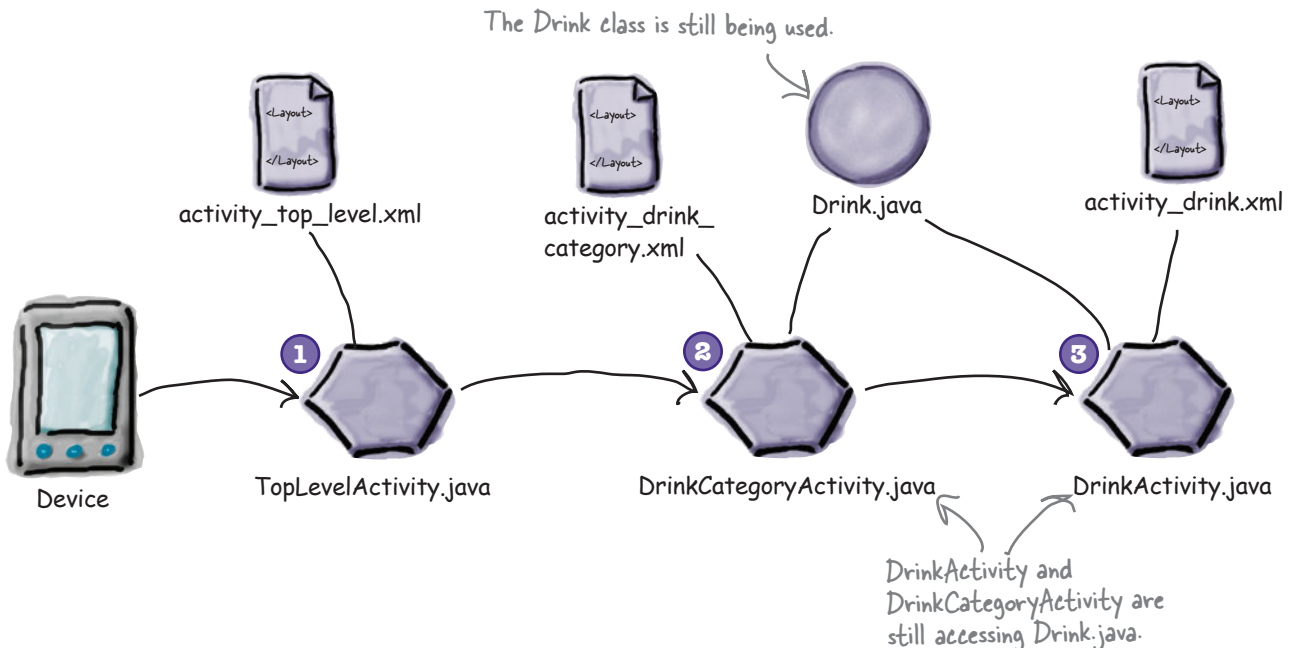
In Chapter 15, you created a SQLite helper for Starbuzz Coffee. The SQLite helper creates a Starbuzz database, adds a DRINK table to it, and populates the table with drinks.

The activities in the Starbuzz app currently get their data from the Java `Drink` class. We're going to change the app so the activities get data from the SQLite database instead.

Here's a reminder of how the app currently works:

- 1 TopLevelActivity displays a list of options for Drinks, Food, and Stores.**
- 2 When the user clicks on the Drinks option, it launches DrinkCategoryActivity.**
This activity displays a list of drinks that it gets from the Java `Drink` class.
- 3 When the user clicks on a drink, its details get displayed in DrinkActivity.**
`DrinkActivity` gets details of the drink from the Java `Drink` class.

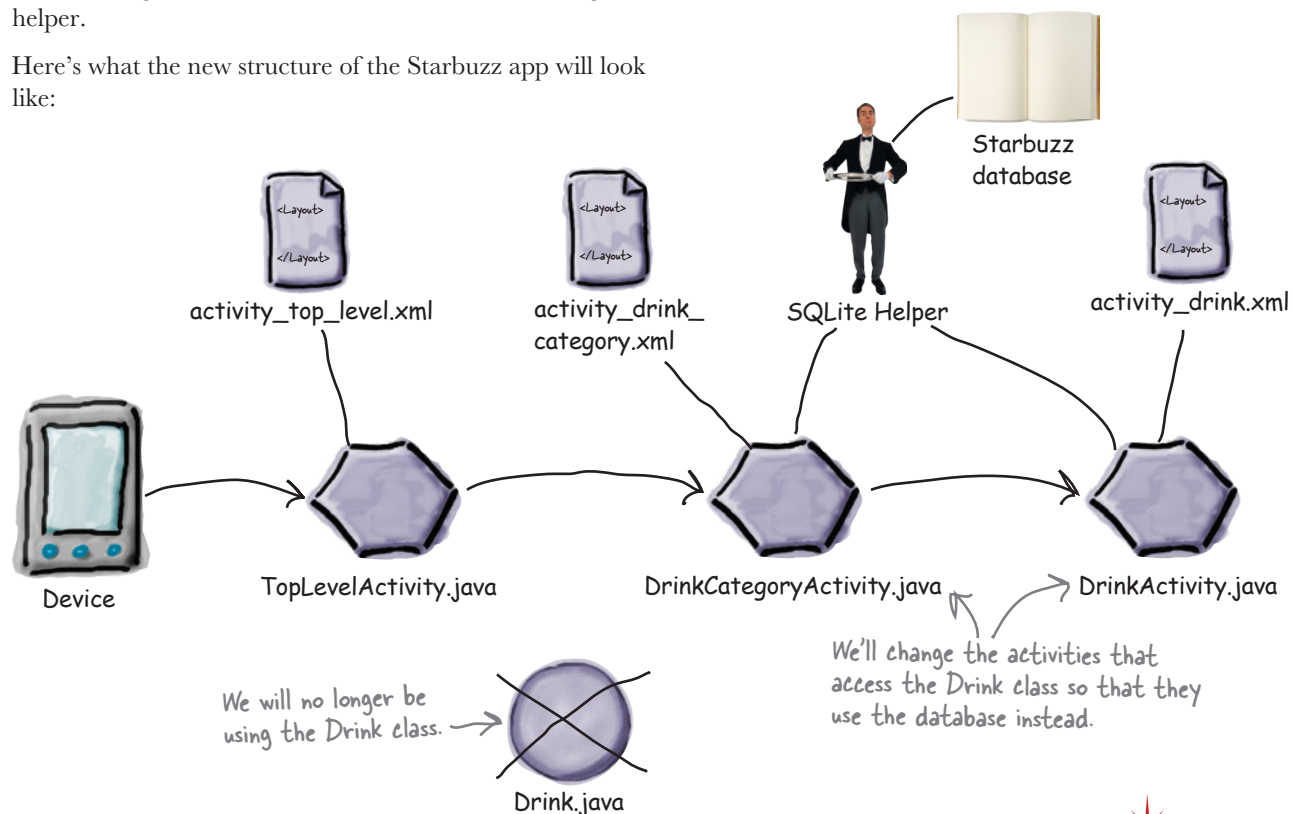
We've created the SQLite helper and added code so it can create the Starbuzz database. It's not being used by any activities yet.



The new Starbuzz app structure

There are two activities that use the Drink class: DrinkActivity and DrinkCategoryActivity. We need to change these activities so that they read data from the SQLite database with assistance from the SQLite helper.

Here's what the new structure of the Starbuzz app will look like:



We'll start by updating DrinkActivity, and we'll change DrinkCategoryActivity later on in the chapter.

Do this!

We're going to update the Starbuzz app in this chapter, so open your Starbuzz project in Android Studio.

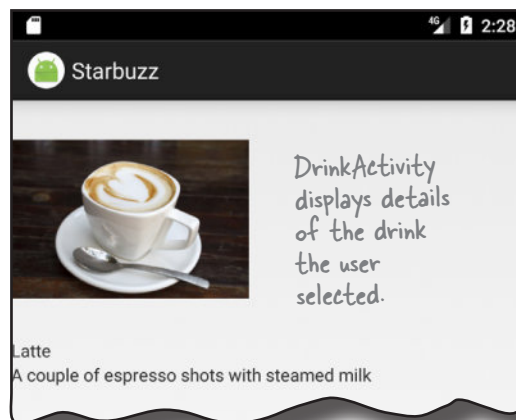
What we'll do to change DrinkActivity to use the Starbuzz database

There are a number of steps we need to go through to change `DrinkActivity` so that it uses the Starbuzz database:

- 1 **Get a reference to the Starbuzz database.**
We'll do this using the Starbuzz SQLite helper we created in Chapter 15.



- 2 **Create a cursor to read drink data from the database.**
We need to read the data held in the Starbuzz database for the drink the user selects in `DrinkCategoryActivity`. The cursor will give us access to this data. (We'll explain cursors soon.)
- 3 **Navigate to the drink record.**
Before we can use the data retrieved by the cursor, we need to explicitly navigate to it.
- 4 **Display details of the drink in DrinkActivity.**
Once we've navigated to the drink record in the cursor, we need to read the data and display it in `DrinkActivity`.



Before we begin, let's remind ourselves of the `DrinkActivity.java` code we created in Chapter 7.

The current DrinkActivity code

Below is a reminder of the current *DrinkActivity.java* code. The `onCreate()` method gets the drink ID selected by the user, gets the details of that drink from the *Drink* class, and then populates the activity's views using the drink attributes. We need to change the code in the `onCreate()` method to get the data from the Starbuzz database.

```
package com.hfad.starbuzz;
```

... ← We're not showing you the import statements.

```
public class DrinkActivity extends Activity {
```

```
    public static final String EXTRA_DRINKID = "drinkId";
```

```
    @Override
```

```
    protected void onCreate(Bundle savedInstanceState) {
```

```
        super.onCreate(savedInstanceState);
```

```
        setContentView(R.layout.activity_drink);
```

```
        //Get the drink from the intent
```

```
        int drinkId = (Integer) getIntent().getExtras().get(EXTRA_DRINKID);
```

```
        Drink drink = Drink.drinks[drinkId];
```

← Use the drink ID from the intent to get the drink details from the *Drink* class. We'll need to change this so the drink comes from the database.

```
        //Populate the drink name
```

```
        TextView name = (TextView) findViewById(R.id.name);
```

```
        name.setText(drink.getName());
```

```
        //Populate the drink description
```

```
        TextView description = (TextView) findViewById(R.id.description);
```

```
        description.setText(drink.getDescription());
```

```
        //Populate the drink image
```

```
        ImageView photo = (ImageView) findViewById(R.id.photo);
```

```
        photo.setImageResource(drink.getImageResourceId());
```

```
        photo.setContentDescription(drink.getName());
```

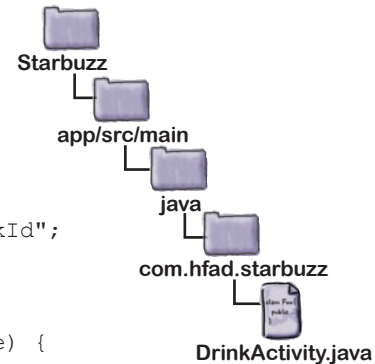
```
    }
```

```
}
```

We need to populate the views in the layout with values from the database, not from the *Drink* class.



Database reference
Create cursor
Navigate to record
Display drink



This is the drink the user selected.



Use the drink ID from the intent to get the drink details from the *Drink* class. We'll need to change this so the drink comes from the database.

Get a reference to the database

The first thing we need is to get a reference to the Starbuzz database using the SQLite helper we created in the last chapter. To do that, you start by getting a reference to the SQLite helper:

```
SQLiteOpenHelper starbuzzDatabaseHelper = new StarbuzzDatabaseHelper(this);
```

You then call the SQLite helper's `getReadableDatabase()` or `getWritableDatabase()` to get a reference to the database. You use the `getReadableDatabase()` method if you need read-only access to the database, and the `getWritableDatabase()` method if you need to perform any updates. Both methods return a `SQLiteDatabase` object that you can use to access the database:

```
SQLiteDatabase db = starbuzzDatabaseHelper.getReadableDatabase();
```

or:

```
SQLiteDatabase db = starbuzzDatabaseHelper.getWritableDatabase();
```

If Android fails to get a reference to the database, a `SQLiteException` is thrown. This can happen if, for example, you call the `getWritableDatabase()` to get read/write access to the database, but you can't write to the database because the disk is full.

In our particular case, we only need to read data from the database, so we'll use the `getReadableDatabase()` method. If Android can't get a reference to the database and a `SQLiteException` is thrown, we'll use a `Toast` (a pop-up message) to tell the user that the database is unavailable:

```
SQLiteOpenHelper starbuzzDatabaseHelper = new StarbuzzDatabaseHelper(this);
try {
    SQLiteDatabase db = starbuzzDatabaseHelper.getReadableDatabase();
    //Code to read data from the database
} catch (SQLiteException e) {
    Toast toast = Toast.makeText(this,
                                "Database unavailable",
                                Toast.LENGTH_SHORT);
    toast.show(); ← This line displays the toast.
}
```

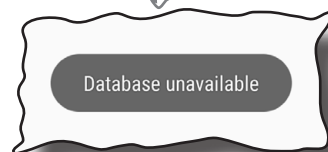
Once you have a reference to the database, you can get data from it using a cursor. We'll look at cursors next.



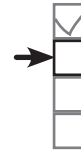
Database reference
Create cursor
Navigate to record
Display drink

↑
 This is a Context, in this case the current activity.

These lines create a Toast that will display the message "Database unavailable" for a few seconds.

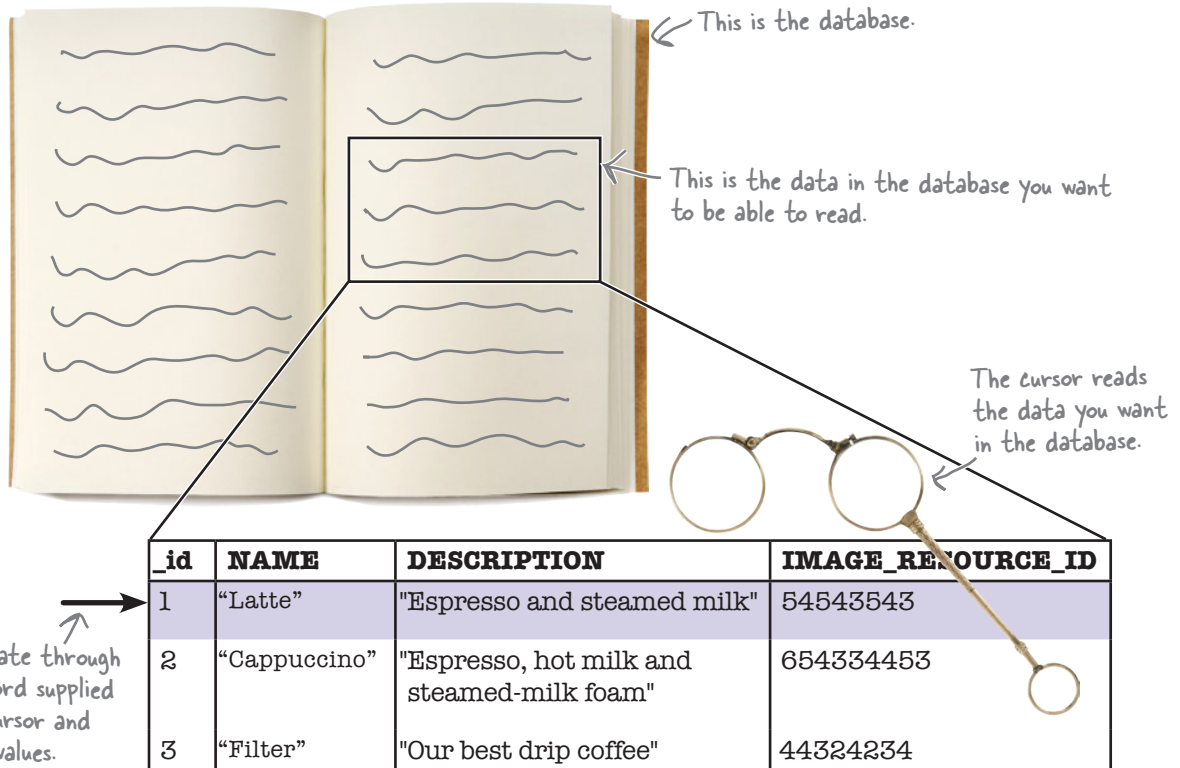


Get data from the database with a cursor



- Database reference
- Create cursor
- Navigate to record
- Display drink

As we said in Chapter 15, a **cursor** lets you read from and write to the database. You specify what data you want access to, and the cursor brings back the relevant records from the database. You can then navigate through the records supplied by the cursor.



You create a cursor using a **database query**. A database query lets you specify which records you want access to from the database. As an example, you can say you want to access all the records from the DRINK table, or just one particular record. These records are then returned in the cursor.

You create a cursor using the `SQLiteDatabase.query()` method:

The `query()` method returns a `Cursor` object. → `Cursor cursor = db.query(...);`

The `query()` method parameters go here. We'll look at them over the next few pages.

There are many overloaded versions of this method with different parameters, so rather than go into each variation, we're only going to show you the most common ways of using it.

Return all the records from a table

The simplest type of database query is one that returns all the records from a particular table in the database. This is useful if, for instance, you want to display all the records in a list in an activity. As an example, here's how you'd return the values in the `_id`, `NAME`, and `DESCRIPTION` columns for every record in the `DRINK` table:

This is the name of the table. →

```
Cursor cursor = db.query("DRINK",
```

new String[] {"_id", "NAME", "DESCRIPTION"},

Set these parameters to null → when you want to return all records from a table. → null, null, null, null, null);

We want to return the values in these columns. →

The query returns all the data from the `_id`, `NAME`, and `DESCRIPTION` columns in the `DRINK` table. →

<code>_id</code>	<code>NAME</code>	<code>DESCRIPTION</code>
1	"Latte"	"Espresso and steamed milk"
2	"Cappuccino"	"Espresso, hot milk and steamed-milk foam"
3	"Filter"	"Our best drip coffee"

To return all the records from a particular table, you pass the name of the table as the `query()` method's first parameter, and a `String` array of the column names as the second. You set all of the other parameters to `null`, as you don't need them for this type of query.

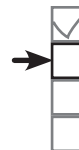
```
Cursor cursor = db.query("TABLE_NAME",
```

new String[] {"COLUMN1", "COLUMN2"},

There are five extra parameters → you need to set to null. → null, null, null, null, null);

You specify each column whose value you want to return in an array of `Strings`. →

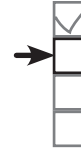
Next we'll look at how you can return the records in a particular order.



Database reference
Create cursor
Navigate to record
Display drink

Behind the scenes, Android uses the `query()` method to construct an SQL `SELECT` statement.

Database reference
 Create cursor
 Navigate to record
 Display drink



Return records in a particular order

If you want to display data in your app in a particular order, you use the query to sort the data by a particular column. This can be useful if, for example, you want to display drink names in alphabetical order.

By default, the data in the table appears in `_id` order, as this was the order in which data was entered:

<code>_id</code>	<code>NAME</code>	<code>DESCRIPTION</code>	<code>IMAGE_RESOURCE_ID</code>	<code>FAVORITE</code>
1	"Latte"	"Espresso and steamed milk"	54543543	0
2	"Cappuccino"	"Espresso, hot milk and steamed-milk foam"	654334453	0
3	"Filter"	"Our best drip coffee"	44324234	1

If you wanted to retrieve data from the `_id`, `NAME`, and `FAVORITE` column in ascending `NAME` order, you would use the following:

```
Cursor cursor = db.query("DRINK",
    new String[] {"_id", "NAME", "FAVORITE"},
    null, null, null, null,
    "NAME ASC");
```

← Order by NAME in ascending order.

<code>_id</code>	<code>NAME</code>	<code>FAVORITE</code>
2	"Cappuccino"	0
3	"Filter"	1
1	"Latte"	0

The `ASC` keyword means that you want to order that column in ascending order. Columns are ordered in ascending order by default, so if you want you can omit the `ASC`. To order the data in descending order instead, you'd use `DESC`.

You can sort by multiple columns too. As an example, here's how you'd order by `FAVORITE` in descending order, followed by `NAME` in ascending order:

```
Cursor cursor = db.query("DRINK",
    new String[] {"_id", "NAME", "FAVORITE"},
    null, null, null, null,
    "FAVORITE DESC, NAME");
```

← Order by FAVORITE in descending order, then NAME in ascending order.

<code>_id</code>	<code>NAME</code>	<code>FAVORITE</code>
3	"Filter"	1
2	"Cappuccino"	0
1	"Latte"	0

Next we'll look at how you return selected records from the database.

Return selected records

You can filter your data by declaring conditions the data must meet, just as you did in Chapter 15. As an example, here's how you'd return records from the DRINK table where the name of the drink is "Latte":

```
Cursor cursor = db.query("DRINK",
    new String[] {"_id", "NAME", "DESCRIPTION"},
    "NAME = ?",
    new String[] {"Latte"},
    null, null, null);
```

These are the columns we want to return.

We want to return records where the value of the NAME column is "Latte".

The third and fourth parameters in the query describe the conditions the data must meet.

The third parameter specifies the column in the condition. In the above example we want to return records where the value of the NAME column is "Latte", so we use "NAME = ?". We want the value in the NAME column to be equal to some value, and the ? symbol is a placeholder for this value.

The fourth parameter is an array of Strings that specifies what the value of the condition should be. In the above example we want to update records where the value of the NAME column is "Latte", so we use:

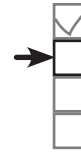
```
new String[] {"Latte"};
```

The value of the condition must be an array of Strings, even if the column you're applying the condition to contains some other type of data. As an example, here's how you'd return records from the DRINK table where the drink_id is 1:

```
Cursor cursor = db.query("DRINK",
    new String[] {"_id", "NAME", "DESCRIPTION"},
    "_id = ?",
    new String[] {Integer.toString(1)},
    null, null, null);
```

Convert the int 1 to a String value.

You've now seen the most common ways of using the query() method to create a cursor, so try the following exercise to construct the cursor we need for DrinkActivity.java.



- Database reference**
- Create cursor**
- Navigate to record**
- Display drink**

<u>_id</u>	NAME	DESCRIPTION
1	"Latte"	"Espresso and steamed milk"

The query returns all the data from the NAME and DESCRIPTION columns in the DRINK table where the value of the NAME column is "Latte".

To find out more ways of using the query() method, visit the SQLiteDatabase documentation:
<https://developer.android.com/reference/android/database/sqlite/SQLiteDatabase.html>



Code Magnets

In our code for `DrinkActivity`, we want to get the name, description, and image resource ID for the drink ID passed to it in an intent. Can you create a cursor that will do that?

```
...
int drinkId = (Integer) getIntent().getExtras().get(EXTRA_DRINKID);

//Create a cursor
SQLiteOpenHelper starbuzzDatabaseHelper = new StarbuzzDatabaseHelper(this);
try {
    SQLiteDatabase db = starbuzzDatabaseHelper.getReadableDatabase();

    Cursor cursor = db.query(.....,

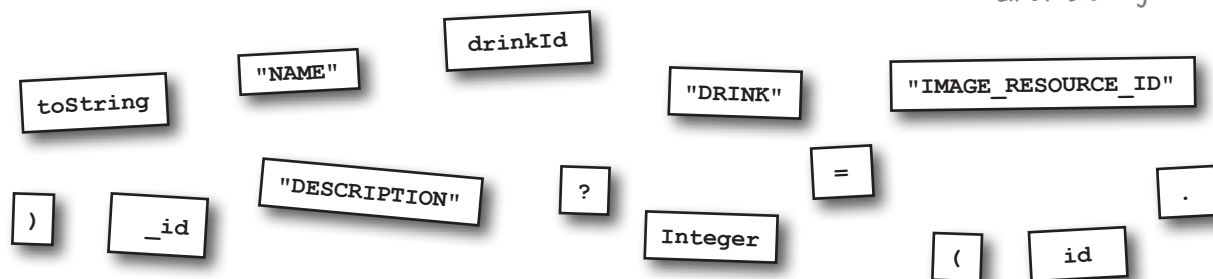
        new String[] { .....', .....', .....'},

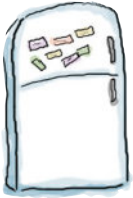
        ".....",

        new String[] { .....'},

        null, null, null);
} catch (SQLException e) {
    Toast toast = Toast.makeText(this, "Database unavailable", Toast.LENGTH_SHORT);
    toast.show();
}
...
```

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





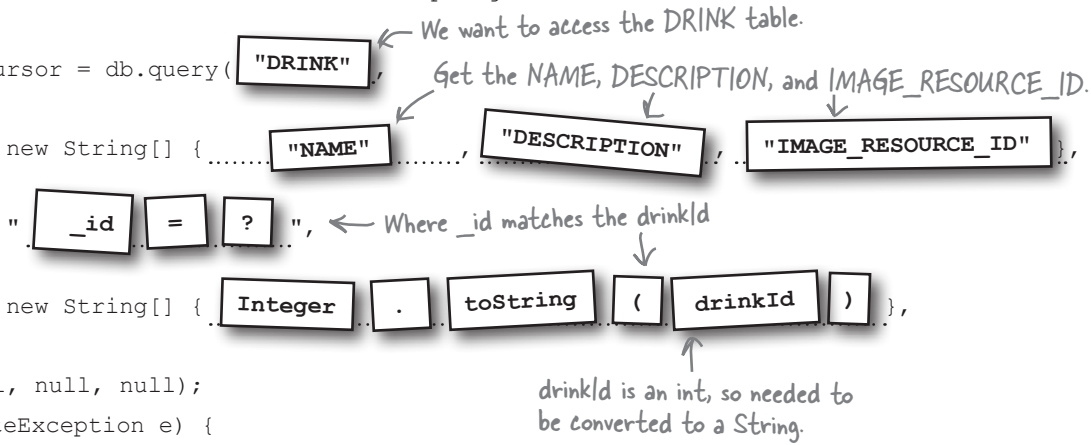
Code Magnets Solution

In our code for DrinkActivity we want to get the name, description, and image resource ID for the drink passed to it in an intent. Can you create a cursor that will do that?

...

```
int drinkId = (Integer) getIntent().getExtras().get(EXTRA_DRINKID);

//Create a cursor
SQLiteOpenHelper starbuzzDatabaseHelper = new StarbuzzDatabaseHelper(this);
try {
    SQLiteDatabase db = starbuzzDatabaseHelper.getReadableDatabase();
    Cursor cursor = db.query("DRINK",
        new String[] { "NAME", "DESCRIPTION", "IMAGE_RESOURCE_ID" },
        "_id = ?",
        new String[] { Integer.toString(drinkId) },
        null, null, null);
} catch (SQLException e) {
    Toast toast = Toast.makeText(this, "Database unavailable", Toast.LENGTH_SHORT);
    toast.show();
}
...
```



You didn't need to use this magnet.

- Database reference
- Create cursor
- Navigate to record
- Display drink



The DrinkActivity code so far

We want to change *DrinkActivity.java*'s `onCreate()` method so that *DrinkActivity* gets its drink data from the Starbuzz database instead of from the *Drink* Java class. Here's the code so far (we suggest you wait until we show you the *full* code—a few pages ahead—before you update your version of *DrinkActivity.java*):

```

package com.hfad.starbuzz;
...
public class DrinkActivity extends Activity {

    public static final String EXTRA_DRINKID = "drinkId";

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

        //Get the drink from the intent
        int drinkId = (Integer) getIntent().getExtras().get(EXTRA_DRINKID);
        Drink drink = Drink.drinks[drinkId];

        //Create a cursor
        SQLiteOpenHelper starbuzzDatabaseHelper = new StarbuzzDatabaseHelper(this);
        try {
            SQLiteDatabase db = starbuzzDatabaseHelper.getReadableDatabase();
            Cursor cursor = db.query("DRINK",
                new String[] {"NAME", "DESCRIPTION", "IMAGE_RESOURCE_ID"},
                "_id = ?",
                new String[] {Integer.toString(drinkId)},
                null, null, null);
        } catch (SQLException e) {
            Toast toast = Toast.makeText(this, "Database unavailable", Toast.LENGTH_SHORT);
            toast.show();
        }
        ...
    }
}

```

Our Starbuzz code uses the Activity class, but we could have changed the code to use AppCompatActivity if we'd wanted to.

We'll add the code to the onCreate() method.

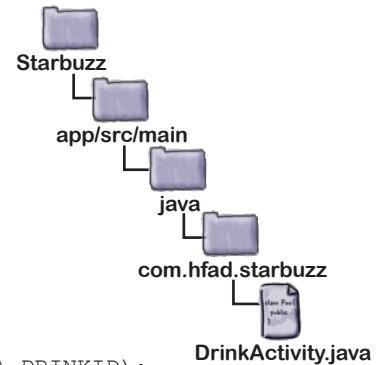
We're no longer getting the drinks from Drink.java.

Get a reference to the database.

Create a cursor to get the name, description, and image resource ID of the drink the user selected.

Display a pop-up message if a SQLException is thrown.

There's more code we haven't changed yet, but you don't need to see it right now.



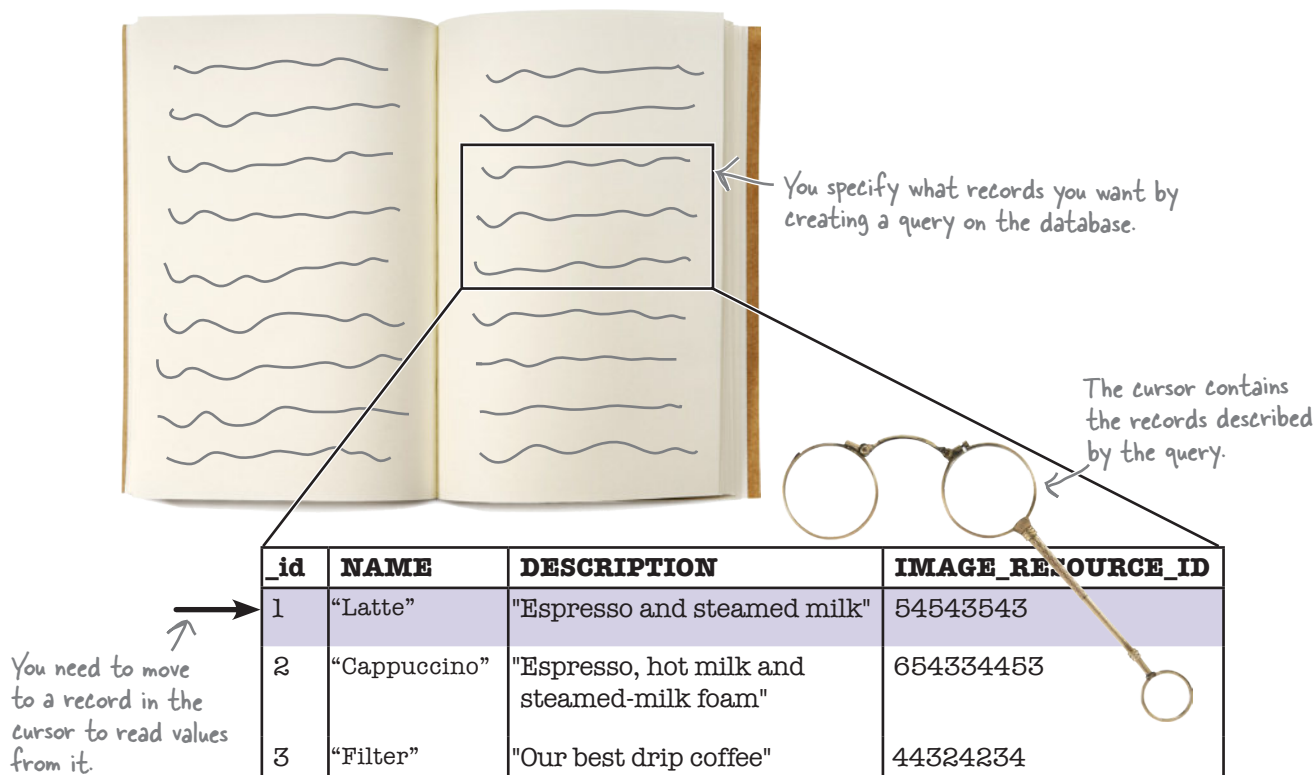
Now that we've created our cursor, the next thing we need to do is get the drink name, description, and image resource ID from the cursor so that we can update *DrinkActivity*'s views.

To read a record from a cursor, you first need to navigate to it



You've now seen how to create a cursor; you get a reference to a SQLiteDatabase, and then use its `query()` method to say what data you want the cursor to return. But that's not the end of the story—once you have a cursor, you need to read values from it.

Whenever you need to retrieve values from a particular record in a cursor, you first need to navigate to that record.



In our particular case, we have a cursor that's composed of a single record that contains details of the drink the user selected. We need to navigate to that record in order to read details of the drink.

Navigate cursors

There are four main methods you use to navigate through the records in a cursor: `moveToFirst()`, `moveToLast()`, `moveToPrevious()`, and `moveToNext()`.

To go to the first record in a cursor, you use its `moveToFirst()` method. This method returns a value of `true` if it finds a record, and `false` if the cursor hasn't returned any records):

```
if (cursor.moveToFirst()) {
    //Do something
};
```

To go to the last record, you use the `moveToLast()` method. Just like the `moveToFirst()` method, it returns a value of `true` if it finds a record, and `false` if it doesn't:

```
if (cursor.moveToLast()) {
    //Do something
};
```

To move through the records in the cursor, you use the `moveToPrevious()` and `moveToNext()` methods.

The `moveToPrevious()` method moves you to the previous record in the cursor. It returns `true` if it succeeds in moving to the previous record, and `false` if it fails (for example, if it's already at the first record):

```
if (cursor.moveToPrevious()) {
    //Do something
};
```

The `moveToNext()` method works in a similar way to the `moveToPrevious()` method, except that it moves you to the next record in the cursor instead:

```
if (cursor.moveToNext()) {
    //Do something
};
```

In our case, we want to read values from the first (and only) record in the cursor, so we'll use the `moveToFirst()` method to navigate to this record.

Once you've navigated to a record in your cursor, you can access its values. We'll look at how to do that next.



Database reference
Create cursor
Navigate to record
Display drink

Move to the first row.

NAME	DESCRIPTION
"Latte"	"Espresso and steamed milk"
Cappuccino	"Espresso, hot milk and steamed-milk foam"
Filter	"Our best drip coffee"

NAME	DESCRIPTION
"Latte"	"Espresso and steamed milk"
Cappuccino	"Espresso, hot milk and steamed-milk foam"
Filter	"Our best drip coffee"

Move to the last row.

NAME	DESCRIPTION
"Latte"	"Espresso and steamed milk"
Cappuccino	"Espresso, hot milk and steamed-milk foam"
Filter	"Our best drip coffee"

Move to the previous row.

NAME	DESCRIPTION
"Latte"	"Espresso and steamed milk"
Cappuccino	"Espresso, hot milk and steamed-milk foam"
Filter	"Our best drip coffee"

Move to the next row.

Get cursor values

You retrieve values from a cursor's current record using the cursor's `get*()` methods: `getString()`, `getInt()`, and so on. The exact method you use depends on the type of value you want to retrieve. To get a String value, for example, you'd use the `getString()` method, and to get an int value you'd use `getInt()`. Each of the methods takes a single parameter: the index of the column whose value you want to retrieve, starting at 0.

As an example, we're using the following query to create our cursor:

```
Cursor cursor = db.query ("DRINK",
    new String[] {"NAME", "DESCRIPTION", "IMAGE_RESOURCE_ID"},
    "_id = ?",
    new String[] {Integer.toString(1)},
    null, null, null);
```

The cursor has three columns: NAME, DESCRIPTION, and IMAGE_RESOURCE_ID. The first two columns, NAME and DESCRIPTION, contain Strings, and the third column, IMAGE_RESOURCE_ID, contains int values.

Suppose you wanted to get the value of the NAME column for the current record. NAME is the first column in the cursor, and contains String values. You'd therefore use the `getString()` method, passing it a parameter of 0 for the column index:

```
String name = cursor.getString(0); ← NAME is column 0 and contains Strings.
```

Similarly, suppose you wanted to get the contents of the IMAGE_RESOURCE_ID column. This has a column index of 2 and contains int values, so you'd use the code:

```
int imageResource = cursor.getInt(2); ← IMAGE_RESOURCE_ID is column 2 and contains ints.
```

Finally, close the cursor and the database

Once you've finished retrieving values from the cursor, you need to close the cursor and the database in order to release their resources. You do this by calling the cursor and database `close()` methods:

```
cursor.close();
db.close();
```

These lines close the cursor and the database.

We've now covered all the code you need to replace the code in `DrinkActivity` so that it gets its data from the Starbuzz database. Let's look at the revised code in full.



Database reference
Create cursor
Navigate to record
Display drink

These are the cursor's columns.

Column 0	Column 1	Column 2
NAME	DESCRIPTION	IMAGE_RESOURCE_ID
"Latte"	"Espresso and steamed milk"	54543543

You can find details of all the cursor get methods in <http://developer.android.com/reference/android/database/Cursor.html>.

- Database reference
- Create cursor
- Navigate to record
- Display drink



The DrinkActivity code

Here's the full code for *DrinkActivity.java* (apply the changes in bold to your code, then save your work):

```
package com.hfad.starbuzz;

import android.app.Activity;
import android.os.Bundle;
import android.widget.ImageView;
import android.widget.TextView;
import android.widget.Toast;
import android.database.Cursor;
import android.database.sqlite.SQLiteDatabase;
import android.database.sqlite.SQLiteException;
import android.database.sqlite.SQLiteOpenHelper;

public class DrinkActivity extends Activity {

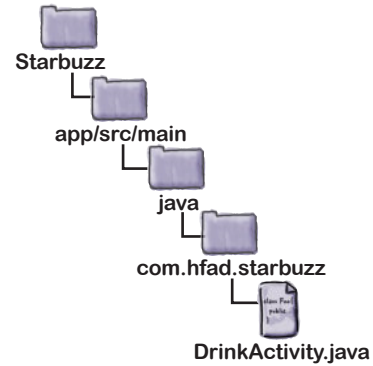
    public static final String EXTRA_DRINKID = "drinkId";

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

        //Get the drink from the intent
        int drinkId = (Integer) getIntent().getExtras().get(EXTRA_DRINKID);
        Drink drink = Drink.drinks[drinkId];

        //Create a cursor
        SQLiteOpenHelper starbuzzDatabaseHelper = new StarbuzzDatabaseHelper(this);
        try {
            SQLiteDatabase db = starbuzzDatabaseHelper.getReadableDatabase();
            Cursor cursor = db.query ("DRINK",
                new String[] { "NAME", "DESCRIPTION", "IMAGE_RESOURCE_ID" },
                "_id = ?",
                new String[] { Integer.toString(drinkId) },
                null, null, null);
```

We're using these extra classes in the code.



This is the ID of the drink the user chose.

We're no longer getting our data from the drinks array, so we need to delete this line.

Create a cursor that gets the NAME, DESCRIPTION, and IMAGE_RESOURCE_ID data from the DRINK table where _id matches drinkId.

The code continues on the next page. →

The DrinkActivity code (continued)



Database reference
Create cursor
Navigate to record
Display drink

```
//Move to the first record in the Cursor
```

```
if (cursor.moveToFirst()) { ← There's only one record in the cursor,
                             but we still need to move to it.
```

The name of the drink is the first item in the cursor, the description is the second column, and the image resource ID is the third. That's because we told the cursor to use the NAME, DESCRIPTION, and IMAGE_RESOURCE_ID columns from the database in that order.

```
//Get the drink details from the cursor
```

```
String nameText = cursor.getString(0);
```

```
String descriptionText = cursor.getString(1);
```

```
int photoId = cursor.getInt(2);
```

```
//Populate the drink name
```

```
TextView name = (TextView)findViewById(R.id.name);
```

```
name.setText(drink.getName());
```

```
name.setText(nameText);
```

← Set the drink name to the value from the database.

```
//Populate the drink description
```

```
TextView description = (TextView)findViewById(R.id.description);
```

```
description.setText(drink.getDescription());
```

```
description.setText(descriptionText);
```

← Use the drink description from the database.

```
//Populate the drink image
```

```
ImageView photo = (ImageView)findViewById(R.id.photo);
```

```
photo.setImageResource(drink.getImageResourceId());
```

```
photo.setContentDescription(drink.getName());
```

```
photo.setImageResource(photoId);
```

```
photo.setContentDescription(nameText);
```

← Set the image resource ID and description to the values from the database.

```
}
```

```
cursor.close();
```

```
db.close(); ← Close the cursor and database.
```

```
} catch (SQLException e) {
```

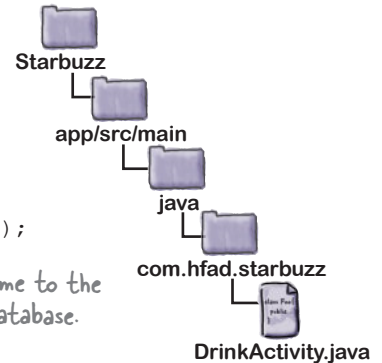
```
    Toast toast = Toast.makeText(this,
```

```
        "Database unavailable",
```

```
        Toast.LENGTH_SHORT);
```

```
    toast.show();
```

↑
If a `SQLException` is thrown, this means there's a problem with the database. In this case, we'll use a toast to display a message to the user.



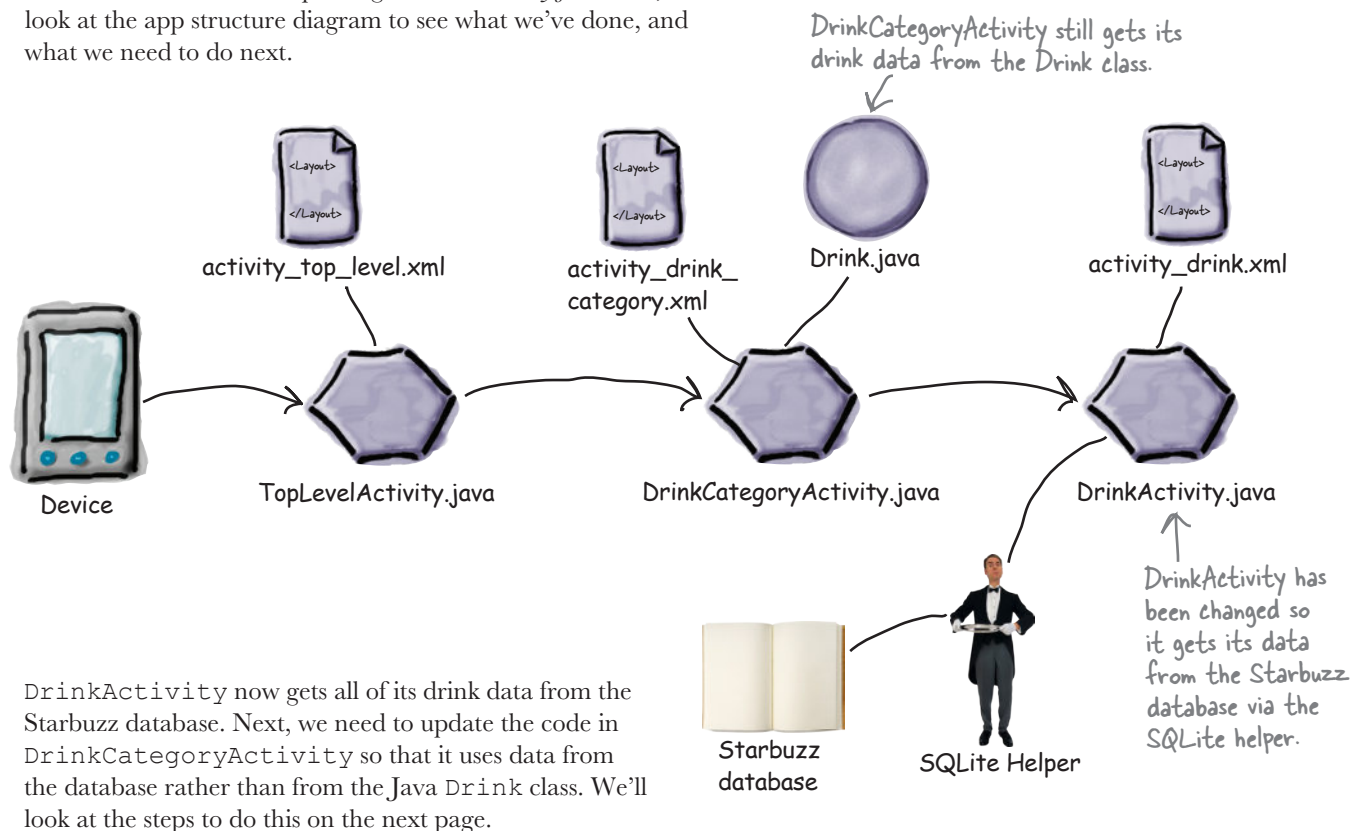
Connecting your activities to a database takes more code than using a Java class.

But if you take your time working through the code in this chapter, you'll be fine.

So that's the complete `DrinkActivity` code. Let's review where we've got to, and what we need to do next.

What we've done so far

Now that we've finished updating the *DrinkActivity.java* code, let's look at the app structure diagram to see what we've done, and what we need to do next.



DrinkActivity now gets all of its drink data from the Starbuzz database. Next, we need to update the code in *DrinkCategoryActivity* so that it uses data from the database rather than from the Java *Drink* class. We'll look at the steps to do this on the next page.

there are no Dumb Questions

Q: How much SQL do I need to know to create cursors?

A: It's useful to have an understanding of SQL `SELECT` statements, as behind the scenes the `query()` method translates to one. In general, your queries probably won't be too complex, but SQL knowledge is a useful skill.

If you want to learn more about SQL, we suggest getting a copy of *Head First SQL* by Lynn Beighley.

Q: You said that if the database can't be accessed, a `SQLiteException` is thrown. How should I deal with it?

A: First, check the exception details. The exception might be caused by an error in SQL syntax, which you can then rectify.

How you handle the exception depends on the impact it has on your app. As an example, if you can get read access to the database but can't write to it, you can still give the user read-only access to the database, but you might want to tell the user that you can't save their changes. Ultimately, it all depends on your app.

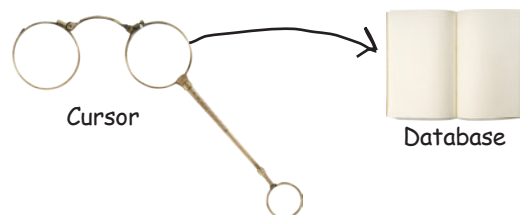
What we'll do to change `DrinkCategoryActivity` to use the Starbuzz database

When we updated `DrinkActivity` to get it to read data from the Starbuzz database, we created a cursor to read data for the drink the user selected, and then we used the values from the cursor to update `DrinkActivity`'s views.

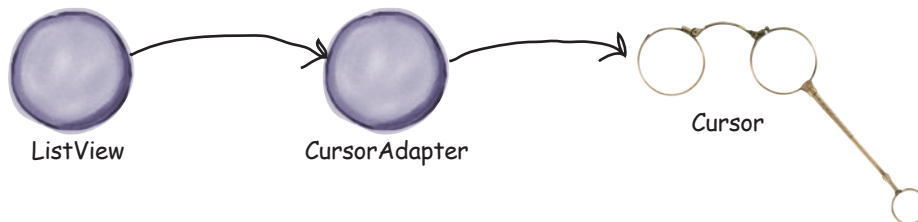
The steps we need to go through to update `DrinkCategoryActivity` are slightly different. This is because `DrinkCategoryActivity` displays a list view that uses the drink data as its source. We need to switch the source of this data to be the Starbuzz database.

Here are the steps we need to go through to change `DrinkCategoryActivity` so that it uses the Starbuzz database:

- 1 Create a cursor to read drink data from the database.**
As before, we need to get a reference to the Starbuzz database. Then we'll create a cursor to retrieve the drink names from the `DRINK` table.



- 2 Replace the list view's array adapter with a cursor adapter.**
The list view currently uses an array adapter to get its drink names. This is because the data's held in an array in the `Drink` class. Because we're now accessing the data using a cursor, we'll use a cursor adapter instead.



Before we get started on these tasks, let's remind ourselves of the `DrinkCategoryActivity.java` code we created in Chapter 7.



The current DrinkCategoryActivity code

Here's a reminder of what the current *DrinkCategoryActivity.java* code looks like. The `onCreate()` method populates a list view with drinks using an array adapter. The `onListItemClick()` method adds the drink the user selects to an intent, and then starts *DrinkActivity*:

```
package com.hfad.starbuzz;

...

public class DrinkCategoryActivity extends Activity {

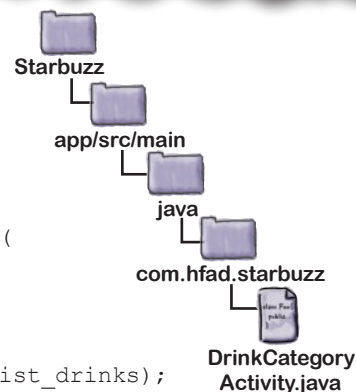
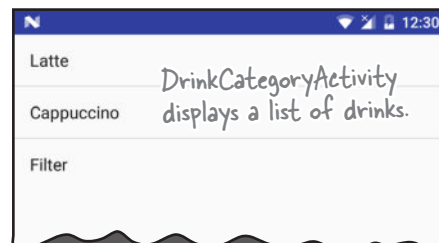
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_drink_category);
        ArrayAdapter<Drink> listAdapter = new ArrayAdapter<>(
            this,
            android.R.layout.simple_list_item_1,
            Drink.drinks);
        ListView listDrinks = (ListView) findViewById(R.id.list_drinks);
        listDrinks.setAdapter(listAdapter);

        //Create a listener to listen for clicks in the list view
        AdapterView.OnItemClickListener itemClickListener =
            new AdapterView.OnItemClickListener() {
                public void onItemClick(AdapterView<?> listDrinks,
                    View itemView,
                    int position,
                    long id) {

                    //Pass the drink the user clicks on to DrinkActivity
                    Intent intent = new Intent(DrinkCategoryActivity.this,
                        DrinkActivity.class);
                    intent.putExtra(DrinkActivity.EXTRA_DRINKID, (int) id);
                    startActivity(intent);
                }
            };

        //Assign the listener to the list view
        listDrinks.setOnItemClickListener(itemClickListener);
    }
}
```

At the moment, we're using an *ArrayAdapter* to bind an array to the *ListView*. We need to replace this code so that the data comes from a database instead.





Get a reference to the Starbuzz database...

We need to change `DrinkCategoryActivity` so that it gets its data from the Starbuzz database. Just as before, this means that we need to create a cursor to return the data we need.

We start by getting a reference to the database. We only need to read the drink data and not update it, so we'll use the `getReadableDatabase()` method as we did before:

```
SQLiteOpenHelper starbuzzDatabaseHelper = new StarbuzzDatabaseHelper(this);
SQLiteDatabase db = starbuzzDatabaseHelper.getReadableDatabase();
```

...then create a cursor that returns the drinks

To create the cursor, we need to specify what data we want it to contain. We want to use the cursor to display a list of drink names, so the cursor needs to include the `NAME` column. We'll also include the `_id` column to get the ID of the drink; we need to pass the ID of the drink the user chooses to `DrinkActivity` so that `DrinkActivity` can display its details. Here's the cursor:

```
cursor = db.query("DRINK",
    new String[]{"_id", "NAME"},
    null, null, null, null, null);
```

We get a reference to the database in exactly the same way that we did earlier in this chapter.

This cursor returns the `_id` and `NAME` of every record in the `DRINK` table.

Putting this together, here's the code to get a reference to the database and create the cursor (you'll add this code to `DrinkCategoryActivity.java` later when we show you the full code listing):

```
SQLiteOpenHelper starbuzzDatabaseHelper = new StarbuzzDatabaseHelper(this);
try {
    SQLiteDatabase db = starbuzzDatabaseHelper.getReadableDatabase();
    cursor = db.query("DRINK",
        new String[]{"_id", "NAME"},
        null, null, null, null, null);
    //Code to use data from the database
} catch (SQLException e) {
    Toast toast = Toast.makeText(this,
        "Database unavailable",
        Toast.LENGTH_SHORT);
    toast.show();
}
```

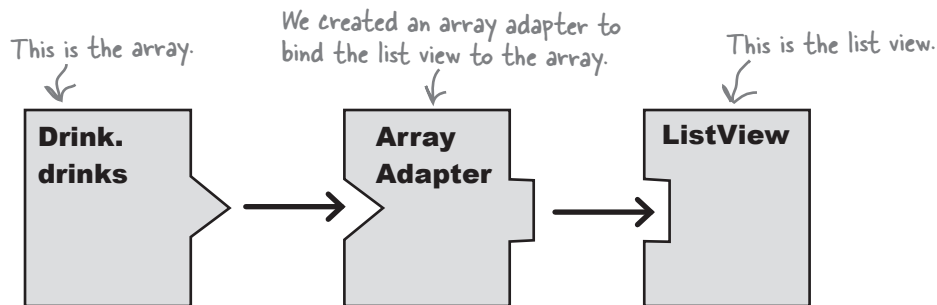
If the database is unavailable, a `SQLException` gets thrown. If this happens, we'll use a toast to display a pop-up message as before.

Next we'll use the cursor's data to populate `DrinkCategoryActivity`'s list view.

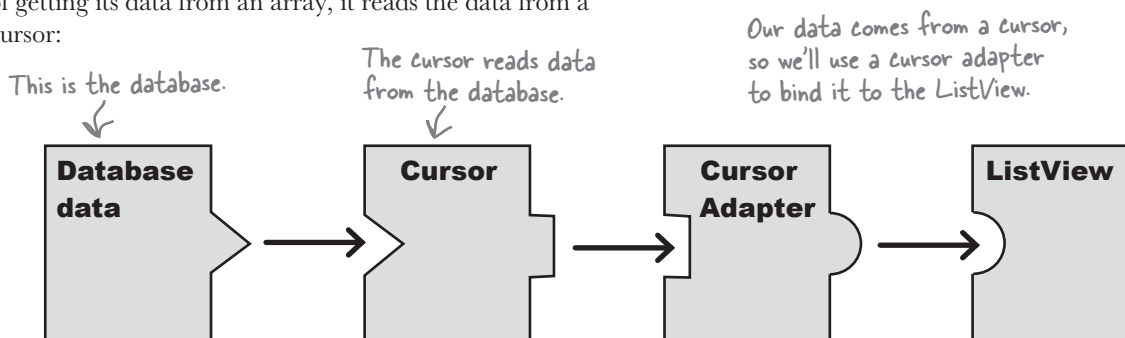


How do we replace the array data in the list view?

When we wanted `DrinkCategoryActivity`'s list view to display data from the `Drink.drinks` array, we used an array adapter. As you saw back in Chapter 7, an array adapter is a type of adapter that works with arrays. It acts as a bridge between the data in an array and a list view:



Now that we're getting our data from a cursor, we'll use a **cursor adapter** to bind the data to our list view. A cursor adapter is just like an array adapter, except that instead of getting its data from an array, it reads the data from a cursor:



We'll look at this in more detail on the next page.

ListViews and Spinners can use any subclass of the Adapter class for their data. This includes ArrayAdapter, CursorAdapter, and SimpleCursorAdapter (a subclass of CursorAdapter).



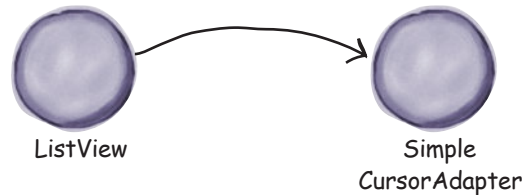
A simple cursor adapter maps cursor data to views

We're going to create a simple cursor adapter, a type of cursor adapter that can be used in most cases where you need to display cursor data in a list view. It takes columns from a cursor, and maps them to text views or image views, for example in a list view.

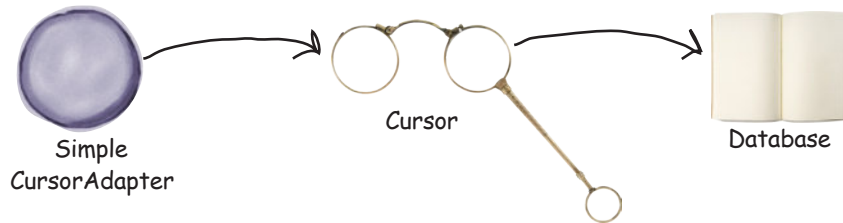
In our case, we want to display a list of drink names. We'll use a simple cursor adapter to map the name of each drink returned by our cursor to `DrinkCategoryActivity`'s list view.

Here's how it will work:

- 1 The list view asks the adapter for data.

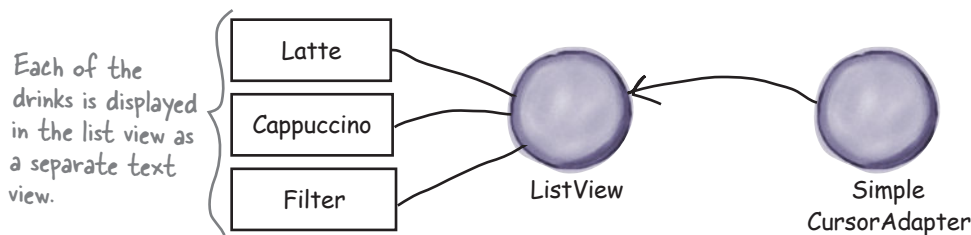


- 2 The adapter asks the cursor for data from the database.



- 3 The adapter returns the data to the list view.

The name of each drink is displayed in the list view as a separate text view.



Let's construct the simple cursor adapter.



How to use a simple cursor adapter

You use a simple cursor adapter in a similar way to how you use an array adapter: you initialize the adapter, then attach it to the list view.

We're going to create a simple cursor adapter to display a list of drink names from the DRINK table. To do this, we'll create a new instance of the `SimpleCursorAdapter` class, passing in parameters to tell the adapter what data to use and how it should be displayed. Finally, we'll assign the adapter to the list view.

Here's the code (you'll add it to `DrinkCategoryActivity.java` later in the chapter):

```
SimpleCursorAdapter listAdapter = new SimpleCursorAdapter(this,
    android.R.layout.simple_list_item_1,
    cursor,
    new String[]{"NAME"},
    new int[]{android.R.id.text1},
    0);
listDrinks.setAdapter(listAdapter);
```

Handwritten annotations:

- "this" is the current activity. (points to `this`)
- This is the same layout we used with the array adapter. It displays a single value for each row in the list view. (points to `android.R.layout.simple_list_item_1`)
- This is the cursor. (points to `cursor`)
- Display the contents of the NAME column in the ListView text views. (points to `new String[]{"NAME"}`)
- Use `setAdapter()` to connect the adapter to the list view. (points to `listDrinks.setAdapter(listAdapter);`)

The general form of the `SimpleCursorAdapter` constructor looks like this:

```
SimpleCursorAdapter adapter = new SimpleCursorAdapter(Context context,
    int layout,
    Cursor cursor,
    String[] fromColumns,
    int[] toViews,
    int flags)
```

Handwritten annotations:

- This is usually the current activity. (points to `Context context`)
- How to display the data (points to `int layout`)
- The cursor you create. The cursor should include the `_id` column, and the data you want to appear. (points to `Cursor cursor`)
- Which columns in the cursor to match to which views (points to `String[] fromColumns`)
- Used to determine the behavior of the cursor (points to `int flags`)

The `context` and `layout` parameters are exactly the same ones you used when you created an array adapter: `context` is the current context, and `layout` says how you want to display the data. Instead of saying which array we need to get our data from, we need to specify which cursor contains the data. You then use `fromColumns` to specify which columns in the cursor you want to use, and `toViews` to say which views you want to display them in.

The `flags` parameter is usually set to 0, which is the default. The alternative is to set it to `FLAG_REGISTER_CONTENT_OBSERVER` to register a content observer that will be notified when the content changes. We're not covering this alternative here, as it can lead to memory leaks (you'll see how to deal with changing content in the next chapter).

Any cursor you use with a cursor adapter MUST include the `_id` column or it won't work.



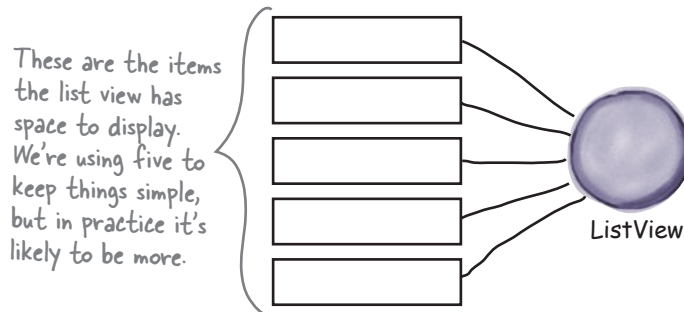
Close the cursor and database

When we introduced you to cursors earlier in the chapter, we said that you needed to close the cursor and database after you'd finished with it in order to release their resources. In our `DrinkActivity` code, we used a cursor to retrieve drink details from the database, and once we'd used these values with our views, we immediately closed the cursor and database.

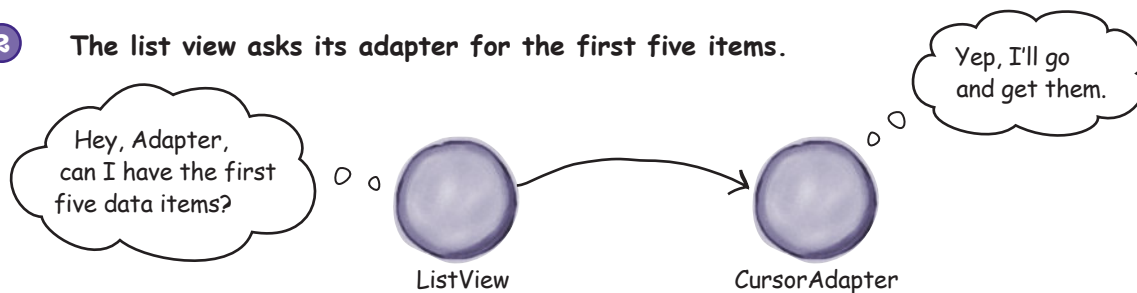
When you use a cursor adapter, (including a simple cursor adapter) it works slightly differently; the cursor adapter needs the cursor to stay open in case it needs to retrieve more data from it. Let's look in more detail at how cursor adapters work to see why this might happen.

1 The list view gets displayed on the screen.

When the list is first displayed, it will be sized to fit the screen. Let's say it has space to show five items.

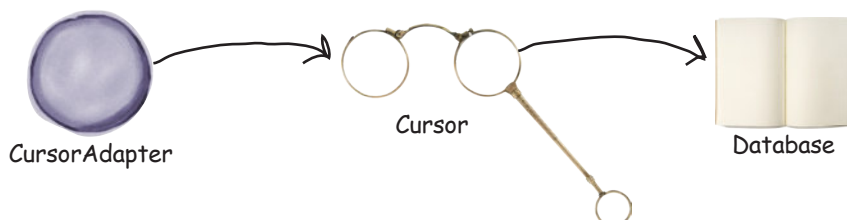


2 The list view asks its adapter for the first five items.



3 The cursor adapter asks its cursor to read five rows from the database.

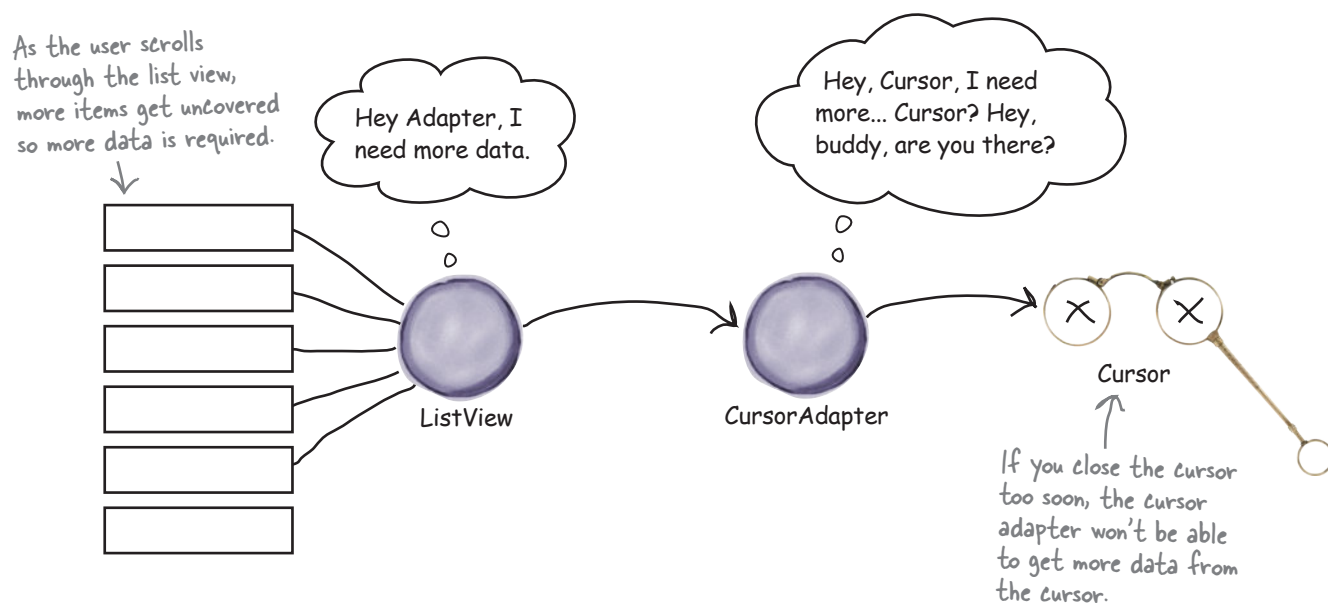
No matter how many rows the database table contains, the cursor only needs to read the first five rows.



The story continues

4 The user scrolls the list.

As the user scrolls the list, the adapter asks the cursor to read more rows from the database. This works fine if the cursor's still open. But if the cursor's already been closed, the cursor adapter can't get any more data from the database.



This means that you can't immediately close the cursor and database once you've used the `setAdapter()` method to connect the cursor adapter to your list view. Instead, we'll close the cursor and database in the activity's `onDestroy()` method, which gets called just before the activity is destroyed. Because the activity's being destroyed, there's no further need for the cursor or database connection to stay open, so they can be closed:

```
public void onDestroy() {
    super.onDestroy();
    cursor.close();
    db.close();
}
```

← Close the cursor and database when the activity is destroyed.

That's everything you need to know in order to update the code for `DrinkCategoryActivity`, so have a go at the exercise on the next page.

Pool Puzzle



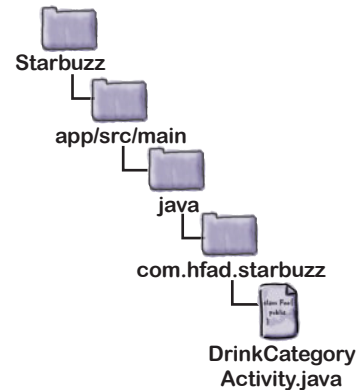
Your **job** is to take code segments from the pool and place them into the blank lines in *DrinkCategoryActivity.java*. You may **not** use the same code segment more than once, and you won't need to use all the code segments. Your **goal** is to populate the list view with a list of drinks from the database.

```
public class DrinkCategoryActivity extends Activity {

    private SQLiteDatabase db;
    private Cursor cursor;

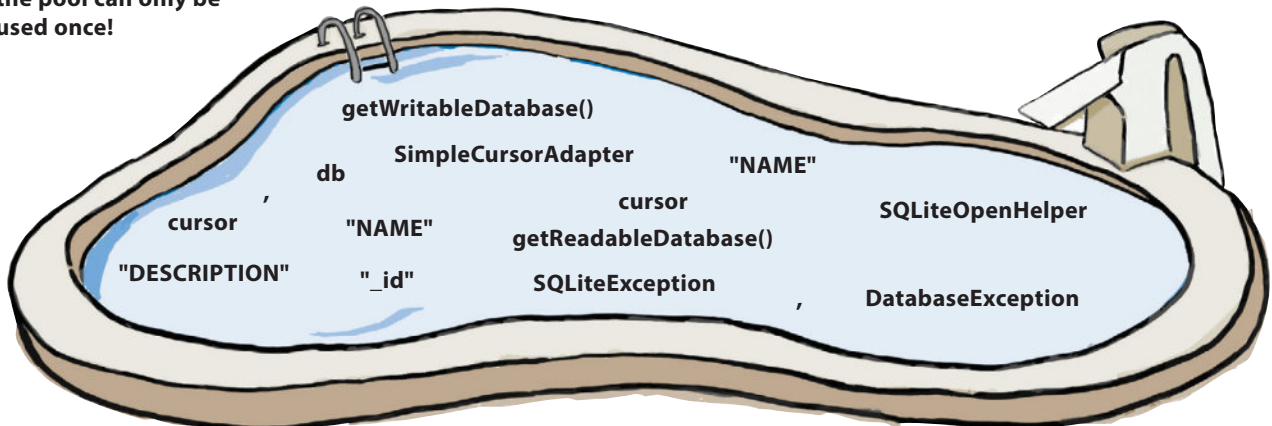
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_drink_category);
        ListView listDrinks = (ListView) findViewById(R.id.list_drinks);

        ..... starbuzzDatabaseHelper = new StarbuzzDatabaseHelper(this);
        try {
            db = starbuzzDatabaseHelper. .... ;
            cursor = db.query("DRINK",
                new String[]{ ..... },
                null, null, null, null, null);
        }
    }
}
```



The code continues → on the next page

Note: each segment in the pool can only be used once!



```

SimpleCursorAdapter listAdapter = new ..... (this,
    android.R.layout.simple_list_item_1,
    ..... '
    new String[]{ ..... },
    new int[]{android.R.id.text1},
    0);

listDrinks.setAdapter(listAdapter);
} catch( ..... e) {
    Toast toast = Toast.makeText(this, "Database unavailable", Toast.LENGTH_SHORT);
    toast.show();
}

//Create the listener
AdapterView.OnItemClickListener itemClickListener =
    new AdapterView.OnItemClickListener() {
        public void onItemClick(AdapterView<?> listDrinks,
            View itemView,
            int position,
            long id) {

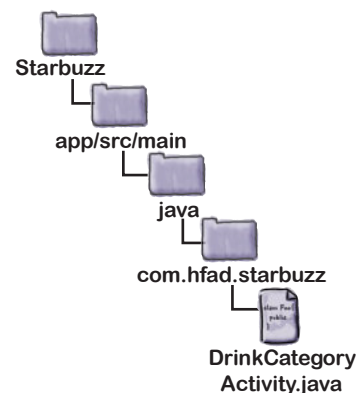
            //Pass the drink the user clicks on to DrinkActivity
            Intent intent = new Intent(DrinkCategoryActivity.this,
                DrinkActivity.class);
            intent.putExtra(DrinkActivity.EXTRA_DRINKID, (int) id);
            startActivity(intent);

        }
    };

//Assign the listener to the list view
listDrinks.setOnItemClickListener(itemClickListener);
}

@Override
public void onDestroy() {
    super.onDestroy();
    ..... .close();
    ..... .close();
}
}

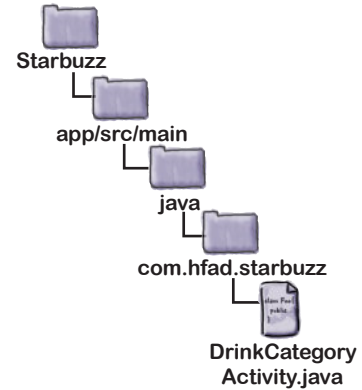
```



Pool Puzzle Solution



Your **job** is to take code segments from the pool and place them into the blank lines in `DrinkCategoryActivity.java`. You may **not** use the same code segment more than once, and you won't need to use all the code segments. Your **goal** is to populate the list view with a list of drinks from the database.



```
public class DrinkCategoryActivity extends Activity {
```

```
private SQLiteDatabase db;
private Cursor cursor;
```

```
@Override
```

```
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_drink_category);
    ListView listDrinks = (ListView) findViewById(R.id.list_drinks);
```

← You get a reference to the database using a `SQLiteOpenHelper`.

```
SQLiteOpenHelper..... starbuzzDatabaseHelper = new StarbuzzDatabaseHelper(this);
```

```
try {
```

```
db = starbuzzDatabaseHelper.getReadableDatabase() ;
```

```
cursor = db.query("DRINK",
```

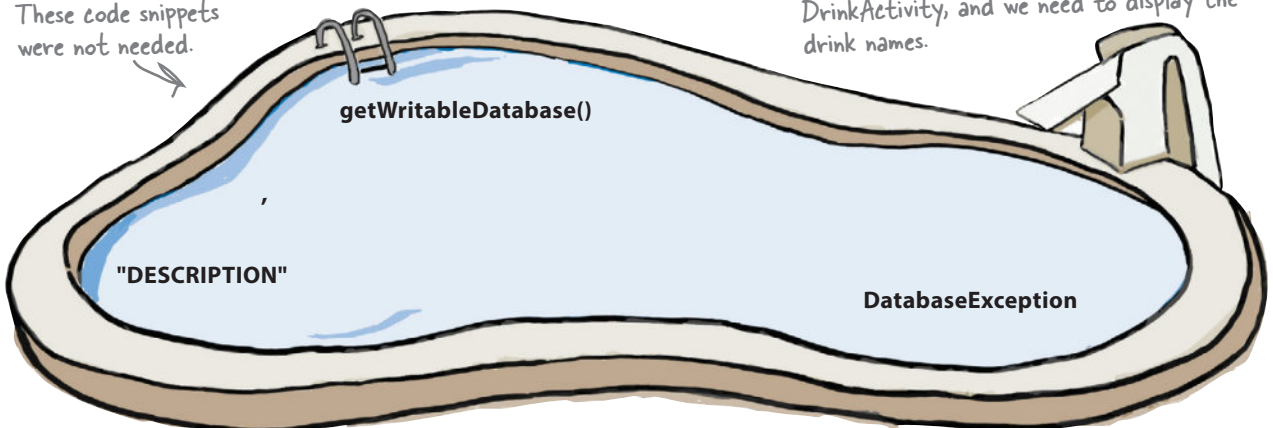
```
new String[]{"_id", "NAME"},
```

```
null, null, null, null, null);
```

← We're reading from the database, so we just need read-only access.

← The cursor must include the `_id` and `NAME` columns. We want to pass the drink ID to `DrinkActivity`, and we need to display the drink names.

These code snippets were not needed.




```

SimpleCursorAdapter listAdapter = new SimpleCursorAdapter (this,
    android.R.layout.simple_list_item_1,
    Use the cursor we just created. → cursor ,
    Display the contents of the NAME column. ← new String[] { "NAME" },
    new int[] { android.R.id.text1 },
    0);

listDrinks.setAdapter(listAdapter);
} catch ( SQLiteException ← e) { ← If the database is unavailable, we'll catch the SQLiteException.
    Toast toast = Toast.makeText(this, "Database unavailable", Toast.LENGTH_SHORT);
    toast.show();
}

//Create the listener
AdapterView.OnItemClickListener itemClickListener =
    new AdapterView.OnItemClickListener() {
        public void onItemClick(AdapterView<?> listDrinks,
            View itemView,
            int position,
            long id) {

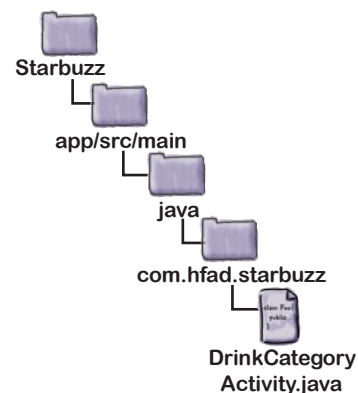
            //Pass the drink the user clicks on to DrinkActivity
            Intent intent = new Intent(DrinkCategoryActivity.this,
                DrinkActivity.class);
            intent.putExtra(DrinkActivity.EXTRA_DRINKID, (int) id);
            startActivity(intent);

        }
    };

//Assign the listener to the list view
listDrinks.setOnItemClickListener(itemClickListener);
}

@Override
public void onDestroy() {
    super.onDestroy();
    cursor .close(); ← Close the cursor before you
    db .close(); ← close the database.
}
}

```



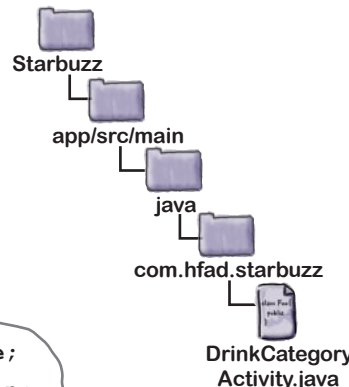


The revised code for DrinkCategoryActivity

Here's the full code for *DrinkCategoryActivity.java*, with the array adapter replaced by a cursor adapter (the changes are in bold); update your code to match ours:

```
package com.hfad.starbuzz;

import android.app.Activity;
import android.os.Bundle;
import android.widget.ListView;
import android.view.View;
import android.content.Intent;
import android.widget.AdapterView;
import android.database.Cursor;
import android.database.sqlite.SQLiteDatabase;
import android.database.sqlite.SQLiteOpenHelper;
import android.widget.SimpleCursorAdapter;
import android.widget.Toast;
```



We're using these extra classes, so you need to import them.

```
public class DrinkCategoryActivity extends Activity {
```

```
private SQLiteDatabase db;
private Cursor cursor;
```

We're adding these as private variables so we can close the database and cursor in our onDestroy() method.

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_drink_category);
    ArrayAdapter<Drink> listAdapter = new ArrayAdapter<>(
        this,
        android.R.layout.simple_list_item_1,
        Drink.drinks);
```

We're no longer using an array adapter, so delete these lines of code.

```
    ListView listDrinks = (ListView) findViewById(R.id.list_drinks);
    SQLiteOpenHelper starbuzzDatabaseHelper = new StarbuzzDatabaseHelper(this);
    try {
        db = starbuzzDatabaseHelper.getReadableDatabase();
        cursor = db.query("DRINK",
            new String[]{"_id", "NAME"},
            null, null, null, null, null);
```

Create the cursor.

Get a reference to the database.

The code continues on the next page.



The DrinkCategoryActivity code (continued)

```

SimpleCursorAdapter listAdapter = new SimpleCursorAdapter(this,
    android.R.layout.simple_list_item_1,
    cursor,
    new String[]{"NAME"},
    new int[]{android.R.id.text1},
    0);
listDrinks.setAdapter(listAdapter);
} catch(SQLiteException e) {
    Toast toast = Toast.makeText(this, "Database unavailable", Toast.LENGTH_SHORT);
    toast.show();
}

//Create a listener to listen for clicks in the list view
AdapterView.OnItemClickListener itemClickListener =
    new AdapterView.OnItemClickListener() {
        public void onItemClick(AdapterView<?> listDrinks,
            View itemView,
            int position,
            long id) {
            //Pass the drink the user clicks on to DrinkActivity
            Intent intent = new Intent(DrinkCategoryActivity.this,
                DrinkActivity.class);
            intent.putExtra(DrinkCategoryActivity.EXTRA_DRINKID, (int) id);
            startActivity(intent);
        }
    };

//Assign the listener to the list view
listDrinks.setOnItemClickListener(itemClickListener);
}

@Override
public void onDestroy() {
    super.onDestroy();
    cursor.close();
    db.close();
}
}

```

Map the contents of the NAME column to the text in the ListView.

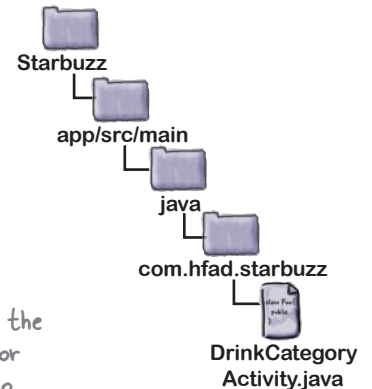
Create the cursor adapter.

Set the adapter to the ListView.

Display a message to the user if a SQLiteException gets thrown.

We didn't need to change any of the listener code.

We're closing the database and cursor in the activity's onDestroy() method. The cursor will stay open until the cursor adapter no longer needs it.



Let's try running our freshly updated app.



Test drive the app

When you run the app, `TopLevelActivity` gets displayed.

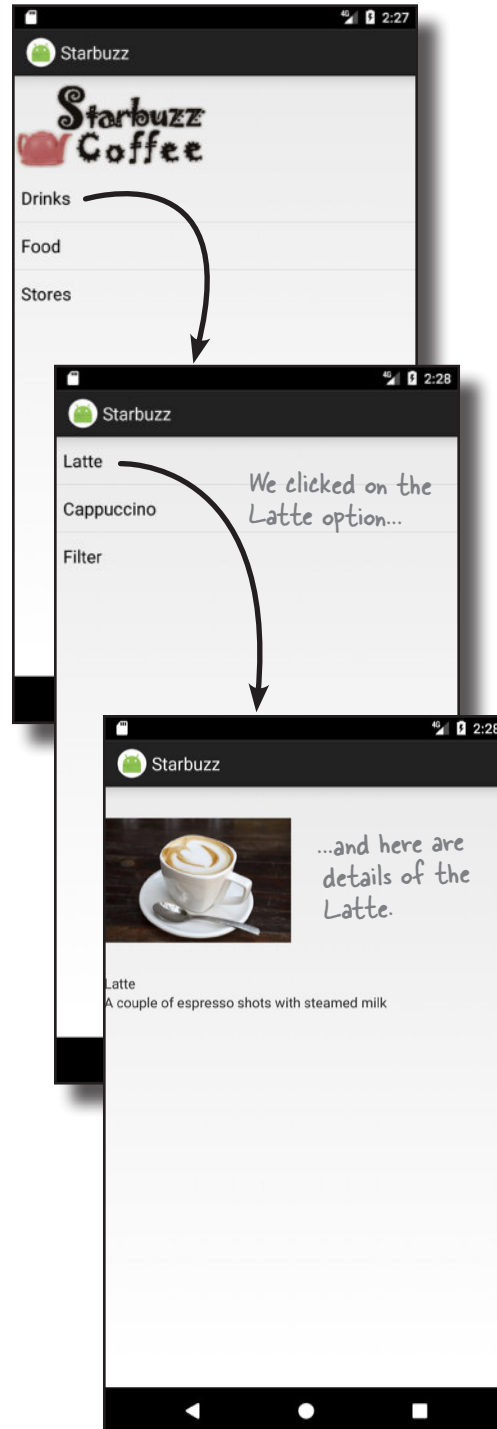
When you click on the Drinks item, `DrinkCategoryActivity` is launched. It displays all the drinks from the Starbuzz database.

When you click on one of the drinks, `DrinkActivity` is launched and details of the selected drink are displayed.

The app looks exactly the same as before, but now the data is being read from the database. In fact, you can now delete the `Drink.java` code, because we no longer need the array of drinks. Every piece of data we need is now coming from the database.



Create cursor
Cursor adapter





Your Android Toolbox

You've got Chapter 16 under your belt and now you've added basic cursors to your toolbox.

You can download the full code for the chapter from <https://tinyurl.com/HeadFirstAndroid>.



BULLET POINTS

- A **cursor** lets you read from and write to the database.
- You create a cursor by calling the `SQLiteDatabase query()` method. Behind the scenes, this builds a SQL `SELECT` statement.
- The `getWritableDatabase()` method returns a `SQLiteDatabase` object that allows you to read from and write to the database.
- The `getReadableDatabase()` returns a `SQLiteDatabase` object. This gives you read-only access to the database. It *may* also allow you to write to the database, but this isn't guaranteed.
- Navigate through a cursor using the `moveTo*()` methods.
- Get values from a cursor using the `get*()` methods. Close cursors and database connections after you've finished with them.
- A **cursor adapter** is an adapter that works with cursors. Use `SimpleCursorAdapter` to populate a list view with the values returned by a cursor.

17 cursors and async tasks

Staying in the Background



My doInBackground() method's awesome. If I left it all to Mr. Main Event, can you imagine how slow he'd be?

In most apps, you'll need your app to update its data.

So far you've seen how to create apps that read data from a SQLite database. But what if you want to update the app's data? In this chapter you'll see how to get your app to **respond to user input** and **update values in the database**. You'll also find out how to **refresh the data that's displayed** once it's been updated. Finally, you'll see how writing efficient **multithreaded code** with **AsyncTasks** will keep your app speedy.

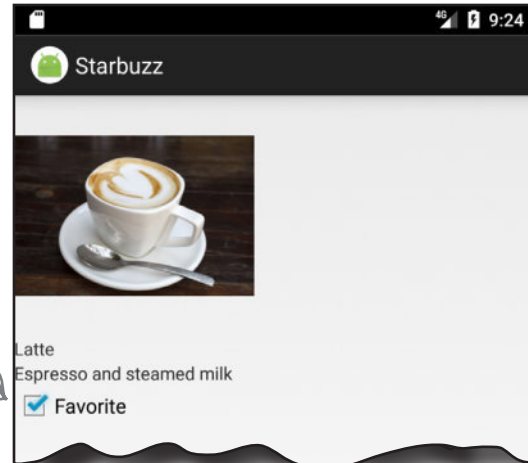
We want our Starbuzz app to update database data

In Chapter 16, you learned how to change your app to read its data from a SQLite database. You saw how to read an individual record (a drink from the Starbuzz data) and display that record's data in an activity. You also learned how to populate a list view with database data (in this case drink names) using a cursor adapter.

In both of these scenarios, you only needed to *read* data from the database. But what if you want users to be able to update the data?

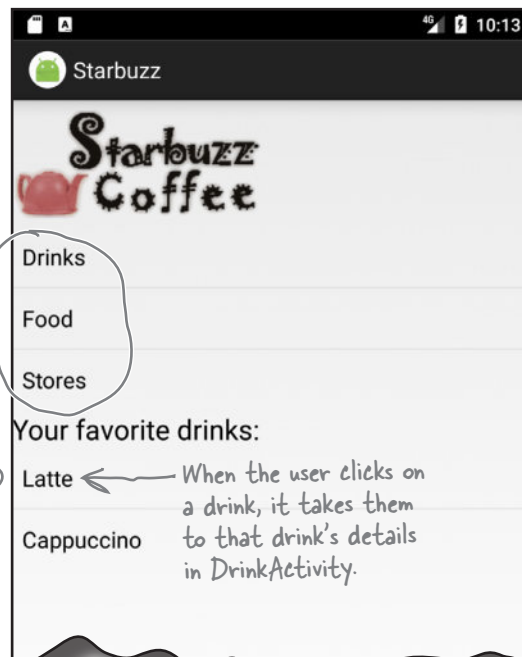
We're going to change the Starbuzz app so that users can record which drinks are their favorites. We'll do this by adding a checkbox to `DrinkActivity`; if it's checked, it means the current drink is one of the user's favorites:

Users can say which drinks are their favorites by checking a checkbox. We need to add this checkbox to `DrinkActivity` and get it to update the database.



We'll also add a new list view to `TopLevelActivity`, which contains the user's favorite drinks:

In the real world, you'd probably want to use tab navigation for these items. We're deliberately keeping the app pretty basic because we want you to focus on databases.



We'll add a `ListView` to `TopLevelActivity`, which contains the user's favorite drinks.

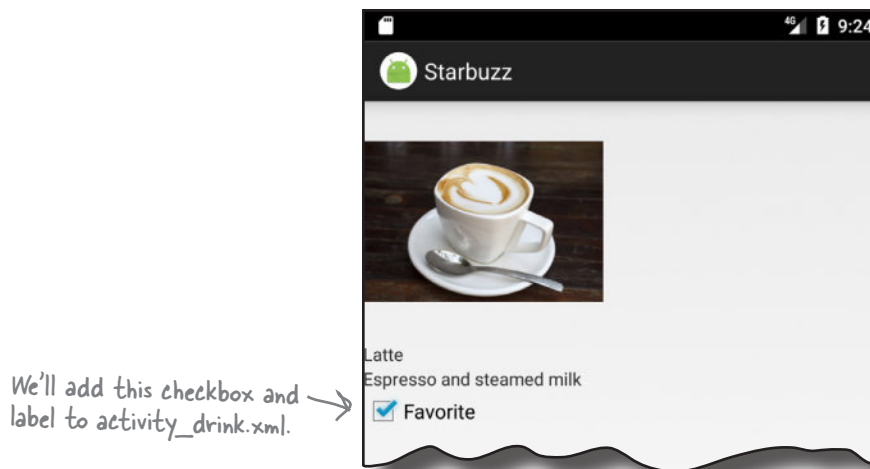
When the user clicks on a drink, it takes them to that drink's details in `DrinkActivity`.

We'll update DrinkActivity first

In Chapter 15, we added a FAVORITE column to the DRINK table in the Starbuzz database. We'll use this column to let users indicate whether a particular drink is one of their favorites. We'll display its value in the new checkbox we're going to add to DrinkActivity, and when the user clicks on the checkbox, we'll update the FAVORITE column with the new value.

Here are the steps we'll go through to update DrinkActivity:

- 1 **Update DrinkActivity's layout to add a checkbox and text label.**



- 2 **Display the value of the FAVORITE column in the checkbox.**
To do this, we'll need to retrieve the value of the FAVORITE column from the Starbuzz database.
- 3 **Update the FAVORITE column when the checkbox is clicked.**
We'll update the FAVORITE column with the value of the checkbox so that the data in the database stays up to date.

Let's get started.

Do this!

We're going to update the Starbuzz app in this chapter, so open your Starbuzz project in Android Studio.

Add a checkbox to DrinkActivity's layout

We'll start by adding a new checkbox to DrinkActivity's layout to indicate whether the current drink is one of the user's favorites. We're using a checkbox, as it's an easy way to display true/false values.

First, add a String resource called "favorite" to strings.xml (we'll use this as a label for the checkbox):

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

Then add the checkbox to activity_drink.xml. We'll give the checkbox an ID of favorite so that we can refer to it in our activity code. We'll also set its android:onClick attribute to "onFavoriteClicked" so that it calls the onFavoriteClicked() method in DrinkActivity when the user clicks on the checkbox. Here's the layout code; update your code to reflect our changes (they're in bold):

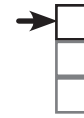
```
<LinearLayout ... >
    <ImageView
        android:id="@+id/photo"
        android:layout_width="190dp"
        android:layout_height="190dp" />
    <TextView
        android:id="@+id/name"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" />
    <TextView
        android:id="@+id/description"
        android:layout_width="match_parent"
        android:layout_height="wrap_content" />
```

These are the photo, name, and description views we added when we first created the activity.

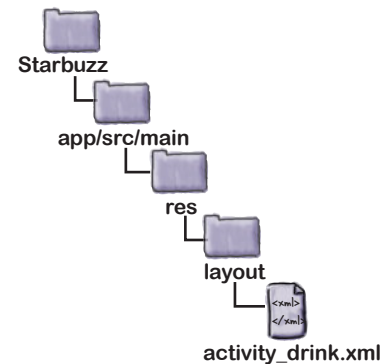
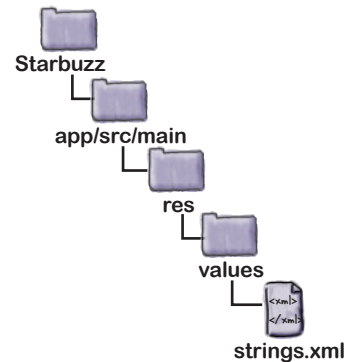
```
<CheckBox android:id="@+id/favorite"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/favorite"
    android:onClick="onFavoriteClicked" />
```

```
</LinearLayout>
```

Next we'll change the DrinkActivity code to get the checkbox to reflect the value of the FAVORITE column from the database.



Update layout
Show favorite
Update favorite



The checkbox has an ID of favorite.
We're giving the checkbox a label.
When the checkbox is clicked, the onFavoriteClicked() method in DrinkActivity will get called. We need to write this method.

Display the value of the FAVORITE column

In order to update the checkbox, we first need to retrieve the value of the FAVORITE column from the database. We can do this by updating the cursor we're using in `DrinkActivity`'s `onCreate()` method to read drink values from the database.

Here's the cursor we're currently using to return data for the drink the user has selected:

```
Cursor cursor = db.query("DRINK",
    new String[]{"NAME", "DESCRIPTION", "IMAGE_RESOURCE_ID"},
    "_id = ?",
    new String[]{Integer.toString(drinkId)},
    null, null, null);
```

To include the FAVORITE column in the data that's returned, we simply add it to the array of column names returned by the cursor:

```
Cursor cursor = db.query("DRINK",
    new String[]{"NAME", "DESCRIPTION", "IMAGE_RESOURCE_ID", "FAVORITE"},
    "_id = ?",
    new String[]{Integer.toString(drinkId)},
    null, null, null);
```

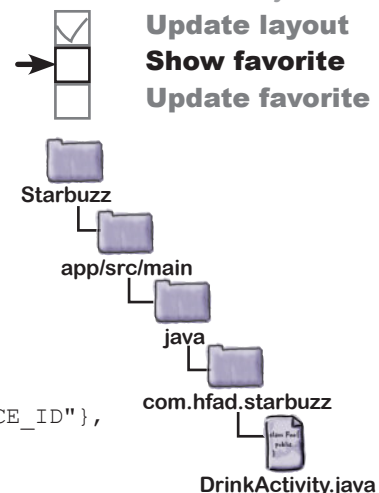
Once we have the value of the FAVORITE column, we can update the `favorite` checkbox accordingly. To get this value, we first navigate to the first (and only) record in the cursor using:

```
cursor.moveToFirst();
```

We can then get the value of the column for the current drink. The FAVORITE column contains numeric values, where 0 is false and 1 is true. We want the `favorite` checkbox to be ticked if the value is 1 (true), and unticked if the value is 0 (false), so we'll use the following code to update the checkbox:

```
boolean isFavorite = (cursor.getInt(3) == 1);
CheckBox favorite = (CheckBox) findViewById(R.id.favorite);
favorite.setChecked(isFavorite);
```

That's all the code we need to reflect the value of the FAVORITE column in the `favorite` checkbox. Next, we need to get the checkbox to respond to clicks so that it updates the database when its value changes.



Add the FAVORITE column to the cursor.

You don't need to update your version of the code yet. We'll show you the full set of changes for `DrinkActivity.java` soon.

Get the value of the FAVORITE column. It's stored in the database as 1 for true, 0 for false.

Set the value of the favorite checkbox.

Respond to clicks to update the database



Update layout
Show favorite
Update favorite

When we added the favorite checkbox to *activity_drink.xml*, we set its `android:onClick` attribute to `onFavoriteClicked()`. This means that whenever the user clicks on the checkbox, the `onFavoriteClicked()` method in the activity will get called. We'll use this method to update the database with the current value of the checkbox. If the user checks or unchecks the checkbox, the `onFavoriteClicked()` method will save the user's change to the database.

In Chapter 15, you saw how to use `SQLiteDatabase` methods to change the data held in a `SQLite` database: the `insert()` method to insert data, the `delete()` method to delete data, and the `update()` method to update existing records.

You can use these methods to change data from within your activity. As an example, you could use the `insert()` method to add new drink records to the `DRINK` table, or the `delete()` method to delete them. In our case, we want to update the `DRINK` table's `FAVORITE` column with the value of the checkbox using the `update()` method.

As a reminder, the `update()` method looks like this:

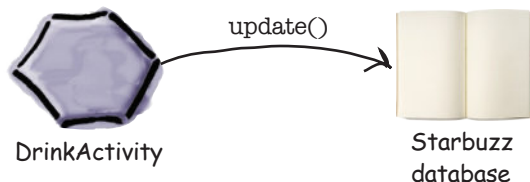
```
db.update(String table,
          ContentValues values,
          String conditionClause,
          String[] conditionArguments);
```

Handwritten annotations with arrows pointing to the code:

- ← The table whose data you want to update (points to `String table`)
- ← The new values (points to `ContentValues values`)
- ← The criteria for updating the data (points to `String conditionClause` and `String[] conditionArguments`)

where `table` is the name of the table you want to update, and `values` is a `ContentValues` object containing name/value pairs of the columns you want to update and the values you want to set them to. The `conditionClause` and `conditionArguments` parameters lets you specify which records you want to update.

You already know everything you need to get `DrinkActivity` to update the `FAVORITE` column for the current drink when the checkbox is clicked, so have a go at the following exercise.





Code Magnets

In our code for `DrinkActivity` we want to update the `FAVORITE` column in the database with the value of the `favorite` checkbox. Can you construct the `onFavoriteClicked()` method so that it will do that?

```
public class DrinkActivity extends Activity {
...
    //Update the database when the checkbox is clicked

    public void onFavoriteClicked(.....){

        int drinkId = (Integer) getIntent().getExtras().get(EXTRA_DRINKID);
        CheckBox favorite = (CheckBox) findViewById(R.id.favorite);

        ..... drinkValues = new .....;

        drinkValues.put(....., favorite.isChecked());

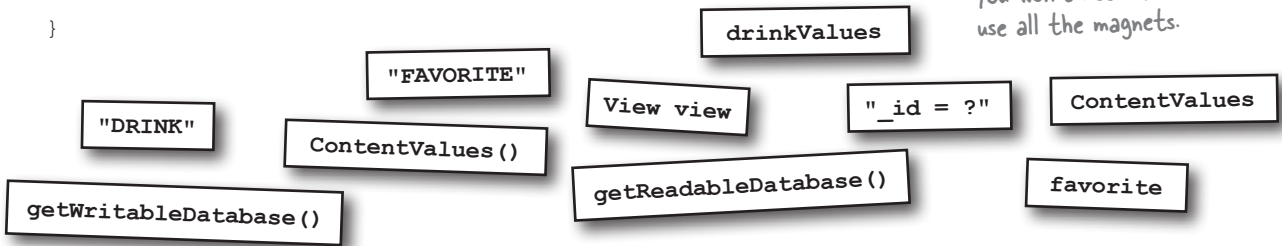
        SQLiteOpenHelper starbuzzDatabaseHelper = new StarbuzzDatabaseHelper(this);
        try {

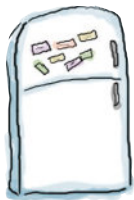
            SQLiteDatabase db = starbuzzDatabaseHelper.....;

            db.update(.....,.....,
                ....., new String[] {Integer.toString(drinkId)});

            db.close();
        } catch(SQLiteException e) {
            Toast toast = Toast.makeText(this, "Database unavailable", Toast.LENGTH_SHORT);
            toast.show();
        }
    }
}
```

You won't need to use all the magnets.





Code Magnets Solution

In our code for `DrinkActivity` we want to update the `FAVORITE` column in the database with the value of the `favorite` checkbox. Can you construct the `onFavoriteClicked()` method so that it will do that?

```
public class DrinkActivity extends Activity {
...
    //Update the database when the checkbox is clicked

    public void onFavoriteClicked(View view) {

        int drinkId = (Integer) getIntent().getExtras().get(EXTRA_DRINKID);
        CheckBox favorite = (CheckBox) findViewById(R.id.favorite);

        ContentValues drinkValues = new ContentValues();

        drinkValues.put("FAVORITE", favorite.isChecked());

        SQLiteOpenHelper starbuzzDatabaseHelper = new StarbuzzDatabaseHelper(this);
        try {

            SQLiteDatabase db = starbuzzDatabaseHelper.getWritableDatabase();

            db.update("DRINK", drinkValues,
                    "_id = ?", new String[] {Integer.toString(drinkId)});

            db.close();
        } catch (SQLException e) {
            Toast toast = Toast.makeText(this, "Database unavailable", Toast.LENGTH_SHORT);
            toast.show();
        }
    }
}
```

We need read/write access to the database to update it.

You didn't need to use these magnets.

getReadableDatabase()

favorite



The full DrinkActivity.java code

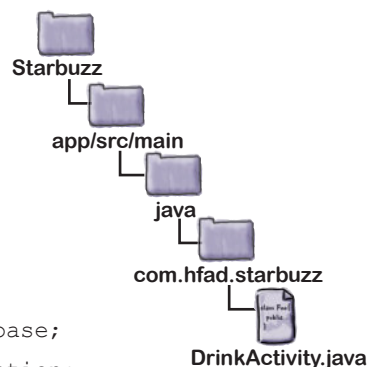
We've now done everything we need to change `DrinkActivity` so that it reflects the contents of the FAVORITE column in the favorite checkbox. It then updates the value of the column in the database if the user changes the value of the checkbox.

Here's the full code for `DrinkActivity.java`, so update your version of the code so that it reflects ours (our changes are in bold):

```
package com.hfad.starbuzz;

import android.app.Activity;
import android.os.Bundle;
import android.widget.ImageView;
import android.widget.TextView;
import android.widget.Toast;
import android.database.Cursor;
import android.database.sqlite.SQLiteDatabase;
import android.database.sqlite.SQLiteException;
import android.database.sqlite.SQLiteOpenHelper;
import android.view.View;
import android.widget.CheckBox;
import android.content.ContentValues;

public class DrinkActivity extends Activity {
```



We're using these extra classes, so you need to import them.

```
    public static final String EXTRA_DRINKID = "drinkId";

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

        //Get the drink from the intent
        int drinkId = (Integer) getIntent().getExtras().get(EXTRA_DRINKID);
```

The code continues
on the next page. →

DrinkActivity.java (continued)



Update layout
Show favorite
Update favorite

```
//Create a cursor
SQLiteOpenHelper starbuzzDatabaseHelper = new StarbuzzDatabaseHelper(this);
try {
    SQLiteDatabase db = starbuzzDatabaseHelper.getReadableDatabase();
    Cursor cursor = db.query("DRINK",
        new String[]{"NAME", "DESCRIPTION", "IMAGE_RESOURCE_ID", "FAVORITE"},
        "_id = ?",
        new String[]{Integer.toString(drinkId)},
        null, null, null);

    //Move to the first record in the Cursor
    if (cursor.moveToFirst()) {
        //Get the drink details from the cursor
        String nameText = cursor.getString(0);
        String descriptionText = cursor.getString(1);
        int photoId = cursor.getInt(2);
        boolean isFavorite = (cursor.getInt(3) == 1);

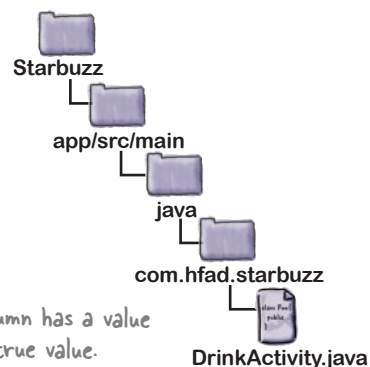
        //Populate the drink name
        TextView name = (TextView) findViewById(R.id.name);
        name.setText(nameText);

        //Populate the drink description
        TextView description = (TextView) findViewById(R.id.description);
        description.setText(descriptionText);

        //Populate the drink image
        ImageView photo = (ImageView) findViewById(R.id.photo);
        photo.setImageResource(photoId);
        photo.setContentDescription(nameText);

        //Populate the favorite checkbox
        CheckBox favorite = (CheckBox) findViewById(R.id.favorite);
        favorite.setChecked(isFavorite);
    }
}
```

↑
Add the FAVORITE
column to the cursor.



↑
If the FAVORITE column has a value
of 1, this indicates a true value.

↙
If the drink is a favorite, put a
checkmark in the favorite checkbox.

The code continues →
on the next page.

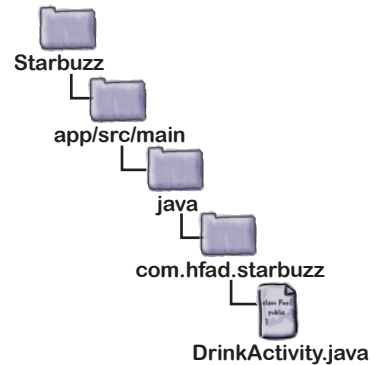


DrinkActivity.java (continued)

```

        cursor.close();
        db.close();
    } catch (SQLiteException e) {
        Toast toast = Toast.makeText(this,
            "Database unavailable",
            Toast.LENGTH_SHORT);
        toast.show();
    }
}

```



//Update the database when the checkbox is clicked

```

public void onFavoriteClicked(View view) {
    int drinkId = (Integer) getIntent().getExtras().get(EXTRA_DRINKID);

    //Get the value of the checkbox
    CheckBox favorite = (CheckBox) findViewById(R.id.favorite);
    ContentValues drinkValues = new ContentValues();
    drinkValues.put("FAVORITE", favorite.isChecked());
    //Get a reference to the database and update the FAVORITE column
    SQLiteOpenHelper starbuzzDatabaseHelper = new StarbuzzDatabaseHelper(this);
    try {
        SQLiteDatabase db = starbuzzDatabaseHelper.getWritableDatabase();
        db.update("DRINK",
            drinkValues,
            "_id = ?",
            new String[] {Integer.toString(drinkId)});
        db.close();
    } catch (SQLiteException e) {
        Toast toast = Toast.makeText(this, "Database unavailable", Toast.LENGTH_SHORT);
        toast.show();
    }
}
}

```

Add the value of the favorite checkbox to the drinkValues ContentValues object.

Update the drink's FAVORITE column in the database to the value of the checkbox.

Display a message if there's a problem with the database.

Let's check what happens when we run the app.

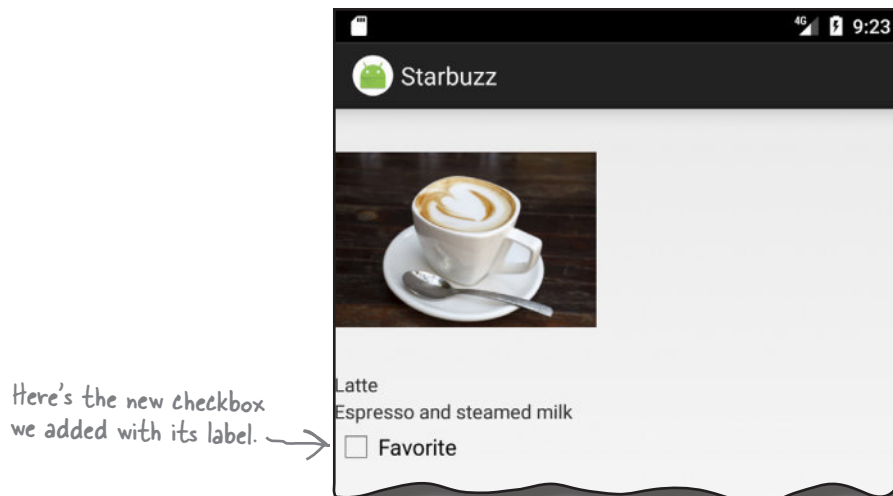


Test drive the app

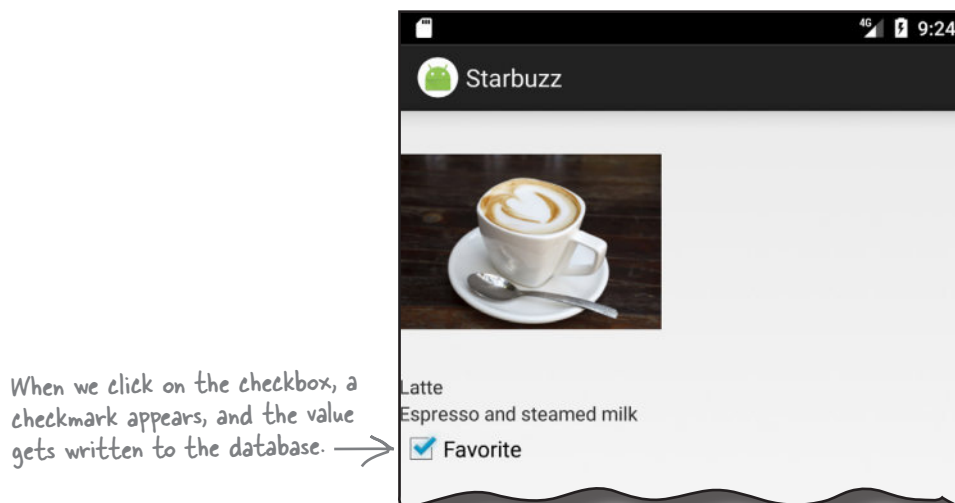


Update layout
Show favorite
Update favorite

When we run the app and navigate to a drink, the new favorite checkbox is displayed (unchecked):



When we click on the checkbox, a checkmark appears to indicate that the drink is one of our favorites:



When we close the app and navigate back to the drink, the checkmark remains. The value of the checkbox has been written to the database.

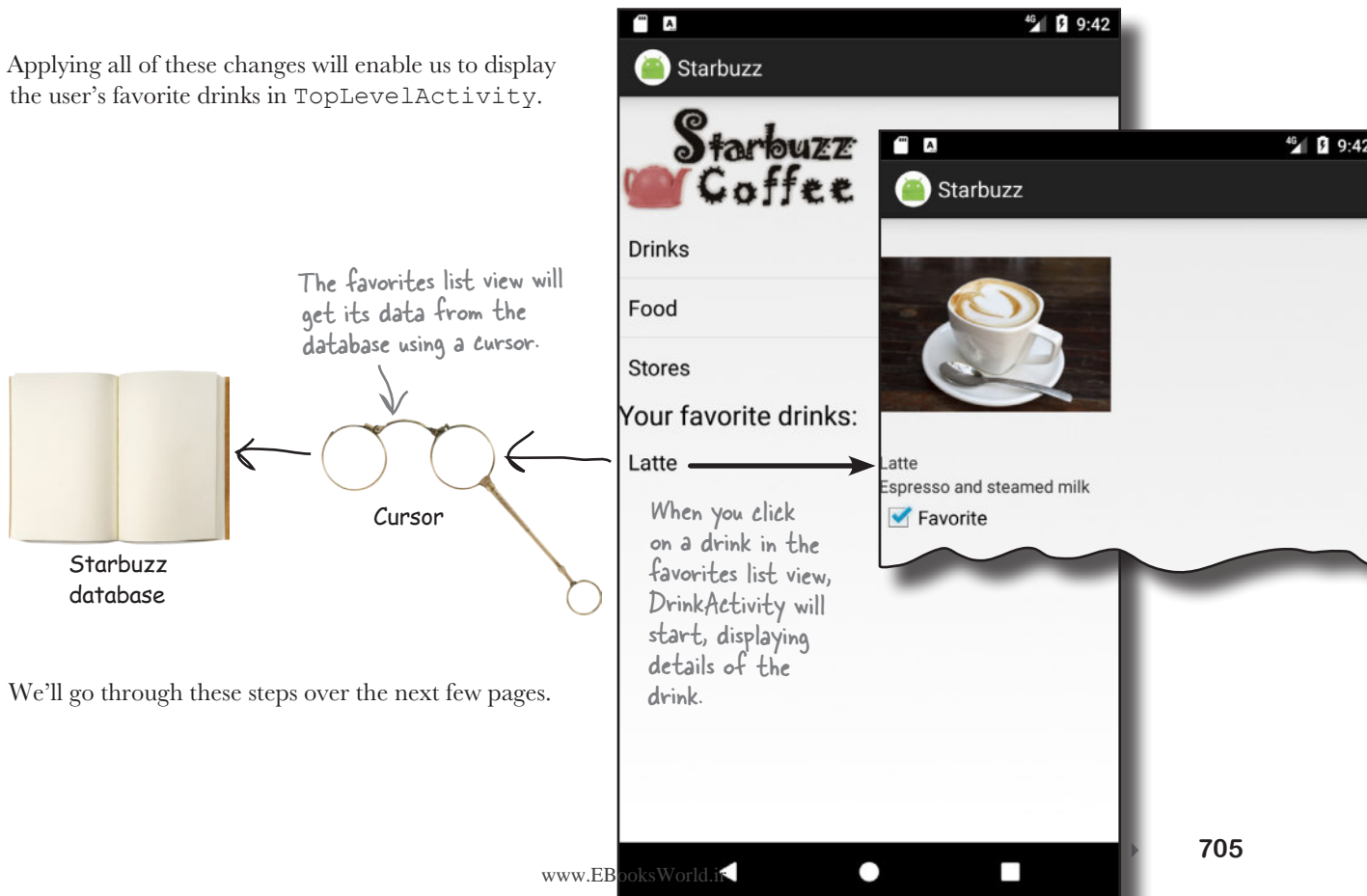
That's everything we need to display the value of the FAVORITE column from the database, and update the database with any changes to it.

Display favorites in TopLevelActivity

The next thing we need to do is display the user's favorite drinks in `TopLevelActivity`. Here are the steps we'll go through to do this:

- 1 **Add a list view and text view to `TopLevelActivity`'s layout.**
- 2 **Populate the list view and get it to respond to clicks.**
We'll create a new cursor that retrieves the user's favorite drinks from the database, and attach it to the list view using a cursor adapter. We'll then create an `onItemClickListener` so that we can get `TopLevelActivity` to start `DrinkActivity` when the user clicks on one of the drinks.
- 3 **Refresh the list view data when we choose a new favorite drink.**
If we choose a new favorite drink in `DrinkActivity`, we want it to be displayed in `TopLevelActivity`'s list view when we navigate back to it.

Applying all of these changes will enable us to display the user's favorite drinks in `TopLevelActivity`.



Display the favorite drinks in activity_top_level.xml

As we said on the previous page, we're going to add a list view to *activity_top_level.xml*, which we'll use to display a list of the user's favorite drinks. We'll also add a text view to display a heading for the list.

First, add the following String resource to *strings.xml* (we'll use this for the text view's text):

```
<string name="favorites">Your favorite drinks:</string>
```

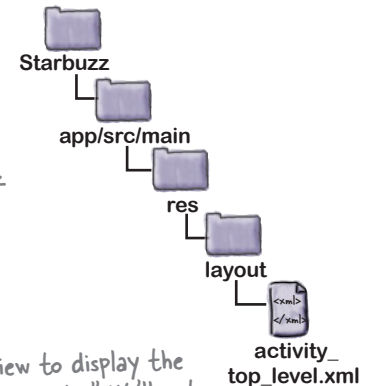
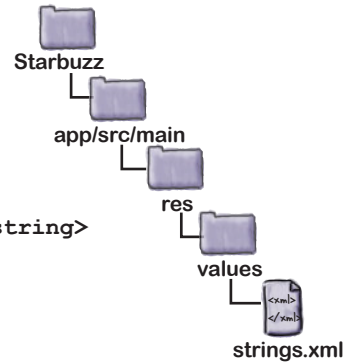
Next, we'll add the new text view and list view to the layout. Here's our code for *activity_top_level.xml*; update your version of the code to match our changes:

```
<LinearLayout ... >
  <ImageView
    android:layout_width="200dp"
    android:layout_height="100dp"
    android:src="@drawable/starbuzz_logo"
    android:contentDescription="@string/starbuzz_logo" />

  <ListView
    android:id="@+id/list_options"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:entries="@array/options" />

  <TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:textAppearance="?android:attr/textAppearanceLarge"
    android:text="@string/favorites" />

  <ListView
    android:id="@+id/list_favorites"
    android:layout_width="match_parent"
    android:layout_height="wrap_content" />
</LinearLayout>
```

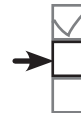


The layout already contains the Starbuzz logo and list view.

We'll add a text view to display the text "Your favorite drinks." We'll put this in a String called favorites.

The list_favorites ListView will display the user's favorite drinks.

Those are all the changes we need to make to *activity_top_level.xml*. Next, we'll update *TopLevelActivity.java*.



Refactor TopLevelActivity.java

Before we write any code for our new list view, we're going to refactor our existing `TopLevelActivity` code. This will make the code a lot easier to read later on. We'll move the code relating to the options list view into a new method called `setupOptionsListView()`. We'll then call this method from the `onCreate()` method.

Here's our code for `TopLevelActivity.java` (update your version of the code to reflect our changes).

```
package com.hfad.starbuzz;
...
public class TopLevelActivity extends Activity {

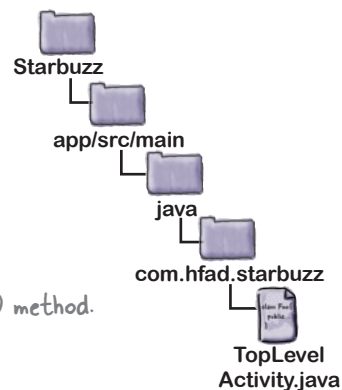
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_top_level);
        setupOptionsListView(); ← Call the new setupOptionsListView() method.
    }

    private void setupOptionsListView() {
        //Create an OnItemClickListener
        AdapterView.OnItemClickListener itemClickListener =
            new AdapterView.OnItemClickListener() {
                public void onItemClick(AdapterView<?> listView,
                    View itemView,
                    int position,
                    long id) {

                    if (position == 0) {
                        Intent intent = new Intent(TopLevelActivity.this,
                            DrinkCategoryActivity.class);
                        startActivity(intent);
                    }
                }
            };
        //Add the listener to the list view
        ListView listView = (ListView) findViewById(R.id.list_options);
        listView.setOnItemClickListener(itemClickListener);
    }
}
}
```

All of this code was in the `onCreate()` method. We're putting it in a new method to make the code tidier.

↑ If the Drink option in the `list_options` list view is clicked, start `DrinkCategoryActivity`.



What changes are needed for TopLevelActivity.java

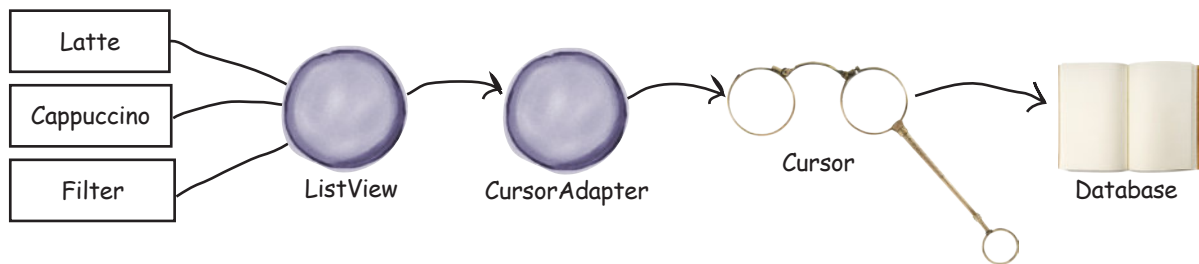


Update layout
Populate list view
Refresh data

We need to display the user’s favorite drinks in the `list_favorites` list view we added to the layout, and get it to respond to clicks. To do this, we need to do the following:

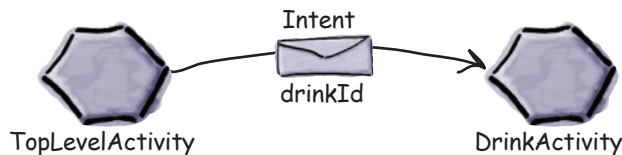
1 Populate the list_favorites list view using a cursor.

The cursor will return all drinks where the FAVORITE column has been set to 1—all drinks that the user has flagged as being a favorite. Just as we did in our code for `DrinkCategoryActivity`, we can connect the cursor to the list view using a cursor adapter.



2 Create an onItemClickListener so that the list_favorites list view can respond to clicks.

If the user clicks on one of their favorite drinks, we can create an intent that starts `DrinkActivity`, passing it the ID of the drink that was clicked. This will show the user details of the drink they’ve just chosen.



You’ve already seen all the code that’s needed to do this. In fact, it’s almost identical to the code we wrote in earlier chapters to control the list of drinks in `DrinkCategoryActivity`. The only difference is that this time, we only want to display drinks with a value of 1 in the FAVORITE column.

We’ve decided to put the code that controls a list view in a new method called `setupFavoritesListView()`. We’ll show you this method on the next page before adding it to `TopLevelActivity.java`.



Ready Bake Code

The `setupFavoritesListView()` method populates the `list_favorites` list view with the names of the user's favorite drinks. Make sure you understand the code below before turning the page.



cursors and asyncTasks

Update layout
Populate list view
Refresh data

```
private void setupFavoritesListView() {
    //Populate the list_favorites ListView from a cursor
    ListView listFavorites = (ListView) findViewById(R.id.list_favorites);
    try{
        SQLiteOpenHelper starbuzzDatabaseHelper = new StarbuzzDatabaseHelper(this);
        db = starbuzzDatabaseHelper.getReadableDatabase();
        favoritesCursor = db.query("DRINK",
            new String[] { "_id", "NAME"},
            "FAVORITE = 1",
            null, null, null, null);
        CursorAdapter favoriteAdapter =
            new SimpleCursorAdapter(TopLevelActivity.this,
                android.R.layout.simple_list_item_1,
                favoritesCursor,
                new String[]{"NAME"},
                new int[]{android.R.id.text1}, 0);
        listFavorites.setAdapter(favoriteAdapter);
    } catch(SQLiteException e) {
        Toast toast = Toast.makeText(this, "Database unavailable", Toast.LENGTH_SHORT);
        toast.show();
    }

    //Navigate to DrinkActivity if a drink is clicked
    listFavorites.setOnItemClickListener(new AdapterView.OnItemClickListener() {
        @Override
        public void onItemClick(AdapterView<?> listView, View v, int position, long id) {
            Intent intent = new Intent(TopLevelActivity.this, DrinkActivity.class);
            intent.putExtra(DrinkActivity.EXTRA_DRINKID, (int)id);
            startActivity(intent);
        }
    });
}
```

Get the `list_favorites` list view.

Create a cursor that gets the values of the `_id` and `NAME` columns where `FAVORITE=1`.

Get the names of the user's favorite drinks.

Create a new cursor adapter.

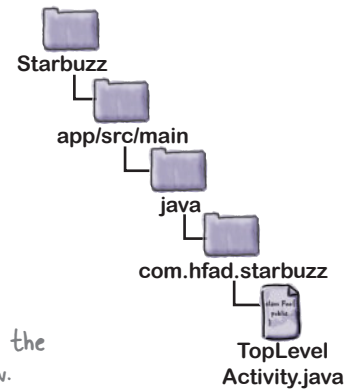
Use the cursor in the cursor adapter.

Display the names of the drinks in the list view.

Display a message if there's a problem with the database.

This will get called if an item in the list view is clicked.

If the user clicks on one of the items in the `list_favorites` list view, create an intent to start `DrinkActivity`, including the ID of the drink as extra information.



The new TopLevelActivity.java code

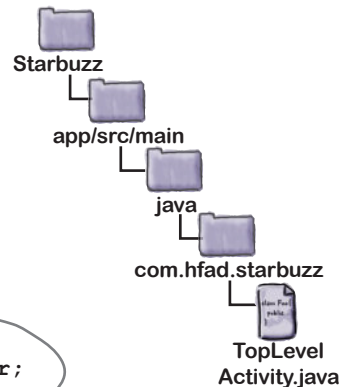


Update layout
Populate list view
Refresh data

We've updated `TopLevelActivity` to populate the `list_favorites` list view and make it respond to clicks. Update your version of `TopLevelActivity.java` to match ours (there's a lot of new code, so go through it carefully and take your time):

```
package com.hfad.starbuzz;

import android.app.Activity;
import android.os.Bundle;
import android.content.Intent;
import android.widget.AdapterView;
import android.widget.AdapterView;
import android.widget.ListView;
import android.view.View;
import android.database.Cursor;
import android.database.sqlite.SQLiteOpenHelper;
import android.database.sqlite.SQLiteException;
import android.database.sqlite.SQLiteDatabase;
import android.widget.SimpleCursorAdapter;
import android.widget.CursorAdapter;
import android.widget.Toast;
```



We're using all these extra classes, so we need to import them.

```
public class TopLevelActivity extends Activity {
```

```
    private SQLiteDatabase db;
    private Cursor favoritesCursor;
```

We're adding the database and cursor as private variables so that we can access them in the `setUpFavoritesListView()` and `onDestroy()` methods.

```
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_top_level);
        setupOptionsListView();
        setupFavoritesListView();
    }
```

Call the `setupFavoritesListView()` method from the `onCreate()` method.

The code continues on the next page.

The TopLevelActivity.java code (continued)



cursors and asyncTasks

Update layout
Populate list view
Refresh data

↙ We don't need to change this method.

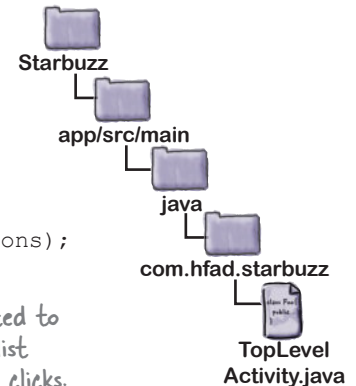
```
private void setupOptionsListView() {  
    //Create an OnItemClickListener  
    AdapterView.OnItemClickListener itemClickListener =  
        new AdapterView.OnItemClickListener(){  
        public void onItemClick(AdapterView<?> listView,  
            View itemView,  
            int position,  
            long id) {  
            if (position == 0) {  
                Intent intent = new Intent(TopLevelActivity.this,  
                    DrinkCategoryActivity.class);  
                startActivity(intent);  
            }  
        }  
    };  
};
```

```
//Add the listener to the list view  
ListView listView = (ListView) findViewById(R.id.list_options);  
listView.setOnItemClickListener(itemClickListener);  
}
```

↙ This is the method we created to populate the list_favorites list view and make it respond to clicks.

```
private void setupFavoritesListView() {  
    //Populate the list_favorites ListView from a cursor  
    ListView listFavorites = (ListView) findViewById(R.id.list_favorites);  
    try{  
        SQLiteOpenHelper starbuzzDatabaseHelper = new StarbuzzDatabaseHelper(this);  
        db = starbuzzDatabaseHelper.getReadableDatabase();  
        favoritesCursor = db.query("DRINK",  
            new String[] { "_id", "NAME",  
                "FAVORITE = 1",  
                null, null, null, null});  
    }  
};
```

↖ The list_favorites list view will use this cursor for its data.



The code continues →
on the next page.

The TopLevelActivity.java code (continued)



Update layout
Populate list view
Refresh data

```

CursorAdapter favoriteAdapter =
    new SimpleCursorAdapter(TopLevelActivity.this,
        android.R.layout.simple_list_item_1,
        favoritesCursor,
        new String[]{"NAME"},
        new int[]{android.R.id.text1}, 0);

listFavorites.setAdapter(favoriteAdapter);
} catch (SQLException e) {
    Toast toast = Toast.makeText(this, "Database unavailable", Toast.LENGTH_SHORT);
    toast.show();
}

//Navigate to DrinkActivity if a drink is clicked
listFavorites.setOnItemClickListener(new AdapterView.OnItemClickListener() {
    @Override
    public void onItemClick(AdapterView<?> listView, View v, int position, long id) {
        Intent intent = new Intent(TopLevelActivity.this, DrinkActivity.class);
        intent.putExtra(DrinkActivity.EXTRA_DRINKID, (int)id);
        startActivity(intent);
    }
});

//Close the cursor and database in the onDestroy() method
@Override
public void onDestroy() {
    super.onDestroy();
    favoritesCursor.close();
    db.close();
}
}

```

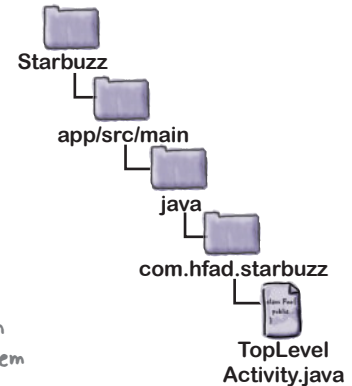
Use the cursor in a cursor adapter.

Set the cursor adapter to the list view.

Get the list_favorites list view to respond to clicks.

Start DrinkActivity, passing it the ID of the drink that was clicked on.

The onDestroy() method gets called just before the activity is destroyed. We'll close the cursor and database in this method, as we no longer need them if the activity's being destroyed.



The above code populates the `list_favorites` list view with the user's favorite drinks. When the user clicks on one of these drinks, an intent starts `DrinkActivity` and passes it the ID of the drink. Let's take the app for a test drive and see what happens.



Test drive the app

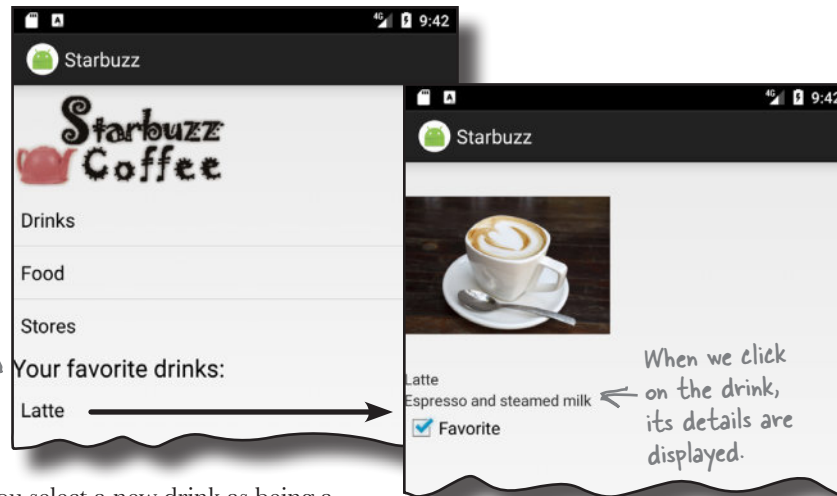
cursors and asyncTasks

Update layout
Populate list view
Refresh data

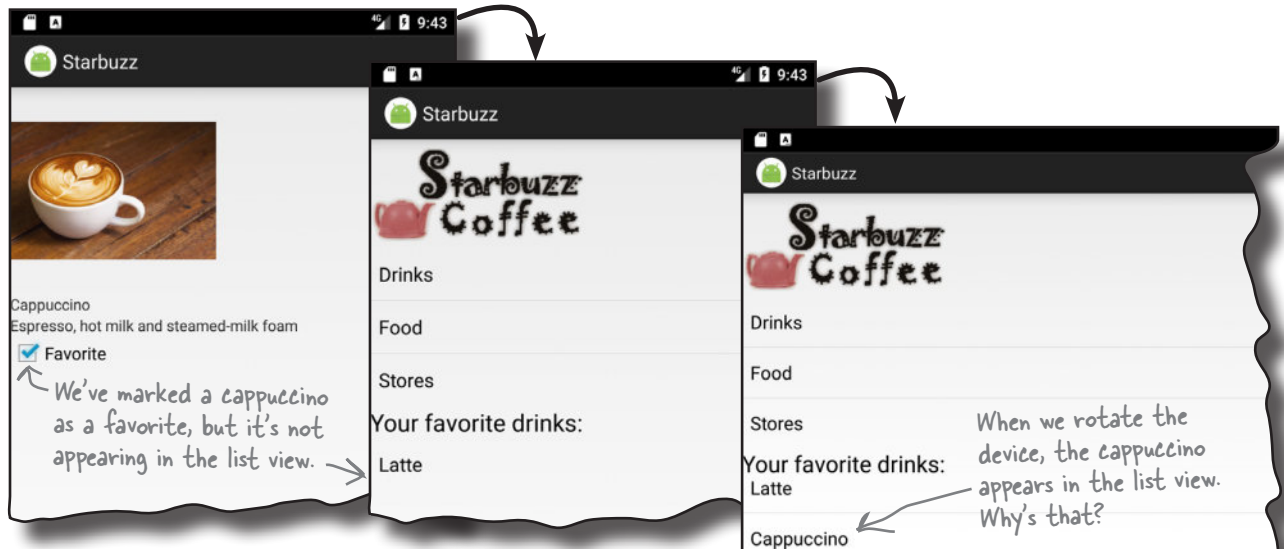


When you run the app, the new text view and `list_favorites` list view are displayed in `TopLevelActivity`. If you've marked a drink as being a favorite, it appears in the list view.

If you click on that drink, `DrinkActivity` starts and details of the drink are displayed.



But there's a problem. If you select a new drink as being a favorite, when you go back to `TopLevelActivity` the `list_favorites` list view doesn't include the new drink. The new drink is only included in the list view if you rotate the device.



Why do you think the new drink we chose as a favorite doesn't appear in the list view until we rotate the device? Give this some thought before turning the page.

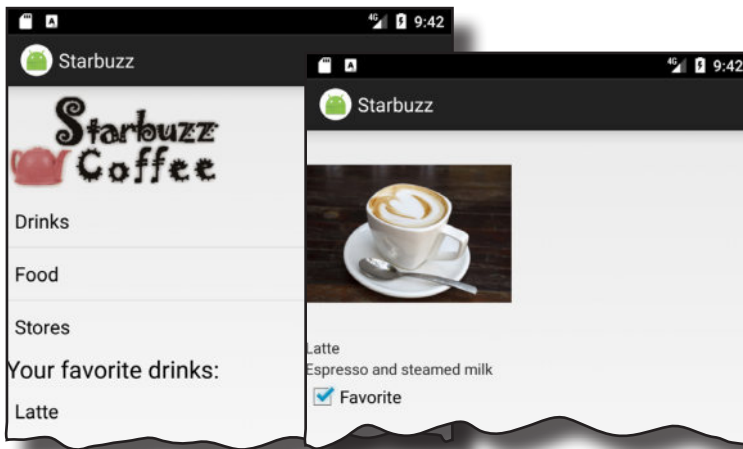
Cursors don't automatically refresh



Update layout
Populate list view
Refresh data

If the user chooses a new favorite drink by navigating through the app to `DrinkActivity`, the new favorite drink isn't automatically displayed in the `list_favorites` list view in `TopLevelActivity`. This is because **cursors retrieve data when the cursor gets created**.

In our case, the cursor is created in the activity `onCreate()` method, so it gets its data when the activity is created. When the user navigates through the other activities, `TopLevelActivity` is stopped. It's not destroyed and recreated, so neither is the cursor.



When you start a second activity, the second activity is stacked on top of the first. The first activity isn't destroyed. Instead, it's paused and then stopped, as it loses the focus and stops being visible to the user.

Cursors don't automatically keep track of whether the underlying data in the database has changed. So if the underlying data changes after the cursor's been created, the cursor doesn't get updated: it still contains the original records, and none of the changes. That means that if the user marks a new drink as being a favorite after the cursor is created, the cursor will be out of date.

If you update the data in the database...
 ...the cursor won't see the new data if the cursor's already been created.

<u>id</u>	NAME	DESCRIPTION	IMAGE_RESOURCE_ID	FAVORITE		
1	"Latte"	"Espresso and steamed milk"	54543543	1		
3	"Filter"	1	"Latte"	"Espresso and steamed milk"	54543543	0
		2	"Cappuccino"	"Espresso, hot milk and steamed-milk foam"	654334453	0
		3	"Filter"	"Our best drip coffee"	44324234	0

So how do we get around this?



Change the cursor with `changeCursor()`

The solution is to change the underlying cursor used by the `list_favorites` list view to an updated version. To do this, you define a new version of the cursor, get a reference to the list view's cursor adapter, and then call the cursor adapter's `changeCursor()` method to change the cursor. Here are the details:

1. Define the cursor

You define the cursor in exactly the same way as you did before. In our case, we want the query to return the user's favorite drinks, so we use:

```
Cursor newCursor = db.query("DRINK",
    new String[] { "_id", "NAME"},
    "FAVORITE = 1",
    null, null, null, null);
```

This is the same query that we had before.

2. Get a reference to the cursor adapter

You get a reference to the list view's cursor adapter by calling the list view's `getAdapter()` method. This method returns an object of type `Adapter`. As our list view is using a cursor adapter, we can cast the adapter to a `CursorAdapter`:

```
ListView listFavorites = (ListView) findViewById(R.id.list_favorites);
CursorAdapter adapter = (CursorAdapter) listFavorites.getAdapter();
```

3. Change the cursor using `changeCursor()`

You change the cursor used by the cursor adapter by calling its `changeCursor()` method. This method takes one parameter, the new cursor:

```
adapter.changeCursor(newCursor); ← Change the cursor used by the cursor adapter to the new one.
```

You get the ListView's adapter using the `getAdapter()` method.

The `changeCursor()` method replaces the cursor adapter's current cursor with the new one. It then closes the old cursor, so you don't need to do this yourself.

We're going to change the cursor used by the `list_favorites` list view in `TopLevelActivity`'s `onRestart()` method. This means that the data in the list view will get refreshed when the user returns to `TopLevelActivity`. Any new favorite drinks the user has chosen will be displayed, and any drinks that are no longer flagged as favorites will be removed from the list.

We'll show you the full code for `TopLevelActivity.java` over the next few pages.



The TopLevelActivity.java code (continued)

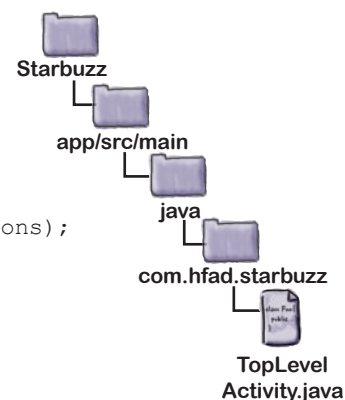
```
private void setupOptionsListView() {
    //Create an OnItemClickListener
    AdapterView.OnItemClickListener itemClickListener =
        new AdapterView.OnItemClickListener() {
        public void onItemClick(AdapterView<?> listView,
            View itemView,
            int position,
            long id) {

            if (position == 0) {
                Intent intent = new Intent(TopLevelActivity.this,
                    DrinkCategoryActivity.class);

                startActivity(intent);
            }
        }
    };

    //Add the listener to the list view
    ListView listView = (ListView) findViewById(R.id.list_options);
    listView.setOnItemClickListener(itemClickListener);
}
```

You don't need to
change any of the
code on this page.



```
private void setupFavoritesListView() {
    //Populate the list_favorites ListView from a cursor
    ListView listFavorites = (ListView) findViewById(R.id.list_favorites);
    try{
        SQLiteOpenHelper starbuzzDatabaseHelper = new StarbuzzDatabaseHelper(this);
        db = starbuzzDatabaseHelper.getReadableDatabase();
        favoritesCursor = db.query("DRINK",
            new String[] { "_id", "NAME"},
            "FAVORITE = 1",
            null, null, null, null);

        CursorAdapter favoriteAdapter =
            new SimpleCursorAdapter(TopLevelActivity.this,
                android.R.layout.simple_list_item_1,
                favoritesCursor,
                new String[]{"NAME"},
                new int[]{android.R.id.text1}, 0);
        listFavorites.setAdapter(favoriteAdapter);
    }
```

The code continues →
on the next page.

The TopLevelActivity.java code (continued)



Update layout
Populate list view
Refresh data

```

} catch(SQLiteException e) {
    Toast toast = Toast.makeText(this, "Database unavailable", Toast.LENGTH_SHORT);
    toast.show();
}

//Navigate to DrinkActivity if a drink is clicked
listFavorites.setOnItemClickListener(new AdapterView.OnItemClickListener() {
    @Override
    public void onItemClick(AdapterView<?> listView, View v, int position, long id) {
        Intent intent = new Intent(TopLevelActivity.this, DrinkActivity.class);
        intent.putExtra(DrinkActivity.EXTRA_DRINKID, (int)id);
        startActivity(intent);
    }
});
}

```

@Override

public void onRestart() {

super.onRestart();

Cursor newCursor = db.query("DRINK",

new String[] { "_id", "NAME"},

"FAVORITE = 1",

null, null, null, null);

ListView listFavorites = (ListView) findViewById(R.id.list_favorites);

CursorAdapter adapter = (CursorAdapter) listFavorites.getAdapter();

adapter.setCursor(newCursor); ← Switch the cursor being used by the list_favorites list view to the new cursor.

favoritesCursor = newCursor;

}

 Change the value of favoritesCursor to the new cursor so we can close it in the activity's onDestroy() method.

//Close the cursor and database in the onDestroy() method

@Override

public void onDestroy(){

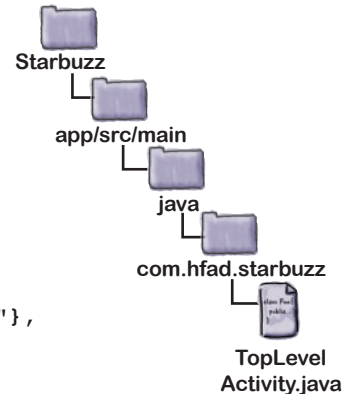
super.onDestroy();

favoritesCursor.close();

db.close();

}

}



Create a new version of the cursor.

Add the onRestart() method. This will get called when the user navigates back to TopLevelActivity.

Switch the cursor being used by the list_favorites list view to the new cursor.

Change the value of favoritesCursor to the new cursor so we can close it in the activity's onDestroy() method.

Let's see what happens when we run the app now.



Test drive the app

When we run the app, our favorite drinks are displayed in `TopLevelActivity` as before. When we click on one of the drinks, its details are displayed in `DrinkActivity`. If we uncheck the `favorite` checkbox for that drink and return to `TopLevelActivity`, the data in the `list_favorites` list view is refreshed and the drink is no longer displayed.

cursors and asyncTasks

Update layout
Populate list view
Refresh data



I've been thinking... Using databases in my app clearly has a lot of advantages, but doesn't opening and reading from the database slow the app down?

Databases are powerful, but they can be slow.

That means that even though our app works, we need to keep an eye on performance...

Databases can make your app go in sloooow-moooo....

Think about what your app has to do when it opens a database. It first needs to go searching for the database file. If the database file isn't there, it needs to create a blank database. Then it needs to run all of the SQL commands to create tables inside the database and any initial data it needs. Finally, it needs to fire off some queries to get the data out of there.

That all takes time. For a tiny database like the one used in the Starbuzz app, it's not a lot of time. But as a database gets bigger and bigger, that time will increase and increase. Before you know it, your app will lose its mojo and will be slower than YouTube on Thanksgiving.

There's not a lot you can do about the speed of creating and reading from a database, but you *can* prevent it from slowing down your interface.

Life is better when threads work together

The big problem with accessing a slow database is that can make your app feel unresponsive. To understand why, you need to think about how threads work in Android. Since Lollipop, there are three kinds of threads you need to think about:



The main event thread

This is the real workhorse in Android. It listens for intents, it receives touch messages from the screen, and it calls all of the methods inside your activities.



The render thread

You don't normally interact with this thread, but it reads a list of requests for screen updates and then calls the device's low-level graphics hardware to repaint the screen and make your app look pretty.



Any other threads that you create

If you're not careful, your app will do almost all of its work on the main event thread because this thread runs your event methods. If you just drop your database code into the `onCreate()` method (as we did in the Starbuzz app), then the main event thread will be busy talking to the database, instead of rushing off to look for any events from the screen or other apps. If your database code takes a long time, users will feel like they're being ignored or wonder if the app has crashed.

So the trick is to **move your database code off the main event thread and run it in a custom thread in the background**. We'll go through how you do this using the `DrinkActivity` code we wrote earlier in the chapter. As a reminder, the code updates the `FAVORITE` column in the Starbuzz database when the user clicks on the `favorite` checkbox, and displays a message if the database is unavailable.



We're going to run the `DrinkActivity` code to update the database in a background thread, but before we rush off and start hacking code, let's think about what we need to do.

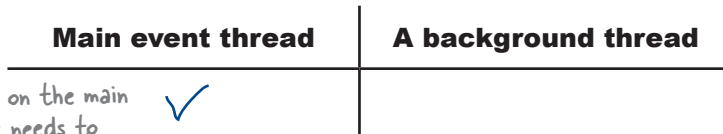
The code that we have at the moment does three different things. Choose the type of thread you think each should run on. We've completed the first one to start you off.

A Set up the interface.

```
int drinkId = (Integer) getIntent().getExtras().get(EXTRA_DRINKID);
CheckBox favorite = (CheckBox) findViewById(R.id.favorite);
ContentValues drinkValues = new ContentValues();
drinkValues.put("FAVORITE", favorite.isChecked());
```

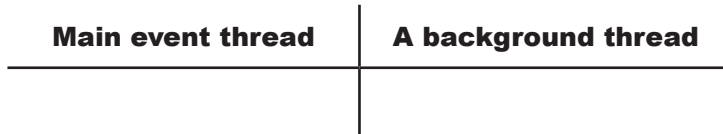


This code must run on the main event thread, as it needs to access the activity's views. ✓



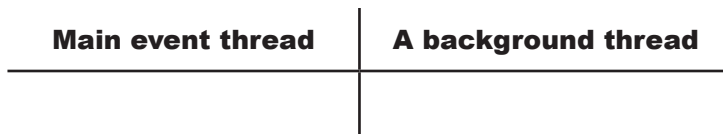
B Talk to the database.

```
SQLiteOpenHelper starbuzzDatabaseHelper = new StarbuzzDatabaseHelper(this);
SQLiteDatabase db = starbuzzDatabaseHelper.getWritableDatabase();
db.update("DRINK", ...);
```



C Update what's displayed on the screen.

```
Toast toast = Toast.makeText(...);
toast.show();
```



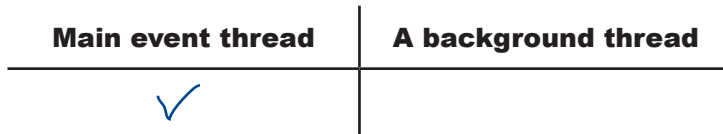
Sharpen your pencil Solution

We're going to run the `DrinkActivity` code to update the database in a background thread, but before we rush off and start hacking code, let's think about what we need to do.

The code that we have at the moment does three different things. Choose the type of thread you think each should run on. We've completed the first one to start you off.

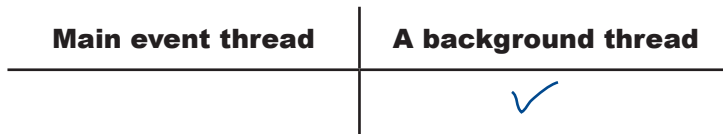
A Set up the interface.

```
int drinkId = (Integer) getIntent().getExtras().get(EXTRA_DRINKID);
CheckBox favorite = (CheckBox) findViewById(R.id.favorite);
ContentValues drinkValues = new ContentValues();
drinkValues.put("FAVORITE", favorite.isChecked());
```



B Talk to the database.

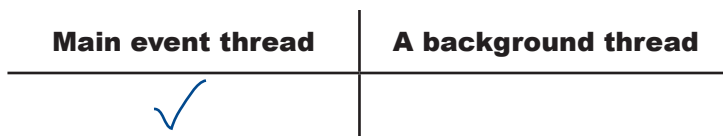
```
SQLiteOpenHelper starbuzzDatabaseHelper = new StarbuzzDatabaseHelper(this);
SQLiteDatabase db = starbuzzDatabaseHelper.getWritableDatabase();
db.update("DRINK", ...);
```



↑
We want to run the database code in the background.

C Update what's displayed on the screen.

```
Toast toast = Toast.makeText(...); ← We must run the code to display a message on the screen on the main event thread; otherwise, we'll get an exception.
toast.show();
```



What code goes on which thread?

When you use databases in your app, it's a good idea to run database code in a background thread, and update views with the database data in the main event thread. We're going to work through the `onFavoriteClicked()` method in the `DrinkActivity` code so that you can see how to approach this sort of problem.

Here's the code for the method (we've split it into sections, which we'll describe below):

```
//Update the database when the checkbox is clicked
public void onFavoriteClicked(View view){
```

1

```
int drinkId = (Integer) getIntent().getExtras().get(EXTRA_DRINKID);
CheckBox favorite = (CheckBox) findViewById(R.id.favorite);
ContentValues drinkValues = new ContentValues();
drinkValues.put("FAVORITE", favorite.isChecked());
```

2

```
SQLiteOpenHelper starbuzzDatabaseHelper = new StarbuzzDatabaseHelper(this);
try {
    SQLiteDatabase db = starbuzzDatabaseHelper.getWritableDatabase();
    db.update("DRINK", drinkValues,
            "_id = ?", new String[] {Integer.toString(drinkId)});
    db.close();
} catch (SQLiteException e) {
```

3

```
    Toast toast = Toast.makeText(this, "Database unavailable", Toast.LENGTH_SHORT);
    toast.show();
}
}
```

1

Code that needs to be run before the database code

The first few lines of code get the value of the `favorite` checkbox, and put it in the `drinkValues` `ContentValues` object. This code must be run before the database code.

2

Database code that needs to be run on a background thread

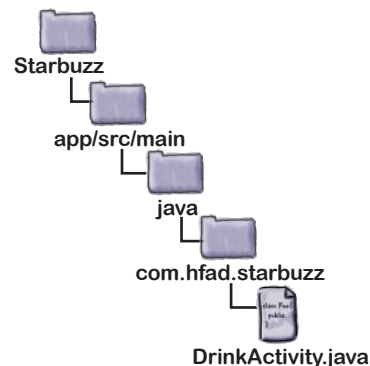
This updates the `DRINK` table.

3

Code that needs to be run after the database code

If the database is unavailable, we want to display a message to the user. This must run on the main event thread.

We're going to implement the code using an `AsyncTask`. What's that, you ask?



AsyncTask performs asynchronous tasks

An AsyncTask lets you perform operations in the background. When they've finished running, it then allows you to update views in the main event thread. If the task is repetitive, you can even use it to publish the progress of the task while it's running.

You create an AsyncTask by extending the AsyncTask class, and implementing its `doInBackground()` method. The code in this method runs in a background thread, so it's the perfect place for you to put database code. The AsyncTask class also has an `onPreExecute()` method that runs before `doInBackground()`, and an `onPostExecute()` method that runs afterward. There's an `onProgressUpdate()` method if you need to publish task progress.

Here's what an AsyncTask looks like:

```
private class MyAsyncTask extends AsyncTask<Params, Progress, Result>
```

```
protected void onPreExecute() {
    //Code to run before executing the task
}
```

```
protected Result doInBackground(Params... params) {
    //Code that you want to run in a background thread
}
```

```
protected void onProgressUpdate(Progress... values) {
    //Code that you want to run to publish the progress of your task
}
```

```
protected void onPostExecute(Result result) {
    //Code that you want to run when the task is complete
}
```

```
}
```

You add your AsyncTask class as an inner class to the activity that needs to use it.

This method is optional. It runs before the code you want to run in the background.

You must implement this method. It contains the code you want to run in the background.

This method is optional. It lets you publish progress of the code running in the background.

This method is also optional. It runs after the code has finished running in the background.

AsyncTask is defined by three generic parameters: Params, Progress, and Results. Params is the type of object used to pass any task parameters to the `doInBackground()` method, Progress is the type of object used to indicate task progress, and Result is the type of the task result. You can set any of these to `Void` if you're not going to use them.

We'll go through this over the next few pages by creating a new AsyncTask called `UpdateDrinkTask` we can use to update drinks in the background. Later on, we'll add this to our `DrinkActivity` code as an inner class.

The onPreExecute() method

We'll start with the `onPreExecute()` method. This gets called before the background task begins, and it's used to set up the task. It's called on the main event thread, so it has access to views in the user interface. The `onPreExecute()` method takes no parameters, and has a `void` return type.



In our case, we're going to use the `onPreExecute()` method to get the value of the favorite checkbox, and put it in the `drinkValues` `ContentValues` object. This is because we need access to the checkbox view in order to do this, and it must be done before any of our database code can be run. We're using a separate attribute outside the method for the `drinkValues` `ContentValues` object so that other methods in the class can access the `ContentValues` object (we'll look at these methods over the next few pages).

Here's the code:

```
private class UpdateDrinkTask extends AsyncTask<Params, Progress, Result> {

    private ContentValues drinkValues;

    protected void onPreExecute() {
        CheckBox favorite = (CheckBox) findViewById(R.id.favorite);
        drinkValues = new ContentValues();
        drinkValues.put("FAVORITE", favorite.isChecked());
    }

    ...

}
```

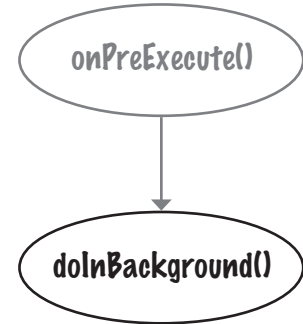
Before we run the database code, we need to get the value of the favorite checkbox.

Next, we'll look at the `doInBackground()` method.

The doInBackground() method

The `doInBackground()` method runs in the background immediately after `onPreExecute()`. You define what type of parameters the task should receive, and what the return type should be.

We're going to use the `doInBackground()` method for our database code so that it runs in a background thread. We'll pass it the ID of the drink we need to update; because the drink ID is an `int` value, we need to specify that the `doInBackground()` method receives `Integer` objects. We'll use a `Boolean` return value so we can tell whether the code ran successfully:



```
private class UpdateDrinkTask extends AsyncTask<Integer, Progress, Boolean> {  
    private ContentValues drinkValues;  
    ...  
    This code runs in a background thread.  
    protected Boolean doInBackground(Integer... drinks) {  
        int drinkId = drinks[0];  
        SQLiteOpenHelper starbuzzDatabaseHelper =  
            new StarbuzzDatabaseHelper(DrinkActivity.this);  
        try {  
            SQLiteDatabase db = starbuzzDatabaseHelper.getWritableDatabase();  
            db.update("DRINK", drinkValues,  
                "_id = ?", new String[] {Integer.toString(drinkId)});  
            db.close();  
            return true;  
        } catch(SQLiteException e) {  
            return false;  
        }  
    }  
    ...  
}
```

You change this to `Integer` to match the parameter of the `doInBackground()` method.

You change this to `Boolean` to match the return type of the `doInBackground()` method.

This is an array of `Integers`, but we'll just include one item, the drink ID.

The `update()` method uses the `drinkValues` object that the `onPreExecute()` method created.

Next, we'll look at the `onProgressUpdate()` method.

The onProgressUpdate() method

The `onProgressUpdate()` method is called on the main event thread, so it has access to views in the user interface. You can use this method to display progress to the user by updating views on the screen. You define what type of parameters the method should have.

The `onProgressUpdate()` method runs if a call to `publishProgress()` is made by the `doInBackground()` method like this:

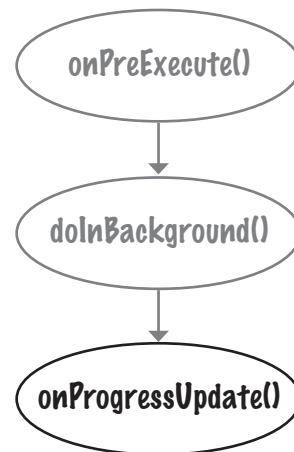
```
protected Boolean doInBackground(Integer... count) {
    for (int i = 0; i < count; i++) {
        publishProgress(i); ← This calls the onProgressUpdate()
                               method, passing in a value of i.
    }
}

protected void onProgressUpdate(Integer... progress) {
    setProgress(progress[0]);
}
```

In our app, we're not publishing the progress of our task, so we don't need to implement this method. We'll indicate that we're not using any objects for task progress by changing the signature of `UpdateDrinkTask`:

```
private class UpdateDrinkTask extends AsyncTask<Integer, Void, Boolean> {
    ...
}
```

Finally, we'll look at the `onPostExecute()` method.



We're not using the `onProgressUpdate()` method, so this is `Void`.

The onPostExecute() method

The `onPostExecute()` method is called after the background task has finished. It's called on the main event thread, so it has access to views in the user interface. You can use this method to present the results of the task to the user. The `onPostExecute()` method gets passed the results of the `doInBackground()` method, so it must take parameters that match the `doInBackground()` return type.

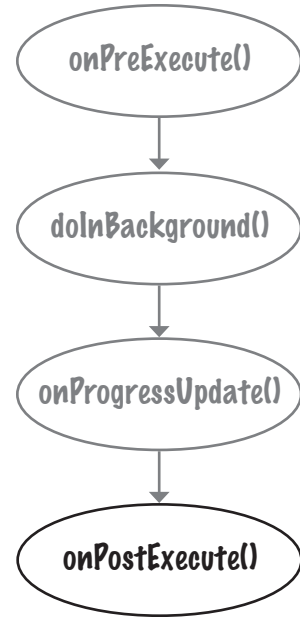
We're going to use the `onPostExecute()` method to check whether the database code in the `doInBackground()` method ran successfully. If it didn't, we'll display a message to the user. We're doing this in the `onPostExecute()` method, as this method can update the user interface; the `doInBackground()` method runs in a background thread, so it can't update views.

Here's the code:

```
private class UpdateDrinkTask extends AsyncTask<Integer, Void, Boolean> {  
  
    ...  
  
    protected void onPostExecute(Boolean success) {  
        if (!success) {  
            Toast toast = Toast.makeText(DrinkActivity.this,  
                "Database unavailable", Toast.LENGTH_SHORT);  
            toast.show();  
        }  
    }  
}
```

This is set to Boolean, as our doInBackground() method returns a Boolean.

Pass the toast the DrinkActivity context.



Now that we've written the code for our `AsyncTask` methods, let's revisit the `AsyncTask` class parameters.

The AsyncTask class parameters

When we first introduced the AsyncTask class, we said it was defined by three generic parameters: Params, Progress, and Results. You specify what these are by looking at the type of parameters used by your doInBackground(), onProgressUpdate(), and onPostExecute() methods. Params is the type of the doInBackground() parameters, Progress is the type of the onProgressUpdate() parameters, and Result is the type of the onPostExecute() parameters:

```
private class MyAsyncTask extends AsyncTask<Params, Progress, Result>

    protected void onPreExecute() {
        //Code to run before executing the task
    }

    protected Result doInBackground(Params... params) {
        //Code that you want to run in a background thread
    }

    protected void onProgressUpdate(Progress... values) {
        //Code that you want to run to publish the progress of your task
    }

    protected void onPostExecute(Result result) {
        //Code that you want to run when the task is complete
    }
}
```

In our example, doInBackground() takes Integer parameters, onPostExecute() takes a Boolean parameter, and we're not using the onProgressUpdate() method. This means that in our example, Params is Integer, Progress is Void, and Result is Boolean:

```
private class UpdateDrinkTask extends AsyncTask<Integer, Void, Boolean> {
    ...
    protected Boolean doInBackground(Integer... drinks) {
        ...
    }

    protected void onPostExecute(Boolean... success) {
        ...
    }
}
```

This is Void because we didn't implement the onProgressUpdate() method.

We'll show you the full UpdateDrinkTask class on the next page.

The full UpdateDrinkTask class

Here's the full code for the UpdateDrinkTask class. It needs to be added to DrinkActivity as an inner class, but we suggest you wait to do that until we show you how to execute it and show you the full *DrinkActivity.java* code listing.

```
private class UpdateDrinkTask extends AsyncTask<Integer, Void, Boolean> {
    private ContentValues drinkValues;
    protected void onPreExecute() {
        CheckBox favorite = (CheckBox) findViewById(R.id.favorite);
        drinkValues = new ContentValues();
        drinkValues.put("FAVORITE", favorite.isChecked());
    }
    protected Boolean doInBackground(Integer... drinks) {
        int drinkId = drinks[0];
        SQLiteOpenHelper starbuzzDatabaseHelper =
            new StarbuzzDatabaseHelper(DrinkActivity.this);
        try {
            SQLiteDatabase db = starbuzzDatabaseHelper.getWritableDatabase();
            db.update("DRINK", drinkValues,
                "_id = ?", new String[] {Integer.toString(drinkId)});
            db.close();
            return true;
        } catch (SQLiteException e) {
            return false;
        }
    }
    protected void onPostExecute(Boolean success) {
        if (!success) {
            Toast toast = Toast.makeText(DrinkActivity.this,
                "Database unavailable", Toast.LENGTH_SHORT);
            toast.show();
        }
    }
}
```

We've defined `drinkValues` as a private variable, as it's used by the `onExecute()` and `doInBackground()` methods.

Before we run the database code, we need to put the value of the favorite checkbox in the `drinkValues` `ContentValues` object.

Our database code goes in the `doInBackground()` method.

After the database code has run in the background, check whether it ran successfully. If it didn't, display a message.

We have to put the code to display a message in the `onPostExecute()` method, as it needs to be run on the main event thread to update the screen.

Execute the AsyncTask...

You run the AsyncTask by calling the AsyncTask `execute()` method and passing it any parameters required by the `doInBackground()` method. As an example, we want to pass the drink the user chose to our AsyncTask's `doInBackground()` method, so we call it using:

```
int drinkId = (Integer) getIntent().getExtras().get(EXTRA_DRINKID);
new UpdateDrinkTask().execute(drinkId); ← Execute the AsyncTask and pass it the drink ID.
```

The type of parameter you pass with the `execute()` method must match the type of parameter expected by the AsyncTask `doInBackground()` method. We're passing an integer value (the drink ID), which matches the type of parameter expected by our `doInBackground()` method:

```
protected Boolean doInBackground(Integer... drinks) {
    ...
}
```

...in DrinkActivity's onFavoritesClicked() method

Our `UpdateDrinkTask` class (the AsyncTask we created) needs to update the FAVORITE column in the Starbuzz database whenever the favorite checkbox in `DrinkActivity` is clicked. We therefore need to execute it in `DrinkActivity`'s `onFavoritesClicked()` method. Here's what the new version of the method looks like:

```
//Update the database when the checkbox is clicked
public void onFavoriteClicked(View view) {
    int drinkId = (Integer) getIntent().getExtras().get(EXTRA_DRINKID);
    new UpdateDrinkTask().execute(drinkId); ← The new version of the
}                                             onFavoritesClicked() method no
                                             longer contains code to update the
                                             FAVORITE column. Instead, it calls
                                             the AsyncTask, which performs the
                                             update in the background.
```

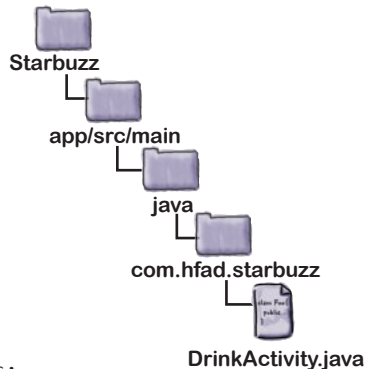
We'll show you the new `DrinkActivity.java` code over the next few pages.

The full DrinkActivity.java code

Here's the complete code for *DrinkActivity.java*; update your version of the code to reflect our changes:

```
package com.hfad.starbuzz;

import android.app.Activity;
import android.os.Bundle;
import android.widget.ImageView;
import android.widget.TextView;
import android.widget.Toast;
import android.database.Cursor;
import android.database.sqlite.SQLiteDatabase;
import android.database.sqlite.SQLiteException;
import android.database.sqlite.SQLiteOpenHelper;
import android.view.View;
import android.widget.CheckBox;
import android.content.ContentValues;
import android.os.AsyncTask; ← We're using the AsyncTask class, so we need to import it.
```



```
public class DrinkActivity extends Activity {

    public static final String EXTRA_DRINKID = "drinkId";

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

        //Get the drink from the intent
        int drinkId = (Integer) getIntent().getExtras().get(EXTRA_DRINKID);

        //Create a cursor
        SQLiteOpenHelper starbuzzDatabaseHelper = new StarbuzzDatabaseHelper(this);
        try {
            SQLiteDatabase db = starbuzzDatabaseHelper.getReadableDatabase();
            Cursor cursor = db.query("DRINK",
                new String[]{"NAME", "DESCRIPTION", "IMAGE_RESOURCE_ID", "FAVORITE"},
                "_id = ?",
                new String[]{Integer.toString(drinkId)},
                null, null, null);
```

← We don't need to change the onCreate() method, we're just showing it for completeness.

The code continues → on the next page.

The full DrinkActivity.java code (continued)

```

//Move to the first record in the Cursor
if (cursor.moveToFirst()) {
    //Get the drink details from the cursor
    String nameText = cursor.getString(0);
    String descriptionText = cursor.getString(1);
    int photoId = cursor.getInt(2);
    boolean isFavorite = (cursor.getInt(3) == 1);

    //Populate the drink name
    TextView name = (TextView) findViewById(R.id.name);
    name.setText(nameText);

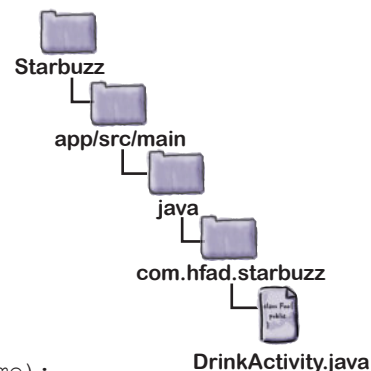
    //Populate the drink description
    TextView description = (TextView) findViewById(R.id.description);
    description.setText(descriptionText);

    //Populate the drink image
    ImageView photo = (ImageView) findViewById(R.id.photo);
    photo.setImageResource(photoId);
    photo.setContentDescription(nameText);

    //Populate the favorite checkbox
    CheckBox favorite = (CheckBox) findViewById(R.id.favorite);
    favorite.setChecked(isFavorite);
}
cursor.close();
db.close();
} catch (SQLException e) {
    Toast toast = Toast.makeText(this,
        "Database unavailable",
        Toast.LENGTH_SHORT);
    toast.show();
}
}

```

None of the code on this page needs to change.



The code continues  on the next page.

The full DrinkActivity.java code (continued)

```

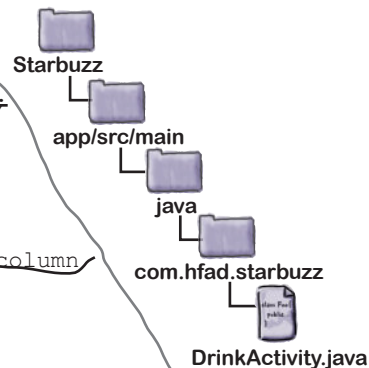
//Update the database when the checkbox is clicked
public void onFavoriteClicked(View view){
    int drinkId = (Integer) getIntent().getExtras().get(EXTRA_DRINKID);


    //Get the value of the checkbox
    CheckBox favorite = (CheckBox) findViewById(R.id.favorite);
    ContentValues drinkValues = new ContentValues();
    drinkValues.put("FAVORITE", favorite.isChecked());

    //Get a reference to the database and update the FAVORITE column
    SQLiteOpenHelper starbuzzDatabaseHelper =
        new StarbuzzDatabaseHelper(this);
    try {
        SQLiteDatabase db = starbuzzDatabaseHelper.getWritableDatabase();
        db.update("DRINK",
            
                drinkValues,
                "_id = ?",
                new String[] {Integer.toString(drinkId)};
            
            db.close();
    } catch (SQLiteException e) {
        Toast toast = Toast.makeText(this, "Database unavailable", Toast.LENGTH_SHORT);
        toast.show();
    }


    new UpdateDrinkTask().execute(drinkId); ← Execute the task.
}

```



Delete all these lines of code, as we're now using an AsyncTask for these actions.

The code continues →
on the next page.

The full DrinkActivity.java code (continued)

```

//Inner class to update the drink.
private class UpdateDrinkTask extends AsyncTask<Integer, Void, Boolean> {
    private ContentValues drinkValues;

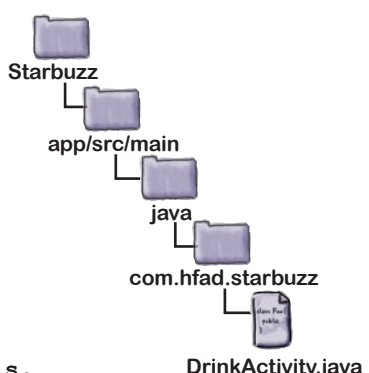
    protected void onPreExecute() {
        CheckBox favorite = (CheckBox) findViewById(R.id.favorite);
        drinkValues = new ContentValues();
        drinkValues.put("FAVORITE", favorite.isChecked());
    }

    protected Boolean doInBackground(Integer... drinks) {
        int drinkId = drinks[0];
        SQLiteOpenHelper starbuzzDatabaseHelper =
            new StarbuzzDatabaseHelper(DrinkActivity.this);
        try {
            SQLiteDatabase db = starbuzzDatabaseHelper.getWritableDatabase();
            db.update("DRINK", drinkValues,
                "_id = ?", new String[] {Integer.toString(drinkId)});
            db.close();
            return true;
        } catch (SQLiteException e) {
            return false;
        }
    }

    protected void onPostExecute(Boolean success) {
        if (!success) {
            Toast toast = Toast.makeText(DrinkActivity.this,
                "Database unavailable", Toast.LENGTH_SHORT);
            toast.show();
        }
    }
}

```

Add the AsyncTask to the activity as an inner class.
Before the database code runs, put the value of the checkbox in the drinkValues ContentValues object.
Run the database code in a background thread.
Update the value of the FAVORITE column.
If the database code didn't run correctly, display a message to the user.



```

graph TD
    Starbuzz[Starbuzz] --> app_src_main[app/src/main]
    app_src_main --> java[java]
    java --> com_hfad_starbuzz[com.hfad.starbuzz]
    com_hfad_starbuzz --> DrinkActivity_java[DrinkActivity.java]

```

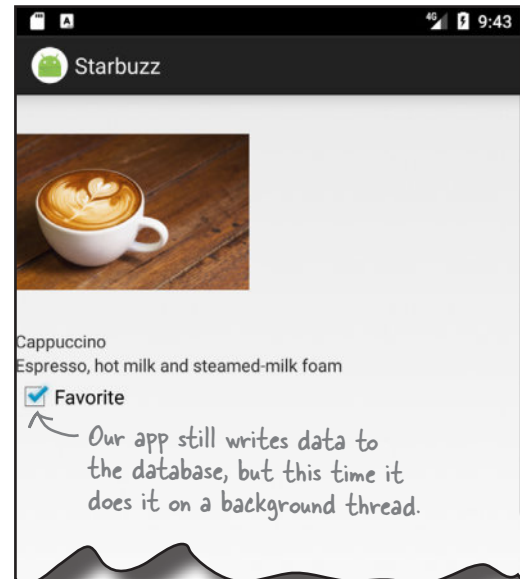
That's everything you need in order to create an AsyncTask.
Let's check what happens when we run the app.



Test drive the app

When we run the app and navigate to a drink, we can indicate that the drink is a favorite drink by checking the “favorite” checkbox. Clicking on the checkbox still updates the FAVORITE column in the database with its value, but this time the code is running on a background thread.

In an ideal world, all of your database code should run in the background. We’re not going to change our other Starbuzz activities to do this, but why not make this change yourself?



there are no Dumb Questions

Q: I’ve written code before that just ran the database code and it was fine. Do I really need to run it in the background?

A: For really small databases, like the one in the Starbuzz app, you probably won’t notice the time it takes to access the database. But that’s just because the database is small. If you use a larger database, or if you run an app on a slower device, the time it takes to access the database will be significant. So yes, you should *always* run database code in the background.

Q: Remind me—why is it bad to update a view from the background thread?

A: The short answer is that it will throw an exception if you try. The longer answer is that multithreaded user interfaces are hugely buggy. Android avoided the problem by simply banning them.

Q: Which part of the database code is slowest: opening the database, or reading data from it?

A: There’s no general way of knowing. If your database has a complex data structure, opening the database for the first time will take a long time because it will need to create all the tables. If you’re running a complex query, that might take a very long time. In general, play it safe and run everything in the background.

Q: If it take a few seconds to read data from the database, what will the user see?

A: The user will see blank views until the database code sets the values.

Q: Why have we put the database code for just one activity in an `AsyncTask`?

A: We wanted to show you how to use `AsyncTasks` in one activity as an example. In the real world, you should do this for the database code in all your activities.



Your Android Toolbox

You've got Chapter 17 under your belt and now you've added writing to SQLite databases to your toolbox.

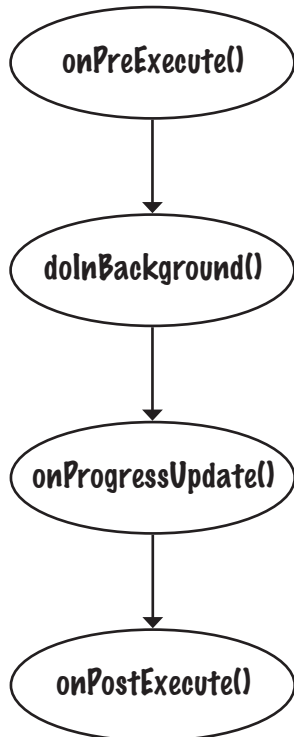
You can download the full code for the chapter from <https://tinyurl.com/HeadFirstAndroid>.



BULLET POINTS

- The `CursorAdapter` `changeCursor()` method replaces the cursor currently used by a cursor adapter with a new cursor that you provide. It then closes the old cursor.
- Run your database code in a background thread using `AsyncTask`.

A summary of the AsyncTask steps



- 1 **onPreExecute()** is used to set up the task. It's called before the background task begins, and runs on the main event thread.
- 2 **doInBackground()** runs in the background thread. It runs immediately after `onPreExecute()`. You can specify what type of parameters it has, and what its return type is.
- 3 **onProgressUpdate()** is used to display progress. It runs in the main event thread when the `doInBackground()` method calls `publishProgress()`.
- 4 **onPostExecute()** is used to display the task outcome to the user when `doInBackground` has finished. It runs in the main event thread and takes the return value of `doInBackground()` as a parameter.

18 started services

At Your Service



There are some operations you want to keep on running, irrespective of which app has the focus. If you start downloading a file, for instance, *you don't want the download to stop when you switch to another app*. In this chapter we'll introduce you to **started services**, components that *run operations in the background*. You'll see how to create a started service using the **IntentService** class, and find out how its lifecycle fits in with that of an activity. Along the way, you'll discover how to **log messages**, and *keep users informed* using Android's built-in **notification service**.

Services work in the background

An Android app is a collection of activities and other components. The bulk of your app's code is there to interact with the user, but sometimes you need to do things in the background, such as download a large file, stream a piece of music, or listen for a message from the server.

These kinds of tasks aren't what activities are designed to do. In simple cases, you can create a thread, but if you're not careful your activity code will start to get complex and unreadable.

That's why **services** were invented. A service is an application component like an activity but without a user interface. They have a simpler lifecycle than activities, and they come with a bunch of features that make it easy to write code that will run in the background while the user is doing something else.

There are three types of service

Services come in three main flavors:



Started services

A started service can run in the background indefinitely, even when the activity that started it is destroyed. If you wanted to download a large file from the Internet, you would probably create it as a started service. Once the operation is done, the service stops.



Bound services

A bound service is bound to another application component such as an activity. The activity can interact with it, send requests, and get results. A bound service runs as long as components are bound to it. When the components are no longer bound, the service is destroyed. If you wanted to create an odometer to measure the distance traveled by a vehicle, for example, you'd probably use a bound service. This way, any activities bound to the service could keep asking the service for updates on the distance traveled.



Scheduled services

A scheduled service is one that's scheduled to run at a particular time. As an example, from API 21, you can schedule jobs to run at an appropriate time.

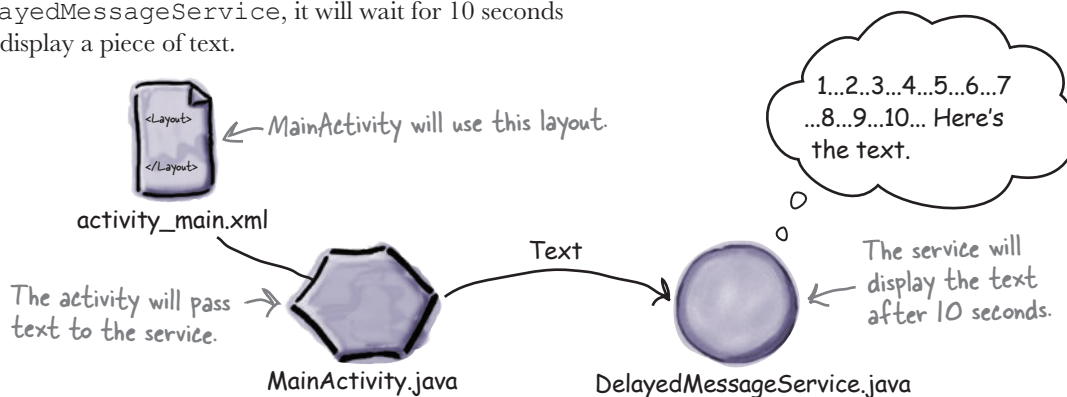
In this chapter, we're going to look at how you create a started service.

In addition to writing your own services, you can use Android's built-in ones.

Built-in services include the notification service, location service, alarm service, and download service.

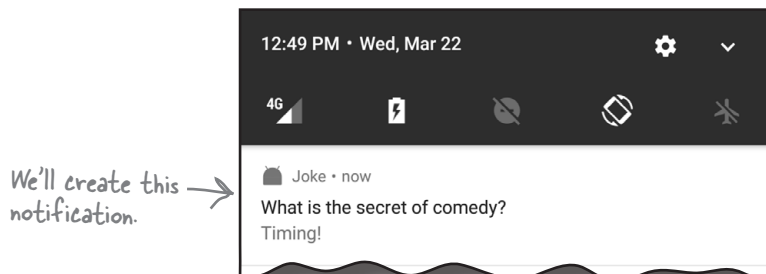
We'll create a **STARTED** service

We're going to create a new project that contains an activity called `MainActivity`, and a started service called `DelayedMessageService`. Whenever `MainActivity` calls `DelayedMessageService`, it will wait for 10 seconds and then display a piece of text.



We're going to do this in two stages:

- 1 **Display the message in Android's log.**
We'll start by displaying the message in Android's log so that we can check that the service works OK. We can look at the log in Android Studio.
- 2 **Display the message in a notification.**
We'll get `DelayedMessageService` to use Android's built-in notification service to display the message in a notification.



Create the project

We'll start by creating the project. Create a new Android project for an application named "Joke" with a company domain of "hfad.com", making the package name `com.hfad.joke`. The minimum SDK should be API 19 so that it will work with most devices. You'll need an empty activity named "MainActivity" and a layout named "activity_main" so that your code matches ours. **Make sure that you uncheck the Backwards Compatibility (AppCompat) option when you create the activity.**

The next thing we need to do is create the service.

Use the IntentService class to create a basic started service

The simplest way of creating a started service is to extend the `IntentService` class, as it provides you with most of the functionality you need. You start it with an intent, and it runs the code that you specify in a separate thread.

We're going to add a new intent service to our project. To do this, switch to the Project view of Android Studio's explorer, click on the `com.hfad.joke` package in the `app/src/main/java` folder, go to `File`→`New...`, and select the `Service` option. When prompted, choose the option to create a new Intent Service. Name the service "DelayedMessageService" and uncheck the option to include helper start methods to minimize the amount of code that Android Studio generates for you. Click on the `Finish` button, then replace the code in `DelayedMessageService.java` with the code here:

```
package com.hfad.joke;

import android.app.IntentService;
import android.content.Intent;

public class DelayedMessageService extends IntentService {

    public DelayedMessageService() {
        super("DelayedMessageService");
    }

    @Override
    protected void onHandleIntent(Intent intent) {
        //Code to do something
    }
}
```

Extend the `IntentService` class.

Put the code you want the service to run in the `onHandleIntent()` method.

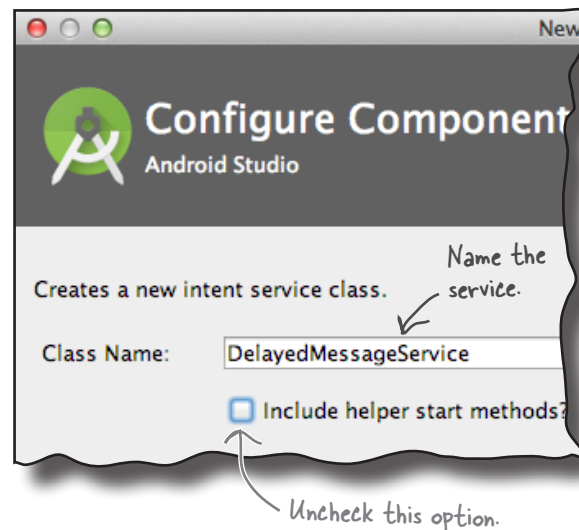
The above code is all you need to create a basic intent service. You extend the `IntentService` class, add a public constructor, and implement the `onHandleIntent()` method.

The `onHandleIntent()` method should contain the code you want to run each time the service is passed an intent. It runs in a separate thread. If it's passed multiple intents, it deals with them one at a time.

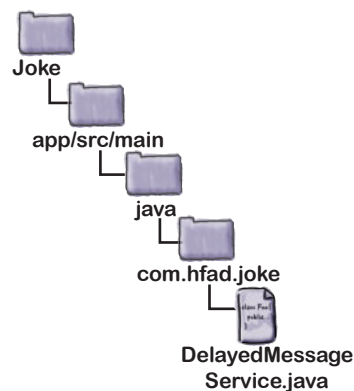
We want `DelayedMessageService` to display a message in Android's log, so let's look at how you log messages.



Log
Display notification



Some versions of Android Studio may ask you what the source language should be. If prompted, select the option for Java.





How to log messages

Adding messages to a log can be a useful way of checking that your code works the way you want. You tell Android what to log in your Java code, and when the app's running, you check the output in Android's log (a.k.a. logcat).

You log messages using one of the following methods in the `android.util.Log` class:

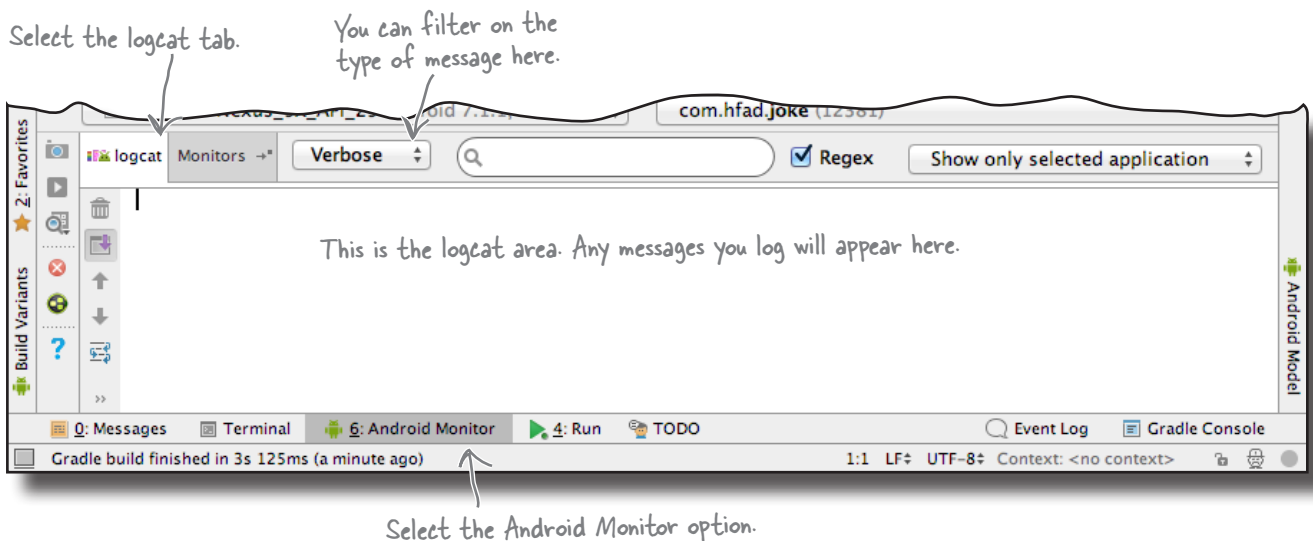
<code>Log.v(String tag, String message)</code>	Logs a verbose message.
<code>Log.d(String tag, String message)</code>	Logs a debug message.
<code>Log.i(String tag, String message)</code>	Logs an information message.
<code>Log.w(String tag, String message)</code>	Logs a warning message.
<code>Log.e(String tag, String message)</code>	Logs an error message.

There's also a `Log.wtf()` method you can use to report exceptions 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 AVD 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. As an example, to log a verbose message that's come from `DelayedMessageService`, you use the `Log.v()` method like this:

```
Log.v("DelayedMessageService", "This is a message");
```

You can view the logcat in Android Studio and filter by the different types of message. To see the logcat, select the Android Monitor option at the bottom of your project screen in Android Studio and then select the logcat tab:





The full DelayedMessageService code

We want our service to get a piece of text from an intent, wait for 10 seconds, then display the piece of text in the log. To do this, we'll add a `showText()` method to log the text, and then call it from the `onHandleIntent()` method after a delay of 10 seconds.

Here's the full code for *DelayedMessageService.java* (update your version of the code to reflect our changes):

```
package com.hfad.joke;

import android.app.IntentService;
import android.content.Intent;
import android.util.Log; ← We're using the Log class, so we need to import it.

public class DelayedMessageService extends IntentService {

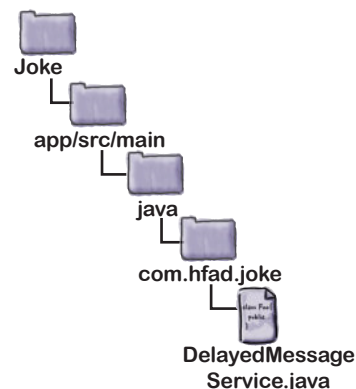
    public static final String EXTRA_MESSAGE = "message"; ← Use a constant to pass
                                                           a message from the
                                                           activity to the service.

    public DelayedMessageService() {
        super("DelayedMessageService"); ← Call the super constructor.
    }

    This method contains the code you want to
    run when the service receives an intent.
    @Override
    protected void onHandleIntent(Intent intent) {
        synchronized (this) {
            try {
                wait(10000); ← Wait 10 seconds.
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        Get the text from the intent
        String text = intent.getStringExtra(EXTRA_MESSAGE);
        showText(text);
    }

    Call the showText() method.
    private void showText(final String text) {
        Log.v("DelayedMessageService", "The message is: " + text);
    }
}

This logs a piece of text so we can see it in
the logcat through Android Studio.
```





You declare services in AndroidManifest.xml

Just like activities, each service needs to be declared in *AndroidManifest.xml* so that Android can call it; if a service isn't declared in this file, Android won't know it's there and won't be able to call it.

Android Studio should update *AndroidManifest.xml* for you automatically whenever you create a new service by adding a new `<service>` element. Here's what our *AndroidManifest.xml* code looks like:

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


    <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/AppTheme">
        <activity android:name=".MainActivity">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>

        <service
            android:name=".DelayedMessageService"
            android:exported="false">
        </service>
    </application>
</manifest>
```

Don't worry if this code looks different than yours.

You declare a service in *AndroidManifest.xml* like this. Android Studio should do this for you automatically.

The service name has a "." in front of it so that Android can combine it with the package name to derive the fully qualified class name.



The `<service>` element contains two attributes: `name` and `exported`. The `name` attribute tells Android what the name of the service is—in our case, `DelayedMessageService`. The `exported` attribute tells Android whether the service can be used by other apps. Setting it to `false` means that the service will only be used within the current app.

Now that we have a service, let's get `MainActivity` to start it.

Add a button to activity_main.xml

We're going to get `MainActivity` to start `DelayedMessageService` whenever a button is clicked, so we'll add the button to `MainActivity`'s layout.

First, add the following values to `strings.xml`:

```
<string name="question">What is the secret of comedy?</string>
<string name="response">Timing!</string>
```

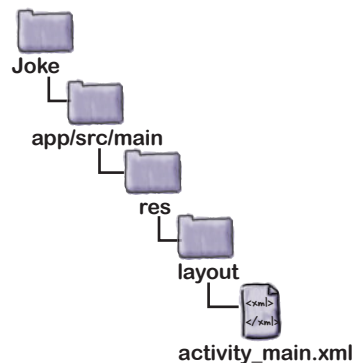
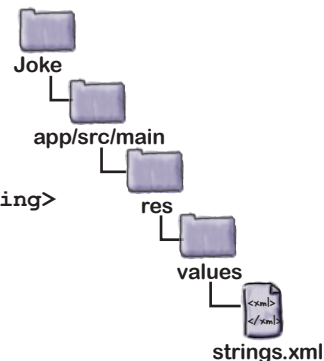
Then, replace your `activity_main.xml` code with ours below so that `MainActivity` displays a button:

```
<?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="com.hfad.joke.MainActivity">

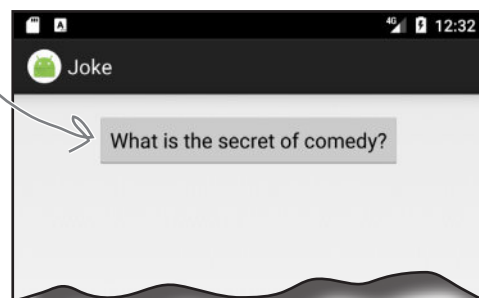
    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="center_horizontal"
        android:text="@string/question"
        android:id="@+id/button"
        android:onClick="onClick"/>

</LinearLayout>
```

The button will call an `onClick()` method whenever the user clicks it, so we'll add this method to `MainActivity`.



This creates a button. When it's clicked, the `onClick()` method in the activity will get called.





You start a service using startService()

We'll use `MainActivity`'s `onClick()` method to start `DelayedMessageService` whenever the user clicks on the button. You start a service from an activity in a similar way to how you start an activity. You create an explicit intent that's directed at the service you want to start, then use the `startService()` method in your activity to start it:

```
Intent intent = new Intent(this, DelayedMessageService.class);
startService(intent);
```

Starting a service is just like starting an activity, except you use `startService()` instead of `startActivity()`.

Here's our code for `MainActivity.java`; update your version to match ours:

```
package com.hfad.joke;
```

```
import android.app.Activity;
import android.content.Intent;
import android.os.Bundle;
import android.view.View;
```

We're using these classes, so we need to import them.

```
public class MainActivity extends Activity {
```

```
@Override
```

```
protected void onCreate(Bundle savedInstanceState) {
```

```
    super.onCreate(savedInstanceState);
```

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

```
}
```

```
public void onClick(View view) {
```

```
    Intent intent = new Intent(this, DelayedMessageService.class);
```

```
    intent.putExtra(DelayedMessageService.EXTRA_MESSAGE,
```

```
        getResources().getString(R.string.response));
```

```
    startService(intent);
```

```
}
```

```
}
```

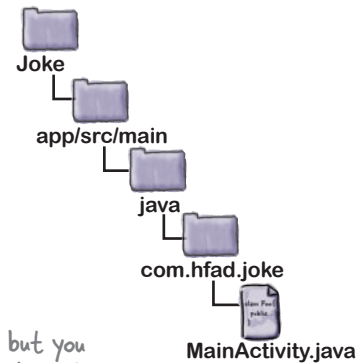
Start the service.

This will run when the button gets clicked.

Create the intent.

We're using `Activity` here, but you could use `AppCompatActivity` instead.

Add text to the intent.



That's all the code we need to get our activity to start the service. Before we take it for a test drive, let's go through what happens when the code runs.

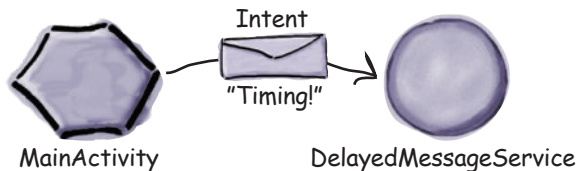


What happens when you run the app

Here's what the code does when we run the app:

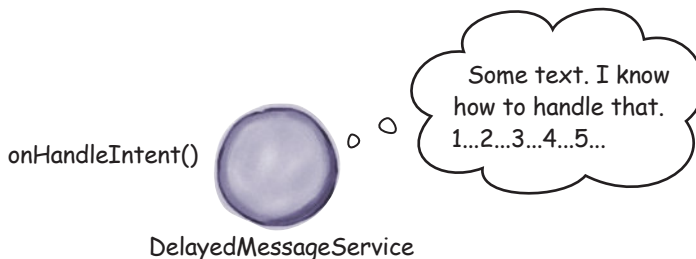
- 1 MainActivity starts DelayedMessageService by calling startService() and passing it an intent.**

The intent contains the message MainActivity wants DelayedMessageService to display, in this case "Timing!".

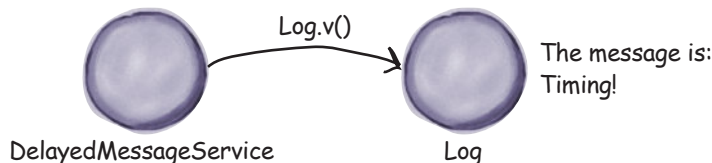


- 2 When DelayedMessageService receives the intent, its onHandleIntent() method runs.**

DelayedMessageService waits for 10 seconds.



- 3 DelayedMessageService logs the message.**



- 4 When DelayedMessageService has finished running, it's destroyed.**

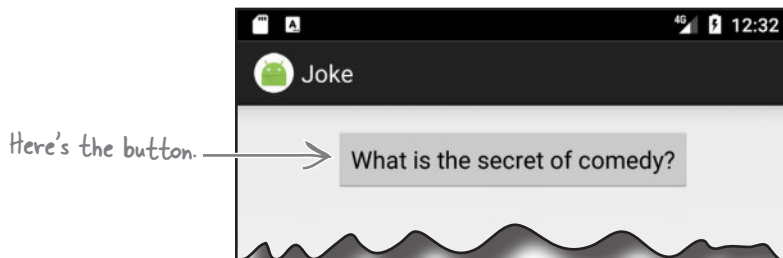


Let's take the app for a test drive so we can see it working.

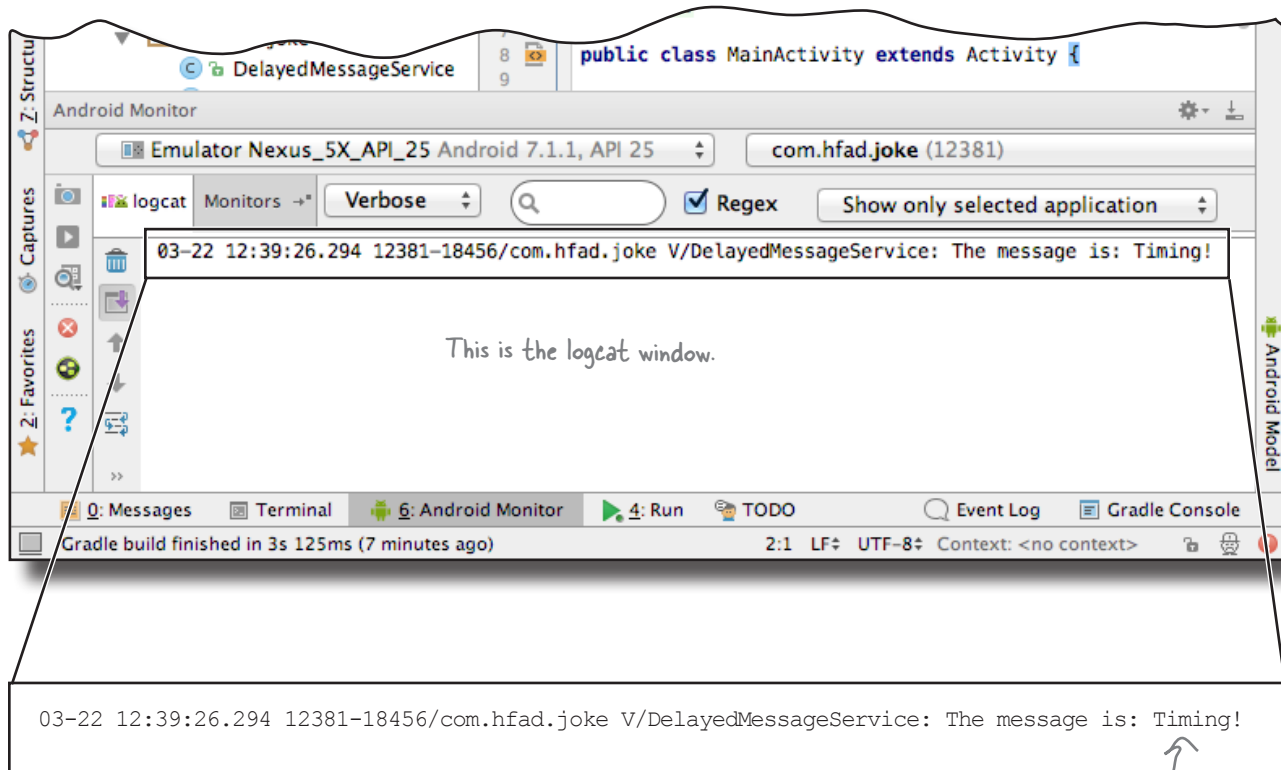


Test drive the app

When you run the app, MainActivity is displayed. It contains a single button:



Press the button, switch back to Android Studio, and watch the logcat output in the bottom part of the IDE. After 10 seconds, the message “Timing!” should appear in the logcat.



Now that you've seen DelayedMessageService running, let's look in more detail at how started services work.

After a 10-second delay, the message is displayed in the log.

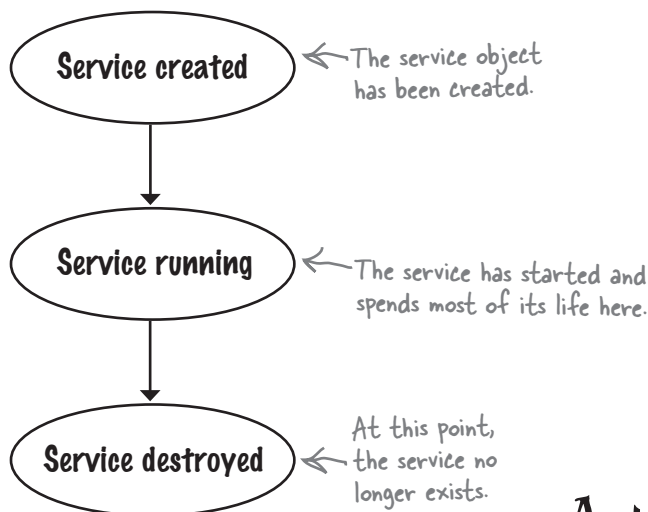


Log
Display notification

The states of a started service

When an application component (such as an activity) starts a service, the service moves from being created to running to being destroyed.

A started service spends most of its life in a **running** state; it's been started by another component such as an activity, and it runs code in the background. It continues to run even if the component that started it is destroyed. When the service has finished running code, it's **destroyed**.



Just like an activity, when a service moves from being created to being destroyed, it triggers key service lifecycle methods, which it inherits.

When the service is created, its `onCreate()` method gets called. You override this method if you want to perform any tasks needed to set up the service.

When the service is ready to start, its `onStartCommand()` method is called. If you're using an `IntentService` (which is usually the case for a started service), you don't generally override this method. Instead, you add any code you want the service to run to its `onHandleIntent()` method, which is called after `onStartCommand()`.

The `onDestroy()` method is called when the started service is no longer running and it's about to be destroyed. You override this method to perform any final cleanup tasks, such as freeing up resources.

We'll take a closer look at how these methods fit into the service states on the next page.

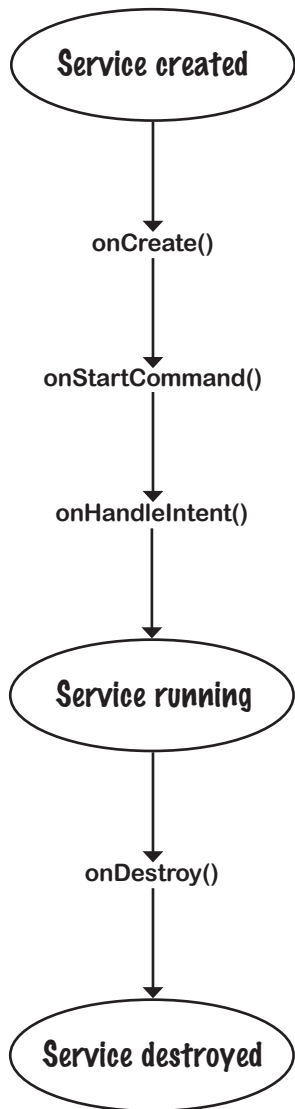
A started service runs after it's been started.

`onCreate()` gets called when the service is first created, and it's where you do any service setup.

`onDestroy()` gets called just before the service gets destroyed.

The started service lifecycle: from create to destroy

Here's an overview of the started service lifecycle from birth to death.

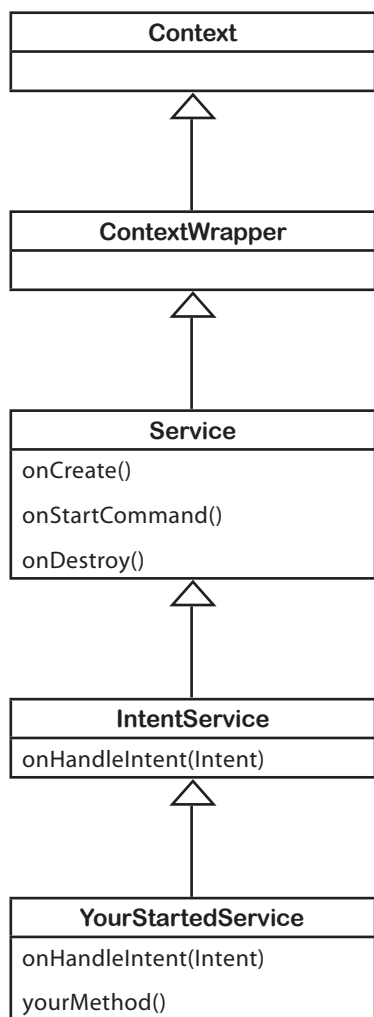


- 1 **The component calls `startService()` and the service gets created.**
- 2 **The `onCreate()` method runs immediately after the service is created.**
The `onCreate()` method is where any service initialization code should go, as this method always gets called after the service has launched but before it starts running.
- 3 **The `onStartCommand()` method runs when the service is about to start.**
If your started service extends the `IntentService` class (which is usually the case), the `onStartCommand()` method creates a separate thread and `onHandleIntent()` is called. You add any code you want the service to run in the background to `onHandleIntent()`.
- 4 **The service spends most of its life running.**
- 5 **The `onDestroy()` method runs when the service has finished running, immediately before it's destroyed.**
The `onDestroy()` method enables you to perform any final cleanup tasks such as freeing up resources.
- 6 **After the `onDestroy()` method has run, the service is destroyed.**
The service ceases to exist.

The `onCreate()`, `onStartCommand()`, and `onDestroy()` methods are three of the main service lifecycle methods. So where do these methods come from?

Your service inherits the lifecycle methods

As you saw earlier in the chapter, the started service you created extends the `android.app.IntentService` class. This class gives your service access to the Android lifecycle methods. Here's a diagram showing the class hierarchy:



Context abstract class

(`android.content.Context`)

An interface to global information about the application environment. Allows access to application resources, classes, and operations.

ContextWrapper class

(`android.content.ContextWrapper`)

A proxy implementation for the `Context`.

Service class

(`android.app.Service`)

The `Service` class implements default versions of the lifecycle methods. You'll find out more about it in the next chapter.

IntentService class

(`android.app.IntentService`)

The `IntentService` class gives you an easy way of creating started services. It includes a method, `onHandleIntent()`, which handles any intents it's given on a background thread.

YourStartedService class

(`com.hfad.foo`)

Most of the behavior of your started service is handled by superclass methods your service inherits. All you do is override the methods you need and add a public constructor.

Now that you understand more about how started services work behind the scenes, have a go at the following exercise. After that, we'll look at how we can make `DelayedMessageService` display its message in a notification.



Service Magnets

Below is most of the code needed to create a started service called `WombleService` that plays a `.mp3` file in the background, and an activity that uses it. See if you can finish off the code.

```
public class WombleService extends .....{

    public WombleService() {
        super("WombleService");
    }

    @Override
    protected void ..... (Intent intent) {
        MediaPlayer mediaPlayer =
            MediaPlayer.create(getApplicationContext(), R.raw.wombling_song);
        mediaPlayer.start();
    }
}
```

← This is the service.

← This uses the Android MediaPlayer class to play a file called `wombling_song.mp3`. The file is located in the `res/raw` folder.

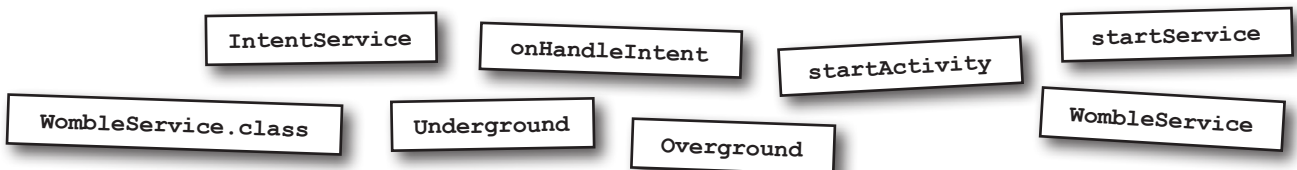
```
public class MainActivity extends Activity {

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

    public void onClick(View view) {
        Intent intent = new Intent(this, .....);
        ..... (intent);
    }
}
```

← This is the activity.

You won't need to use all of these magnets.





Service Magnets Solution

Below is most of the code needed to create a started service called `WombleService` that plays a `.mp3` file in the background, and an activity that uses it. See if you can finish off the code.

```
public class WombleService extends IntentService {
    // This is the service. It extends the IntentService class.

    public WombleService() {
        super("WombleService");
    }

    // The code needs to run in the onHandleIntent() method.
    @Override
    protected void onHandleIntent(Intent intent) {
        MediaPlayer mediaPlayer =
            MediaPlayer.create(getApplicationContext(), R.raw.wombling_song);
        mediaPlayer.start();
    }
}
```

```
public class MainActivity extends Activity {
    // This is the activity.

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

    public void onClick(View view) {
        Intent intent = new Intent(this, WombleService.class);
        // Create an explicit intent directed at WombleService.class.

        startService(intent);
        // Start the service.
    }
}
```

You didn't need to use these magnets.

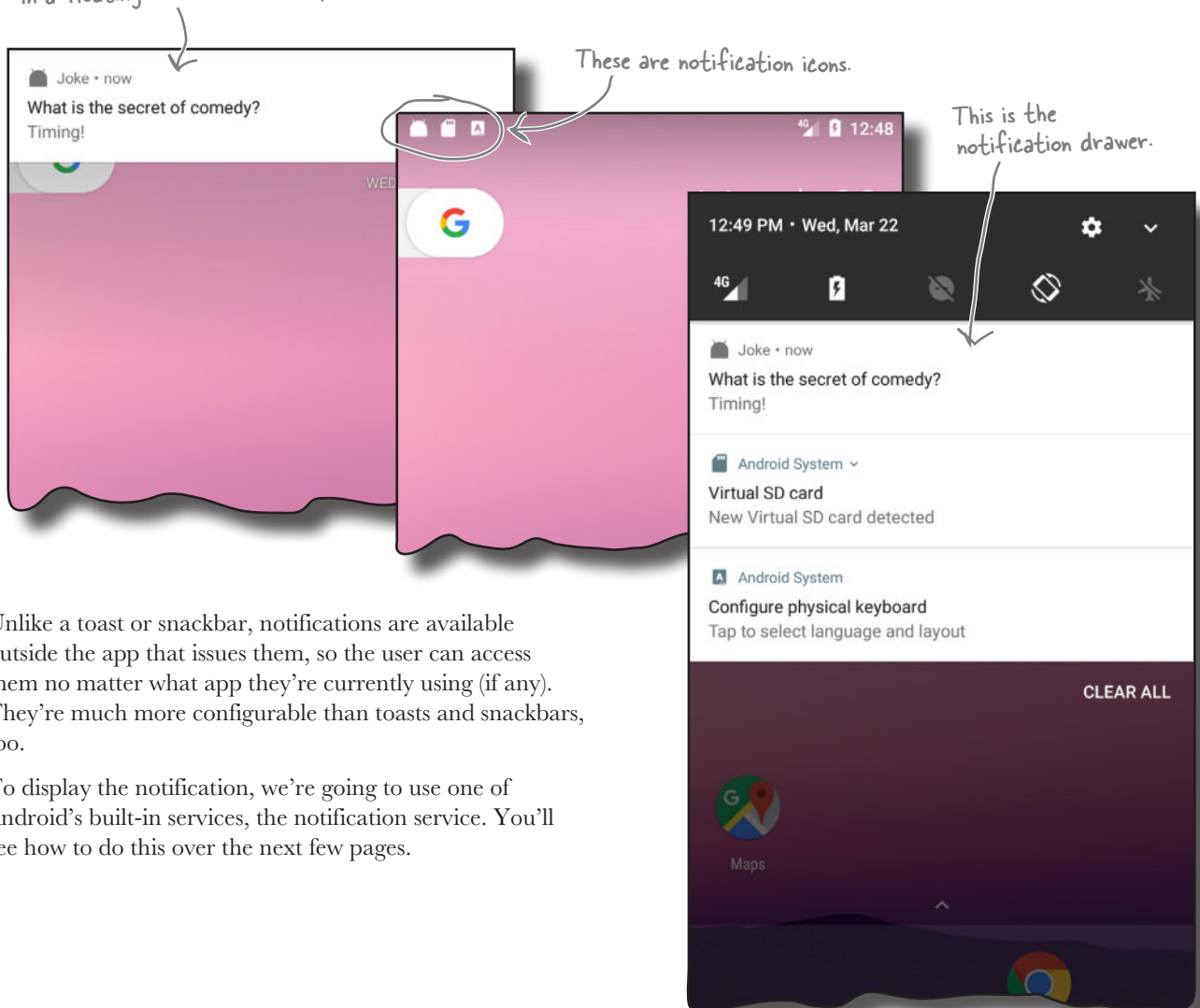




Android has a built-in notification service

We're going to change our Joke app so that our message gets displayed in a **notification**. Notifications are messages that are displayed outside the app's user interface. When a notification gets issued, it's displayed as an icon in the notification area of the status bar. You can see details of the notification in the notification drawer, which you access by swiping down from the top of the screen:

Heads-up notifications are temporarily displayed in a floating window at the top of the screen.



Unlike a toast or snackbar, notifications are available outside the app that issues them, so the user can access them no matter what app they're currently using (if any). They're much more configurable than toasts and snackbars, too.

To display the notification, we're going to use one of Android's built-in services, the notification service. You'll see how to do this over the next few pages.

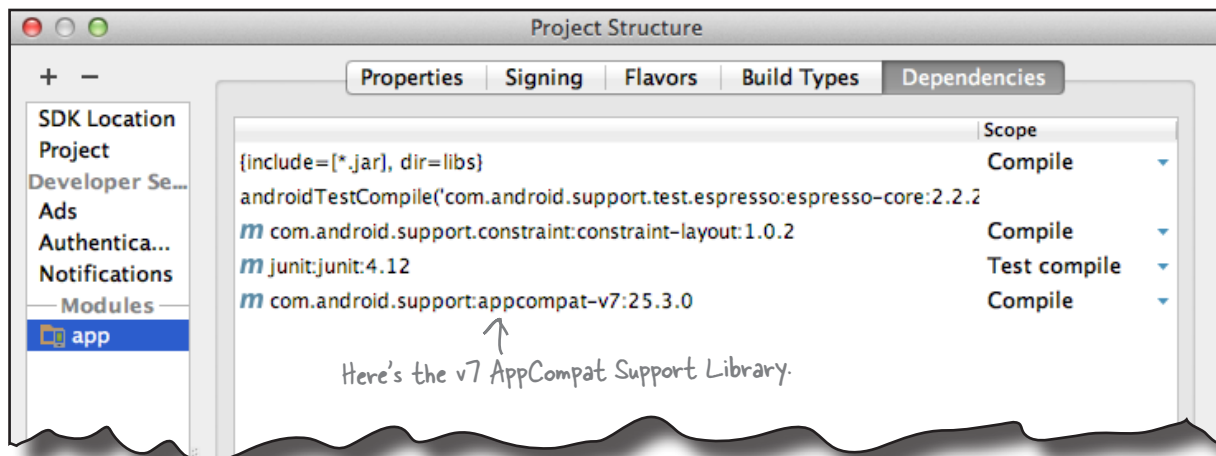


We'll use notifications from the AppCompat Support Library

We're going to create notifications using classes from the AppCompat Support Library so that our notifications will work consistently across a wide range of Android versions. While it's possible to create notifications using classes from the main release of Android, recent changes to these classes mean that the newest features won't be available on older versions.

Before we can use the notification classes from the Support Library, we need to add it to our project as a dependency. To do this, choose File→Project Structure, then click on the app module and choose Dependencies. Android Studio may have already added the AppCompat Support Library for you automatically. If so, you will see it listed as `appcompat-v7`. If it hasn't been added, you will need to add it yourself. Click on the "+" button at the bottom or right side of the screen, choose the Library Dependency option, select the `appcompat-v7` library, and then click on the OK button. Click on OK again to save your changes and close the Project Structure window.

Use notifications from the AppCompat Support Library to allow apps running on older versions of Android to include the newest features.



To get `DelayedMessageService` to display a notification, there are three things we need to do: create a notification builder, tell the notification to start `MainActivity` when it's clicked, and issue the notification. We'll build up the code over the next few pages, then show you the full code at the end.



First create a notification builder

The first thing we need to do is create a notification builder. This enables you to build a notification with specific content and features.

Each notification you create must include a small icon, a title, and some text as a bare minimum. Here's the code to do that:

The NotificationCompat class comes from the AppCompatActivity Support Library.

```
NotificationCompat.Builder builder = new NotificationCompat.Builder(this)
```

```
    .setSmallIcon(android.R.drawable.sym_def_app_icon)
    .setContentTitle(getString(R.string.question))
    .setContentText(text);
```

This displays a small icon, in this case a built-in Android icon.

Set the title and text.

To add more features to the notification, you simply add the appropriate method call to the builder. As an example, here's how you additionally specify that the notification should have a high priority, vibrate the device when it appears, and disappear when the user clicks on it:

```
NotificationCompat.Builder builder = new NotificationCompat.Builder(this)
    .setSmallIcon(android.R.drawable.sym_def_app_icon)
    .setContentTitle(getString(R.string.question))
    .setContentText(text)
    .setPriority(NotificationCompat.PRIORITY_HIGH)
    .setVibrate(new long[] {0, 1000})
    .setAutoCancel(true);
```

Make it a high priority and vibrate the device.

This makes the notification disappear when the user clicks on it.

Wait for 0 milliseconds before vibrating the device for 1,000 milliseconds.

Simply chain the method calls together to add more features to the notification.

These are just some of the properties that you can set. You can also set properties such as the notification's visibility to control whether it appears on the device lock screen, a number to display in case you want to send many notifications from the same app, and whether it should play a sound. You can find out more about these properties (and many others) here:

<https://developer.android.com/reference/android/support/v4/app/NotificationCompat.Builder.html>

Next, we'll add an action to the notification to tell it which activity to start when it's clicked.

You create a heads-up notification (one that appears in a small floating window) by setting its priority to high, and making it vibrate the device or play a sound.



Add an action to tell the notification which activity to start when clicked

When you create a notification, it's a good idea to add an action to it, specifying which activity in your app should be displayed when the user clicks on the notification. As an example, an email app might issue a notification when the user receives a new email, and display the contents of that email when the user clicks on it. In our particular case, we're going to start `MainActivity`.

You add an action by creating a **pending intent** to start an activity, which you then add to the notification. A pending intent is an intent that your app can pass to other applications. The application can then submit the intent on your app's behalf at a later time.

To create a pending intent, you first create an explicit intent directed to the activity you want to start when the notification is clicked. In our case, we want to start `MainActivity`, so we use:

This is a normal intent that starts `MainActivity`.

```
Intent actionIntent = new Intent(this, MainActivity.class);
```

We then use that intent to create a pending intent using the `PendingIntent.getActivity()` method.

```
PendingIntent actionPendingIntent = PendingIntent.getActivity(
```

This is a flag that's used if you ever need to retrieve the pending intent. We don't need to, so we're setting it to 0.

`this`, ← A context, in this case the current service.

`0`,

`actionIntent`, ← This is the intent we created above.

```
PendingIntent.FLAG_UPDATE_CURRENT);
```

↑ This means that if there's a matching pending intent, it will be updated.

The `getActivity()` method takes four parameters: a context (usually `this`), an `int` request code, the explicit intent we defined above, and a flag that specifies the pending intent's behavior. In the above code, we've used a flag of `FLAG_UPDATE_CURRENT`. This means that if a matching pending intent already exists, its extra data will be updated with the contents of the new intent. Other options are `FLAG_CANCEL_CURRENT` (cancel any existing matching pending intents before generating a new one), `FLAG_NO_CREATE` (don't create the pending intent if there's no matching existing one), and `FLAG_ONE_SHOT` (you can only use the pending intent once).

Once you've created the pending intent, you add it to the notification using the notification builder `setContentIntent()` method:

```
builder.setContentIntent(actionPendingIntent);
```

← This adds the pending intent to the notification.

This tells the notification to start the activity specified in the intent when the user clicks on the notification.



Issue the notification using the built-in notification service

Finally, you issue the notification using Android's notification service.

To do this, you first need to get a **NotificationManager**. You do this by calling the `getSystemService()` method, passing it a parameter of `NOTIFICATION_SERVICE`:

```
NotificationManager notificationManager =
    (NotificationManager) getSystemService(NOTIFICATION_SERVICE);
```

This gives you access to Android's notification service.

You then use the notification manager to issue the notification by calling its **notify()** method. This takes two parameters: a notification ID and a `Notification` object.

The notification ID is used to identify the notification. If you send another notification with the same ID, it will replace the current notification. This is useful if you want to update an existing notification with new information.

You create the `Notification` object by calling the notification builder's **build()** method. The notification it builds includes all the content and features you've specified via the notification builder.

Here's the code to issue the notification:

```
public static final int NOTIFICATION_ID = 5453;

...

NotificationManager notificationManager =
    (NotificationManager) getSystemService(NOTIFICATION_SERVICE);
notificationManager.notify(NOTIFICATION_ID, builder.build());
```

This is an ID we'll use for the notification. It's a random number we made up.

That's everything we need to create and issue notifications. We'll show you the full code for `DelayedMessageService` on the next page.

Use the notification service to display the notification we created.

The full code for DelayedMessageService.java

Here's the full code for *DelayedMessageService.java*. It now uses a notification to display a message to the user. Update your code to match ours:

```
package com.hfad.joke;

import android.app.IntentService;
import android.content.Intent;
import android.util.Log; ← Delete this line.
import android.support.v4.app.NotificationCompat;
import android.app.PendingIntent;
import android.app.NotificationManager;

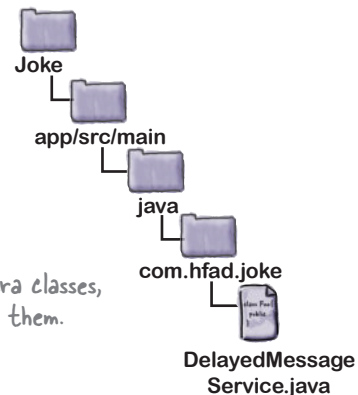
public class DelayedMessageService extends IntentService {

    public static final String EXTRA_MESSAGE = "message";
    public static final int NOTIFICATION_ID = 5453;

    public DelayedMessageService() {
        super("DelayedMessageService");
    }

    @Override
    protected void onHandleIntent(Intent intent) {
        synchronized (this) {
            try {
                wait(10000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        String text = intent.getStringExtra(EXTRA_MESSAGE);
        showText(text);
    }
}
```

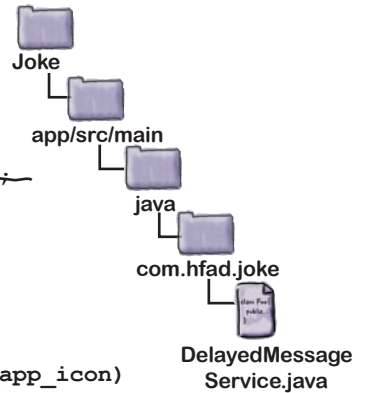
We're using these extra classes, so we need to import them.



This is used to identify the notification. It could be any number; we just decided on 5453.

The code continues → on the next page.

The DelayedMessageService.java code (continued)



```

private void showText(final String text) {
    Log.v("DelayedMessageService", "The message is: " + text);

    //Create a notification builder
    NotificationCompat.Builder builder =
        new NotificationCompat.Builder(this)
            .setSmallIcon(android.R.drawable.sym_def_app_icon)
            .setContentTitle(getString(R.string.question))
            .setContentText(text)
            .setPriority(NotificationCompat.PRIORITY_HIGH)
            .setVibrate(new long[] {0, 1000})
            .setAutoCancel(true);

    //Create an action
    Intent actionIntent = new Intent(this, MainActivity.class);
    PendingIntent actionPendingIntent = PendingIntent.getActivity(
        this,
        0,
        actionIntent,
        PendingIntent.FLAG_UPDATE_CURRENT);
    builder.setContentIntent(actionPendingIntent);

    //Issue the notification
    NotificationManager notificationManager =
        (NotificationManager) getSystemService(NOTIFICATION_SERVICE);
    notificationManager.notify(NOTIFICATION_ID, builder.build());
}
}

```

Use a notification builder to specify the content and features of the notification.

Use the intent to create a pending intent.

Create an intent.

Add the pending intent to the notification.

Display the notification using a notification manager.

That's all the code we need for our started service. Let's go through what happens when the code runs.

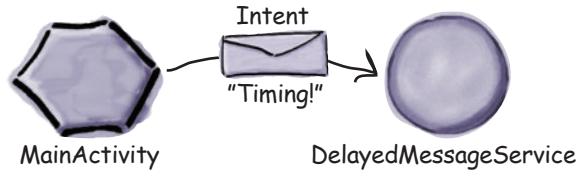


What happens when you run the code

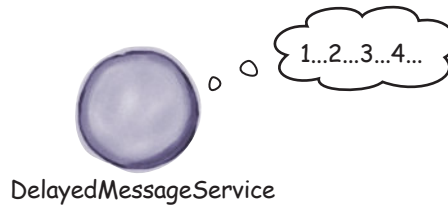
Before you try running the updated app, let's go through what happens when the code runs:

- 1 **MainActivity starts DelayedMessageService by calling startService() and passing it an intent.**

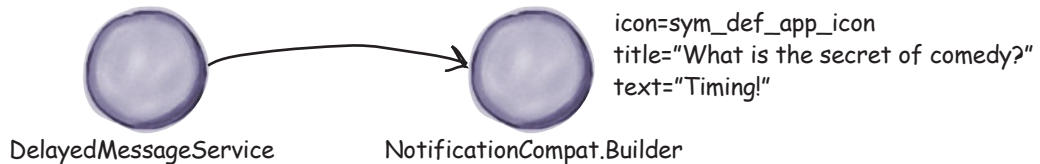
The intent contains the message MainActivity wants DelayedMessageService to display.



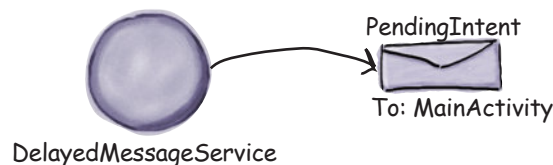
- 2 **DelayedMessageService waits for 10 seconds.**



- 3 **DelayedMessageService creates a notification builder and sets details of how the notification should be configured.**



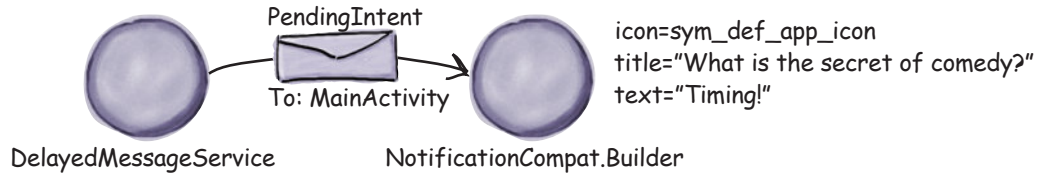
- 4 **DelayedMessageService creates an intent for MainActivity, which it uses to create a pending intent.**





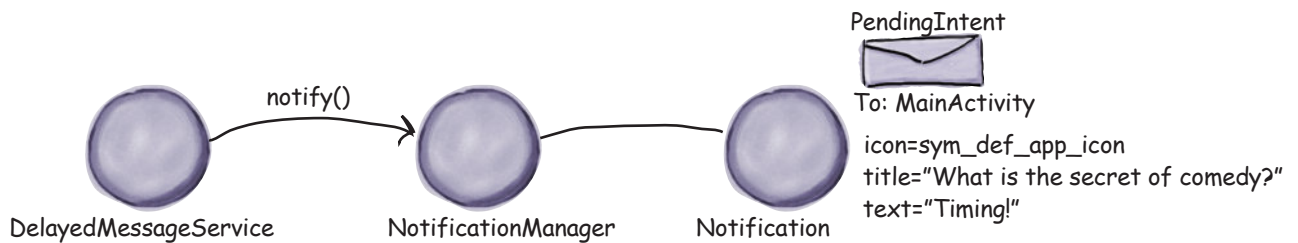
The story continues

- 5 **DelayedMessageService** adds the pending intent to the notification builder.

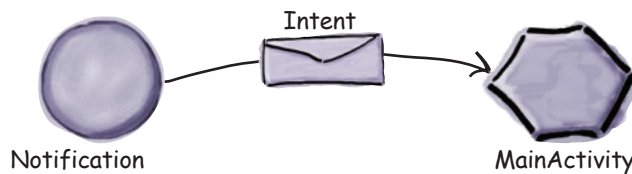


- 6 **DelayedMessageService** creates a **NotificationManager** object and calls its **notify()** method.

The notification service displays the notification built by the notification builder.



- 7 **When the user clicks on the notification, the notification uses its pending intent to start MainActivity.**



Now that we've gone through what the code does, let's take the app for a test drive.

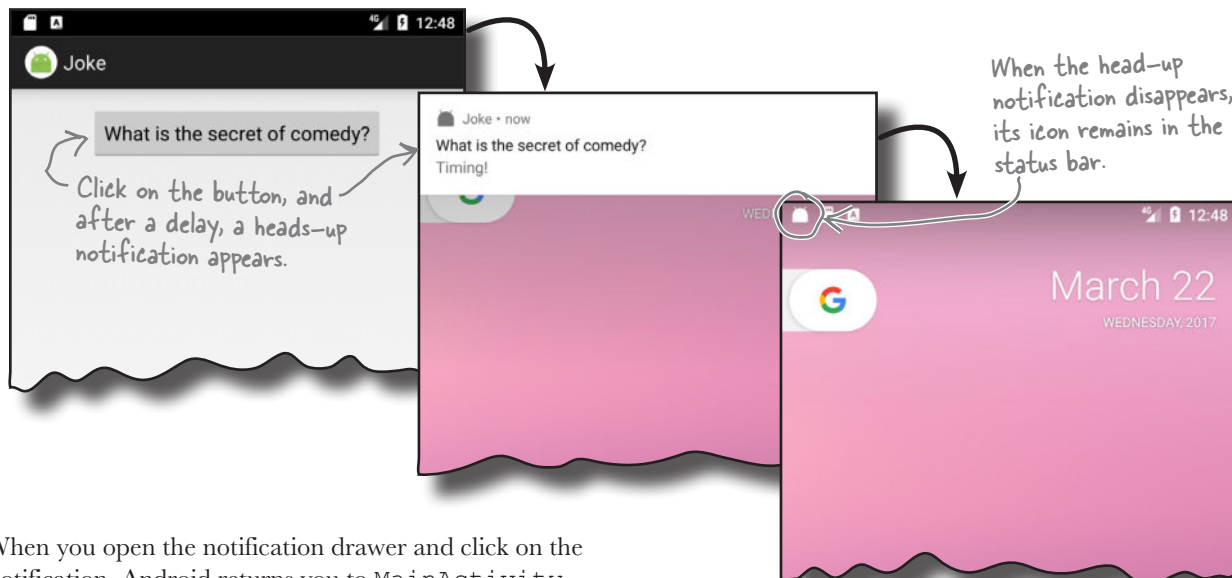


Test drive the app

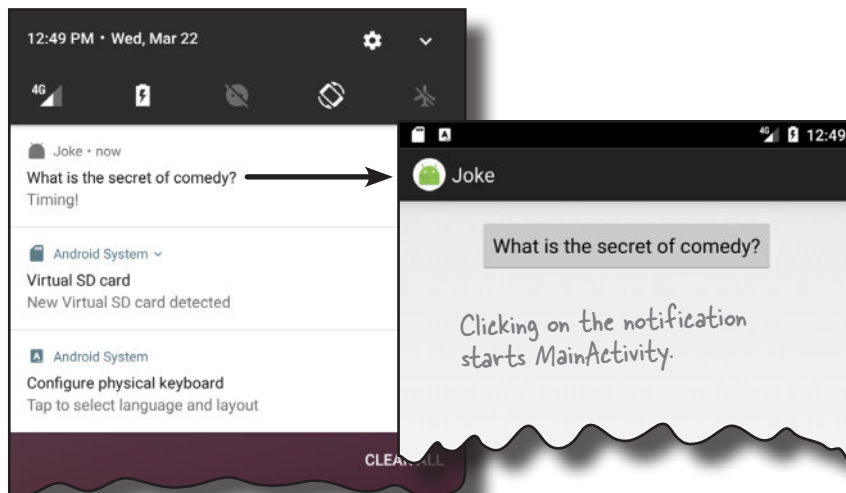


Log
Display notification

When you click on the button in MainActivity, a notification is displayed after 10 seconds. You'll receive the notification irrespective of which app you're in.



When you open the notification drawer and click on the notification, Android returns you to MainActivity.



You now know how to create a started service that displays a notification using the Android notification service. In the next chapter, we'll look at how you create a bound service.



Your Android Toolbox

You've got Chapter 18 under your belt and now you've added started services to your toolbox.



BULLET POINTS

- A **service** is an application component that can perform tasks in the background. It doesn't have a user interface.
- A **started service** can run in the background indefinitely, even when the activity that started it is destroyed. Once the operation is done, it stops itself.
- A **bound service** is bound to another component such as an activity. The activity can interact with it and get results.
- A **scheduled service** is one that's scheduled to run at a particular time.
- You can create a simple started service by extending the `IntentService` class, overriding its `onHandleIntent()` method and adding a public constructor.
- You declare services in `AndroidManifest.xml` using the `<service>` element.
- You start a started service using the `startService()` method.
- When a started service is created, its `onCreate()` method gets called, followed by `onStartCommand()`. If the service is an `IntentService`, `onHandleIntent()` is then called in a separate thread. When the service has finished running, `onDestroy()` gets called before the service is destroyed.
- The `IntentService` class inherits lifecycle methods from the `Service` class.
- You log messages using the `Android.util.Log` class. You can view these messages in the logcat in Android Studio.
- You create a notification using a **notification builder**. Each notification must include a small icon, a title, and some text as a bare minimum.
- A **heads-up** notification has its priority set to high, and vibrates the device or plays a sound when it's issued.
- You tell the notification which activity to start when it's clicked by creating a **pending intent** and adding it to the notification as an action.
- You issue the notification using a **notification manager**. You create a notification manager using Android's notification service.

You can download the full code for the chapter from <https://tinyurl.com/HeadFirstAndroid>.

19 bound services and permissions

Bound Together



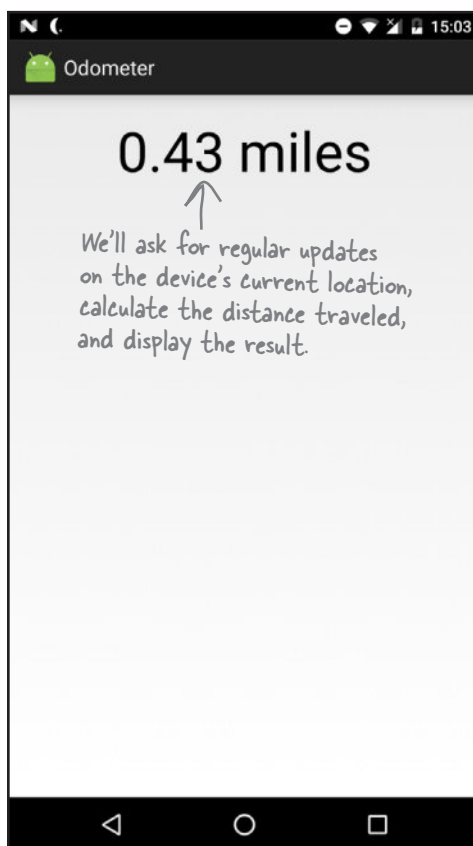
Started services are great for background operations, but what if you need a service that's more interactive? In this chapter you'll discover how to create a **bound service**, a type of service your activity can interact with. You'll see how to **bind** to the service when you need it, and how to **unbind** from it when you're done to save resources. You'll find out how to use **Android's Location Services** to get *location updates from your device GPS*. Finally, you'll discover how to use **Android's permission model**, including *handling runtime permission requests*.

Bound services are bound to other components

As you saw in Chapter 18, a started service is one that starts when it's passed an intent. It runs code in the background, and stops when the operation is complete. It continues running even if the component that starts it gets destroyed.

A **bound service** is one that's bound to another application component, such as an activity. Unlike a started service, the component can interact with the bound service and call its methods.

To see this in action, we're going to create a new odometer app that uses a bound service. We'll use Android's location service to track the distance traveled:



On the next page we'll look at the steps we'll go through to create the app.

Here's what we're going to do

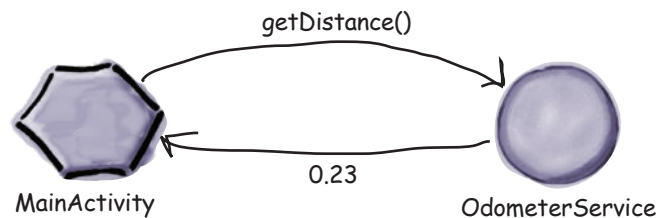
We're going to build the app in three main steps:

- 1 **Create a basic version of a bound service called `OdometerService`.**
We'll add a method to it, `getDistance()`, which will return a random number.

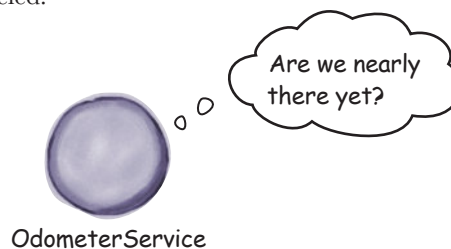


- 2 **Get an activity, `MainActivity`, to bind to `OdometerService` and call its `getDistance()` method.**

We'll call the method every second, and update a text view in `MainActivity` with the results.



- 3 **Update `OdometerService` to use Android's Location Services.**
The service will get updates on the user's current location, and use these to calculate the distance traveled.



Create a new Odometer project

We'll start by creating the project. Create a new Android project for an application named "Odometer" with a company domain of "hfad.com", making the package name `com.hfad.odometer`. The minimum SDK should be API 19 so that it will work with most devices. You'll need an empty activity named "MainActivity" and a layout named "activity_main" so that your code matches ours. **Make sure that you uncheck the Backwards Compatibility (AppCompat) option when you create the activity.**

Create a new service

You create a bound service by extending the **Service** class. This class is more general than the **IntentService** class we used in Chapter 18, which is used for started services. Extending **Service** gives you more flexibility, but requires more code.

We're going to add a new bound service to our project, so switch to the Project view of Android Studio's explorer, click on the `com.hfad.odometer` package in the `app/src/main/java` folder, go to File→New..., and select the Service option. When prompted, choose the option to create a new Service (not an Intent Service), and name the service "OdometerService". Uncheck the "Exported" option, as this only needs to be true if you want services outside this app to access the service. Make sure that the "checkedenabled" option is checked; if it isn't, the activity won't be able to run the app. Then replace the code in `OdometerService.java` with this (shown here in bold):

```
package com.hfad.odometer;

import android.app.Service;
import android.content.Intent;
import android.os.IBinder;

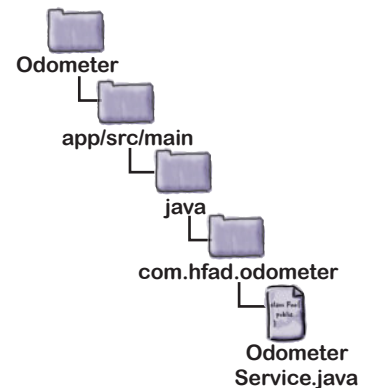
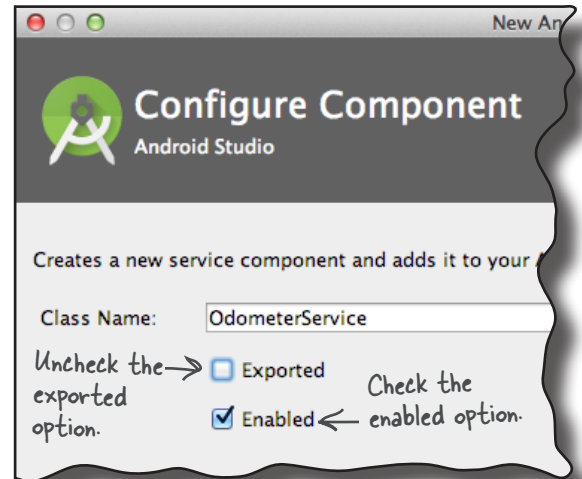
public class OdometerService extends Service {

    @Override
    public IBinder onBind(Intent intent) {
        //Code to bind the service
    }
}
```

Some versions of Android Studio may ask you what the source language should be. If prompted, select the option for Java.

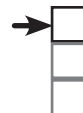
The class extends the Service class.

The `onBind()` method is called when a component wants to bind to the service.



The above code implements one method, `onBind()`, which gets called when a component, such as an activity, wants to bind to the service. It has one parameter, an `Intent`, and returns an `IBinder` object.

`IBinder` is an interface that's used to bind your service to the activity, and you need to provide an implementation of it in your service code. We'll look at how you do this next.



Implement a binder

You implement the `IBinder` by adding a new inner class to your service code that extends the `Binder` class (which implements the `IBinder` interface). This inner class needs to include a method that activities can use to get a reference to the bound service.

We're going to define a binder called `OdometerBinder` that `MainActivity` can use to get a reference to `OdometerService`. Here's the code we'll use to define it:

```
public class OdometerBinder extends Binder {
    OdometerService getOdometer() {
        return OdometerService.this;
    }
}
```

When you create a bound service, you need to provide a `Binder` implementation.
The activity will use this method to get a reference to the `OdometerService`.

We need to return an instance of the `OdometerBinder` in `OdometerService`'s `onBind()` method. To do this, we'll create a new private variable for the binder, instantiate it, and return it in the `onBind()` method. Update your `OdometerService.java` code to include our changes below:

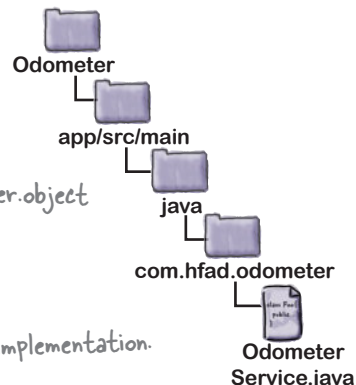
```
...
import android.os.Binder;

public class OdometerService extends Service {
    private final IBinder binder = new OdometerBinder();

    public class OdometerBinder extends Binder {
        OdometerService getOdometer() {
            return OdometerService.this;
        }
    }

    @Override
    public IBinder onBind(Intent intent) {
        return binder;
    }
}
```

We're using this extra class, so we need to import it.
We're using a private final variable for our `IBinder` object.
This is our `IBinder` implementation.
Return the `IBinder`.



We've now written all the service code we need to allow `MainActivity` to bind to `OdometerService`. Next, we'll add a new method to the service to make it return a random number.

Add a getDistance() method to the service



OdometerService
MainActivity
Location Services

We're going to add a method to OdometerService called `getDistance()`, which our activity will call. We'll get it to return a random number for now, and later on we'll update it to use Android's location services.

Here's the full code for `OdometerService.java` including this change; update your version to match ours:

```
package com.hfad.odometer;

import android.app.Service;
import android.content.Intent;
import android.os.IBinder;
import android.os.Binder;
import java.util.Random;

public class OdometerService extends Service {

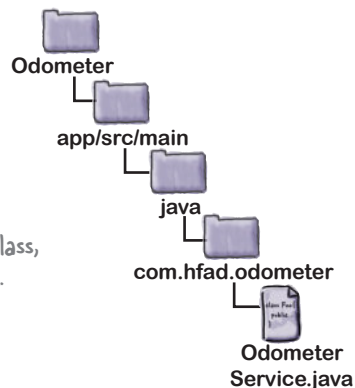
    private final IBinder binder = new OdometerBinder();
    private final Random random = new Random();

    public class OdometerBinder extends Binder {
        OdometerService getOdometer() {
            return OdometerService.this;
        }
    }

    @Override
    public IBinder onBind(Intent intent) {
        return binder;
    }

    public double getDistance() {
        return random.nextDouble();
    }
}
```

We're using this extra class, so we need to import it.

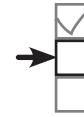


We'll use a `Random()` object to generate random numbers.

Add the `getDistance()` method.

Return a random double.

Next we'll update `MainActivity` so that it uses `OdometerService`.



Update MainActivity's layout

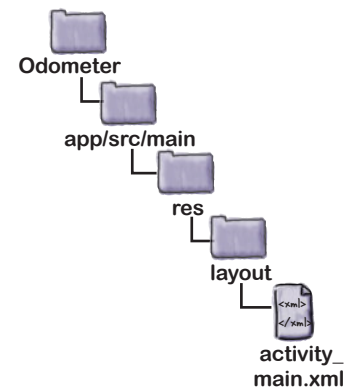
The next step in creating our app is to get MainActivity to bind to OdometerService and call its `getDistance()` method. We're going to start by adding a text view to MainActivity's layout. This will display the number returned by OdometerService's `getDistance()` method.

Update your version of `activity_main.xml` to reflect our changes:

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

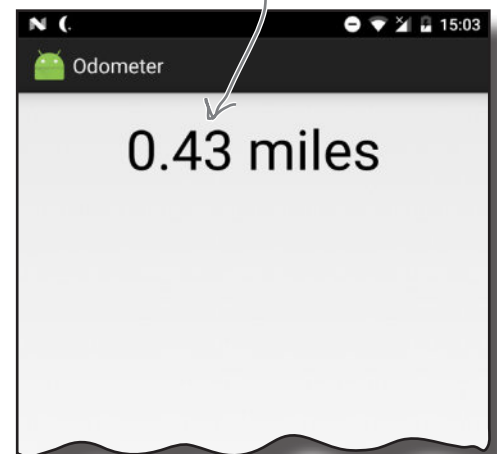
    <TextView
        android:id="@+id/distance"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:textSize="48sp"
        android:layout_gravity="center_horizontal"
        android:textAppearance="?android:attr/textAppearanceLarge" />

</LinearLayout>
```

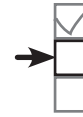


We'll use the `TextView` to display the number returned by the `OdometerService.getDistance()` method.

Now that we've added a text view to MainActivity's layout, we'll update its activity code. Let's go through the changes we need to make.



What MainActivity needs to do

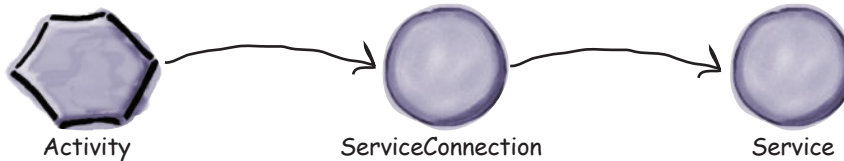


OdometerService
MainActivity
Location Services

To get an activity to connect to a bound service and call its methods, there are a few steps you need to perform:

1 Create a ServiceConnection.

This uses the service's `IBinder` object to form a connection with the service.



2 Bind the activity to the service.

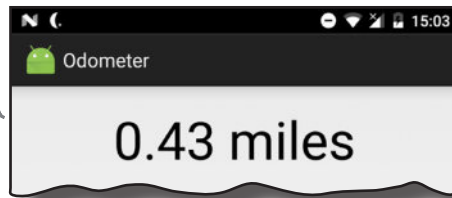
Once you've bound it to the service, you can call the service's methods directly.



3 Interact with the service.

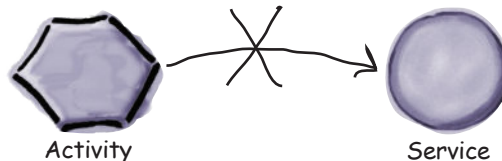
In our case, we'll use the service's `getDistance()` method to update the activity's text view.

When your activity is bound to the service, you can use it to update your activity.

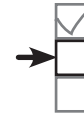


4 Unbind from the service when you've finished with it.

When the service is no longer used, Android destroys the service to free up resources.



We'll go through these steps with `MainActivity`, starting with creating the `ServiceConnection`.



Create a ServiceConnection

A **ServiceConnection** is an interface that enables your activity to bind to a service. It has two methods that you need to define: `onServiceConnected()` and `onServiceDisconnected()`. The `onServiceConnected()` method is called when a connection to the service is established, and `onServiceDisconnected()` is called when it disconnects.

We need to add a **ServiceConnection** to **MainActivity**. Here's what the basic code looks like; update your version of *MainActivity.java* to match ours:

```
package com.hfad.odometer;
```

```
import android.app.Activity;
import android.os.Bundle;
import android.content.ServiceConnection;
import android.os.IBinder;
import android.content.ComponentName;
```

```
public class MainActivity extends Activity {
```

```
    private ServiceConnection connection = new ServiceConnection() {
        @Override
        public void onServiceConnected(ComponentName componentName, IBinder binder) {
            //Code that runs when the service is connected
        }
        @Override
        public void onServiceDisconnected(ComponentName componentName) {
            //Code that runs when the service is disconnected
        }
    };
```

Create a Service Connection object.

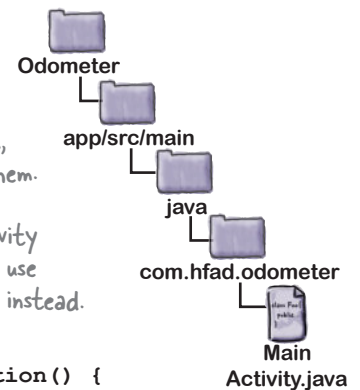
You need to define these methods.

We're using these classes, so we need to import them.

We're using an Activity here, but you could use AppCompatActivity instead.

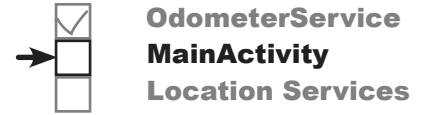
```
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }
}
```

Add MainActivity's onCreate() method.



We'll update the `onServiceConnected()` and `onServiceDisconnected()` methods on the next page.

The onServiceConnected() method



As we said on the previous page, the `onServiceConnected()` method is called when a connection is established between the activity and the service. It takes two parameters: a `ComponentName` object that describes the service that's been connected to, and an `IBinder` object that's defined by the service:

```
@Override
public void onServiceConnected(ComponentName componentName, IBinder binder) {
    //Code that runs when the service is connected
}
```

The `ComponentName` identifies the service. It includes the service package and class names.

This is an `IBinder` defined by the service. We added one to `OdometerService` earlier.

There are two things we need the `onServiceConnected()` method to do:

- ★ Use its `IBinder` parameter to get a reference to the service we're connected to, in this case `OdometerService`. We can do this by casting the `IBinder` to an `OdometerService.OdometerBinder` (as this is the type of `IBinder` we defined in `OdometerService`) and calling its `getOdometer()` method.
- ★ Record that the activity is bound to the service.

Here's the code to do these things (update your version of `MainActivity.java` to include our changes):

```
public class MainActivity extends Activity {

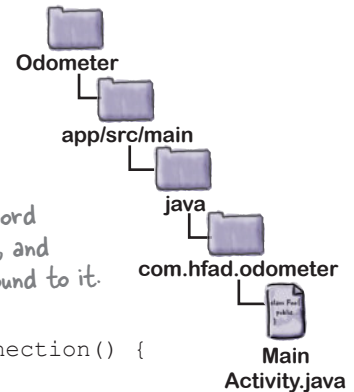
    private OdometerService odometer;
    private boolean bound = false;

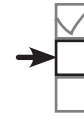
    private ServiceConnection connection = new ServiceConnection() {
        @Override
        public void onServiceConnected(ComponentName componentName, IBinder binder) {
            OdometerService.OdometerBinder odometerBinder =
                (OdometerService.OdometerBinder) binder;
            odometer = odometerBinder.getOdometer();
            bound = true;
        }
        ...
    };
    ...
}
```

Add these variables to record a reference to the service, and whether the activity is bound to it.

The activity is bound to the service, so set the bound variable to true.

Use the `IBinder` to get a reference to the service.





The onServiceDisconnected() method

The `onServiceDisconnected()` method is called when the service and the activity are disconnected. It takes one parameter, a `ComponentName` object that describes the service:

```
@Override
public void onServiceDisconnected(ComponentName componentName) {
    //Code that runs when the service is disconnected
}
```

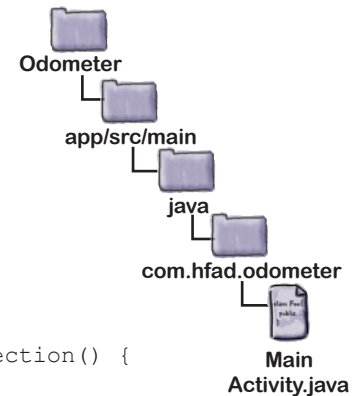
There's only one thing we need the `onServiceDisconnected()` method to do when it's called: record that the activity is no longer bound to the service. Here's the code to do that; update your version of *MainActivity.java* to match ours:

```
public class MainActivity extends Activity {

    private OdometerService odometer;
    private boolean bound = false;

    private ServiceConnection connection = new ServiceConnection() {
        @Override
        public void onServiceConnected(ComponentName componentName, IBinder binder) {
            OdometerService.OdometerBinder odometerBinder =
                (OdometerService.OdometerBinder) binder;
            odometer = odometerBinder.getOdometer();
            bound = true;
        }
        @Override
        public void onServiceDisconnected(ComponentName componentName) {
            bound = false;
        }
    };
    ...
}
```

↑
Set bound to false, as MainActivity is no longer bound to OdometerService.



Next we'll look at how you bind to and unbind from the service.

Use bindService() to bind the service



OdometerService
MainActivity
Location Services

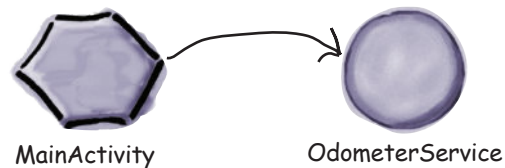
When you bind your activity to a service, you usually do it in one of two places:

- ★ In the activity's `onStart()` method when the activity becomes visible. This is appropriate if you only need to interact with the service when it's visible.
- ★ In the activity's `onCreate()` method when the activity gets created. Do this if you need to receive updates from the service even when the activity's stopped.

You don't usually bind to a service in the activity's `onResume()` method in order to keep the processing done in this method to a minimum.

In our case, we only need to display updates from `OdometerService` when `MainActivity` is visible, so we'll bind to the service in its `onStart()` method.

To bind to the service, you first create an explicit intent that's directed at the service you want to bind to. You then use the activity's `bindService()` method to bind to the service, passing it the intent, the `ServiceConnection` object defined by the service, and a flag to describe how you want to bind.



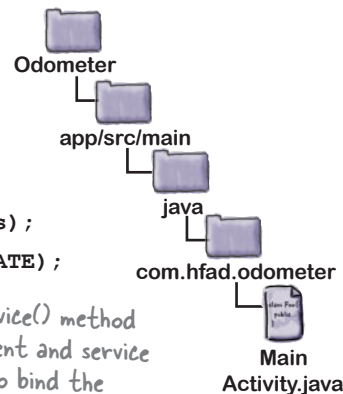
To see how you do this, here's the code you'd use to bind `MainActivity` to `OdometerService` (we'll add this code to `MainActivity.java` a few pages ahead):

```
@Override
protected void onStart() {
    super.onStart();
    Intent intent = new Intent(this, OdometerService.class);
    bindService(intent, connection, Context.BIND_AUTO_CREATE);
}
```

This is the `ServiceConnection` object.

This is an intent directed to the `OdometerService`.

The `bindService()` method uses the intent and service connection to bind the activity to the service.



In the above code, we've used the flag `Context.BIND_AUTO_CREATE` to tell Android to create the service if it doesn't already exist. There are other flags you can use instead; you can see all the available ones in the Android documentation here:

<https://developer.android.com/reference/android/content/Context.html>

Next, we'll look at how you unbind the activity from the service.

Use `unbindService()` to unbind from the service

When you unbind your activity from a service, you usually add the code to do so to your activity's `onStop()` or `onDestroy()` method. The method you use depends on where you put your `bindService()` code:

- ★ If you bound to the service in your activity's `onStart()` method, unbind from it in the `onStop()` method.
- ★ If you bound to the service in your activity's `onCreate()` method, unbind from it in the `onDestroy()` method.

In our case, we used `MainActivity`'s `onStart()` method to bind to `OdometerService`, so we'll unbind from it in the activity's `onStop()` method.

You unbind from a service using the `unbindService()` method. The method takes one parameter, the `ServiceConnection` object. Here's the code that we need to add to `MainActivity` (we'll add this code to `MainActivity.java` a few pages ahead):

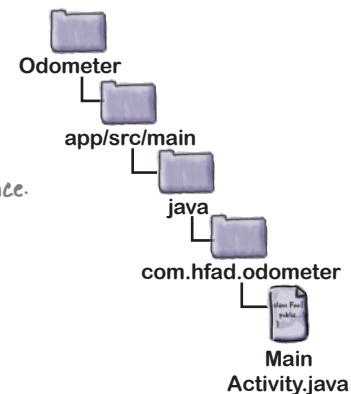
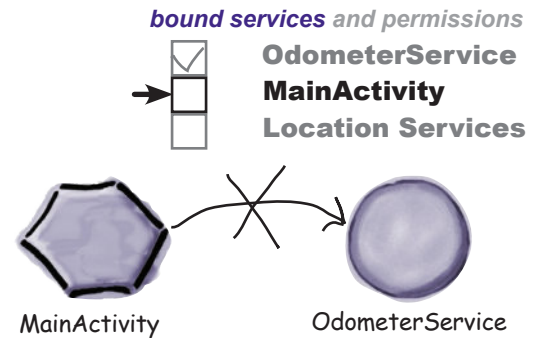
```
@Override
protected void onStop() {
    super.onStop();
    if (bound) {
        unbindService(connection);
        bound = false;
    }
}
```

This uses the ServiceConnection object to unbind from the service.

When we unbind, we'll set bound to false.

In the above code we're using the value of the `bound` variable to test whether or not we need to unbind from the service. If `bound` is `true`, this means `MainActivity` is bound to `OdometerService`. We need to unbind the service, and set the value of `bound` to `false`.

So far we have an activity that binds to the service when the activity starts, and unbinds from it when the activity stops. The final thing we need to do is get `MainActivity` to call `OdometerService`'s `getDistance()` method, and display its value.



Call OdometerService's getDistance() method



OdometerService
MainActivity
Location Services

Once your activity is bound to the service, you can call its methods. We're going to call the OdometerService's `getDistance()` method every second, and update MainActivity's text view with its value.

To do this, we're going to write a new method called `displayDistance()`. This will work in a similar way to the `runTimer()` code we used in Chapters 4 and 11.

Here's our `displayDistance()` method. We'll add it to *MainActivity.java* a couple of pages ahead:

```
private void displayDistance() {
    final TextView distanceView = (TextView) findViewById(R.id.distance);
    final Handler handler = new Handler();
    handler.post(new Runnable() {
        @Override
        public void run() {
            double distance = 0.0;
            if (bound && odometer != null) {
                distance = odometer.getDistance();
            }
            String distanceStr = String.format(Locale.getDefault(),
                "%1$, .2f miles", distance);
            distanceView.setText(distanceStr);
            handler.postDelayed(this, 1000);
        }
    });
}
```

← Get the TextView.

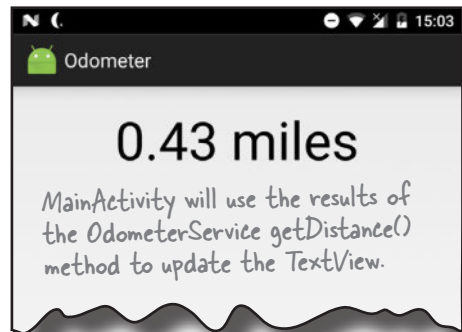
← Create a new Handler.

← Call the Handler's post() method, passing in a new Runnable.

If we've got a reference to the OdometerService and we're bound to it, call getDistance().

You could use a String resource for "miles", but we've hardcoded it here for simplicity.

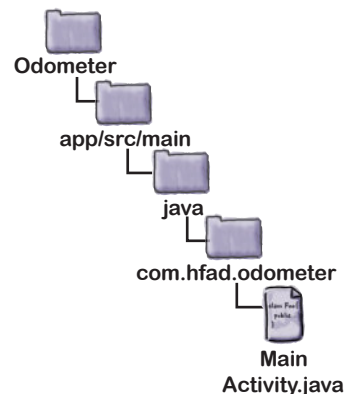
Post the code in the Runnable to be run again after a delay of 1 second. As this line of code is included in the Runnable run() method, it will run every second (with a slight lag).



We'll call the `displayDistance()` method in MainActivity's `onCreate()` method so that it starts running when the activity gets created (we'll add this code to *MainActivity.java* on the next page):

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    ...
    displayDistance();
}
```

← Call displayDistance() in MainActivity's onCreate() method to kick it off.



We'll show you the full code for MainActivity on the next page.

The full MainActivity.java code

Here's the complete code for *MainActivity.java*; make sure your version of the code matches ours:

```
package com.hfad.odometer;

import android.app.Activity;
import android.os.Bundle;
import android.content.ServiceConnection;
import android.os.IBinder;
import android.content.ComponentName;
import android.content.Context;
import android.content.Intent;
import android.os.Handler;
import android.widget.TextView;
import java.util.Locale;
```

```
public class MainActivity extends Activity {
```

```
    private OdometerService odometer;
    private boolean bound = false;
    private ServiceConnection connection = new ServiceConnection() {
        @Override
        public void onServiceConnected(ComponentName componentName, IBinder binder) {
            OdometerService.OdometerBinder odometerBinder =
                (OdometerService.OdometerBinder) binder;
            odometer = odometerBinder.getOdometer();
            bound = true;
        }
        @Override
        public void onServiceDisconnected(ComponentName componentName) {
            bound = false;
        }
    };

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        displayDistance();
    }
```

We're using these extra classes, so we need to import them.

Use this for the OdometerService.

Use this to store whether or not the activity's bound to the service.

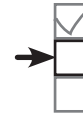
We need to define a ServiceConnection.

Get a reference to the OdometerService when the service is connected.

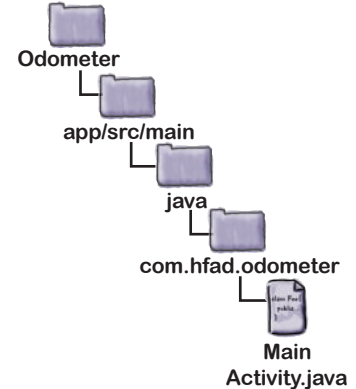
Call the displayDistance() method when the activity is created.

The code continues on the next page.

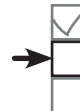
bound services and permissions



OdometerService
MainActivity
Location Services



The MainActivity.java code (continued)



OdometerService
MainActivity
Location Services

```

@Override
protected void onStart() {
    super.onStart();
    Intent intent = new Intent(this, OdometerService.class);
    bindService(intent, connection, Context.BIND_AUTO_CREATE);
}

@Override
protected void onStop() {
    super.onStop();
    if (bound) {
        unbindService(connection);
        bound = false;
    }
}

private void displayDistance() {
    final TextView distanceView = (TextView) findViewById(R.id.distance);
    final Handler handler = new Handler();
    handler.post(new Runnable() {
        @Override
        public void run() {
            double distance = 0.0;
            if (bound && odometer != null) {
                distance = odometer.getDistance();
            }
            String distanceStr = String.format(Locale.getDefault(),
                "%1$, .2f miles", distance);
            distanceView.setText(distanceStr);
            handler.postDelayed(this, 1000);
        }
    });
}

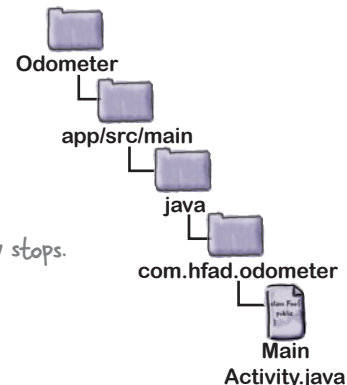
```

Bind the service when the activity starts.

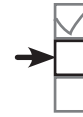
Unbind the service when the activity stops.
Display the value returned by the service's `getDistance()` method.

Call OdometerService's `getDistance()` method.

Update the TextView's value every second.



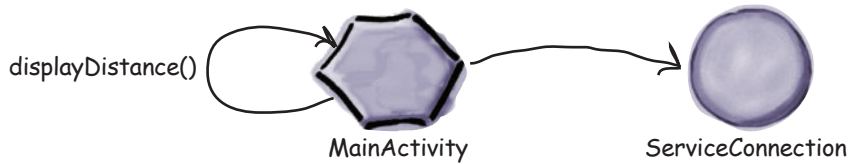
That's all the code you need to get MainActivity to use OdometerService. Let's go through what happens when you run the code.



What happens when you run the code

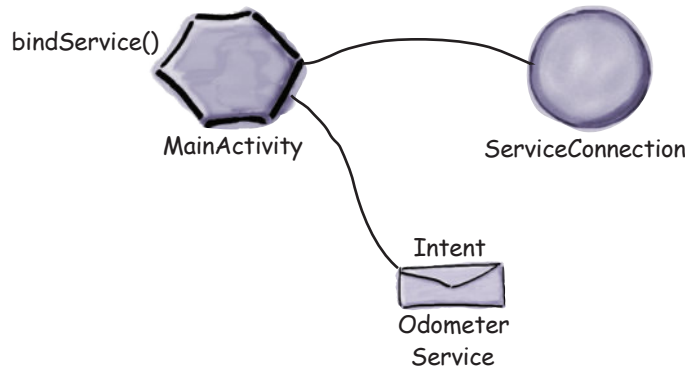
Before you see the app up and running, let's walk through what the code does.

- 1 When MainActivity is created, it creates a ServiceConnection object and calls the displayDistance() method.

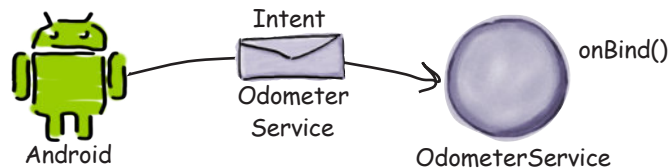


- 2 MainActivity calls bindService() in its onStart() method.

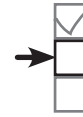
The bindService() method includes an intent meant for OdometerService, and a reference to the ServiceConnection.



- 3 Android creates an instance of the OdometerService, and passes it the intent by calling its onBind() method.

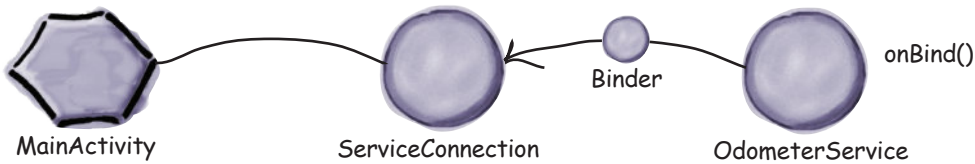


The story continues

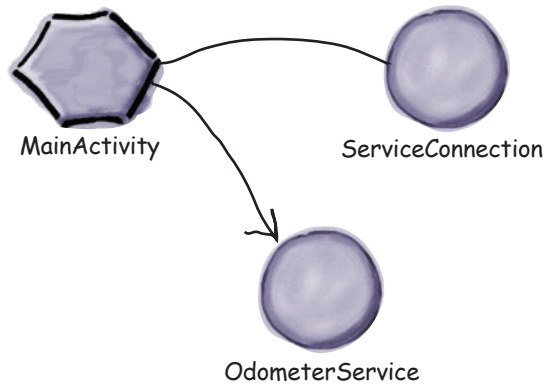


OdometerService
MainActivity
Location Services

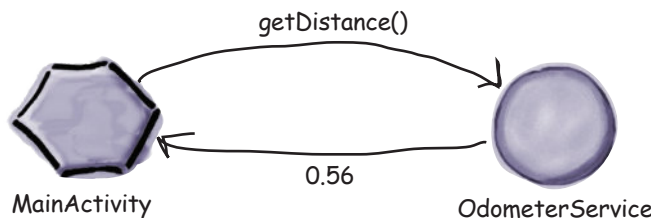
- 4 **OdometerService's onBind() method returns a Binder.**
The Binder is passed to MainActivity's ServiceConnection.



- 5 **The ServiceConnection uses the Binder to give MainActivity a reference to OdometerService.**

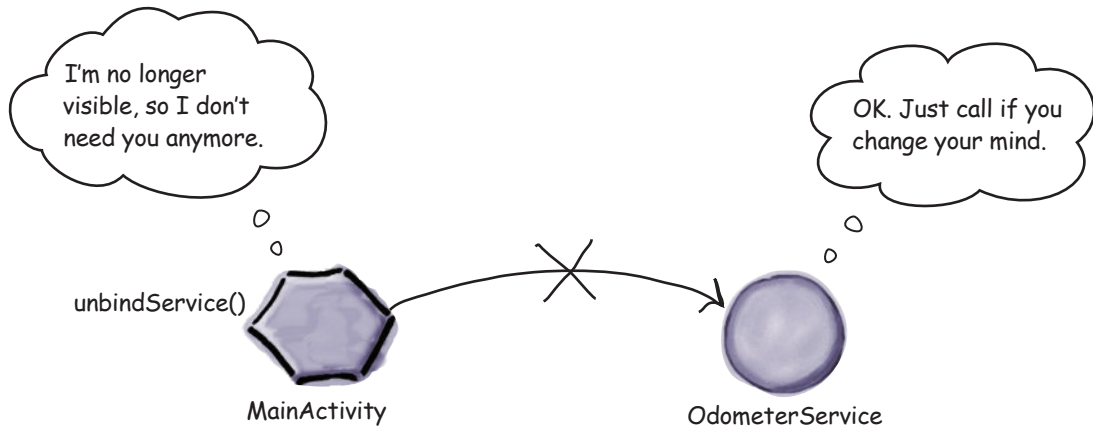


- 6 **MainActivity's displayDistance() method calls OdometerService's getDistance() method every second.**
OdometerService returns a random number to MainActivity, in this case 0.56.



The story continues

- 7 When MainActivity stops, it disconnects from OdometerService by calling `unbindService()`.



- 8 OdometerService is destroyed when MainActivity is no longer bound to it.



Now that you understand what happens when the code runs, let's take the app for a test drive.



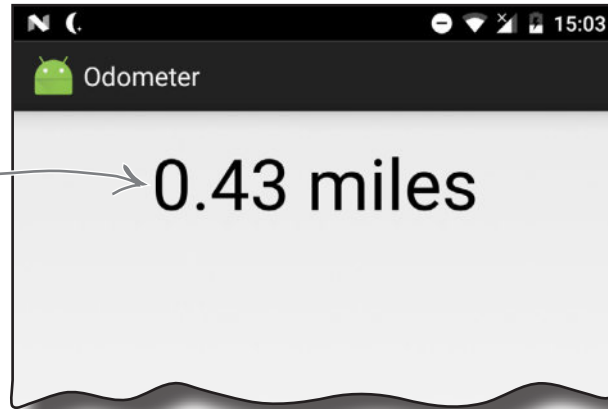
Test drive the app

When we run the app, a random number is displayed in `MainActivity`. This number changes every second.



OdometerService
MainActivity
Location Services

This is a random number generated by `OdometerService`.



We now have a working service that `MainActivity` can bind to. We still need to change the service so that the `getDistance()` method returns the distance traveled instead of a random number. Before we do that, however, we're going to take a closer look at how bound services work behind the scenes.

there are no Dumb Questions

Q: What's the difference between a started service and a bound service again?

A: A started service is created when an activity (or some other component) calls `startService()`. It runs code in the background, and when it finishes running, the service is destroyed.

A bound service is created when the activity calls `bindService()`. The activity can interact with the service by calling its methods. The service is destroyed when no components are bound to it.

Q: Can a service be both started and bound?

A: Yes. In such cases, the service is created when `startService()` or `bindService()` is called. It's only destroyed when the code it was asked to run in the background has stopped running, and there are no components bound to it.

Creating this kind of started-and-bound service is more complicated than creating a service that's only started or bound. You can find out how to do it in the Android documentation: <https://developer.android.com/guide/components/services.html>.

Q: What's the difference between a `Binder` and an `IBinder`?

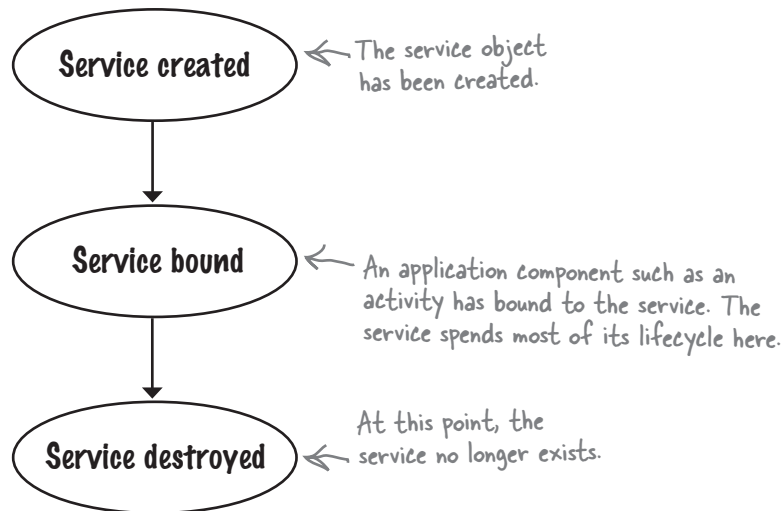
A: An `IBinder` is an *interface*. A `Binder` is a class that *implements* the `IBinder` interface.

Q: Can other apps use a service I create?

A: Yes, but only if you set its `exported` attribute to `true` in `AndroidManifest.xml`.

The states of a bound service

When an application component (such as an activity) binds to a service, the service moves between three states: being created, being bound, and being destroyed. A bound service spends most of its time in a bound state.



Just like a started service, when a bound service is created, its `onCreate()` method gets called. As before, you override this method if you want to perform any tasks needed to set up the service.

The `onBind()` method runs when a component binds to the service. You override this method to return an `IBinder` object to the component, which it uses to get a reference to the service.

When all components have unbound from the service, its `onUnbind()` method is called.

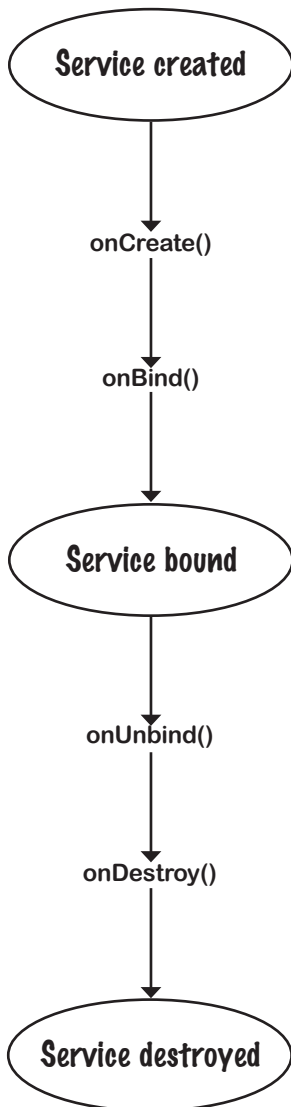
Finally, the `onDestroy()` method is called when no components are bound to the service and it's about to be destroyed. As before, you override this method to perform any final cleanup tasks and free up resources.

We'll take a closer look at how these methods fit into the service states on the next page.

A bound service is destroyed when no components are bound to it.

The bound service lifecycle: from create to destroy

Here's a more detailed overview of the bound service lifecycle from birth to death.



- 1** The component calls `bindService()` and the service gets created.
- 2** The `onCreate()` method runs immediately after the service is created.
The `onCreate()` method is where any service initialization code should go, as this method always gets called after the service has launched but before any components have bound to it.
- 3** The `onBind()` method runs when the component binds to the service.
You override this method to return an `IBinder` object, which the component can use to get a reference to the service and call its methods.
- 4** The service is bound for most of its life.
- 5** The `onUnbind()` method runs when all components have unbound from the service.
- 6** The `onDestroy()` method is called when no components are bound to the service and it's about to be destroyed.
You override this method if you want to perform any final cleanup tasks such as freeing up resources.
- 7** After the `onDestroy()` method has run, the service is destroyed.
The service ceases to exist.

Now that you have a better understanding of how bound services work, let's change our Odometer app so that it displays the actual distance traveled by the user.



We'll use Android's Location Services to return the distance traveled

We need to get our `OdometerService` to return the distance traveled in its `getDistance()` method. To do this, we'll use Android's Location Services. These allow you to get the user's current location, request periodic updates, and ask for an intent to be fired when the user comes within a certain radius of a particular location.

In our case, we're going to use the Location Services to get periodic updates on the user's current location. We'll use these to calculate the distance the user has traveled.

To do this, we'll perform the following steps:

- 1 **Declare we need permission to use the Location Services.**
Our app can only use the Location Services if the user grants our app permission to do so.
- 2 **Set up a location listener when the service is created.**
This will be used to listen for updates from the Location Services.
- 3 **Request location updates.**
We'll create a location manager, and use it to request updates on the user's current location.
- 4 **Calculate the distance traveled.**
We'll keep a running total of the distance traveled by the user, and return this distance in the `OdometerService`'s `getDistance()` method.
- 5 **Remove location updates just before the service is destroyed.**
This will free up system resources.

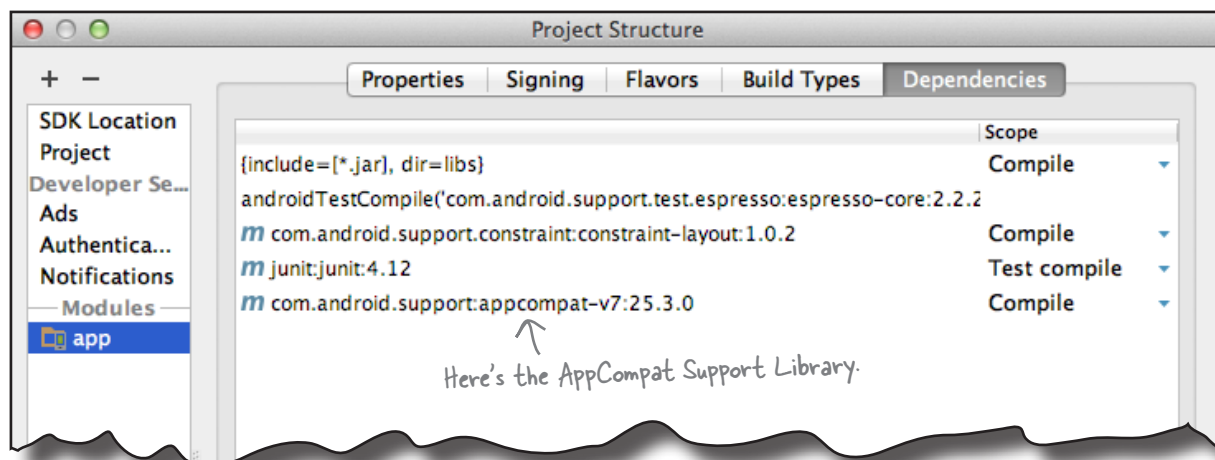
Before we start, we'll add the `AppCompatActivity` Support Library to our project, as we'll need to use it in our code.

Add the AppCompat Support Library



OdometerService
MainActivity
Location Services

To get our location code working properly, there are a couple of classes we need to use from the AppCompat Support Library, so we'll add it to our project as a dependency. You do this in the same way that you did in earlier chapters. Choose File→Project Structure, then click on the app module and choose Dependencies. You'll be presented with the following screen:



Android Studio may have already added the AppCompat Support Library automatically. If so, you will see it listed as appcompat-v7, as shown above.

If the AppCompat Library isn't listed, you will need to add it yourself. To do this, click on the "+" button at the bottom or right side of the screen, choose the Library Dependency option, select the appcompat-v7 library, then click on the OK button. Click on OK again to save your changes and close the Project Structure window.

Next, we'll look at how to declare we need permission to use Android's Location Services.



Declare the permissions you need

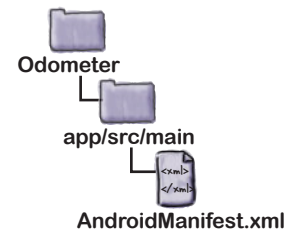
Android allows you to perform many actions by default, but there are some that the user needs to give permission for in order for them to work. This can be because they use the user's private information, or could affect stored data or the way in which other apps function. Location Services is one of those things that the user needs to grant your app permission to use.

You declare the permissions your app requires in *AndroidManifest.xml* using the `<uses-permission>` element, which you add to the root `<manifest>` element. In our case, we need to access the user's precise location in order to display the distance traveled, so we need to declare the `ACCESS_FINE_LOCATION` permission. To do this, you add the following declaration to *AndroidManifest.xml* (update your version of the file to reflect this change):

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.hfad.odometer">
    <uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />
    <application
        ...
    </application>
</manifest>
```

Declare we need a permission.

We need to know the user's precise location.



How the app uses the above declaration depends on your app's target SDK (usually the most recent version of Android) and the API level of the user's device:

- ★ If your target SDK is API level 23 or above, *and* the user's device is running 23 or above, **the app requests permission at runtime**. The user can deny or revoke permission, so whenever your code wants to use the thing that requires permission, it needs to check that permission is still granted. You'll find out how to do this later in the chapter.
- ★ If your target SDK is API level 22 or below, *or* the user's device is running 22 or below, **the app requests permission when it's installed**. If the user denies permission, the app isn't installed. Once granted, permission can't be revoked except by uninstalling the app.

Now that we've declared that our app needs to know the user's location, let's get to work on `OdometerService`.

Add a location listener to OdometerService



OdometerService
MainActivity
Location Services

You create a location listener by implementing the **LocationListener** interface. It has four methods that you need to define: `onLocationChanged()`, `onProviderEnabled()`, `onProviderDisabled()`, and `onStatusChanged()`. `onLocationChanged()` gets called when the user's location has changed. We'll use this method later in the chapter to track the distance the user has traveled. The `onProviderEnabled()`, `onProviderDisabled()`, and `onStatusChanged()` methods are called when the location provider is enabled, when it is disabled, or when its status has changed, respectively.

We'll look at location providers on the next page.

We need to set up a location listener when `OdometerService` is first created, so we'll implement the interface in `OdometerService`'s `onCreate()` method. Update your version of `OdometerService.java` to include our changes below:

```
...
import android.os.Bundle;
import android.location.LocationListener;
import android.location.Location;

public class OdometerService extends Service {
    ...
    private LocationListener listener;

    @Override
    public void onCreate() {
        super.onCreate();
        listener = new LocationListener() {
            @Override
            public void onLocationChanged(Location location) {
                //Code to keep track of the distance
            }

            @Override
            public void onProviderDisabled(String arg0) {}

            @Override
            public void onProviderEnabled(String arg0) {}

            @Override
            public void onStatusChanged(String arg0, int arg1, Bundle bundle) {}
        };
    }
    ...
}
```

We're using these extra classes, so we need to import them.

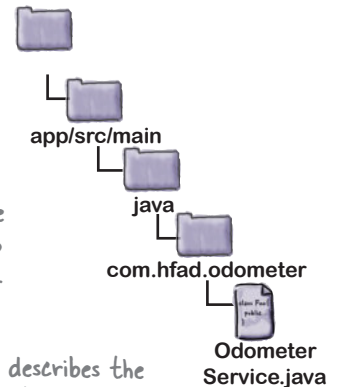
We're using a private variable for the `LocationListener` so other methods can access it.

Set up the `LocationListener`.

The `Location` parameter describes the current location. We'll use this later.

We'll complete this code later in the chapter.

We won't use any of these methods in our `OdometerService` code, but we still need to declare them.





OdometerService
MainActivity
Location Services

We need a location manager and location provider

To get location updates, we need to do three things: create a location manager to get access to Android's Location Services, specify a location provider, and request that the location provider sends regular updates on the user's current location to the location listener we added on the previous page. We'll start by getting a location manager.

Create a location manager

You create a location manager in a similar way to how you created a notification manager in Chapter 18: using the `getSystemService()` method. Here's the code to create a location manager that you can use to access Android's Location Services (we'll add the code to `OdometerService`'s `onCreate()` method later on):

```
LocationManager locationManager = (LocationManager) getSystemService(Context.LOCATION_SERVICE);
```

This is how you access the Android location service.

We used the `getSystemService()` method in Chapter 18 to get access to Android's notification service.

Specify the location provider

Next, we need to specify the location provider, which is used to determine the user's location. There are two main options: GPS or network. The GPS option uses the device's GPS sensor to establish the user's location, whereas the network option looks at Wi-Fi, Bluetooth, or mobile networks.

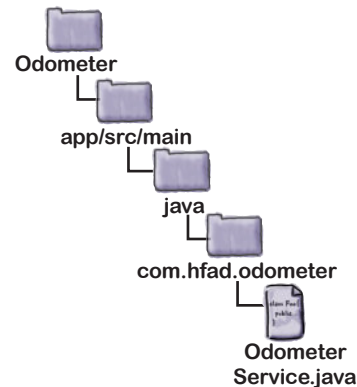
Not all devices have both types of location provider, so you can use the location manager's `getBestProvider()` method to get the most accurate location provider on the device. This method takes two parameters: a `Criteria` object you can use to specify criteria such as power requirements, and a flag to indicate whether it should currently be enabled on the device.

We want to use the location provider on the device with the greatest accuracy, so we'll use the following (we'll add it to `OdometerService` later):

```
String provider = locationManager.getBestProvider(new Criteria(), true);
```

Next we'll get the location provider to send location updates to the location listener.

This gets the most accurate location provider that's available on the device.



Request location updates...



OdometerService
MainActivity
Location Services

You get the location provider to send updates to the location listener using the location manager's `requestLocationUpdates()` method. It takes four parameters: the location provider, the minimum time interval between updates in milliseconds, the minimum distance between location updates in meters, and the location listener you want to receive the updates. As an example, here's how you'd request location updates from the location provider every second when the device has moved more than a meter:

```
locManager.requestLocationUpdates(provider, 1000, 1, listener);
```

The location provider → `provider`
 The time in milliseconds → `1000`
 The distance in meters → `1`
 The LocationListener you want to receive updates → `listener`

...but first check that your app has permission

If your app's target SDK is API level 23 or above, you need to check at runtime whether the user has granted you permission to get their current location. (As we said earlier in the chapter, if your target SDK is API level 23 or above, and the user's device is running one of these versions, the user may have installed the app without granting Location Services permission. You therefore have to check whether permission's been granted before running any code that requires Location Services, or your code won't compile.)

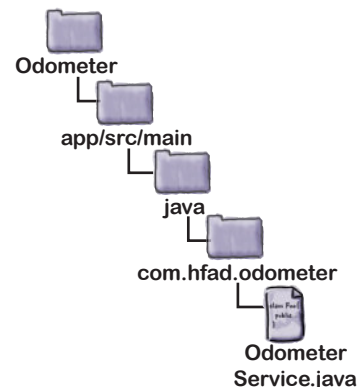
You check whether permission's been granted using the `ContextCompat.checkSelfPermission()` method. `ContextCompat` is a class from the AppCompat Support Library that provides backward compatibility with older versions of Android. Its `checkSelfPermission()` method takes two parameters: the current `Context` (usually `this`) and the permission you want to check. It returns a value of `PackageManager.PERMISSION_GRANTED` if permission has been granted.

In our case, we want to check whether the app's been granted `ACCESS_FINE_LOCATION` permission. Here's the code to do that:

```
if (ContextCompat.checkSelfPermission(this,
    android.Manifest.permission.ACCESS_FINE_LOCATION)
    == PackageManager.PERMISSION_GRANTED) {
    locManager.requestLocationUpdates(provider, 1000, 1, listener);
}
```

...before requesting location updates.

You can check your app's target SDK version by choosing `File -> Project Structure`, clicking on the app option, and then choosing `Flavors`.



Check if the `ACCESS_FINE_LOCATION` permission has been granted...

On the next page we'll show you all the code you need to add to `OdometerService.java` to request location updates.



Here's the updated OdometerService code

Here's the full code to request location updates (update your version of *OdometerService.java* to include our changes):

```

...
import android.content.Context;
import android.location.LocationManager;
import android.location.Criteria;
import android.support.v4.content.ContextCompat;
import android.content.pm.PackageManager;

public class OdometerService extends Service {
    ...
    private LocationManager locManager;
    public static final String PERMISSION_STRING
        = android.Manifest.permission.ACCESS_FINE_LOCATION;
    ...

    @Override
    public void onCreate() {
        super.onCreate();
        listener = new LocationListener() {
            ...
        };
        locManager = (LocationManager) getSystemService (Context.LOCATION_SERVICE);
        if (ContextCompat.checkSelfPermission(this, PERMISSION_STRING)
            == PackageManager.PERMISSION_GRANTED) {
            String provider = locManager.getBestProvider(new Criteria(), true);
            if (provider != null) {
                locManager.requestLocationUpdates(provider, 1000, 1, listener);
            }
        }
    }
    ...
}

```

We're using these extra classes, so import them.

We're using a private variable for the LocationManager so we can access it from other methods.

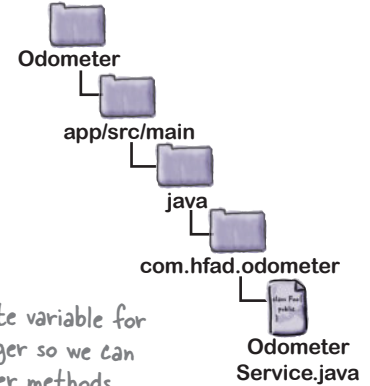
We're adding the permission String as a constant.

Get the LocationManager.

Check whether we have permission.

Get the most accurate location provider.

Request updates from the location provider.



Next we'll get our location listener to deal with the location updates.

Calculate the distance traveled



OdometerService
MainActivity
Location Services

So far, we've requested that the location listener be notified when the user's current location changes. When this happens, the listener's `onLocationChanged()` method gets called.

This method has one parameter, a `Location` object representing the user's current location. We can use this object to calculate the distance traveled by keeping a running total of the distance between the user's current location and their last.

You find the distance in meters between two locations using the `Location` object's `distanceTo()` method. As an example, here's how you'd find the distance between two locations called `location` and `lastLocation`:

```
double distanceInMeters = location.distanceTo(lastLocation);
```

This gets the distance between `location` and `lastLocation`.

Here's the code we need to add to `OdometerService` to record the distance traveled by the user (update your version of `OdometerService.java` to match ours):

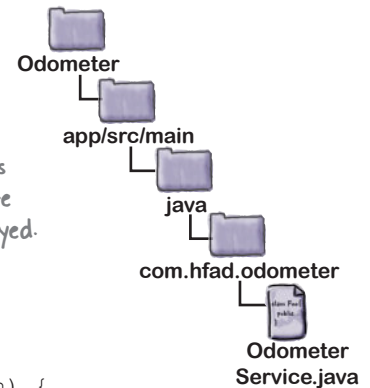
```
public class OdometerService extends Service {
    private static double distanceInMeters;
    private static Location lastLocation = null;
    ...

    @Override
    public void onCreate() {
        super.onCreate();
        listener = new LocationListener() {
            @Override
            public void onLocationChanged(Location location) {
                if (lastLocation == null) {
                    lastLocation = location;
                }
                distanceInMeters += location.distanceTo(lastLocation);
                lastLocation = location;
            }
        };
        ...
    }
    ...
}
```

We're using static variables to store the distance traveled and the user's last location so that their values are retained when the service is destroyed.

Set the user's starting location.

Update the distance traveled and the user's last location.



We'll use this code to return the distance traveled to `MainActivity`.



Return the miles traveled

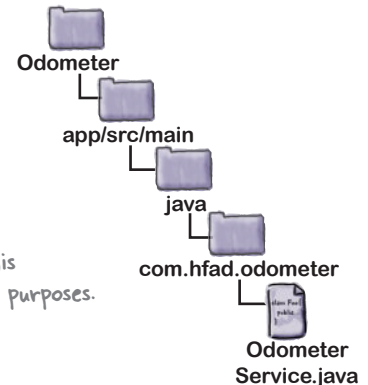
To tell MainActivity how far the user has traveled, we need to update OdometerService's `getDistance()` method. It currently returns a random number, so we'll change it to convert the value of the `distanceInMeters` variable to miles and return that value. Here's the new version of the `getDistance()` method; update your version of it to match ours:

```
public double getDistance() {
    return random.nextDouble();
    return this.distanceInMeters / 1609.344;
}
```

Delete this line.

This converts the distance traveled in meters into miles. We could make this calculation more precise if we wanted to, but it's accurate enough for our purposes.

Finally, we'll stop the listener getting location updates when the service is about to be destroyed.



Stop the listener getting location updates

We're going to stop the location listener getting updates using the `OdometerService`'s `onDestroy()` method, as this gets called just before the service is destroyed.

You stop the updates by calling the location manager's `removeUpdates()` method. It takes one parameter, the listener you wish to stop receiving the updates:

```
locManager.removeUpdates(listener);
```

This stops the location listener getting updates.

If your app's target SDK is API level 23 or above, you need to check whether the user has granted `ACCESS_FINE_LOCATION` permission before calling the `removeUpdates()` method. This is because you can only use this method if the user's granted this permission, and Android Studio will complain if you don't check first. You check whether permission's been granted in the same way we did earlier, by checking the return value of the `ContextCompat.checkSelfPermission()` method:

```
if (ContextCompat.checkSelfPermission(this, PERMISSION_STRING)
    == PackageManager.PERMISSION_GRANTED) {
    locManager.removeUpdates(listener);
}
```

We can only remove the updates if we have permission.

Over the next few pages we'll show you the full code for `OdometerService`, including the new `onDestroy()` method.

The full OdometerService.java code



OdometerService
MainActivity
Location Services

We've now done everything we need to get OdometerService to return the distance traveled. Update your version of *OdometerService.java* to match ours.

```
package com.hfad.odometer;

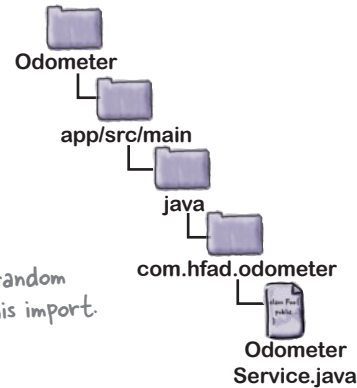
import android.app.Service;
import android.content.Context;
import android.content.Intent;
import android.os.Bundle;
import android.os.IBinder;
import android.os.Binder;
import java.util.Random;
import android.location.LocationListener;
import android.location.Location;
import android.location.LocationManager;
import android.location.Criteria;
import android.support.v4.content.ContextCompat;
import android.content.pm.PackageManager;
```

```
public class OdometerService extends Service {
```

```
    private final IBinder binder = new OdometerBinder();
    private final Random random = new Random();
    private LocationListener listener;
    private LocationManager locationManager;
    private static double distanceInMeters;
    private static Location lastLocation = null;
    public static final String PERMISSION_STRING
        = android.Manifest.permission.ACCESS_FINE_LOCATION;
```

```
    public class OdometerBinder extends Binder {
        OdometerService getOdometer() {
            return OdometerService.this;
        }
    }
}
```

We're no longer returning a random number, so you can delete this import.



Delete the Random() object, as we're no longer using it.

The code continues on the next page.

The OdometerService.java code (continued)

bound services and permissions



OdometerService
MainActivity
Location Services

```
@Override
public void onCreate() {
    super.onCreate();
    listener = new LocationListener() {
        @Override
        public void onLocationChanged(Location location) {
            if (lastLocation == null) {
                lastLocation = location;
            }
            distanceInMeters += location.distanceTo(lastLocation);
            lastLocation = location;
        }
    };

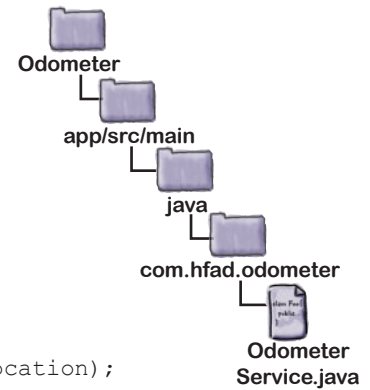
    @Override
    public void onProviderDisabled(String arg0) {
    }

    @Override
    public void onProviderEnabled(String arg0) {
    }

    @Override
    public void onStatusChanged(String arg0, int arg1, Bundle bundle) {
    }
};

locManager = (LocationManager) getSystemService (Context.LOCATION_SERVICE);
if (ContextCompat.checkSelfPermission(this, PERMISSION_STRING)
    == PackageManager.PERMISSION_GRANTED) {
    String provider = locManager.getBestProvider(new Criteria(), true);
    if (provider != null) {
        locManager.requestLocationUpdates(provider, 1000, 1, listener);
    }
}
}
```

*None of the code on
this page has changed.*



*The code continues →
on the next page.*

The OdometerService.java code (continued)



OdometerService
MainActivity
Location Services

```
@Override
public IBinder onBind(Intent intent) {
    return binder;
}
```

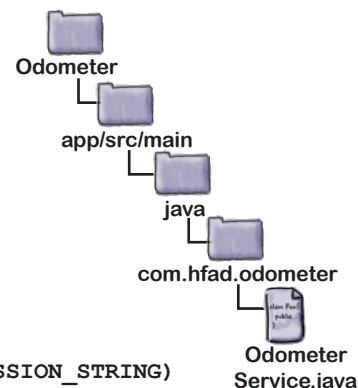
```
@Override
public void onDestroy() {
    super.onDestroy();
    if (locManager != null && listener != null) {
        if (ContextCompat.checkSelfPermission(this, PERMISSION_STRING)
            == PackageManager.PERMISSION_GRANTED) {
            locManager.removeUpdates(listener);
        }
        locManager = null;
        listener = null;
    }
}

public double getDistance() {
    return this.distanceInMeters / 1609.344;
}
```

Add the onDestroy() method.

Stop getting locations updates (if we have permission to remove them).

Set the LocationManager and LocationListener variables to null.



Let's take the app for a test drive.

there are no Dumb Questions

Q: I noticed I can call `checkSelfPermission()` directly in my service without using `ContextCompat`. Why do I have to use the `ContextCompat` version?

A: Because it's simpler to use. A `checkSelfPermission()` method was added to the `Context` class in API level 23, but this means it's not available on devices running an older version of Android.



Test drive the app

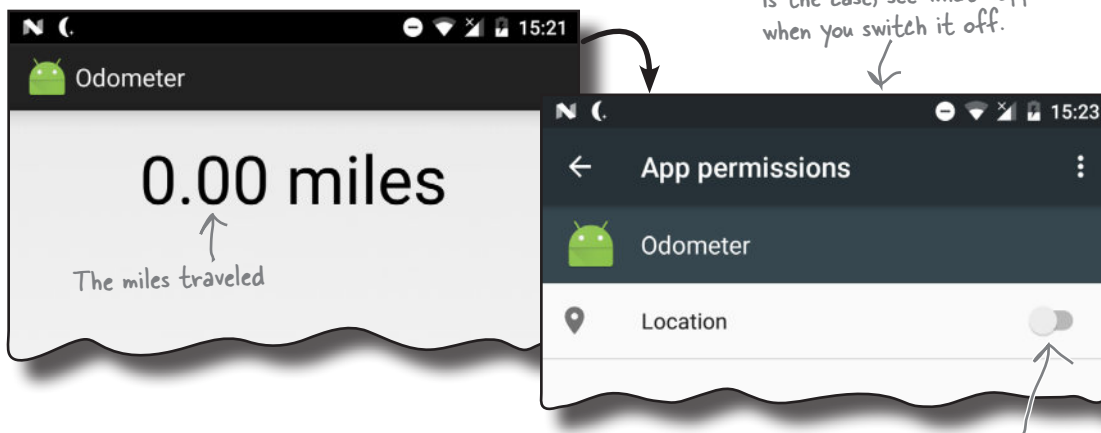
When we launch the app, 0.0 miles is initially displayed. When we check the app permissions, permission hasn't been granted to use Location Services. You can check this on your device by opening the device settings, choosing Apps, selecting the Odometer app, and opening the Permissions section:

bound services and permissions

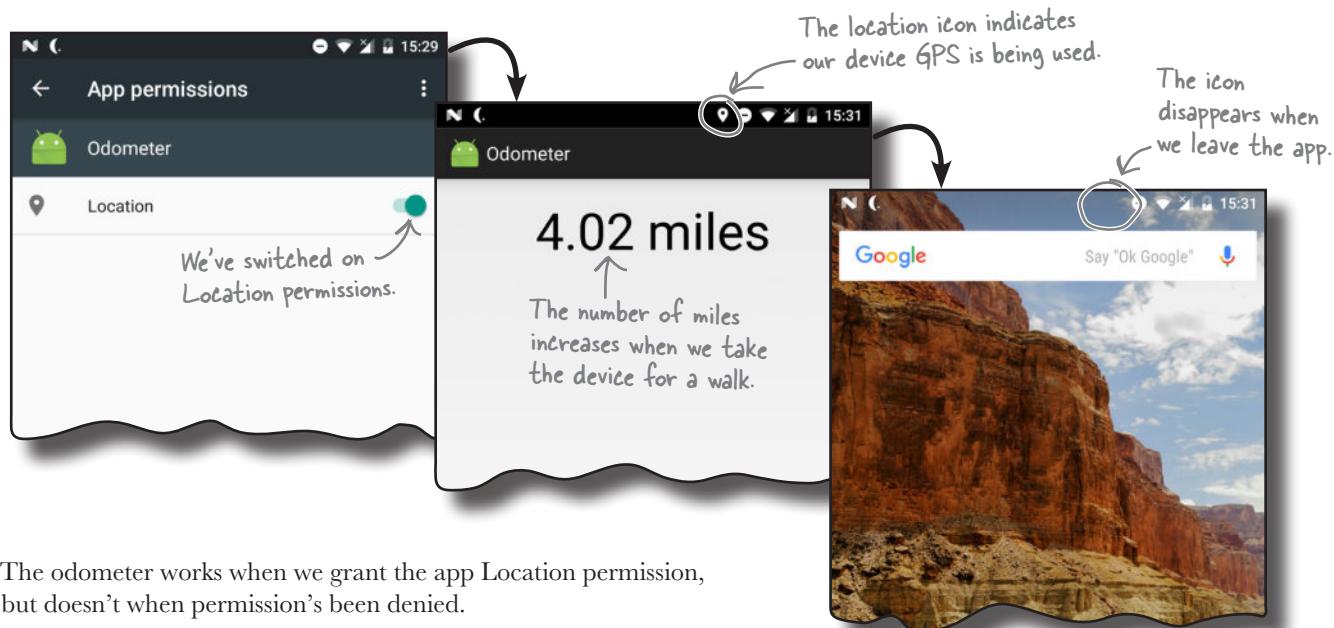


OdometerService
MainActivity
Location Services

Location Services permission for the app may be granted on your device by default. If this is the case, see what happens when you switch it off.



When we grant Location permission for the Odometer app and return to the app, the location icon is displayed in the status bar, and the number of miles traveled increases when we take the device for a walk. The location icon disappears when we leave the app:



The odometer works when we grant the app Location permission, but doesn't when permission's been denied.

Get the app to request permission

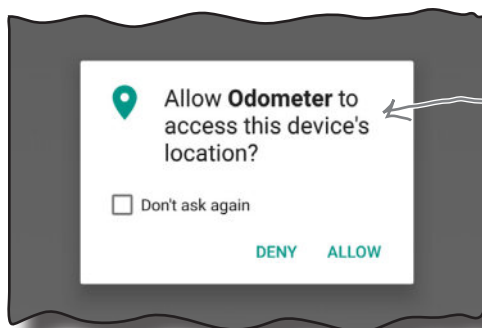
So far, we've made `OdometerService` check whether it has permission to get the user's precise location. If permission's been granted, it uses the Android Location Services to track the distance the user travels. But what if permission *hasn't* been granted?

If the app doesn't have permission to get the user's precise location, `OdometerService` can't use the Location Services we require. Rather than just accepting this, the app would work better if it were to ask the user to grant the permission.

We're going to change `MainActivity` so that if the user hasn't granted the permission we need, we'll request it. To achieve this, we'll get `MainActivity` to do three things:

- 1 **Before `MainActivity` binds to the service, request `ACCESS_FINE_LOCATION` permission if it's not already been granted.**

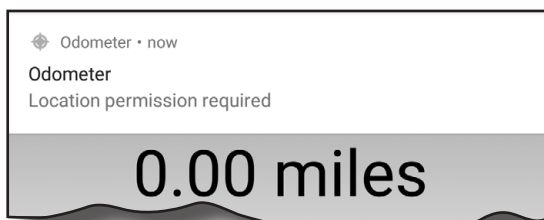
This will present the user with a permission request dialog.



This dialog appears when you request `ACCESS_FINE_LOCATION` permission at runtime.

- 2 **Check the response, and bind to the service if permission is granted.**
- 3 **If permission is denied, issue a notification.**

We'll issue this notification if the user doesn't grant the permission we need.



Let's start by looking at how you make an activity request permissions at runtime.



Check permissions at runtime

Earlier in the chapter, you saw how to check whether the user has granted a particular permission using the `ContextCompat.checkSelfPermission()` method:

```
if (ContextCompat.checkSelfPermission(this, PERMISSION_STRING)
    == PackageManager.PERMISSION_GRANTED) {
    //Run code that needs the user's permission
}
```

This code checks whether the user has granted a permission.

If the user has granted permission, the method returns a value of `PackageManager.PERMISSION_GRANTED`, and the code requiring the permission will run successfully. But what if permission has been denied?

Ask for permissions you don't have

If the user hasn't granted one or more permissions needed by your code, you can use the `ActivityCompat.requestPermissions()` method to request permission at runtime. `ActivityCompat` is a class from the `AppCompatActivity` Support Library that provides backward compatibility with older versions of Android. Its `requestPermissions()` method takes three parameters: a `Context` (usually `this`), a `String` array of permissions you want to check, and an int request code for the permission request. As an example, here's how you'd use the method to request the `ACCESS_FINE_LOCATION` permission:

```
ActivityCompat.requestPermissions(this,
    new String[]{android.Manifest.permission.ACCESS_FINE_LOCATION}, 6854);
```

When the `requestPermissions()` method is called, it displays one or more dialogs asking the user for each permission. The dialog gives the user a choice between denying or allowing the permission, and there's also a checkbox they can check if they don't want to be asked about the permission again. If they check the checkbox and deny the permission, subsequent calls to the `requestPermissions()` method won't display the dialog.

Note that the `requestPermissions()` method can only be called from an activity. You can't request permissions from a service.

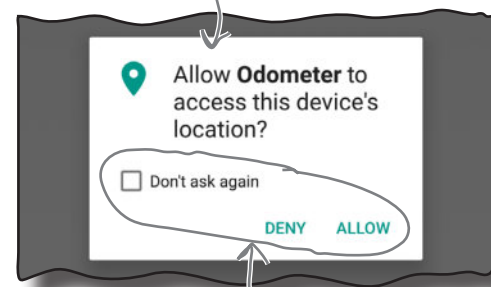
We're going to update `MainActivity` so that it requests permission to get the device's location if it hasn't already been granted.

The `requestPermissions()` method can only be called by an activity. It can't be called from a service.

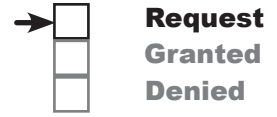
Use this method to request permissions at runtime.

This is the request code for the permissions request. It can be any int. You'll see where this gets used in a couple of pages.

This is the permissions request dialog.



The user can deny or allow permissions using these options.



Check for Location Services permissions in MainActivity's onStart() method

We're currently using MainActivity's onStart() method to bind the activity to OdometerService. We're going to change the code so that MainActivity only binds to the service if the user has granted the permission specified by the PERMISSION_STRING constant we defined in OdometerService. If permission's not been granted, we'll ask for it.

Here's the updated code for MainActivity.java; update your version of the code with our changes:

```

...
import android.content.pm.PackageManager;
import android.support.v4.app.ActivityCompat;
import android.support.v4.content.ContextCompat;

public class MainActivity extends Activity {
    ...
    private final int PERMISSION_REQUEST_CODE = 698;
    ...
    @Override
    protected void onStart() {
        super.onStart();
        if (ContextCompat.checkSelfPermission(this, OdometerService.PERMISSION_STRING)
            != PackageManager.PERMISSION_GRANTED) {
            ActivityCompat.requestPermissions(this,
                new String[]{OdometerService.PERMISSION_STRING},
                PERMISSION_REQUEST_CODE);
        } else {
            Intent intent = new Intent(this, OdometerService.class);
            bindService(intent, connection, Context.BIND_AUTO_CREATE);
        }
    }
}

```

We're using these extra classes, so we need to import them.

We'll use this int for the permission request code.

If permission hasn't already been granted... request it at runtime.

If permission has already been granted, bind to the service.

```

graph TD
    Odometer --> app_src_main[app/src/main]
    app_src_main --> java
    java --> com_hfad_odometer[com.hfad.odometer]
    com_hfad_odometer --> Main_Activity_java[Main Activity.java]

```

Once you've requested permission from the user, you need to check the user's response. We'll do that next.

Check the user's response to the permission request



When you ask the user to grant a permission using the `requestPermissions()` method, you can't determine whether permission was granted by checking its return value. That's because the permission request happens asynchronously so that the current thread isn't blocked while you wait for the user to respond.

Instead, you check the user's response by overriding the activity's `onRequestPermissionsResult()` method. This has three parameters: an int request code to identify the permissions request, a String array of permissions, and an int array for the results of the requests.

To use this method, you first check whether the int request code matches the one you used in the `requestPermissions()` method. If it does, you then check whether the permission has been granted.

The code below checks whether the user granted the permission we requested using the `requestPermissions()` method on the previous page. If permission's been granted, it binds to `OdometerService`. Add this method to your version of `MainActivity.java`:

MainActivity.java:

```

@Override
public void onRequestPermissionsResult(int requestCode,
String permissions[], int[] grantResults) {

    switch (requestCode) {
        case PERMISSION_REQUEST_CODE: {
            if (grantResults.length > 0
                && grantResults[0] == PackageManager.PERMISSION_GRANTED) {
                Intent intent = new Intent(this, OdometerService.class);
                bindService(intent, connection, Context.BIND_AUTO_CREATE);
            } else {
                //Code to run if permission was denied
            }
        }
    }
}

```

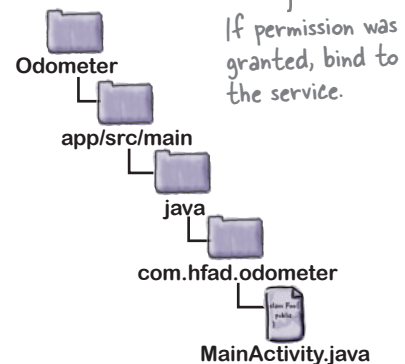
The `onRequestPermissionsResult()` method returns the results of your permissions requests.

Check whether the code matches the one we used in our `requestPermissions()` method.

If the request was cancelled, no results will be returned.

If permission was granted, bind to the service.

We still need to write this code.



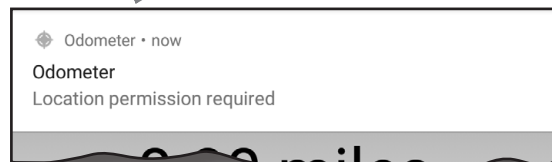
Finally, if the user doesn't give us permission to use their current location, we need to inform them that the odometer won't work.

Issue a notification if we're denied permission



If the user decides not to grant permission to use their current location, the `OdometerService` won't be able to tell how far they've traveled. If this happens, we'll issue a notification to inform the user. We're going to use a notification because it will remain in the notification area until the user has decided what to do. Another advantage of using a notification is that we can get it to start `MainActivity` when it's clicked. Doing this means that `MainActivity`'s `onStart()` method will run, which will again prompt the user to grant the permission (unless the user has previously selected the option not to be asked again).

We'll issue this notification if the user denies permission.



See if you can build the notification we require by having a go at the exercise on the next page.

there are no Dumb Questions

Q: I tried turning off Location permissions for the Odometer app when I was in the middle of using it, and the number of miles displayed went back to 0. Why?

A: When you switch off the Location permissions for the app, Android may kill the process the app is running in. This resets all of the variables.

Q: That sounds drastic. Are there any other times when Android might kill a process?

A: Yes, when it's low on memory, but it will always try to keep alive any processes that are actively being used.

Q: Why aren't we calling the `requestPermissions()` method from `OdometerService`?

A: Because the `requestPermissions()` method is only available for activities, not services.

Q: Can I change the text that's displayed in the `requestPermissions()` dialog?

A: No. The text and options it displays are fixed, so Android won't let you change them.

Q: But I want to give the user more information about why I need a particular permission. Can I do that?

A: One option is to call the `ActivityCompat.shouldShowRequestPermissionRationale()` method before calling `requestPermissions()`. This returns a `true` value if the user has previously denied the permission request, and hasn't checked the checkbox to say they don't want to be asked again. If this is the case, you can give the user more information outside the permission request before requesting the permission again.

Q: What other permissions do I have to declare and ask permission for?

A: Generally, you need the user's permission for any actions that use private data or may affect the way in which other apps function. The online documentation for each class should indicate whether a permission is needed, and Android Studio should highlight this too. You can find a full list of actions requiring permissions here:

<https://developer.android.com/guide/topics/permissions/requesting.html#normal-dangerous>

Q: What if I design an app that performs these kinds of actions and don't ask for permission?

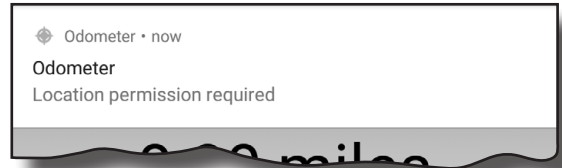
A: If your target SDK is API level 23 or above and you don't request permission, your code won't compile.

Pool Puzzle



Your **goal** is to build and issue a heads-up notification. The notification should start MainActivity when it's clicked, then disappear. Take code snippets from the pool, and place them in 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.

Write the code to create this notification.



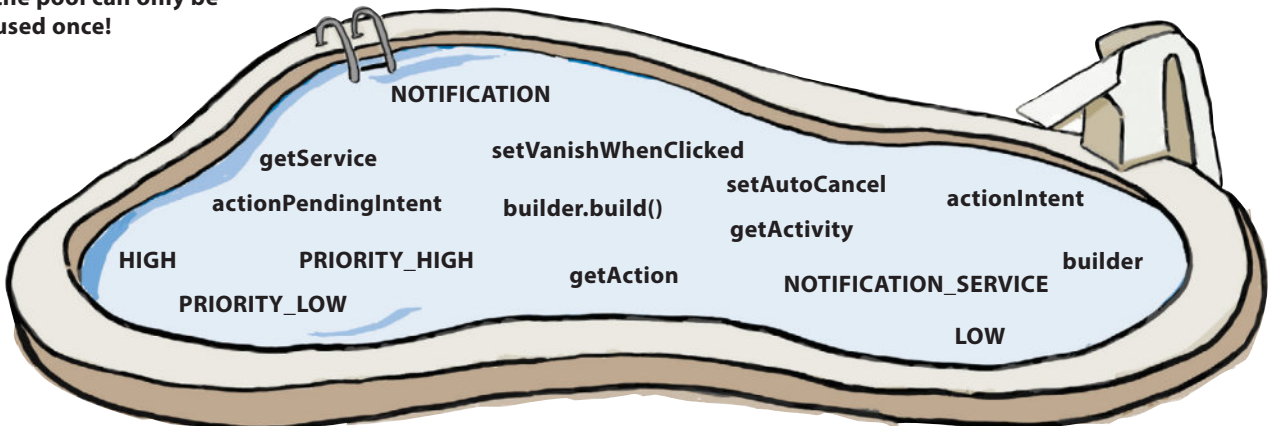
```
NotificationCompat.Builder builder = new NotificationCompat.Builder(this)
    .setSmallIcon(android.R.drawable.ic_menu_compass)
    .setContentTitle("Odometer")
    .setContentText("Location permission required")
    .setPriority(NotificationCompat. ....)
    .setVibrate(new long[] {0, 1000})
    .....(true);

Intent actionIntent = new Intent(this, MainActivity.class);
PendingIntent actionPendingIntent = PendingIntent. ....(this, 0,
    actionIntent, PendingIntent.FLAG_UPDATE_CURRENT);
builder.setContentIntent( .....);

NotificationManager notificationManager =
    (NotificationManager) getSystemService( .....);
notificationManager.notify(43, .....);
```

A built-in drawable to display a compass icon.

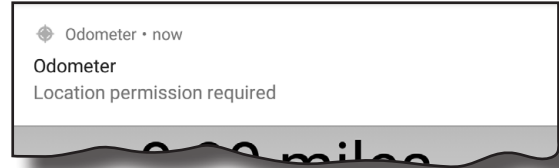
Note: each thing from the pool can only be used once!



Pool Puzzle Solution



Your **goal** is to build and issue a heads-up notification. The notification should start `MainActivity` when it's clicked, then disappear. Take code snippets from the pool, and place them in 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.



```
NotificationCompat.Builder builder = new NotificationCompat.Builder(this)
    .setSmallIcon(android.R.drawable.ic_menu_compass)
    .setContentTitle("Odometer")
    .setContentText("Location permission required")
    .setPriority(NotificationCompat.PRIORITY_HIGH)
    .setVibrate(new long[] {0, 1000})
    .setAutoCancel(true);
```

This makes the notification disappear when it's clicked.

Heads-up notifications need to have a high priority.

```
Intent actionIntent = new Intent(this, MainActivity.class);
PendingIntent actionPendingIntent = PendingIntent.getActivity(this, 0,
    actionIntent, PendingIntent.FLAG_UPDATE_CURRENT);
builder.setContentIntent(actionPendingIntent);
```

Create a `PendingIntent` using the `getActivity()` method.

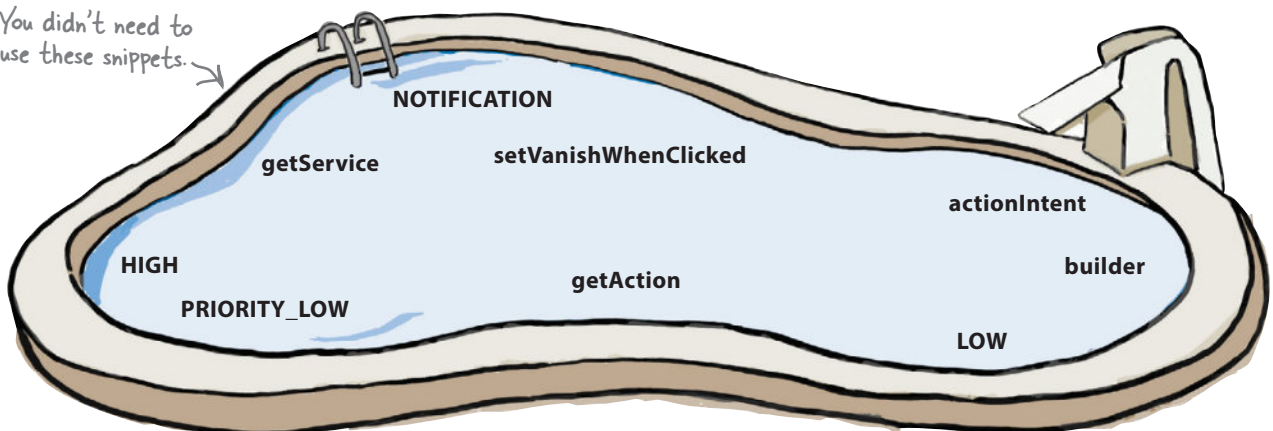
Add the `PendingIntent` to the notification so that it starts `MainActivity` when it's clicked.

```
NotificationManager notificationManager =
    (NotificationManager) getSystemService(NOTIFICATION_SERVICE);
notificationManager.notify(43, builder.build());
```

Build the notification.

Use the notification service.

You didn't need to use these snippets.





Request
Granted
Denied

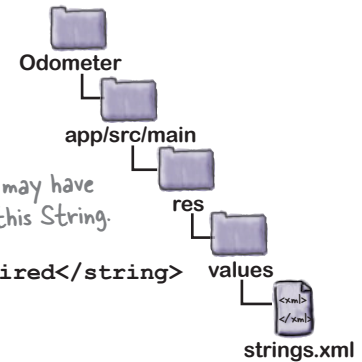
Add notification code to `onRequestPermissionsResult()`

We'll update our `MainActivity` code so that if the user denies our permission request, it issues a heads-up notification.

First, add the following Strings to `Strings.xml`; we'll use these for the notification's title and text:

```
<string name="app_name">Odometer</string>
<string name="permission_denied">Location permission required</string>
```

Android Studio may have already added this String.



Then update your version of `MainActivity.java` with the following code:

```
...
import android.support.v4.app.NotificationCompat;
import android.app.NotificationManager;
import android.app.PendingIntent;

public class MainActivity extends Activity {
    ...
    private final int NOTIFICATION_ID = 423;
    ...
    @Override
    public void onRequestPermissionsResult(int requestCode,
                                         String permissions[], int[] grantResults) {
        switch (requestCode) {
            case PERMISSION_REQUEST_CODE: {
                if (grantResults.length > 0
                    && grantResults[0] == PackageManager.PERMISSION_GRANTED) {
                    ...
                } else {
                    //Create a notification builder
                    NotificationCompat.Builder builder = new NotificationCompat.Builder(this)
                        .setSmallIcon(android.R.drawable.ic_menu_compass)
                        .setContentTitle(getResources().getString(R.string.app_name))
                        .setContentText(getResources().getString(R.string.permission_denied))
                        .setPriority(NotificationCompat.PRIORITY_HIGH)
                        .setVibrate(new long[] {1000, 1000})
                        .setAutoCancel(true);
                }
            }
        }
    }
}
```

We're using these extra classes, so we need to import them.

We'll use this constant for the notification ID.

These settings are needed for all notifications.

Add these to make it a heads-up notification.

This line makes the notification disappear when it's clicked.

The code continues on the next page.



The notification code (continued)

```

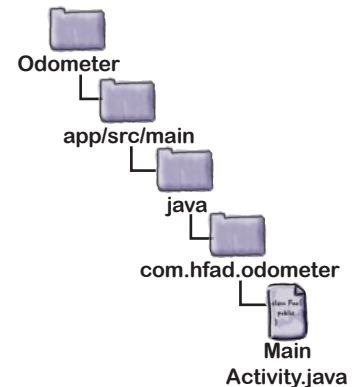
        //Create an action
        Intent actionIntent = new Intent(this, MainActivity.class);
        PendingIntent actionPendingIntent = PendingIntent.getActivity(
            this,
            0,
            actionIntent,
            PendingIntent.FLAG_UPDATE_CURRENT);
        builder.setContentIntent(actionPendingIntent);

        //Issue the notification
        NotificationManager notificationManager =
            (NotificationManager) getSystemService(NOTIFICATION_SERVICE);
        notificationManager.notify(NOTIFICATION_ID, builder.build());
    }
}
...
}

```

Adding a PendingIntent to the notification means that it will start MainActivity when it's clicked.

Build the notification and issue it.



That's all the code we need to display a notification if the user decides to deny us `ACCESS_FINE_LOCATION` permission. We'll show you the full `MainActivity` code over the next few pages, then take the app for a final test drive.



Request
Granted
Denied

The full code for MainActivity.java

Here's our complete code for *MainActivity.java*; make sure your version of the code matches ours:

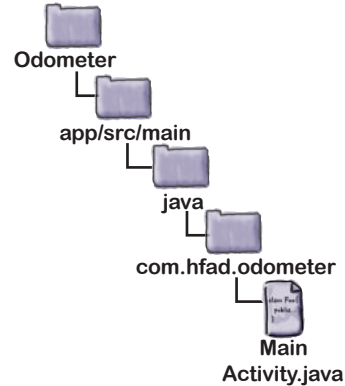
```
package com.hfad.odometer;

import android.app.Activity;
import android.os.Bundle;
import android.content.ServiceConnection;
import android.os.IBinder;
import android.content.ComponentName;
import android.content.Context;
import android.content.Intent;
import android.os.Handler;
import android.widget.TextView;
import java.util.Locale;
import android.content.pm.PackageManager;
import android.support.v4.app.AppCompatActivity;
import android.support.v4.content.ContextCompat;
import android.support.v4.app.NotificationCompat;
import android.app.NotificationManager;
import android.app.PendingIntent;

public class MainActivity extends Activity {

    private OdometerService odometer;
    private boolean bound = false;
    private final int PERMISSION_REQUEST_CODE = 698;
    private final int NOTIFICATION_ID = 423;

    private ServiceConnection connection = new ServiceConnection() {
        @Override
        public void onServiceConnected(ComponentName componentName, IBinder binder) {
            OdometerService.OdometerBinder odometerBinder =
                (OdometerService.OdometerBinder) binder;
            odometer = odometerBinder.getOdometer();
            bound = true;
        }
        @Override
        public void onServiceDisconnected(ComponentName componentName) {
            bound = false;
        }
    };
};
```



These are all classes from the AppCompatActivity Support Library.

We've been using the Activity class, but you can use AppCompatActivity if you prefer.

We need a ServiceConnection in order to bind MainActivity to OdometerService.

The code continues on the next page.

The MainActivity.java code (continued)



```

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

@Override
public void onRequestPermissionsResult(int requestCode,
                                     String permissions[], int[] grantResults) {
    switch (requestCode) {
        case PERMISSION_REQUEST_CODE: {
            if (grantResults.length > 0
                && grantResults[0] == PackageManager.PERMISSION_GRANTED) {
                Intent intent = new Intent(this, OdometerService.class);
                bindService(intent, connection, Context.BIND_AUTO_CREATE);
            } else {
                //Create a notification builder
                NotificationCompat.Builder builder = new NotificationCompat.Builder(this)
                    .setSmallIcon(android.R.drawable.ic_menu_compass)
                    .setContentTitle(getResources().getString(R.string.app_name))
                    .setContentText(getResources().getString(R.string.permission_denied))
                    .setPriority(NotificationCompat.PRIORITY_HIGH)
                    .setVibrate(new long[] { 1000, 1000})
                    .setAutoCancel(true);

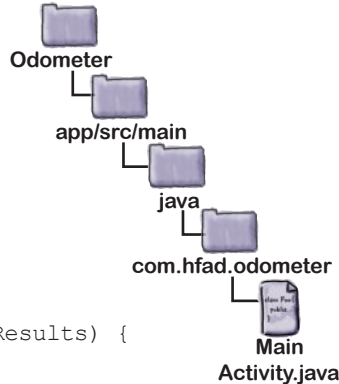
                //Create an action
                Intent actionIntent = new Intent(this, MainActivity.class);
                PendingIntent actionPendingIntent = PendingIntent.getActivity(this, 0,
                    actionIntent, PendingIntent.FLAG_UPDATE_CURRENT);
                builder.setContentIntent(actionPendingIntent);
                //Issue the notification
                NotificationManager notificationManager =
                    (NotificationManager) getSystemService(NOTIFICATION_SERVICE);
                notificationManager.notify(NOTIFICATION_ID, builder.build());
            }
        }
    }
}

```

If we've asked the user for permissions at runtime, check the result.

Bind to the service if the user has granted permission.

Issue a notification if permission's been denied.



The code continues on the next page.



The MainActivity.java code (continued)

```

@Override
protected void onStart() {
    super.onStart();
    if (ContextCompat.checkSelfPermission(this,
        OdometerService.PERMISSION_STRING)
        != PackageManager.PERMISSION_GRANTED) {
        ActivityCompat.requestPermissions(this,
            new String[]{OdometerService.PERMISSION_STRING},
            PERMISSION_REQUEST_CODE);
    } else {
        Intent intent = new Intent(this, OdometerService.class);
        bindService(intent, connection, Context.BIND_AUTO_CREATE);
    }
}

@Override
protected void onStop() {
    super.onStop();
    if (bound) {
        unbindService(connection);
        bound = false;
    }
}

private void displayDistance() {
    final TextView distanceView = (TextView) findViewById(R.id.distance);
    final Handler handler = new Handler();
    handler.post(new Runnable() {
        @Override
        public void run() {
            double distance = 0.0;
            if (bound && odometer != null) {
                distance = odometer.getDistance();
            }
            String distanceStr = String.format(Locale.getDefault(),
                "%1$, .2f miles", distance);
            distanceView.setText(distanceStr);
            handler.postDelayed(this, 1000);
        }
    });
}
}

```

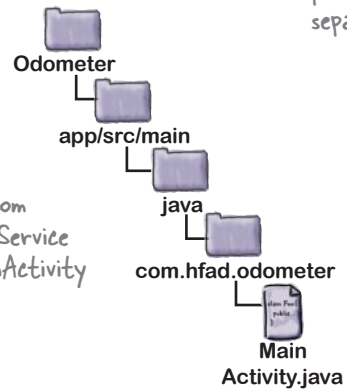
Request ACCESS_FINE_LOCATION permission if we don't have it already.

Bind to OdometerService if we've been granted the permission it requires.

Unbind from OdometerService when MainActivity stops.

Display the distance traveled.

We're binding to OdometerService in two different places, so you could put this code in a separate method.

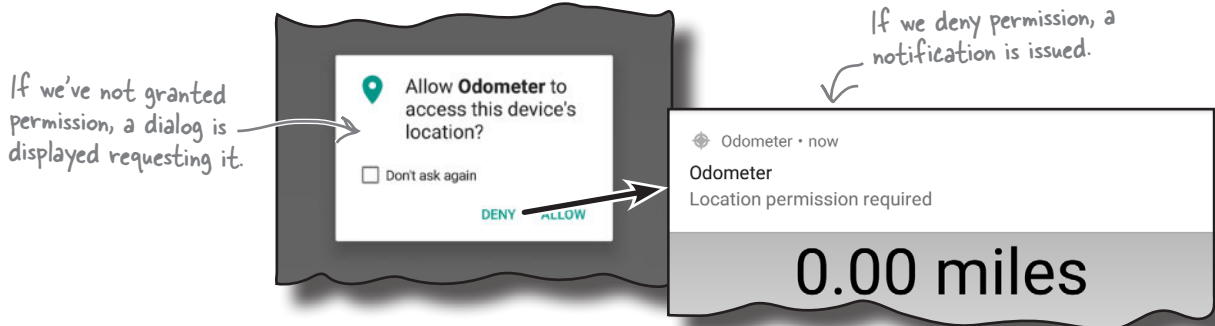




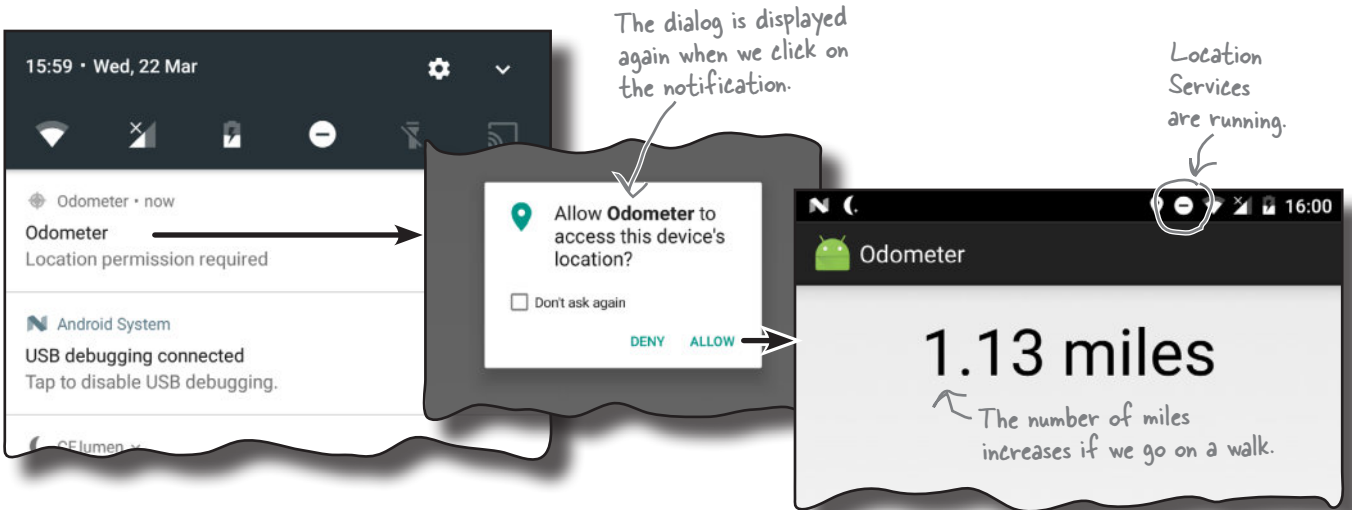
Test drive the app



When we run the app with its Location permission switched off, a permission request dialog is displayed. If we click on the Deny option, a notification is issued:



When we click on the notification, the permission request dialog is displayed again. If we click on the option to allow permission, the Location icon appears in the status bar and the number of miles increases when we take the device on a road trip:



We know you're full of great ideas for improving the Odometer app, so why not try them out? As an example, try adding Start, Stop, and Reset buttons to start, stop, and reset the distance traveled.



Your Android Toolbox

You've got Chapter 19 under your belt and now you've added bound services to your toolbox.

You can download the full code for the chapter from <https://tinyurl.com/HeadFirstAndroid>.



BULLET POINTS

- You create a bound service by extending the **Service** class. You define your own **Binder** object, and override the **onBind()** method.
- Bind a component to a service using the **bindService()** method.
- Use a **ServiceConnection** so that your activity can get a reference to the service when it's bound.
- Unbind a component from a service using the **unbindService()** method.
- When a bound service is created, its **onCreate()** method is called. **onBind()** gets called when a component binds to the service.
- When all components have unbound from the service, its **onUnbind()** method is called.
- A bound service is destroyed when no components are bound to it. Its **onDestroy()** method is called just before the service is destroyed.
- Use the **Android Location Services** to determine the current location of the device.
- To get the current location of the device, you need to declare that the app requires **ACCESS_FINE_LOCATION** permission in *AndroidManifest.xml*.
- Get location updates using a **LocationListener**.
- A **LocationManager** gives you access to Android's Location Services. Get the best location provider available on the device using its **getBestProvider()** method. Request location updates from the provider using **requestLocationUpdates()**.
- Use **removeUpdates()** to stop getting location updates.
- If your target SDK is API level 23 or above, check at runtime whether your app has been granted a permission using the **ContextCompat.checkSelfPermission()** method.
- Request permissions at runtime using **ActivityCompat.requestPermissions()**.
- Check the user's response to a permission request by implementing the activity's **onRequestPermissionsResult()** method.

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 use. There are still a few more gems for you in the back of the book and a handy index, and then it's time to take all these new ideas and put them into practice. Bon voyage!

appendix i: relative and grid layouts

Meet the Relatives



There are two more layouts you will often meet in Androidville.

In this book we've concentrated on using simple *linear and frame layouts*, and introduced you to *Android's new constraint layout*. But there are two more layouts we'd like you to know about: **relative layouts** and **grid layouts**. They've largely been superseded by the constraint layout, but we have a soft spot for them, and we think they'll stay around for a few more years.

A relative layout displays views in relative positions

A relative layout allows you to position views relative to their parent layout, or relative to other views in the layout.

You define a relative layout using the `<RelativeLayout>` element like this:

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

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

There may be other attributes too.

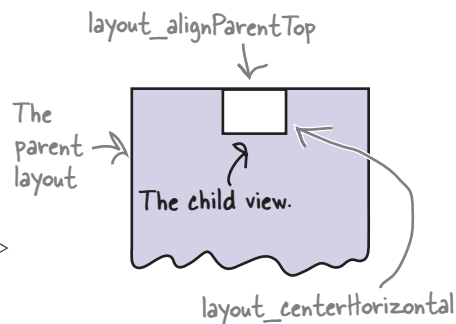
This is where you add any views.

Positioning views relative to the parent layout

If you want a view to always appear in a particular position on the screen, irrespective of the screen size or orientation, you need to position the view relative to its parent. As an example, here's how you'd make sure a button always appears in the top-center of the layout:

```
<RelativeLayout ... >
    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/click_me"
        android:layout_alignParentTop="true"
        android:layout_centerHorizontal="true" />
</RelativeLayout>
```

The layout contains the button, so the layout is the button's parent.



The lines of code:

```
android:layout_alignParentTop="true"
android:layout_centerHorizontal="true"
```

mean that the top edge of the button is aligned to the top edge of the layout, and the button is centered horizontally in the parent layout. This will be the case no matter what the screen size, language, or orientation of your device:

Positioning views to the left or right

You can also position a view to the left or right of the parent layout. There are two ways of doing this.

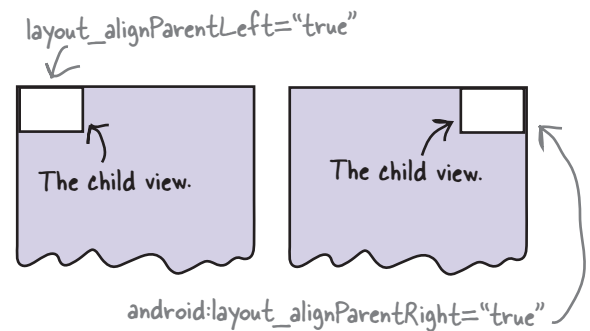
The first way is to explicitly position the view on the left or right using:

```
android:layout_alignParentLeft="true"
```

or:

```
android:layout_alignParentRight="true"
```

These lines of code mean that the left (or right) edge of the view is aligned to the left (or right) edge of the parent layout, regardless of the screen size, orientation, or language being used on the device.



Use start and end to take language direction into account

For apps where the minimum SDK is *at least* API 17, you can position views on the left or right depending on the language setting on the device. As an example, you might want views to appear on the left for languages that are read from left to right such as English. For languages that are read from right to left, you might want them to appear on the right instead so that their position is mirrored.

To do this, you use:

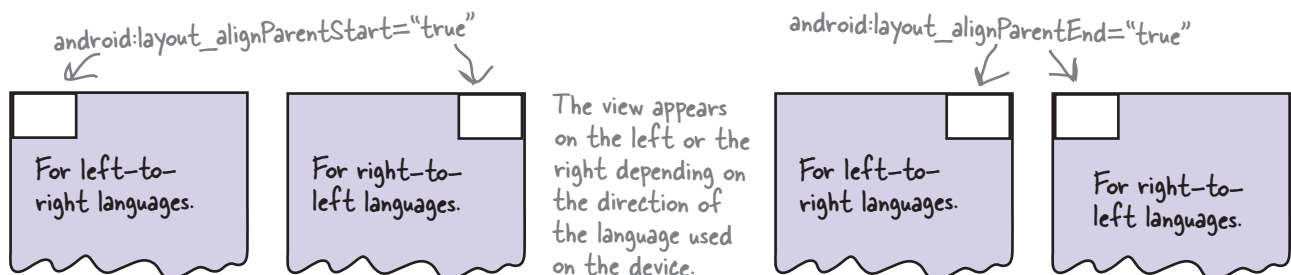
```
android:layout_alignParentStart="true"
```

or:

```
android:layout_alignParentEnd="true"
```

`android:layout_alignParentStart="true"` aligns the start edge of the view with that of its parent. The start edge is on the left for languages that are read from left to right, and the right edge for languages that are read from right to left.

`android:layout_alignParentEnd="true"` aligns the end edge of the view with that of its parent. The end edge is on the right for languages that are read from left to right, and the left edge for languages that are read from right to left.



The view appears on the left or the right depending on the direction of the language used on the device.

Attributes for positioning views relative to the parent layout

Here are some of the most common attributes for positioning views relative to their parent layout. Add the attribute you want to the view you're positioning, then set its value to "true":

```
android:attribute="true"
```

Attribute

What it does

layout_alignParentBottom

Aligns the bottom edge of the view to the bottom edge of the parent.

layout_alignParentLeft

Aligns the left edge of the view to the left edge of the parent.

layout_alignParentRight

Aligns the right edge of the view to the right edge of the parent.

layout_alignParentTop

Aligns the top edge of the view to the top edge of the parent.

layout_alignParentStart

Aligns the start edge of the view to the start edge of the parent.

layout_alignParentEnd

Aligns the end edge of the view to the end edge of the parent.

layout_centerInParent

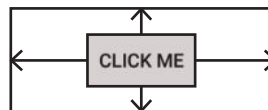
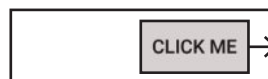
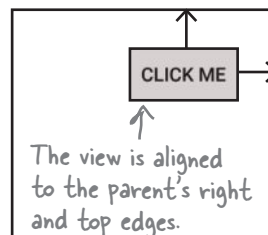
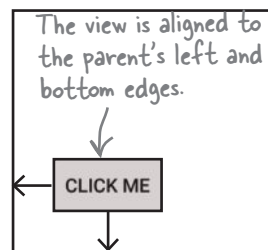
Centers the view horizontally and vertically in the parent.

layout_centerHorizontal

Centers the view horizontally in the parent.

layout_centerVertical

Centers the view vertically in the parent.



The start is on the left and the end is on the right for left-to-right languages. This is reversed for right-to-left languages.

Positioning views relative to other views

In addition to positioning views relative to the parent layout, you can also position views *relative to other views*. You do this when you want views to stay aligned in some way, irrespective of the screen size or orientation.

In order to position a view relative to another view, the view you're using as an anchor must be given an ID using the `android:id` attribute:

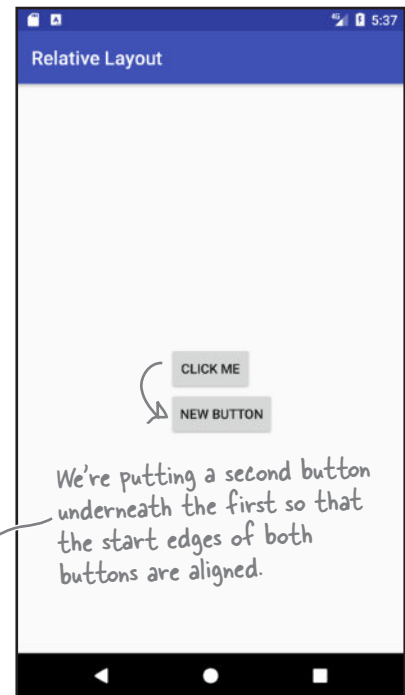
```
android:id="@+id/button_click_me"
```

The syntax “@+id” tells Android to include the ID as a resource in its resource file `R.java`. You must include the “+” whenever you define a new view in the layout. If you don't, Android won't add the ID as a resource and you'll get errors in your code. You can omit the “+” when the ID has already been added as a resource.

Here's how you create a layout with two buttons, with one button centered in the middle of the layout, and the second button positioned underneath the first:

```
<RelativeLayout ... >
  <Button
    android:id="@+id/button_click_me"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_centerInParent="true"
    android:text="Click Me" />

  <Button
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_alignStart="@id/button_click_me"
    android:layout_below="@id/button_click_me"
    android:text="New Button" />
</RelativeLayout>
```



The lines:

```
android:layout_alignStart="@id/button_click_me"
android:layout_below="@id/button_click_me"
```

ensure that the second button has its start edge aligned to the start edge of the first button, and is always positioned beneath it.

Attributes for positioning views relative to other views

Here are attributes you can use when positioning views relative to another view. Add the attribute to the view you're positioning, and set its value to the view you're positioning relative to:

```
android:attribute="@+id/view_id"
```

Remember, you can leave out the "+" if you've already defined the view ID in your layout.

Attribute

What it does

layout_above

Puts the view above the view you're anchoring it to.

layout_below

Puts the view below the view you're anchoring it to.

layout_alignTop

Aligns the top edge of the view to the top edge of the view you're anchoring it to.

layout_alignBottom

Aligns the bottom edge of the view to the bottom edge of the view you're anchoring it to.

layout_alignLeft, layout_alignStart

Aligns the left (or start) edge of the view to the left (or start) edge of the view you're anchoring it to.

layout_alignRight, layout_alignEnd

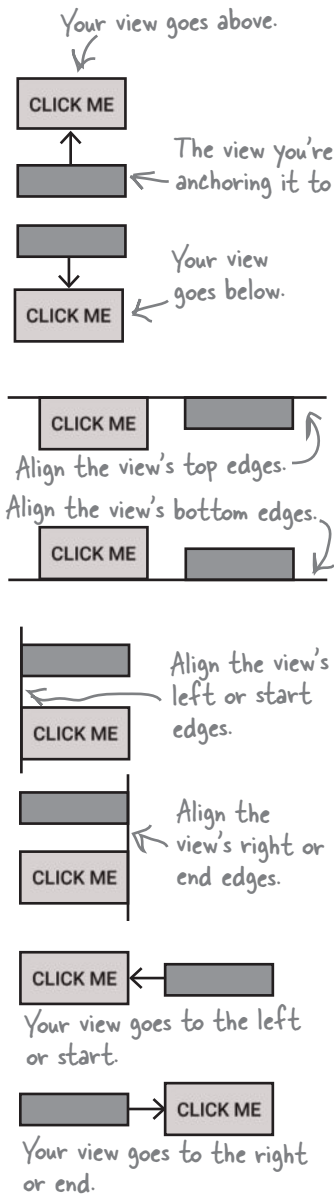
Aligns the right (or end) edge of the view to the right (or end) edge of the view you're anchoring it to.

layout_toLeftOf, layout_toStartOf

Puts the right (or end) edge of the view to the left (or start) of the view you're anchoring it to.

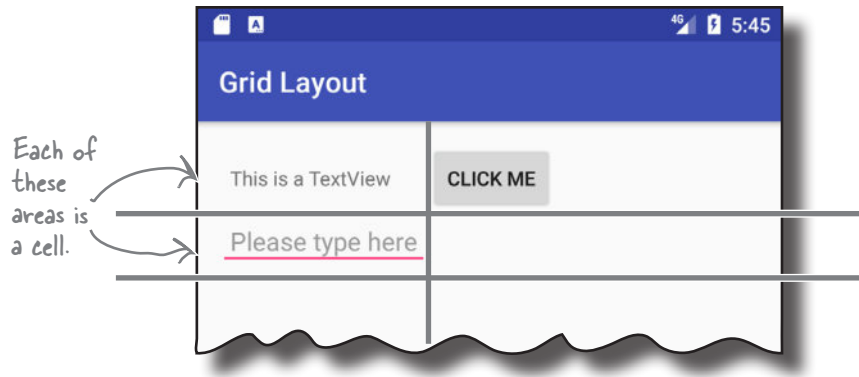
layout_toRightOf, layout_toEndOf

Puts the left (or start) edge of the view to the right (or end) of the view you're anchoring it to.



A grid layout displays views in a grid

A grid layout splits the screen up into a grid of rows and columns, and allocates views to cells:



How you define a grid layout

You define a grid layout in a similar way to how you define the other types of layout, this time using the `<GridLayout>` element:

```

<GridLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:columnCount="2"
    ... >

    ...

</GridLayout>

```

These are the same attributes we used for our other layouts.
How many columns you want your layout to have (in this case, 2)
This is where you add any views.

You specify how many columns you want the grid layout to have using:

```
android:columnCount="number"
```

where `number` is the number of columns. You can also specify a maximum number of rows using:

```
android:rowCount="number"
```

but in practice you can usually let Android figure this out based on the number of views in the layout. Android will include as many rows as is necessary to display the views.

Adding views to the grid layout

You add views to a grid layout in a similar way to how you add views to a linear layout:

```
<GridLayout ... >

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/textview" />

    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/click_me" />

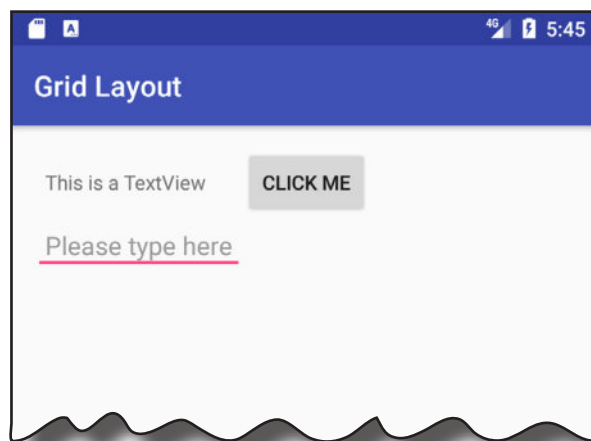
    <EditText
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:hint="@string/edit" />

</GridLayout>
```

Just as with a linear layout, there's no need to give your views IDs unless you're explicitly going to refer to them in your activity code. The views don't need to refer to each other within the layout, so they don't need to have IDs for this purpose.

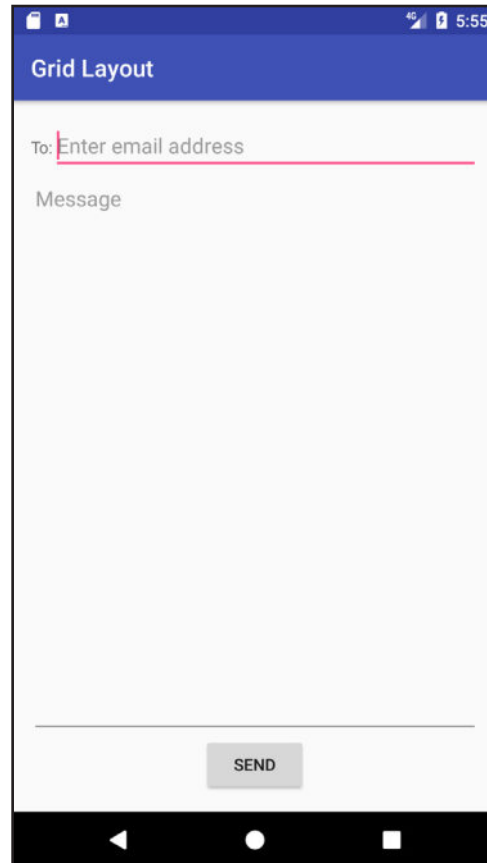
By default, the grid layout positions your views in the order in which they appear in the XML. So if you have a grid layout with two columns, the grid layout will put the first view in the first position, the second view in the second position, and so on.

The downside of this approach is that if you remove one of your views from the layout, it can drastically change the appearance of the layout. To get around this, you can specify where you want each view to appear, and how many columns you want it to span.



Let's create a new grid layout

To see this in action, we'll create a grid layout that specifies which cells we want views to appear in, and how many columns they should span. The layout is composed of a text view containing the text "To", an editable text field that contains hint text of "Enter email address", an editable text field that contains hint text of "Message", and a button labeled "Send":

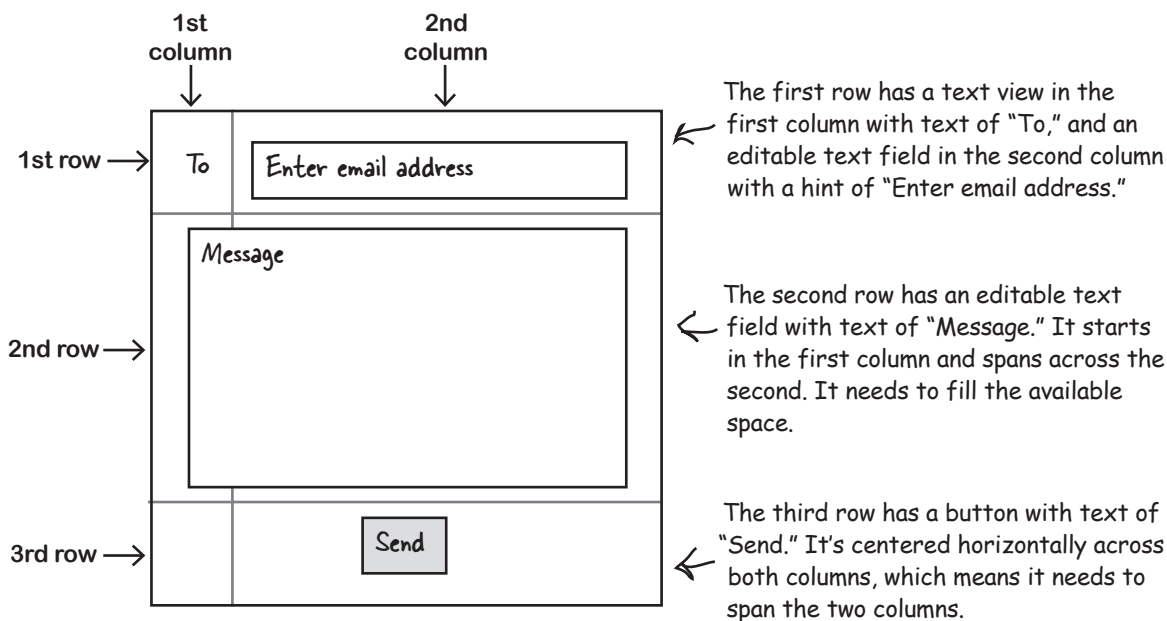


Here's what we're going to do

- 1 Sketch the user interface, and split it into rows and columns.**
This will make it easier for us to see how we should construct our layout.
- 2 Build up the layout row by row.**

We'll start with a sketch

The first thing we'll do to create our new layout is sketch it out. That way we can see how many rows and columns we need, where each view should be positioned, and how many columns each view should span.



The grid layout needs two columns

We can position our views how we want if we use a grid layout with two columns:

```
<GridLayout 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:columnCount="2"
    tools:context="com.hfad.gridlayout.MainActivity" >
</GridLayout>
```

Now that we have the basic grid layout defined, we can start adding views.

Row 0: add views to specific rows and columns

The first row of the grid layout is composed of a text view in the first column, and an editable text field in the second column. You start by adding the views to the layout:

```
<GridLayout...>
  <TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/to" />

  <EditText
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_gravity="fill_horizontal"
    android:hint="@string/to_hint" />
</GridLayout>
```



You can use `android:gravity` and `android:layout_gravity` attributes with grid layouts.

You can use `layout_gravity` in grid layouts too. We're using `fill_horizontal` because we want the editable text field to fill the remaining horizontal space.

Then you use the `android:layout_row` and `android:layout_column` attributes to say which row and column you want each view to appear in. The row and column indices start from 0, so if you want a view to appear in the first column and first row, you use:

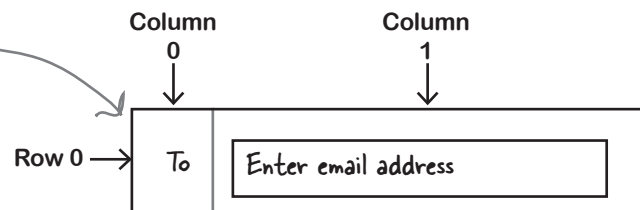
```
android:layout_row="0"
android:layout_column="0"
```

Columns and rows start at 0, so this refers to the first row and first column.

Let's apply this to our layout code by putting the text view in column 0, and the editable text field in column 1.

```
<GridLayout...>
  <TextView
    ...
    android:layout_row="0"
    android:layout_column="0"
    android:text="@string/to" />

  <EditText
    ...
    android:layout_row="0"
    android:layout_column="1"
    android:hint="@string/to_hint" />
</GridLayout>
```



Row and column indices start at 0. `layout_column="n"` refers to column `n+1` in the display.

Row 1: make a view span multiple columns

The second row of the grid layout is composed of an editable text field that starts in the first column and spans across the second. The view takes up all the available space.

To get a view to span multiple columns, you start by specifying which row and column you want the view to start in. We want the view to start in the first column of the second row, so we need to use:

```
android:layout_row="1"
android:layout_column="0"
```

We want our view to go across two columns, and we can do this using the `android:layout_columnSpan` attribute like this:

```
android:layout_columnSpan="number"
```

where `number` is the number of columns we want the view to span across. In our case, this is:

```
android:layout_columnSpan="2"
```

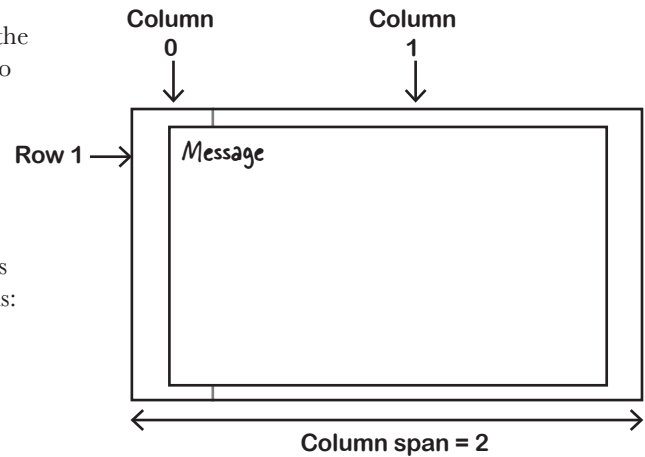
Putting it all together, here's the code for the message view:

```
<GridLayout...>
  <TextView... />
  <EditText.../>
  <EditText
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_gravity="fill"
    android:gravity="top"
    android:layout_row="1"
    android:layout_column="0"
    android:layout_columnSpan="2"
    android:hint="@string/message" />
</GridLayout>
```

These are the views we added on the last page for row 0.

We want the view to fill the available space, and for the text to appear at the top.

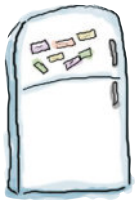
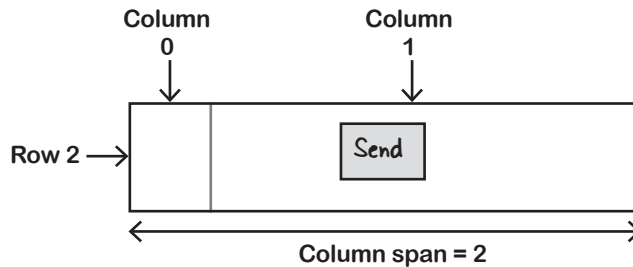
The view starts in column 0, and spans two columns.



Now that we've added the views for the first two rows, all we need to do is add the button.

Row 2: make a view span multiple columns

We need the button to be centered horizontally across the two columns like this:



Layout Magnets

We wrote some code to center the Send button in the third row of the grid layout, but a sudden breeze blew some of it away. See if you can reconstruct the code using the magnets below.

```
<GridLayout...>
  <TextView... />
  <EditText.../>
  <EditText.../>

```

These are the views we've already added.

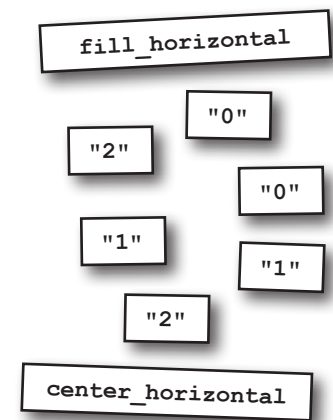
```
<Button
  android:layout_width="wrap_content"
  android:layout_height="wrap_content"

  android:layout_row= .....
  android:layout_column= .....
  android:layout_gravity= .....
  android:layout_columnSpan= .....
  android:text="@string/send" />

```

```
</GridLayout>
```

You won't need to use all of these magnets.





Layout Magnets Solution

We wrote some code to center the Send button in the third row of the grid layout, but a sudden breeze blew some of it away. See if you can reconstruct the code using the magnets below.

```

<GridLayout...>
    <TextView... />
    <EditText.../>
    <EditText.../>

    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"

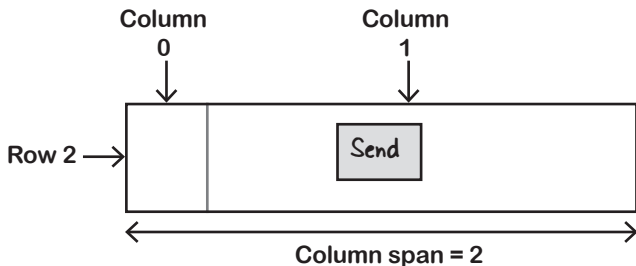
        android:layout_row= "2"
        android:layout_column= "0"
        android:layout_gravity= center_horizontal
        android:layout_columnSpan= "2"
        android:text="@string/send" />
</GridLayout>
    
```

"2" ← The button starts at row 2, column 0.

"0" ← We want to center it horizontally.

center_horizontal

"2" ← It spans two columns.



fill_horizontally

"0"

"1"

"1"

You didn't need to use these magnets.

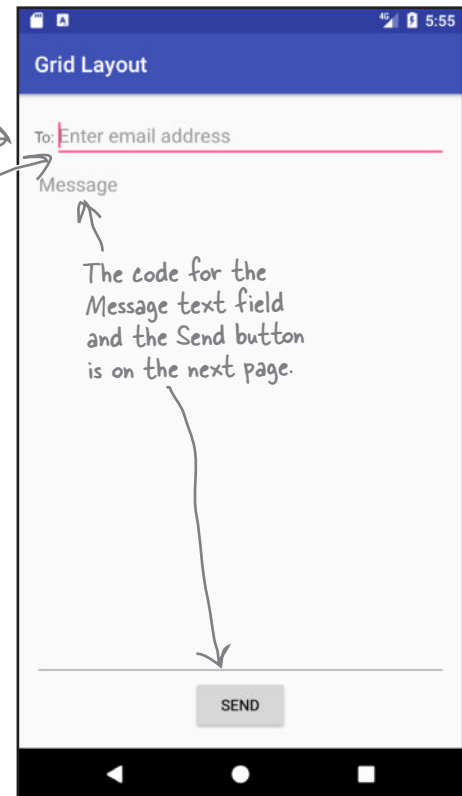
The full code for the grid layout

Here's the full code for the grid layout.

```
<?xml version="1.0" encoding="utf-8"?>
<GridLayout 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:columnCount="2"
    tools:context="com.hfad.gridlayout.MainActivity">

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_row="0"
        android:layout_column="0"
        android:text="@string/to" />

    <EditText
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="fill_horizontal"
        android:layout_row="0"
        android:layout_column="1"
        android:hint="@string/to_hint" />
```



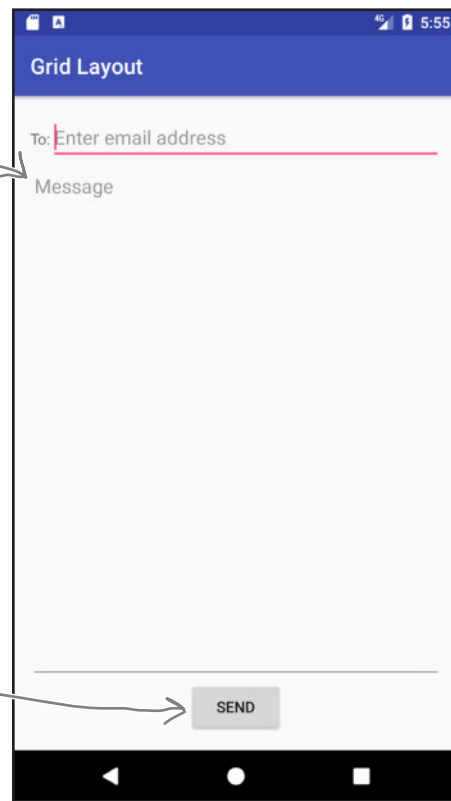
The grid layout code (continued)

```
<EditText
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_gravity="fill"
    android:gravity="top"
    android:layout_row="1"
    android:layout_column="0"
    android:layout_columnSpan="2"
    android:hint="@string/message" />

<Button
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_row="2"
    android:layout_column="0"
    android:layout_gravity="center_horizontal"
    android:layout_columnSpan="2"
    android:text="@string/send" />
</GridLayout>
```

The Message text field

The button spans two columns, starting from row 2 column 0. It's centered horizontally.



The Gradle Build Tool

Take one SDK, add a sprinkling of libraries, mix well, then bake for two minutes.



Most Android apps are created using a build tool called Gradle.

Gradle works behind the scenes to find and download libraries, compile and deploy your code, run tests, clean the grouting, and so on. Most of the time you *might not even realize it's there* because Android Studio provides a graphical interface to it. Sometimes, however, it's helpful to dive into Gradle and **hack it manually**. In this appendix we'll introduce you to some of Gradle's many talents.

has Gradle

What ~~have the Romans~~ ever done for us?

When you click the run button in Android Studio, most of the actual work is done by an external build tool called **Gradle**. Here are some of the things that Gradle does:

- ★ Locates and downloads the correct versions of any third-party libraries you need.
- ★ Calls the correct build tools in the correct sequence to turn all of your source code and resources into a deployable app.
- ★ Installs and runs your app on an Android device.
- ★ A whole bunch of other stuff, like running tests and checking the quality of your code.

It's hard to list all of the things that Gradle does because it's designed to be easily extensible. Unlike other XML-based build tools like Maven or Ant, Gradle is written in a procedural language (Groovy), which is used for both configuring a build and adding extra functionality.

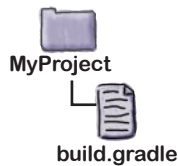
Your project's Gradle files

Every time you create a new project, Android Studio creates two files called *build.gradle*. One of these files is in your project folder, and contains a small amount of information that specifies the basic settings of your app, such as what version of Gradle to use, and which online repository:

```
buildscript {
    repositories {
        jcenter()
    }
    dependencies {
        classpath 'com.android.tools.build:gradle:2.3.0'
    }
}

allprojects {
    repositories {
        jcenter()
    }
}

task clean(type: Delete) {
    delete rootProject.buildDir
}
```



You will normally only need to change the code in this file if you want to install a third-party plug-in, or need to specify another place that contains downloadable libraries.

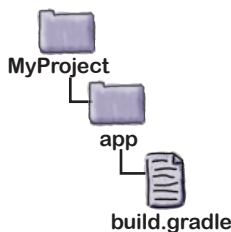
Your app's main Gradle file

The second *build.gradle* file lives inside the *app* folder within your project. This file tells Gradle how to build all of the code in your main Android module. It's where the majority of your application's properties are set, such as the level of the API that you're targeting, and the specifics of which external libraries your app will need:

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

```
android {
    compileSdkVersion 25
    buildToolsVersion "25.0.1"
    defaultConfig {
        applicationId "com.hfad.example"
        minSdkVersion 19
        targetSdkVersion 25
        versionCode 1
        versionName "1.0"
        testInstrumentationRunner "android.support.test.runner.AndroidJUnitRunner"
    }
    buildTypes {
        release {
            minifyEnabled false
            proguardFiles getDefaultProguardFile('proguard-android.txt'), 'proguard-rules.pro'
        }
    }
}

dependencies {
    compile fileTree(dir: 'libs', include: ['*.jar'])
    androidTestCompile('com.android.support.test.espresso:espresso-core:2.2.2', {
        exclude group: 'com.android.support', module: 'support-annotations'
    })
    compile 'com.android.support:appcompat-v7:25.1.1'
    compile 'com.android.support.constraint:constraint-layout:1.0.2'
    testCompile 'junit:junit:4.12'
}
```



Gradle comes built in to your project

Every time you create a new application, Android Studio includes an install of the Gradle build tool. If you look in the project directory, you will see two files called *gradlew* and *gradlew.bat*. These files contains scripts that allow you to build and deploy your app from the command line.

To get more familiar with Gradle, open a command prompt or terminal on your development machine and change into the top-level directory of your project. Then run one of the *gradlew* scripts with the parameter *tasks*. Gradle will tell you some of the tasks that it can perform for you:

We've cut down the actual output, because there are many, many tasks that you can run by default.



```
File Edit Window Help EmacsFTW
$ ./gradlew tasks
Build tasks
-----
assemble - Assembles all variants of all applications and
            secondary packages.
build - Assembles and tests this project.
clean - Deletes the build directory.
compileDebugSources
mockableAndroidJar - Creates a version of android.jar that's
                    suitable for unit tests.

Install tasks
-----
installDebug - Installs the Debug build.
uninstallAll - Uninstall all applications.
uninstallDebug - Uninstalls the Debug build.

Verification tasks
-----
check - Runs all checks.
connectedAndroidTest - Installs and runs instrumentation tests
                       for all flavors on connected devices.
lint - Runs lint on all variants.
test - Run unit tests for all variants.

To see all tasks and more detail, run gradlew tasks --all

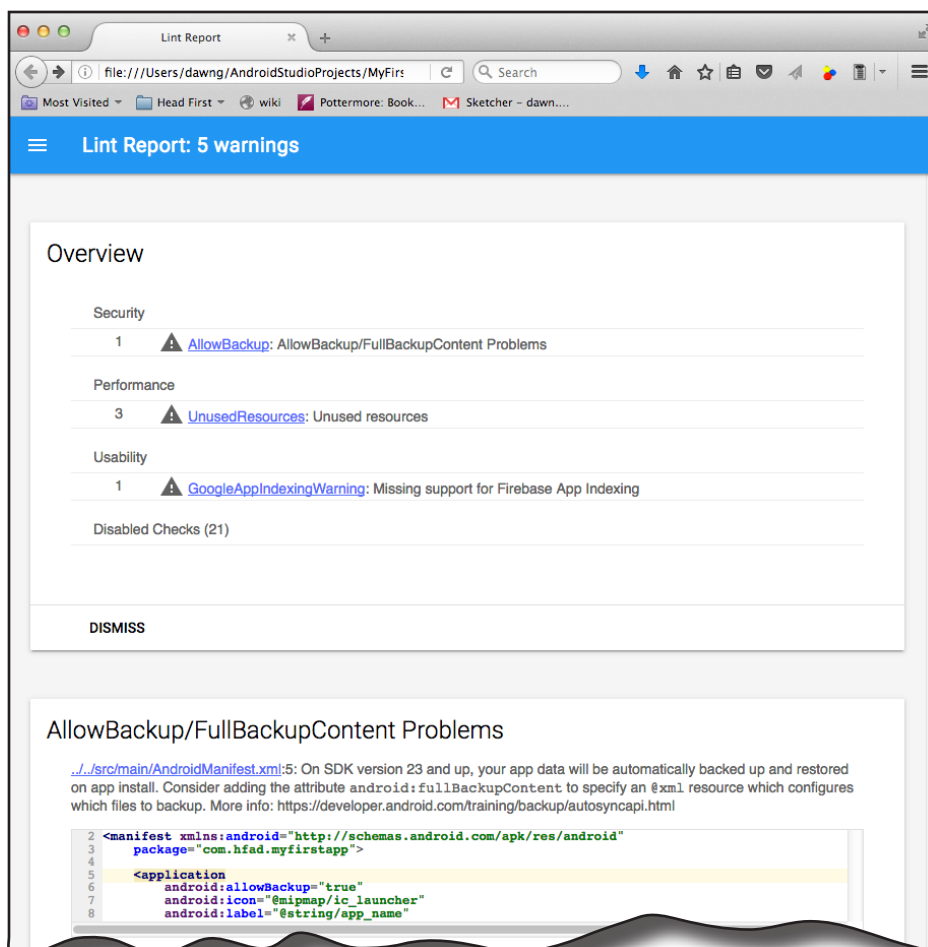
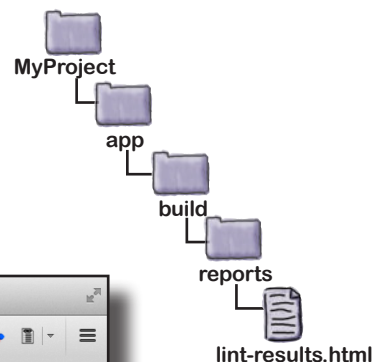
BUILD SUCCESSFUL

Total time: 6.209 secs
$
```

Let's take a quick tour of some of the most useful ones.

The check task

The check task performs static analysis on the source code in your application. Think of it as your coding buddy, who at a moment's notice can scoot through your files looking for coding errors. By default, the check task uses the *lint* tool to look for common Android programming errors. It will generate a report in `app/build/reports/lint-results.html`:



The clean installDebug task

This task will do a complete compile and install of your application on your connected device. You can obviously do this from the IDE, but it can be useful to do this from the command line if, for example, you want to automatically build your application on an Integration Server.

The androidDependencies task

For this task, Gradle will automatically pull down any libraries your application requires, and some of those libraries will automatically pull down other libraries, which might pull down other libraries and... well, you get the idea.

Even though your *app/build.gradle* file might only mention a couple of libraries, your application might need to install many dependent libraries for your app. So it's sometimes useful to see which libraries your application requires, and why. That's what the `androidDependencies` task is for: it displays a tree of all of the libraries in your app:

```
File Edit Window Help EmacsFTW
$ ./gradlew androidDependencies

Incremental java compilation is an incubating feature.
:app:androidDependencies
debug
+--- com.android.support:appcompat-v7:25.1.1@aar
|   +--- com.android.support:support-annotations:25.1.1@jar
|   +--- com.android.support:support-v4:25.1.1@aar
|       +--- com.android.support:support-compat:25.1.1@aar
|           \--- com.android.support:support-annotations:25.1.1@jar
|       +--- com.android.support:support-media-compat:25.1.1@aar
|           +--- com.android.support:support-annotations:25.1.1@jar
|           \--- com.android.support:support-compat:25.1.1@aar
|               \--- com.android.support:support-annotations:25.1.1@jar
|   +--- com.android.support:support-core-utils:25.1.1@aar
|       +--- com.android.support:support-annotations:25.1.1@jar
|       \--- com.android.support:support-compat:25.1.1@aar
...

```


gradlew <your-task-goes-here>

The real reason that Android apps are commonly built with Gradle is because it's easily extended. All Gradle files are written in Groovy, which is a general-purpose language designed to be run by Java. That means you can easily add your own custom tasks.

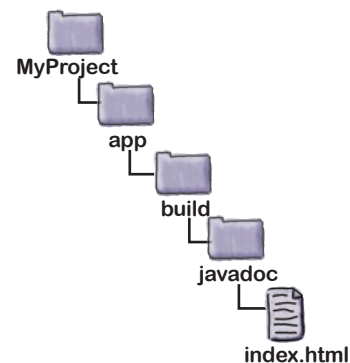
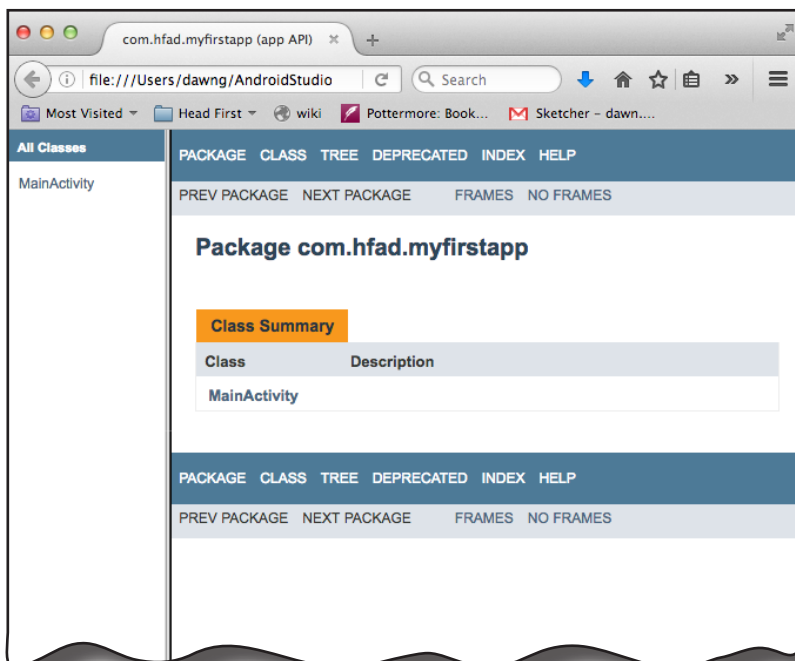
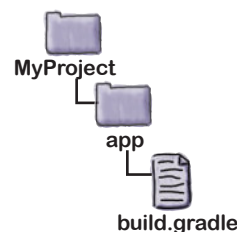
As an example, add the following code to the end of your *app/build.gradle* file:

```
task javadoc(type: Javadoc) {
    source = android.sourceSets.main.java.srcDirs
    classpath += project.files(android.getBootClasspath().join(File.pathSeparator))
    destinationDir = file("$project.buildDir/javadoc/")
    failOnError false
}
```

This code creates a new task called *javadoc*, which generates javadoc for your source code. You can run the task with:

```
./gradlew javadoc
```

The generated files will be published in *app/build/javadoc*:



Gradle plug-ins

As well as writing your own tasks, you can also install Gradle plug-ins. A plug-in can greatly extend your build environment. Want to write Android code in Clojure? Want your build to automatically interact with source control systems like Git? How about spinning up entire servers in Docker and then testing your application on them?

You can do these things, and many, many more by using Gradle plug-ins. For details on what plug-ins are available, see <https://plugins.gradle.org>.

The Android Runtime



Ever wonder how Android apps can run on so many kinds of devices? Android apps run in a virtual machine called the **Android runtime (ART)**, not the Oracle Java Virtual Machine (JVM). This means that your apps are quicker to start on small, low-powered devices and run more efficiently. In this appendix, we'll look at how ART works.

What is the Android runtime?

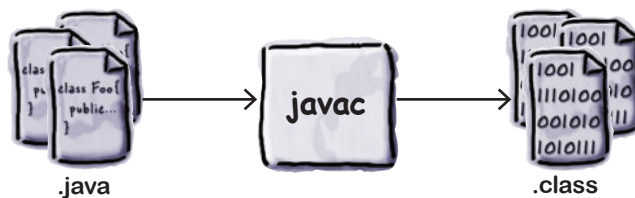
The Android Runtime (ART) is the system that runs your compiled code on an Android device. It first appeared on Android with the release of KitKat and became the standard way of running code in Lollipop.

ART is designed to run your compiled Android apps quickly and efficiently on small, low-powered devices. Android Studio uses the Gradle build system to do all the work of creating and installing apps for you, but it can be useful to understand what happens behind the scenes when you click the Run button. Let's see what really goes on.

You don't need to understand the info in this appendix in order to create cool Android apps. So if you're not into the nitty-gritty of what's going on behind the scenes when an Android device runs an app, feel free to skip this appendix.

Previously, Java code ran on the Oracle JVM

Java has been around for a very long time, and compiled Java applications have almost always been run on the Oracle Java Virtual Machine (JVM). In that scenario, Java source code gets compiled down to `.class` files. One `.class` file gets created for every Java class, interface, or enum in your source code:



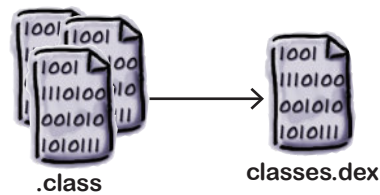
The `.class` files contain Java bytecodes, which can be read and executed by the JVM. The JVM is a software emulation of a Central Processing Unit (CPU), like the chip at the heart of your development machine. But because it's emulated, it can be made to work on almost any device. That's why Java code is designed to be written once, run anywhere.

So is that what happens on Android devices? Well...not quite. The Android Runtime performs the same kind of job as the JVM, but it does it in a very different way.

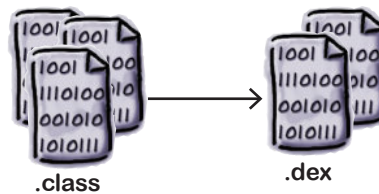
ART compiles code into DEX files

When you do Android development, your Java source code is compiled down into *.dex* files. A *.dex* file serves a similar purpose to a *.class* file, because it contains executable bytecodes. But instead of being JVM bytecodes, they are in a different format called **Dalvik**. DEX stands for **Dalvik EX**ecutable.

Rather than creating a *.dex* file for each and every class file, your app will normally be compiled down into a single file called *classes.dex*. That single *.dex* file will contain bytecodes for all of your source code, and for every library in your app.



The DEX format can only cope with 65,535 methods, so if your app contains a lot of code or some large libraries, your file might need to be converted into multiple *.dex* files:

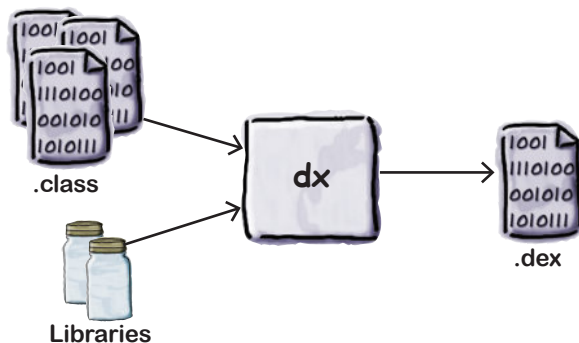


You can find out more about creating multi-DEX apps here:

<https://developer.android.com/studio/build/multidex.html>.

How DEX files get created

When Android builds your app, it uses a tool called dx, which stitches .class files into a DEX file:



It may seem a bit weird that the compilation process involves two stages of compilation: first to .class files, and then from .class files to the DEX format. Why doesn't Google just create a single tool that goes straight from .java source code to DEX bytecodes?

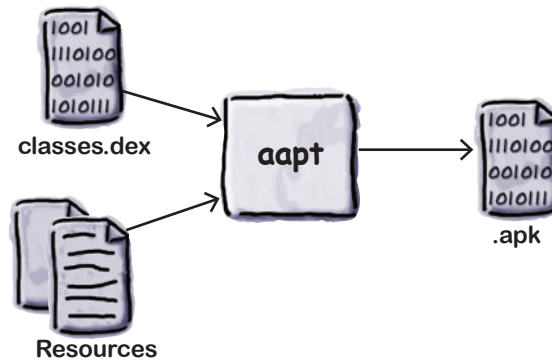
For a while Google was developing a compiler called JACK and an associated linker called JILL that could create DEX code straight from Java code, but there was a problem. Some Java tools don't just work at the source code level; they work directly with .class files, and modify the code that those files contain.

For example, if you use a code coverage tool to see which code your tests are actually running, the coverage tool will likely want to modify the contents of the generated .class files to add in additional bytecodes to keep track of the code as it's executed. If you used the JACK compiler, no .class files were generated.

So back in March 2017, Google announced that it was shelving JACK, and was putting all of its efforts into making the dx tool work really well with the latest Java .class formats. This has the additional advantage that any new Java language features—so long as they don't require new Java byte codes—will automatically be supported by Android.

DEX files are zipped into APK files

But Android apps aren't shipped around as *.dex* files. There's a whole bunch of other files that make up your app: images, sounds, metadata, and so on. All of these resources and your DEX bytecode is wrapped up into a single zip file called an Android Package or *.apk* file. The *.apk* file is created by another tool called the Android Asset Packing Tool, or *aapt*.

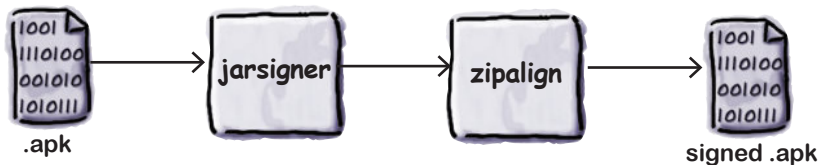


When you download an app from the Google Play Store, what actually gets downloaded is an APK file. In fact, when you run your app from Android Studio, the build system will first build an *.apk* file and then install it onto your Android device.

You may need to sign the .apk file

If you're going to distribute your app through the Google Play Store, you will need to sign it. Signing an app package means that you store an additional file in the *.apk* that is based on a checksum of the contents of the *.apk* and a separately generated private key. The *.apk* file uses the standard *jarsigner* tool that comes as part of Oracle's Java Development Kit. The *jarsigner* tool was created to sign *.jar* files, but it will also work with *.apk* files.

If you sign the *.apk* file, you will then also need to run it through a tool called *zipalign*, which will make sure that the compressed parts of the file are lined up on byte boundaries. Android wants them byte-aligned so it can read them easily without needing to uncompress the file.



Android Studio will do all of this for you if you choose Generate Signed APK from the Build menu.

So that's how your app gets converted from Java source code into an installable file. But how does it get installed and run on your device?

Say hello to the Android Debug Bridge (adb)

All communication between your development machine and your Android device takes place over the Android Debug Bridge. There are two sides to the bridge: a command-line tool on your dev machine called adb, and a daemon process on your Android device called adbd (the Android Debug Bridge Daemon).

The adb command on your development machine will run a copy of itself in the background, called the ADB server. The server receives commands over network port 5037, and sends them to the adbd process on the device. If you want to copy or read a file, install an app, or look at the logcat information for an app, then all of this information passes back and forth across the debug bridge.

So when the build system wants to install your APK file, it does so by sending a command like this to the debug bridge:

```
adb install bitsandpizzas.apk
```

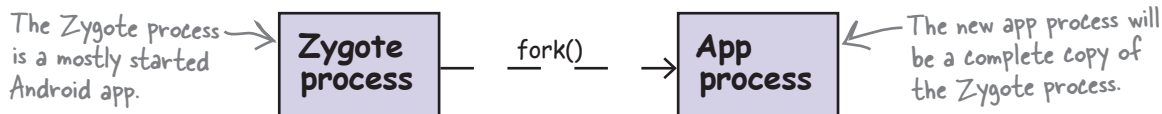
The file will then be transferred to a virtual device, or over a USB cable to a real device, and be installed into the `/data/app/` directory, where it will sit waiting for the app to be run.

How apps come alive: running your APK file

When your app is run, whether it's been started by a user pressing its launch icon or because the IDE has started it, the Android device needs to turn that `.apk` file into a process running in memory.

It does this using a process called Zygote. Zygote is like a half-started process. It's allocated some memory and already contains the main Android libraries. It has everything it needs, in fact, except for the code that's specific to your app.

When Android runs your app, it first creates a copy (a.k.a. fork) of the Zygote process, and then tells the copied process to load your application code. So why does Android leave this half-started process hanging around? Why not just start a fresh process from scratch for each app? It's because of performance. It can take Android a long time to create a new process from scratch, but it can fork a copy of an existing process in a split second.



Android converts the .dex code to OAT format

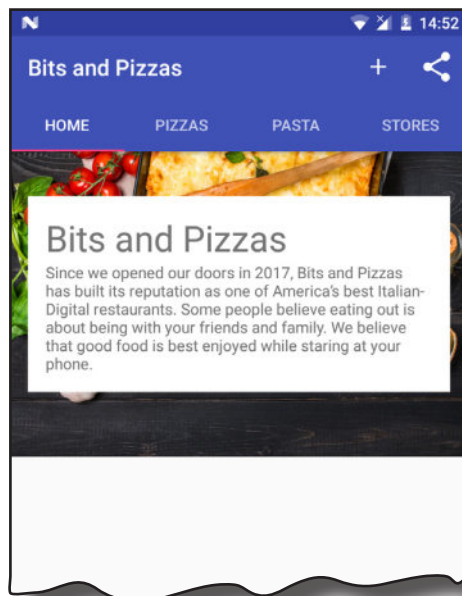
The new app process now needs to load the code that's specific to your app. Remember, your app code is stored in the *classes.dex* file inside your *.apk* package. So the *classes.dex* file is extracted from the *.apk* and placed into a separate directory. But rather than simply use the *classes.dex* file, Android will convert the Dalvik byte codes in *classes.dex* into native machine code. Technically, the *classes.dex* will be converted into an ELF shared object. Android calls this library format OAT, and calls the tool that converts the *classes.dex* file dex2oat.



The converted file is stored into a directory named something like this:

```
/data/dalvik-cache/data@app@com.hfad.bitsandpizzas@base.apk@classes.dex
```

This file can then be loaded as a native library by the application process, and the app appears on the screen.



appendix iv: adb

The Android Debug Bridge

What better gift for the girl who has everything than a new command-line tool?

Why, darling, that's so thoughtful.



In this book, we've focused on using an IDE for all your **Android needs**. But there are times when using a command-line tool can be plain useful, like those times when Android Studio can't see your Android device but you just *know* it's there. In this chapter, we'll introduce you to the **Android Debug Bridge (or adb)**, a command-line tool you can use to communicate with the emulator or Android devices.

adb: your command-line pal

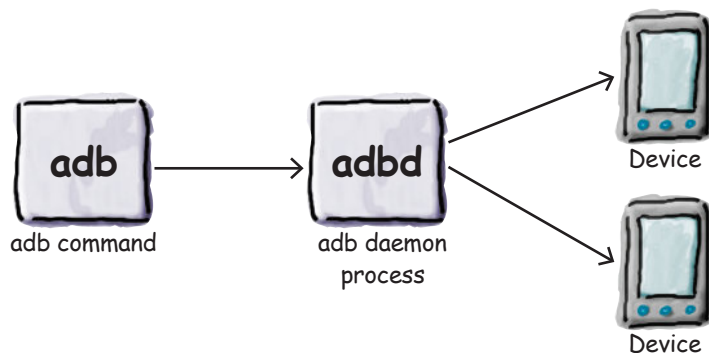
Every time your development machine needs to talk to an Android device, whether it's a real device connected with a USB cable, or a virtual device running in an emulator, it does so by using the **Android Debug Bridge (adb)**. The adb is a process that's controlled by a command that's also called adb.

The adb command is stored in the *platform-tools* directory of the Android System Developer's Kit on your computer. If you add the *platform-tools* directory to your PATH, you will be able to run adb from the command line.

In a terminal or at a command prompt, you can use it like this:

```
Interactive Session
$ adb devices
List of devices attached
emulator-5554      device
$
```

The `adb devices` command means “Tell me which Android devices you are connected to.” The adb command works by talking to an adb server process, which runs in the background. The adb server is sometimes called the *adb daemon* or *adb*. When you enter an adb command in a terminal, a request is sent to network port 5037 on your machine. The adb listens for commands to come in on this port. When Android Studio wants to run an app, or check the log output, or do anything else that involves talking to an Android device, it will do it via port 5037.



When the adb receives a command, it will forward it to a separate adb process that's running in the relevant Android device. This process will then be able to make changes to the Android device or return the requested information.

Sometimes, if the adb server isn't running, the adb command will need to start it:

```
Interactive Session
$ adb devices
* daemon not running. starting it now on port 5037 *
* daemon started successfully *
List of devices attached
emulator-5554      device
$
```

Likewise, if ever you plug in an Android device and Android Studio can't see it, you can manually kill the adb server and restart it:

```
Interactive Session
$ adb devices
List of devices attached
$ adb kill-server
$ adb start-server
* daemon not running. starting it now on port 5037 *
* daemon started successfully *
$ adb devices
List of devices attached
emulator-5554      device
$
```

By killing and restarting the server, you force adb to get back in touch with any connected Android devices.

Running a shell

Most of the time you won't use adb directly; you'll let an IDE like Android Studio do the work for you. But there are times when it can be useful to go to the command line and interact with your devices directly.

One example is if you want to run a shell on your device:

```
Interactive Session
$ adb shell
root@generic_x86:/ #
```

The `adb shell` command will open up an interactive shell directly on the Android device. If you have more than one device attached, you can indicate which device you mean with the `-s` option, followed by the name given by the `adb devices` command. For example, `adb -s emulator-5554 shell` will open a shell on the emulator.

Once you open a shell to your device, you can run a lot of the standard Linux commands:

```
Interactive Session
$ adb shell
root@generic_x86:/ # ls
acct
cache
charger
config
d
data
default.prop
dev
etc
file_contexts
....
|root@generic_x86:/ # df
Filesystem      Size      Used      Free     Blksize
/dev            439.8M    60.0K    439.8M    4096
/mnt/asec       439.8M    0.0K    439.8M    4096
/mnt/obb        439.8M    0.0K    439.8M    4096
/system         738.2M    533.0M   205.2M    4096
/data           541.3M    237.8M   303.5M    4096
/cache          65.0M     4.3M     60.6M    4096
/mnt/media_rw/sdcard 196.9M    4.5K    196.9M    512
/storage/sdcard 196.9M    4.5K    196.9M    512
root@generic_x86:/ #
```

Useful shell commands

If you open a shell to Android, you have access to a whole bunch of command line tools. Here are just a few:

Command	Description	Example (and what it does)
pm	Package management tool.	<pre>pm list packages (this lists all installed apps) pm path com.hfad.bitzandpizzas (find where an app is installed) pm -help (show other options)</pre>
ps	Process status.	<pre>ps (lists all processes and their IDs)</pre>
dexdump	Display details of an APK.	<pre>dexdump -d /data/app/com.hfad. bitzandpizzas-2/base.apk (disassemble an app)</pre>
lsuf	List a process's open files and other connections.	<pre>lsuf -p 1234 (show what process with id 1234 is doing)</pre>
screencap	Take a screenshot.	<pre>screencap -p /sdcard/screenshot.png (save current screenshot to /sdcard/screenshot.png, and get it off the device with adb pull /sdcard/screenshot.png)</pre>
top	Show busiest processes.	<pre>top -m 5 (show the top five processes)</pre>

Each of these examples works from an interactive shell prompt, but you can also pass them direct to the shell command from your development machine. For example, this command will show the apps installed on your device:

```
Interactive Session
$ adb shell pm list packages
```

Kill the adb server

Sometimes the connection can fail between your development machine and your device. If this happens, you can reset the connection by killing the adb server:

```
Interactive Session
$ adb kill-server
```

The next time you run an adb command, the server will restart and a fresh connection will be made.

Get the output from logcat

All of the apps running on your Android device send their output to a central stream called the logcat. You can see the live output from the logcat by running the `adb logcat` command:

```
Interactive Session
$ adb logcat
----- beginning of system
I/Vold ( 936): Vold 2.1 (the revenge) firing up
D/Vold ( 936): Volume sdcard state changing -1
(Initializing) -> 0 (No-Media)
W/DirectVolume( 936): Deprecated implied prefix pattern
detected, please use '/devices/platform/goldfish_mmc.0*'
instead
...
```

The logcat output will keep streaming until you stop it. It can be useful to run `adb logcat` if you want to store the output in a file. The `adb logcat` command is used by Android Studio to produce the output you see in the Devices/logcat panel.

Copying files to/from your device

The `adb pull` and `adb push` commands can be used to transfer files back and forth. For example, here we are copying the `/default.prop/` properties file into a local file called `1.txt`:

```
Interactive Session
$ adb pull /default.prop 1.txt
28 KB/s (281 bytes in 0.009s)
$ cat 1.txt
#
# ADDITIONAL_DEFAULT_PROPERTIES
#
ro.secure=0
ro.allow.mock.location=1
ro.debuggable=1
ro.zygote=zygote32
dalvik.vm.dex2oat-Xms=64m
dalvik.vm.dex2oat-Xmx=512m
dalvik.vm.image-dex2oat-Xms=64m
dalvik.vm.image-dex2oat-Xmx=64m
ro.dalvik.vm.native.bridge=0
persist.sys.usb.config=adb
$
```

And much, much more...

There are many, many commands that you can run using `adb`: you can back up and restore databases (very useful if you need to debug a problem with a database app), start the `adb` server on a different port, reboot machines, or just find out a lot of information about the running devices. To find out all the options available, just type `adb` on the command line:

```
Interactive Session
$ adb
Android Debug Bridge version 1.0.32
  -a interfaces for a connection      - directs adb to listen on all
  -d connected USB device           - directs command to the only
  one USB device is present.        returns an error if more than
  -e running emulator.              - directs command to the only
  returns an error if more than one emulator is ....
```


Speeding Things Up



Ever felt like you were spending all your time waiting for the emulator? There's no doubt that using the Android emulator is useful. It allows you to see how your app will run on devices other than the physical ones you have access to. But at times it can feel a little...sluggish. In this appendix, we'll explain why the emulator can seem slow. Even better, we'll give you a few tips we've learned for **speeding it up**.

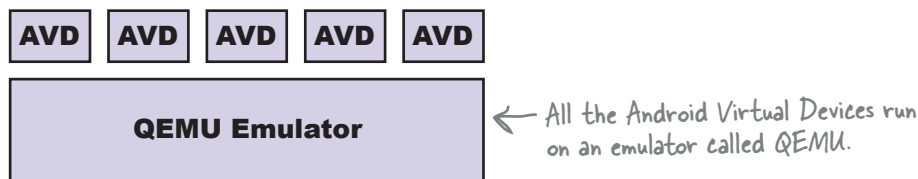
Why the emulator is so slow

When you're writing Android apps, you'll spend a lot of time waiting for the Android emulator to start up or deploy your code. Why is that? Why is the Android emulator so sloooooow? If you've ever written iPhone code, you know how fast the iPhone simulator is. If it's possible for the iPhone, then why not for Android?

There's a clue in the names: the iPhone *simulator* and the Android *emulator*.

The iPhone simulator simulates a device running the iOS operating system. All of the code for iOS is compiled to run natively on the Mac and the iPhone simulator runs at Mac-native speed. That means it can simulate an iPhone boot-up in just a few seconds.

The Android emulator works in a completely different way. It uses an open source application called QEMU (or Quick Emulator) to emulate the entire Android hardware device. It runs code that interprets machine code that's intended to be run by the device's processor. It has code that emulates the storage system, the screen, and pretty much every other piece of physical equipment on an Android device.



An emulator like QEMU creates a much more realistic representation of a virtual device than something like the iPhone simulator does, but the downside is that it has to do far more work for even simple operations like reading from disk or displaying something on a screen. That's why the emulator takes so long to boot up a device. It has to pretend to be every little hardware component inside the device, and it has to interpret every single instruction.

How to speed up your Android development

1. Use a real device

The simplest way to speed up your development process is by using a real device. A real device will boot up much faster than an emulated one, and it will probably deploy and run apps a lot more quickly. If you want to develop on a real device, you may want to go into “Developer options” and check the Stay Awake option. This will prevent your device from locking the screen, which is useful if you are repeatedly deploying to it.

2. Use an emulator snapshot

Booting up is one of the slowest things the emulator does. If you save a snapshot of the device while it’s running, the emulator will be able to reset itself to this state without having to go through the boot-up process. To use a snapshot with your device, open the AVD manager from the Android Studio menu by selecting Tools→Android→AVD Manager, edit the AVD by clicking on the Edit symbol, and then check the “Store a snapshot for faster startup” option.

This will save a snapshot of what the memory looks like when the device is running. The emulator will be able to restore the memory in this state without booting the device.

3. Use hardware acceleration

By default, the QEMU emulator will have to interpret each machine code instruction on the virtual device. That means it’s very flexible because it can pretend to be lots of different CPUs, but it’s one of the main reasons why the emulator is slow. Fortunately, there’s a way to get your development machine to run the machine code instructions directly. There are two main types of Android Virtual Device: ARM machines and x86 machines. If you create an x86 Android device and your development machine is using a particular type of Intel x86 CPU, then you can configure your emulator to run the Android machine code instructions directly on your Intel CPU.

You will need to install Intel’s Hardware Accelerated Execution Manager (HAXM). At the time of writing, you can find HAXM here:

<https://software.intel.com/en-us/android/articles/intel-hardware-accelerated-execution-manager>

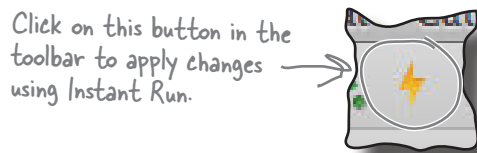
← If it’s moved, a quick web search should track it down.

HAXM is a hypervisor. That means it can switch your CPU into a special mode to run virtual machine instructions directly. However, HAXM will only run on Intel processors that support Intel Virtualization Technology. But if your development machine is compatible, then HAXM will make your AVD run much faster.

4. Use Instant Run

Since Android Studio 2.3, it's been possible to redeploy apps much more quickly using the Instant Run utility. This utility allows Android Studio to recompile just the files that have changed, and then patch the application currently running on the device. This means that instead of waiting for a minute or so while the application is recompiled and deployed, you can see your changes running in just a few seconds.

To use Instant Run, click on the Apply Changes option in the Run menu, or click on the lightning bolt icon in the toolbar:



There are a couple of conditions on using Instant Run. First, your app's target needs to be at least API 15. Second, you need to deploy to a device that runs API 21 or above. So long as you satisfy both these conditions, you'll find that Instant Run is by far the fastest way of getting your code up and running.

The Top Ten Things (we didn't cover)

Oh my, look at
the tasty treats
we have left...



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. Distributing 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 releasing 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 and modifying *AndroidManifest.xml* so that only devices that are able to run your app are able to 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 publicizing your app, selling it, and distributing it.

To release your app on the Play Store, you need to register for a publisher account and use the Developer Console to publish your app. You can find further details here:

<http://developer.android.com/distribute/googleplay/start.html>

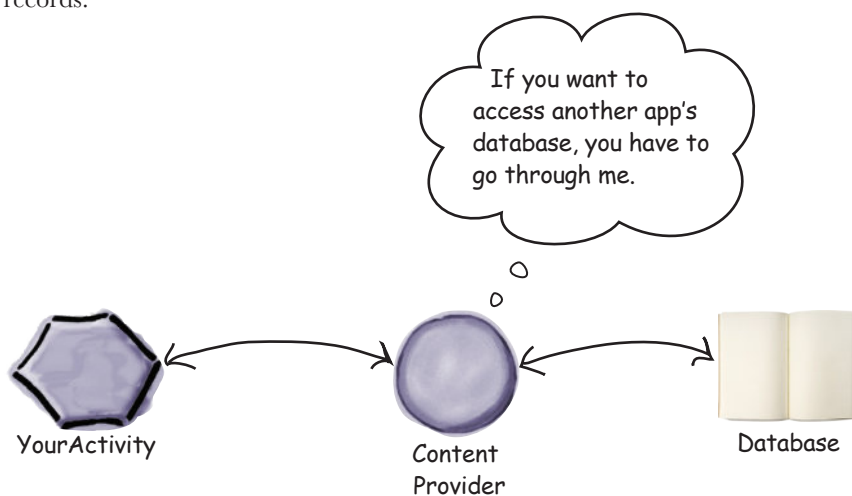
For ideas on how to best target your app to your users and build a buzz about it, we suggest you explore the documents here:

<http://developer.android.com/distribute/index.html>

2. Content providers

You've seen how to use intents to start activities in other apps. As an example, you can start the Messaging app to send the text you pass to it. But what if you want to use another app's data in your own app? For example, what if you want to use Contacts data in your app to perform some task, or insert a new Calendar event?

You can't access another app's data by interrogating its database. Instead, you use a **content provider**, which is an interface that allows apps to share data in a controlled way. It allows you to perform queries to read the data, insert new records, and update or delete existing records.



If you want other apps to use your data, you can create your own content provider.

You can find out more about the concept of content providers here:

<http://developer.android.com/guide/topics/providers/content-providers.html>

Here's a guide on using Contacts data in your app:

<http://developer.android.com/guide/topics/providers/contacts-provider.html>

And here's a guide on using Calendar data:

<http://developer.android.com/guide/topics/providers/calendar-provider.html>

3. Loaders

If you do a lot of work with databases or content providers, sooner or later you'll encounter **loaders**. A loader helps you load data so that it can be displayed in an activity or fragment.

Loaders run on separate threads in the background, and make it easier to manage threads by providing callback methods. They persist and cache data across configuration changes so that if, for example, the user rotates their device, the app doesn't create duplicate queries. You can also get them to notify your app when the underlying data changes so that you can deal with the change in your views.

The Loader API includes a generic `Loader` class, which is the base class for all loaders. You can create your own loader by extending this class, or you can use one of the built-in subclasses: `AsyncTaskLoader` or `CursorLoader`. `AsyncTaskLoader` uses an `AsyncTask`, and `CursorLoader` loads data from a content provider.





You can find out more about loaders here:

<https://developer.android.com/guide/components/loaders.html>

4. Sync adapters

Sync adapters allow you to synchronize data between an Android device and a web server. This allows you to do things like back up the user's data to a web server, for instance, or transfer data to a device so that it can be used offline.

Sync adapters have a number of advantages over designing your own data transfer mechanism.

-  They allow you to automate data transfer based on specific criteria—for example, time of day or data changes.
-  They automatically check for network connectivity, and only run when the device has a network connection.
-  Sync adapter-based data transfers run in batches, which helps improve battery performance.
-  They let you add user credentials or a server login to the data transfer.

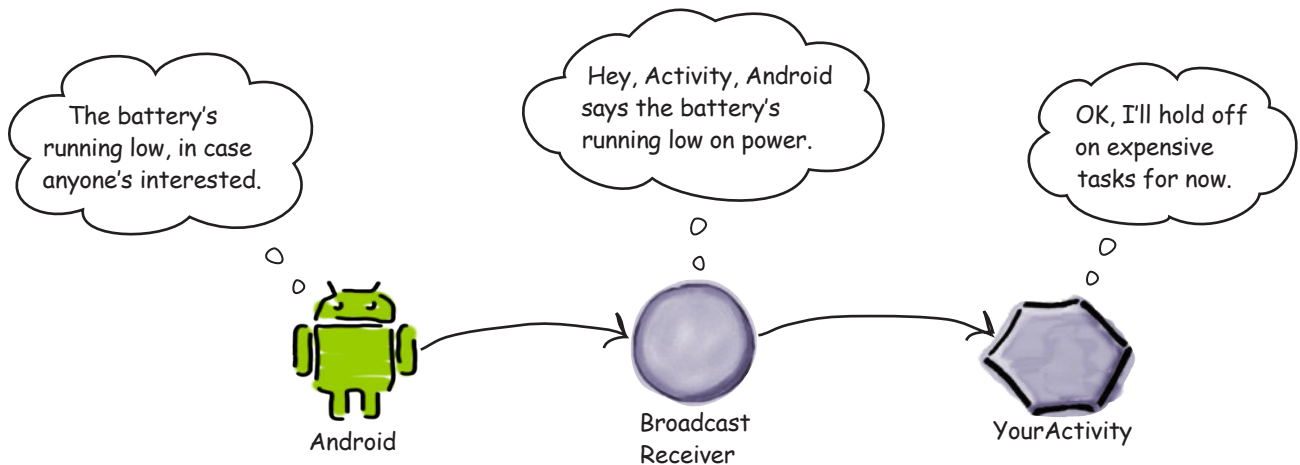
You can find out how to use sync adapters here:

<https://developer.android.com/training/sync-adapters/index.html>

5. Broadcasts

Suppose you want your app to react in some way when a system event occurs. You may, for example, have built a music app, and you want it to stop playing music if the headphones are removed. How can your app tell when these events occur?

Android broadcasts system events when they occur, including things like the device running low on power, a new incoming phone call, or the system getting booted. You can listen for these events by creating a **broadcast receiver**. Broadcast receivers allow you to subscribe to particular broadcast messages so that your app can respond to particular system events.



Your app can also send custom broadcast messages to notify other apps of events.

You can find out more about broadcasts here:

<https://developer.android.com/guide/components/broadcasts.html>

6. The WebView class

If you want to provide your users with access to web content, you have two options. The first option is to open the web content with an external app like Chrome or Firefox. The second option is to display the content inside your app using the `WebView` class.

The `WebView` class allows you to display the contents of a web page inside your activity's layout. You can use it to deliver an entire web app as a client application, or to deliver individual web pages. This approach is useful if there's content in your app you might need to update, such as an end-user agreement or user guide.

You add a `WebView` to your app by including it in your layout:

```
<WebView xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/webview"
    android:layout_width="match_parent"
    android:layout_height="match_parent" />
```

You then tell it which web page it should load by calling its `loadUrl()` method:

```
WebView webView = (WebView) findViewById(R.id.webview);
webView.loadUrl("http://www.oreilly.com/");
```

You also need to specify that the app must have Internet access by adding the `INTERNET` permission to *AndroidManifest.xml*:

```
<manifest ... >
    <uses-permission android:name="android.permission.INTERNET" />
    ...
</manifest>
```

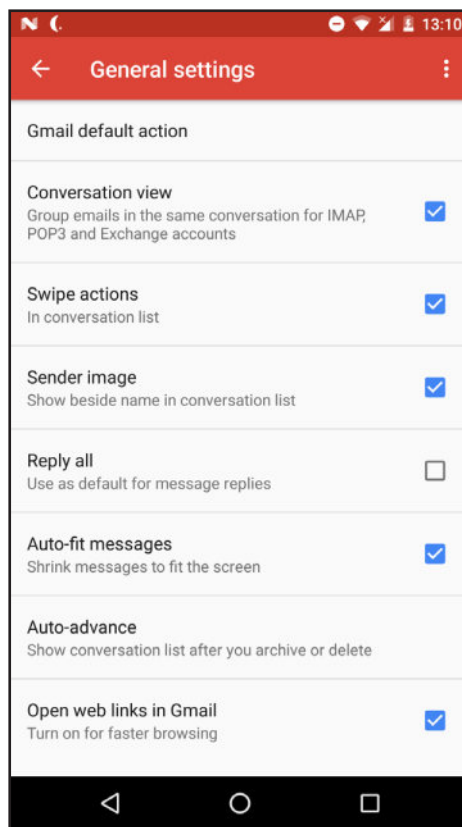
You can find out more about using web content in your apps here:

<http://developer.android.com/guide/webapps/index.html>

7. Settings

Many apps include a settings screen so that the user can record their preferences. As an example, an email app may allow the user to specify whether they want to see a confirmation dialog before sending an email:

This is the settings screen for the Gmail app.



You build a settings screen for your app using the Preferences API. This allows you to add individual preferences, and record a value for each preference. These values are recorded in a shared preferences file for your app.

You can find out more about creating settings screens here:

<https://developer.android.com/guide/topics/ui/settings.html>

8. Animation

As Android devices increasingly take advantage of the power of their built-in graphics hardware, animation is being used more and more to improve the user's app experience.

There are several types of animation that you can perform in Android:

Property animation

Property animation relies on the fact that the visual components in an Android app use a lot of numeric properties to describe their appearance. If you change the value of a property like the height or the width of a view, you can make it animate. That's what property animation is: smoothly animating the properties of visual components over time.

View animations

A lot of animations can be created declaratively as XML resources. So you can have XML files that use a standard set of animations (like scaling, translation, and rotation) to create effects that you can call from your code. The wonderful thing about declarative view animations is that they are decoupled from your Java code, so they are very easy to port from one app project to another.

Activity transitions

Let's say you write an app that displays a list of items with names and images. You click on an item and you're taken to a detail view of it. The activity that shows you more detail will probably use the same image that appeared in the previous list activity.

Activity transitions allow you to animate a view from one activity that will also appear in the next activity. So you can make an image from a list smoothly animate across the screen to the position it takes in the next activity. This will give your app a more seamless feel.

To learn more about Android animations see:

<https://developer.android.com/guide/topics/graphics/index.html>

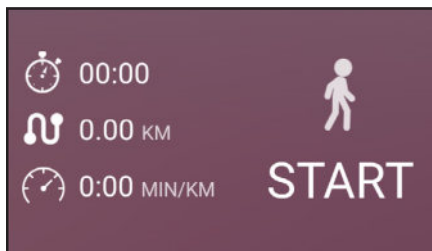
To learn about activity transitions and material design, see:

<https://developer.android.com/training/material/animations.html>

9. App widgets

An app widget is a small application view that you can add to other apps or your home screen. It gives you direct access to an app's core content or functionality from your home screen without you having to launch the app.

Here's an example of an app widget:



← This is an app widget. It gives you direct access to the app's core functionality without you having to launch the app.

To find out how you create your own app widgets, look here:

<http://developer.android.com/guide/topics/appwidgets/index.html>

10. Automated testing

All modern development relies heavily on 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.

There are many ways to automatically test your app, but they generally fall into two categories: **unit testing** and **on-device testing** (sometimes called *instrumentation testing*).

Unit tests

Unit tests run on your development machine, and they check the individual pieces—or units—of your code. The most popular unit testing framework is **JUnit**, and Android Studio will probably bundle JUnit into your project. Unit tests live in the `app/src/test` folder of your project, and a typical test method looks something like this:

```
@Test
public void returnsTheCorrectAmberBeers() {
    BeerExpert beerExpert = new BeerExpert();
    assertEquals(new String[]{"Jack Amber", "Red Moose"},
        beerExpert.getBrands("amber").toArray());
}
```

You can find more out about JUnit here:

<http://junit.org>

On-device tests

On-device tests run inside an emulator or physical device and check the fully assembled app. They work by installing a separate package alongside your app that uses a software layer called *instrumentation* to interact with your app in the same way as a user. An increasingly popular framework for on-device testing is called **Espresso**, and Android Studio will probably bundle it in to your project. On-device tests live in the `app/src/androidTest` folder, and Espresso tests look something like this:

```
@Test
public void ifYouDoNotChangeTheColorThenYouGetAmber() {
    onView(withId(R.id.find_beer)).perform(click());
    onView(withId(R.id.brands)).check(matches(withText(
        "Jail Pale Ale\nGout Stout\n")));
}
```

When you run an on-device test, you will see the app running on your phone or tablet and responding to keypresses and gestures just as if a person were using it.

You can find out more about Espresso testing here:

<https://developer.android.com/training/testing/ui-testing/espresso-testing.html>



Index

A

- action bar, 314. *See also* app bar
- ActionBarActivity class, 298
- ActionBar class, 306, 314
- ActionBarDrawerToggle class, 607
- actions
 - about, 100
 - in app bar
 - adding, 315–317, 319–325
 - icons for, 320, 322
 - menu for, 321, 323
 - method for, 324–325
 - title for, 320, 322
 - category specified with, 106, 117
 - determining activities for, 105–106
 - share actions, 332–335
 - specifying in an intent, 100–104
 - types of, 101
- activities
 - about, 2, 12, 31, 38, 120–121
 - backward compatibility for, 295, 298
 - calling custom Java classes from, 71–74
 - calling methods from, 59–62
 - class hierarchy for, 139, 297
 - class name for, 85
 - converting to fragments, 438–449
 - creating, 13–14, 61, 67–68
 - declaring in manifest file, 85
 - default, 41
 - editing. *See* code editor; design editor
 - location in project, 17
 - main, specifying, 84, 120
 - multiple
 - chaining, 78
 - creating, 78–83
 - passing text to, 90–95
 - retrieving text from, 96–97
 - starting with intents, 86–89
 - navigating. *See* navigation
 - organizing, 249–255, 290–291
 - category activities, 249–250, 252–255, 267–268, 277–278
 - detail/edit activities, 249–250, 253–255, 279–283
 - top-level activities, 249–250, 252–255
 - in other apps
 - determining from actions, 105–106
 - none available, 117
 - starting with intents, 99–104
 - users always choosing, 112–117
 - users choosing default, 111
 - users choosing if multiple, 100, 104, 111
 - parent activity, 328
 - states of. *See* activity lifecycle
- Activity class, 61, 139, 297
- activity lifecycle
 - about, 137–138, 146–147
 - class hierarchy for, 139
 - compared to fragment lifecycle, 439
 - methods associated with, 137–139, 147, 167, 439
 - states in
 - based on app’s focus, 154–157
 - based on app’s visibility, 146–147
 - list of, 137–138, 146–147
 - saving and restoring, 140–141, 145
- ActivityNotFoundException, 103, 117
- activity transitions, 868
- adapters
 - array. *See* ArrayAdapter class
 - decoupling, with interface, 572–574
 - recycler view. *See* recycler view adapter
 - sync adapters, 864
- AdapterView class, 261, 269
- adb (Android Debug Bridge)
 - about, 846, 850–851
 - copying files, 855
 - killing adb server, 854
 - logcat output, 854
 - running a shell, 852–853

- ADD COLUMN command, 650
 - addDrawerListener() method, DrawerLayout, 607
 - ALTER TABLE command, 649
 - Android application package. *See* APK files
 - Android apps. *See also* projects
 - about, 120–121
 - activities in. *See* activities
 - adapting for different device sizes, 340–341, 394–398, 402–412
 - app bar for. *See* app bar
 - back button functionality for, 413–415
 - building, 7–10, 13–15, 39
 - configuration of, affected by rotation, 134–136, 145, 156
 - designing, 248–250
 - devices supported by, 10, 340–341. *See also* fragments
 - distributing, 862
 - examples of. *See* examples
 - focus/visibility of, 146–147, 154–157
 - icons for, 84, 299
 - Java for. *See* Java
 - languages used in, 4
 - layouts for. *See* layouts
 - manifest file for. *See* AndroidManifest.xml file
 - minimum SDK for, 10–11
 - navigating. *See* navigation
 - package name for, 9, 84
 - processes used by, 120–121, 133
 - resource files for. *See* resource files
 - running activities in other apps, 100–108
 - running in emulator, 23–30, 49, 117
 - running on physical devices, 109–111, 117
 - settings screen for, 867
 - structuring, 249–250, 252–255, 290–291
 - theme for. *See* themes
 - Android debug bridge. *See* ADB
 - Android devices
 - about, 2
 - adapting apps for different sizes of, 340–341, 394–398, 402–412. *See also* fragments
 - API level on. *See* Android SDK (API level)
 - rotation of
 - configuration affected by, 134–136, 145, 156
 - saving activity state for, 140–141
 - saving fragment state for, 427–431
 - running apps on, 109–111, 117
 - virtual, for testing. *See* AVD (Android virtual device)
- Android emulator
 - about, 23
 - performance of, 858–860
 - running apps in, 23–30, 49, 117
 - AndroidManifest.xml file
 - about, 17, 84–85
 - activity hierarchy in, 328
 - app bar attributes in, 299–301, 318–319
 - intent filters in, 105–106
 - language settings in, 172
 - main activity in, 120, 132
 - permissions needed in, 791
 - screen size settings in, 403
 - started services in, 745
 - themes in, 589
 - Android platform
 - about, 2–3
 - versions of. *See* Android SDK (API level)
 - Android Runtime (ART), 3, 30, 842–843
 - Android SDK (API level)
 - about, 5
 - features specific to, 171, 173, 293, 297, 298, 314, 328
 - libraries for, 16, 294
 - list of, 11
 - permissions affected by, 791, 794
 - specifying for AVD, 25, 400
 - specifying minimum for apps, 10
 - Android Studio
 - about, 5, 7
 - alternatives to, 7
 - console in, 28
 - editors. *See* code editor; design editor
 - installing, 6
 - projects, creating, 8–10
 - system requirements, 6
 - Android virtual device. *See* AVD
 - animated toolbar. *See* collapsing toolbar; scrolling toolbar
 - animation, 868
 - API level. *See* Android SDK (API level)
 - APK files, 27, 30, 845–846
 - app bar
 - about, 291–293
 - activity titles in, 317
 - adding actions to, 315–317, 319–325
 - attributes in manifest file for, 299–301, 318–319
 - icon in, 299
 - label for, 299, 318–319

- removing, 308
- replacing with toolbar, 292, 306–313
- sharing content on, 331–335
- tabs in, 499–500
- themes required for, 293, 296–298, 299–300
- Up button for, 327–330
- AppBarLayout element, 499–500, 518–519
- AppCompatActivity class, 297–298, 307
- AppCompat Library. *See* v7 AppCompat Library
- app folder, 17
- application framework, 3. *See also* APIs
- application name, 9
- applications, core. *See* core applications
- app_name string resource, 51
- apps. *See* Android apps
- AppTheme attribute, 589–590
- app widgets, 869
- ArrayAdapter class, 269–271, 275, 287, 375–377
- @array reference, 57
- array resources, 56–57, 210, 259
- ART (Android Runtime), 3, 30, 842–843
- AsyncTask class, 724, 729, 731, 737
- ?attr reference, 309
- automated testing, 870
- AVD (Android virtual device)
 - about, 23
 - compared to design editor, 49
 - creating for smartphone, 24–26
 - creating for tablet, 399–401
- AWT, 4

B

- Back button
 - compared to Up button, 327
 - enabling, 413–415
- background, services running in. *See* services
- background thread, 720–731, 736
- back stack, 414–415
- backward compatibility, for activities, 295, 298
- Bates, Bert (Head First Java), 4
- Beer Adviser app, 38–76
 - activities, 61–69
 - button, 59–60
 - Java class, 70–74
 - layout, 41–48
 - project, 40
 - spinner, 56–58
 - String resources, 50–54
- Beighley, Lynn (Head First SQL), 675
- biases for views, 231
- Binder class, 786
- bindService() method, activity, 778
- blueprint tool
 - about, 226
 - aligning views, 238
 - biases, setting, 231
 - centering views, 230
 - horizontal constraints, 227
 - inferring constraints, 240–241, 245
 - margins, setting, 228
 - size of views, changing, 232–233
 - vertical constraints, 228
 - view properties, editing, 241
 - widgets, adding, 226, 240, 242–243
- books and publications
 - Head First Java (Sierra; Bates), 4
 - Head First SQL (Beighley), 675
- bound services
 - about, 740, 768
 - binding to an activity, 771, 774–778
 - calling methods from, 780–785
 - in combination with started services, 786
 - compared to started services, 786
 - creating, 770, 772
 - displaying results of, 773
 - lifecycle of, 787–788
 - other apps using, 786
- broadcasts, 865
- build folder, 17
- build.gradle files, 834–835
- built-in services
 - about, 740
 - location. *See* Location Services
 - notification. *See* notification service
- Bundle class
 - about, 145
 - restoring state data from, 141, 144
 - saving state data in, 140, 143
- Button element. *See also* FAB (floating action button)
 - about, 203
 - adding, 43

- calling methods
 - in activity, 60–62
 - in fragment, 452–460
- code for, 44–47, 80
- images added to, 213
- widget for, in blueprint tool, 226

C

- CalledFromWrongThreadException, 127
- cardCornerRadius attribute, card view, 543
- cardElevation attribute, card view, 543
- CardView element, 543
- CardView Library, 294, 542
- card views
 - about, 542
 - adding data to, 545, 550
 - creating, 543–544
 - displaying in recycler view, 558, 560
- CatChat app, 581–620
 - about, 581–584
 - Drafts option, 586
 - Feedback option, 592
 - header for, 594–595
 - Help option, 591
 - Inbox option, 585
 - menu for, 596–601
 - SentItems option, 587
 - Trash option, 588
- category activities, 249–250, 252–255, 267–268, 277–278, 290
- centering views, 230
- chaining activities, 78
- changeCursor() method, cursor adapter, 715–717
- checkableBehavior attribute, 598
- CheckBox element, 206–207, 696–700
- checkSelfPermission() method, ContextCompat, 794, 800
- check task, Gradle, 837
- choosers, 112–117
- classes. *See* Java classes
- clean task, Gradle, 837
- closeDrawer() method, DrawerLayout, 614
- close() method, cursor, 672, 683
- close() method, SQLiteDatabase, 672, 683

- code editor
 - about, 18, 32
 - activity file in, 21–22
 - layout file in, 19–20, 41, 44–46
- collapsing toolbar, 507, 517–525
- CollapsingToolbarLayout element, 518–521
- color resources, 304
- colors, for themes, 303
- columnCount attribute, GridLayout, 823
- company domain, 9
- constraint layout
 - about, 223
 - aligning views, 238
 - alternatives to, 245
 - biases, setting, 231
 - centering views, 230
 - horizontal constraints, adding, 227
 - inferring constraints, 240–241, 245
 - library for, 224
 - margins, setting, 228–229
 - size of views, changing, 232–233
 - specifying in main activity, 225
 - vertical constraints, adding, 228
 - widgets, adding, 226, 240, 242–243
 - XML code for, 229
- Constraint Layout Library, 224, 294
- contentDescription attribute, ImageView, 212
- content providers, 863
- contentScrim attribute, CollapsingToolbarLayout, 523
- ContentValues object, 632, 645
- Context class, 139, 752
- ContextThemeWrapper class, 139
- ContextWrapper class, 139, 752
- CoordinatorLayout element, 508–511, 518, 527
- core applications, 3
- core libraries, 3. *See also* libraries
- createChooser() method, Intent, 112–117
- CREATE TABLE command, 631
- Cursor class, 624
- cursors, SQLite
 - about, 624, 663
 - adapter for, 679–681, 688–689, 715
 - closing, 672, 682–683
 - creating, 663–666, 678
 - navigating to records in, 670–671

refreshing, 714–718
 retrieving values from, 672–674
 custom Java classes. *See* Java classes

D

data
 adapters for. *See* adapters
 from other apps, content providers for, 863
 loaders for, 864
 sharing with other apps. *See* share action provider
 storing as string resources, 259
 storing in classes, 256, 268–271, 360
 web content, 866

database. *See* SQLite database

delete() method, SQLiteDatabase, 647

density-independent pixels (dp), 171

design editor. *See also* blueprint tool
 about, 18, 32, 42
 adding GUI components, 43
 compared to AVD, 49
 XML changes reflected in, 48

Design Support Library
 about, 294, 506
 AppBarLayout element, 499–500
 collapsing toolbar, 517–525
 FAB (floating action button), 507, 526–529
 navigation drawers, 584
 scrolling toolbar, 508–515
 snackbar, 507, 526, 530–533
 TabLayout element, 499, 501

detail/edit activities, 249–250, 253–255, 279–283, 290

development environment. *See* Android SDK; Android Studio

devices. *See* Android devices

DEX files, 843–845, 847

dimension resources, 174

dimens.xml file, 174

distributing apps, 862

doInBackground() method, AsyncTask, 724, 726, 731

dp (density-independent pixels), 171

drawable attributes, Button, 213
 @drawable reference, 212

drawable resource folders. *See* image resources

DrawerLayout element, 602–603. *See also* navigation drawers

drop-down list. *See* Spinner element

DROP TABLE command, 650

dynamic fragments. *See* fragments: dynamic

E

editors. *See* blueprint tool; code editor; design editor

EditText element
 about, 80, 178, 202
 code for, 80
 hint, 178

email app example. *See* CatChat app

email grid layout example, 825–832

emulator. *See* Android emulator

entries attribute, ListView, 259

entries attribute, Spinner, 57, 210

Espresso tests, 871

event listeners
 about, 199
 for card views, 572–576
 compared to onClick attribute, 264
 for ListView element, 261–263
 for fragments, 455–460
 for list fragments, 385–387
 for ListView element, 708, 710–712
 location listeners, 792
 for navigation drawers, 608
 for snackbar actions, 530

examples
 Beer Adviser app. *See* Beer Adviser app
 CatChat app. *See* CatChat app
 email grid layout, 825–832
 Joke app. *See* Joke app
 My Constraint Layout app. *See* My Constraint Layout app
 My First App. *See* My First App
 My Messenger app. *See* My Messenger app
 Odometer app. *See* Odometer app
 Pizza app. *See* Pizza app
 source code for, xxxvi
 Starbuzz Coffee app. *See* Starbuzz Coffee app
 Stopwatch app. *See* Stopwatch app
 Workout app. *See* Workout app

execSQL() method, SQLiteDatabase, 631, 650

execute() method, AsyncTask, 731

F

FAB (floating action button), 507, 526–529

findViewById() method, view, 63–64, 68

FloatingActionButton element, 527

focus

- activities having, 154–157

- views handling, 199

FragmentActivity class, 354

fragment element, 352, 463

fragment lifecycle, 365–366, 439

fragment manager, 362, 419

FragmentManager class, 491–493

fragments

- about, 342

- activities using

- adding fragment to, 352–353

- button linking main to detail activity, 346–347

- creating, 345

- interactions with, 359–369

- referencing fragment from, 363, 367

- setting values for, 367–369

- adding to project, 348–349

- app structure for, 343–344

- back button functionality with, 413–415

- code for, 349–350

- converting activities to, 438–449

- data for, 360

- dynamic

- about, 434

- adding, 435–449, 469–477

- calling methods from, 450–460

- code for, 440–446, 458–460

- layout for, 447–449

- ID for, 361–362

- layouts for, 349–351, 408–410

- list fragments

- connecting data to, 375–377

- connecting to detail activity, 384–389

- creating, 372–374

- displaying in main activity, 378–381

- listener for, in main activity, 385–387

- methods associated with, 365–366, 439

- nested, 474–477

- replacing programmatically, 416–425

- state of, saving, 427–431, 464–467

- swiping through, 489–493

- for tab navigation, 483–488

- v7 AppCompat Library for, 345, 354

- view for, creating, 350

FragmentManager class, 491, 493

fragment transactions, 419–423, 463–467, 472–475

FrameLayout element

- about, 188–190, 193

- gravity for view placement, 190

- height, 188

- nesting, 191–192

- order of views in, 190

- replacing fragments programmatically using, 416–425, 464–467, 471

- width, 188

G

getActivity() method, PendingIntent, 758

getBestProvider() method, LocationManager, 793

getCheckedRadioButtonId() method, RadioGroup, 208

getChildFragmentManager() method, fragment, 474–475

getContext() method, LayoutInflater, 376

getCount() method, FragmentPagerAdapter, 491–492

getFragmentManager() method

- activity, 362

- fragment, 472–473

getIntent() method, activity, 92, 96, 280

getIntExtra() method, Intent, 92

getItemCount() method, RecyclerView.Adapter, 547

getItem() method, FragmentPagerAdapter, 491–492

get*() methods, cursor, 672

getReadableDatabase() method, SQLiteOpenHelper, 662, 678

getSelectedItem() method, view, 67, 69, 210

getStringExtra() method, Intent, 92, 96

getString() method, string resource, 115

getSupportActionBar() method, activity, 329

getSupportFragmentManager() method, activity, 362

getText() method, EditText, 202

getView() method, fragment, 367, 370

getWritableDatabase() method, SQLiteOpenHelper, 662

GPS location provider, 793

Gradle build tool

- about, 7, 834

- build.gradle files, 834–835

- check task, 837
- custom tasks, 839
- dependencies used by, 838
- gradlew and gradlew.bat files, 836
- plugins, 840
- gradlew and gradlew.bat files, 836
- gravity attribute, view, 182–183
- GridLayout element
 - about, 823
 - adding views to, 824, 827–829
 - creating, 825–832
 - dimensions, defining, 823, 826
- GridLayoutManager class, 556–557
- group element, 598–599
- GUI components. *See also* views; specific GUI components
 - adding, 43
 - inherited from View class, 44, 197–198
 - referencing, 63–64

H

- Handler class, 128–129
- HAXM (Hardware Accelerated Execution Manager), 859
- headerLayout attribute, 602
- Head First Java (Sierra; Bates), 4
- Head First SQL (Beighley), 675
- heads-up notification, 755, 757
- hint attribute, EditText, 178, 202
- horizontal constraints, 227
- HorizontalScrollView element, 215

I

- IBinder interface, 786
- IBinder object, 770–771, 776
- icon attribute, 299, 322
- icons
 - for actions in app bar, 320
 - for app, default, 84, 299
 - built-in icons, 597
- id attribute, view, 44, 175
- IDE
 - Android Studio as. *See* Android Studio
 - not using, 7
- ImageButton element, 214

- image resources. *See also* icons; mipmap resource folders
 - adding, 189, 211, 257, 512
 - for Button elements, 213
 - for ImageButton elements, 214
 - for ImageView element, 212, 258
 - for navigation drawer header, 594
 - R.drawable references for, 212, 257
 - resolution options, folders for, 211
- ImageView element, 211–212, 258, 523
- include tag, 312, 314
- inflate() method, LayoutInflater, 350
- inflators, layout, 350, 365
- inputType attribute, EditText, 202
- insert() method, SQLiteDatabase, 632–633
- installDebug task, Gradle, 837
- Instant Run, 860
- IntelliJ IDEA, 5
- intent filter, 105–106
- intents
 - about, 86
 - alternatives to, 91
 - category for, 106, 117
 - creating, 86–87, 277
 - implicit and explicit, 101, 105, 117
 - passing text using, 90–95
 - resolving activities and actions, 105–106
 - retrieving data from, 280–282
 - retrieving text from, 96–97
 - sharing content using, 331
 - starting activities, 86–89
 - starting activities in other apps, 99–104
- IntentService class, 742, 744, 752
- internationalization. *See* localization
- isChecked() method, CheckBox, 206

J

- Java
 - about, 2
 - activities. *See* activities
 - required knowledge of, xxxvi, 4
 - source file location, 16–17
- Java classes
 - about, 38
 - calling from activities, 71–74
 - custom, creating, 70, 256
 - data stored in, 256, 268–271

java folder, 17
Java Virtual Machine (JVM), 842
JDBC, 624
Joke app
 logging messages, 743–746
 notifications, 755–766
 started services, 741–766
JUnit, 870
JVM (Java Virtual Machine), 842

L

label attribute, application, 299, 314, 318
layout_above attribute, RelativeLayout, 822
layout_align* attributes, RelativeLayout, 820, 822
layout_behavior attribute, ViewPager, 510, 519
layout_below attribute, RelativeLayout, 822
layout_center* attributes, RelativeLayout, 820
layout_collapseMode attribute, Toolbar, 519, 523
layout_column attribute, GridLayout, 827
layout_columnSpan attribute, GridLayout, 828
layout_gravity attribute, view, 47, 184–185, 190
layout_height attribute
 FrameLayout, 188
 LinearLayout, 171
 view, 44, 46, 47, 175, 180, 232–233
LayoutInflator class, 350, 365, 376
layout manager, for recycler view, 556–557
layout_margin attributes, view, 47, 176, 228–229
layout resource folders, 402–408
layout_row attribute, GridLayout, 827
layouts
 about, 2, 12, 31, 38, 170
 code for, 19–20, 33, 41, 44–46, 80
 constraint. *See* constraint layout
 creating, 13–14
 default, 41
 editing, 41–48
 for fragments. *See* fragments
 frame. *See* FrameLayout element
 grid. *See* GridLayout element
 inherited from ViewGroup class, 198, 200
 linear. *See* LinearLayout element
 nesting, 191–192, 222
 relative. *See* RelativeLayout element
 toolbars as, 311–313, 316

layout_scrollFlags attribute, Toolbar, 510, 519
layout_to* attributes, RelativeLayout, 822
layout_weight attribute, view, 179–181
layout_width attribute
 FrameLayout, 188
 LinearLayout, 171
 view, 44, 46, 47, 175, 232–233
libraries. *See also* specific libraries
 about, 3
 adding to project, 224
 location in project, 16
 Support Libraries, list of, 294
LinearLayout element
 about, 41, 45, 171, 187
 dimension resources, 174
 gravity for view contents, 182–183
 gravity for view placement, 184–185
 height, 171
 nesting, 191–192, 222
 order of views in, 175
 orientation, 172
 padding, 173
 weight of views in, 179–181
 width, 171
 xmlns:android attribute, 171
LinearLayoutManager class, 556–557
Linux kernel, 3
Listener interface, 385–387, 572–576
listeners. *See* event listeners
ListFragment class
 connecting data to, 375–377
 connecting to detail activity, 384–389
 creating, 372–374
 displaying in main activity, 378–381
 listener for, in main activity, 385–387
ListView element
 about, 251
 class hierarchy for, 570
 creating, 259–260, 705–709
 event listeners for, 261–263, 708, 710–712
loaders, 864
localization
 right-to-left languages, 172
 String resources, 50, 55
LocationListener class, 792
LocationManager class, 793

Location Services
 about, 789
 distance traveled, calculating, 796–797
 library for, 790
 listener for, 792
 manager for, 793
 permissions needed
 checking for, 794, 800
 declaring, 791
 notification if denied, 806, 809–810
 requesting from user, 802–804, 806
 provider for, 793
 requesting location updates, 794–795
 stopping location updates, 797
 logcat, viewing, 743, 854
 Log class, 743
 logging messages, 743–746

M

main activity, 84, 120
 manifest file. *See* AndroidManifest.xml file
 material design, 506. *See also* Design Support Library
 menu attribute, NavigationView, 602
 menu element, 321
 menu resource folders, 321
 messages. *See also* notification service
 logging, 743–746
 pop-up messages (toasts), 216
 methods. *See also* specific methods
 calling from activities, 59–62, 81
 creating, 62–63, 65–66, 81
 name of, 69
 mipmap resource folders, 299
 moveToFirst() method, cursor, 671
 moveToLast() method, cursor, 671
 moveToNext() method, cursor, 671
 moveToPrevious() method, cursor, 671
 My Constraint Layout app, 223–246
 activities, 225
 layout, 226–243
 library, 224
 String resources, 225
 My First App, 7–36
 activities, 12–15, 31
 editors, 18–22

emulator, 23–30
 folder structure, 16–17
 layout, 31–34
 project, 8–11
 My Messenger app, 79–118
 activities, 82–83, 90
 choosers, 112–117
 intents, 86–89, 92–108
 layout, 80, 82–83, 91
 manifest file, 84–85
 running on device, 109–111
 String resources, 81

N

navigation. *See also* app bar; ListView element; navigation
 drawers; tab navigation; toolbar
 about, 250–251, 253–255, 291
 Back button, 327, 413–415
 Up button, 292, 327–330
 navigation drawers
 about, 580–583
 adding to main activity, 583, 602–604
 closing the drawer, 606, 614
 compared to tab navigation, 580
 drawer toggle for, 606–607
 fragments for, 583, 585–592
 header for, 583, 593–595
 libraries for, 584
 menu click behavior, 606, 608–613
 menu options for, 583, 593, 596–601
 multiple, 614
 submenu in, 600
 theme for, 589–590
 toolbar for, 589
 NavigationView element, 602–603, 608
 NestedScrollView element, 513, 518–519
 network location provider, 793
 NotificationManager class, 759
 Notification object, 759
 notification service
 about, 755, 762–763
 action for, adding, 758
 heads-up notification, 755, 757
 issuing a notification, 759, 806, 809–810
 library for, 756

notification builder for, 757
notification manager for, 759
notify() method, NotificationManager, 759

O

OAT format, 847
Odometer app, 769–816
bound services, 769–788
library, 790
Location Services, 789–800
permissions, 791, 802–816
onActivityCreated() method, fragment, 365
onAttach() method, fragment, 365
onBind() method, Service, 770, 787–788
onBindViewHolder() method, RecyclerView.Adapter, 550, 571
onButtonClicked() method, activity, 203, 214
onCheckboxClicked() method, CheckBox, 207
onClick attribute
Button, 60, 125, 203, 264
CheckBox, 207, 698–700
ImageButton, 214
RadioButton, 209
Switch, 205
ToggleButton, 204
onClickDone() method, activity, 529
OnClickListener interface, 455–460
onClickListener() method, view, 530
onClick() method
fragment, 456
Listener, 572
OnClickListener, 455
onCreate() method
activity, 61, 81, 96, 121, 133, 137, 138, 147, 167, 787–788
fragment, 365, 429, 430
service, 751
SQLiteOpenHelper, 628, 634–635
onCreateOptionsMenu() method, activity, 323
onCreateViewHolder() method, ViewHolder, 549
onCreateView() method, fragment, 349–350, 365, 374, 376
onDestroy() method
activity, 137, 138, 147, 167, 683, 787–788
fragment, 365
service, 751
onDestroyView() method, fragment, 365
onDetach() method, fragment, 365
on-device tests, 871
onDowngrade() method, SQLiteOpenHelper, 637, 641
onHandleEvent() method, IntentService, 742
onHandleIntent() method, IntentService, 744
OnItemClickListener class, 261–262
onItemClick() method, OnItemClickListener, 261–262
onListItemClick() method, list fragment, 373, 386
onLocationChanged() method, LocationListener, 792
onNavigationItemSelected() method, activity, 608–609
onOptionsItemSelected() method, activity, 324
onPause() method
activity, 154–157, 159, 167
fragment, 365
onPostExecute() method, AsyncTask, 724, 728
onPreExecute() method, AsyncTask, 724–725
onProgressUpdate() method, AsyncTask, 727
onProviderDisabled() method, LocationListener, 792
onProviderEnabled() method, LocationListener, 792
onRadioButtonClicked() method, RadioGroup, 209
onRequestPermissionsResult() method, activity, 805
onRestart() method, activity, 146, 147, 153, 167
onResume() method
activity, 154–157, 159, 167
fragment, 365
onSaveInstanceState() method
activity, 140, 142, 143, 146
fragment, 429, 431
onServiceConnected() method, ServiceConnection, 775–776
onServiceDisconnected() method, ServiceConnection, 775, 777
onStartCommand() method, Service, 751
onStart() method
activity, 146, 147, 153, 167
fragment, 365, 367
onStatusChanged() method, LocationListener, 792
onStop() method
activity, 146, 147, 148–150, 167
fragment, 365
onSwitchClicked() method, Switch, 205
onToggleClicked() method, ToggleButton, 204

onUnbind() method, Service, 787–788
 onUpgrade() method, SQLiteOpenHelper, 628, 637, 640, 642–650
 Oracle JVM (Java Virtual Machine), 842
 orderInCategory attribute, 322
 organizing ideas. *See also* Starbuzz Coffee app
 orientation attribute, LinearLayout, 45, 172

P

package name, 84
 padding attributes, LinearLayout, 173
 parent activity, 328
 PendingIntent class, 758
 performance
 of Android emulator, 858–860
 SQLite database affecting, 720
 permissions
 API levels affecting, 791, 794
 checking for permissions granted, 794, 800
 declaring permissions needed, 791
 denied, issuing notification of, 806, 809–810
 requesting from user, 802–804, 806
 Pizza app, 290–338, 482–536, 538–578
 actions, 315–326
 activities, 297–298
 adapters, 571–574
 app bar, 291–293, 299
 card views, 543–546, 550, 560
 collapsing toolbar, 517–525
 color resources, 304
 FABs, 526–529
 fragments, 485–488
 layout, 305
 libraries, 294–296, 307, 506
 recycler views, 538–539, 545–570, 575–576
 scrollbars, 508–515
 share action provider, 331–336
 snackbar, 526, 530–534
 tab navigation, 498–504
 themes and styles, 300–303
 toolbar, 306–314
 Up button, 327–329
 ViewPager, 489–493
 Play Store, releasing apps to, 862
 pop-up messages (toasts), 216

postDelayed() method, Handler, 128–129
 post() method, Handler, 128–129
 Preferences API, 867
 processes, apps using, 120–121, 133. *See also* services;
 threads
 projects. *See also* Android apps
 application name, 9
 company domain, 9
 creating, 8–10, 40–41
 files in, 15–17, 34
 libraries for, adding, 224
 location, 9
 property animation, 868
 publishProgress() method, AsyncTask, 727
 putExtra() method, Intent, 92, 101
 put() method, ContentValues, 632

Q

QEMU (Quick Emulator), 858
 query() method, SQLiteDatabase, 663–666

R

RadioButton element, 208
 RadioGroup element, 208–209
 Random class, 772
 R.drawable reference, 212, 257
 recycler view adapter, 545–550, 554, 571–574
 RecyclerView.Adapter class, 545–546
 RecyclerView element, 554
 RecyclerView Library, 294
 recycler views
 about, 538–539
 class hierarchy for, 570
 creating, 553–555, 562–565
 data for, 547, 550
 layout manager for, 556–557
 responding to clicks, 566–576
 scrollbars for, 554
 views for, 548–549
 RecyclerView-v7 Library, 542
 RelativeLayout element
 about, 818
 positioning relative to other views, 821–822
 positioning relative to parent, 818–820

releasing apps, 862
removeUpdates() method, LocationManager, 797
RENAME TO command, 649
render thread, 720
requestLocationUpdates() method, LocationManager, 794
requestPermissions() method, ActivityCompat, 803–804, 806
res-auto namespace, 543
res folder, 17
resolution of images, 211
resource files. *See also* array resources; image resources; String resources
 about, 2
 color resources, 304
 dimension resources, 174
 layout resources, 402–408
 menu resources, 321
 mipmap resources, 299
 style resources, 300–301
 types of, folders for, 16–17, 403
resources. *See* books and publications; website resources
resources element, 54
R.java file, 17, 63, 69
rotation of device
 configuration changed by, 134–136, 145, 156
 saving activity state for, 140–141
 saving fragment state for, 427–431
roundIcon attribute, 299
rowCount attribute, GridLayout, 823
Runnable class, 128

S

Safari Books Online, xl
scale-independent pixels (sp), 201
scheduled services, 740
screens
 activities in. *See* activities
 density of, images based on, 211
 layouts for. *See* layouts
 size of, adapting apps for, 340–341, 394–398, 402–412
scrollbars attribute, recycler view, 554
scrolling toolbar, 507, 508–515
ScrollView element, 215
SDK. *See* Android SDK (API level)

Service class, 752, 770
ServiceConnection interface, 775–777
service element, 745
services
 about, 740
 bound. *See* bound services
 built-in, 740
 location. *See* Location Services
 notification. *See* notification service
 scheduled, 740
 started. *See* started services
setAction() method, Snackbar, 530
setAdapter() method
 ListView, 270
 RecyclerView, 554
 ViewPager, 493
setContentDescription() method, ImageView, 212, 282
setContentIntent() method, notification builder, 758
setContentView() method, activity, 61, 133, 137, 363
setDisplayHomeAsUpEnabled() method, ActionBar, 329
setImageResource() method, ImageView, 212, 282
setListAdapter() method, fragment, 376
setNavigationItemSelectedListener() method, Navigation-View, 608
setShareIntent() method, ShareActionProvider, 333
setSupportActionBar() method, AppCompatActivity, 313, 314, 317
setText() method, TextView, 64, 201, 282
settings screen, 867
setType() method, Intent, 101
setupWithViewPager() method, TabLayout, 501
share action provider, 292, 331–335
shell, running with adb, 852–853
shortcuts. *See* app bar
showAsAction attribute, 322
Sierra, Kathy (Head First Java), 4
signing APK files, 845
SimpleCursorAdapter class, 681
slider. *See* Switch element
snackbar, 507, 526, 530–533
SnackBar class, 530
Spinner element
 about, 47, 48, 210
 setting values in, 64
 values for, 56–57
sp (scale-independent pixels), 201

- SQLite database
 - about, 623–624
 - accessing in background thread, 720–731, 736
 - adding columns, 649, 650
 - alternatives to, 624
 - closing, 672, 682–683
 - conditions for columns, 647
 - conditions for queries, 666
 - creating, 629, 634–635
 - cursors
 - about, 624, 663
 - adapter for, 679–681, 688–689, 715
 - closing, 672, 682–683
 - creating, 663–666, 678
 - navigating to records in, 670–671
 - refreshing, 714–718
 - retrieving values from, 672–674
 - data types in, 630
 - deleting records, 647
 - deleting tables, 650
 - downgrading, 641
 - DrinkActivity. *See* DrinkActivity
 - files for, 623
 - getting a reference to, 662, 678
 - helper, 624, 626–628, 634–635
 - inserting data in, 632–633
 - location of, 623, 624
 - ordering records from query, 665
 - performance of, 720, 736
 - querying records, 663–666
 - renaming tables, 649
 - security for, 624
 - tables in, 630–631
 - updating records
 - programmatically, 645–647
 - from user input, 695–703, 705–706, 708–712
 - upgrading, 637–640, 642–650
 - version number of, 637–639
- SQLiteDatabase class, 624
- SQLiteException, 662, 675
- SQLiteOpenHelper class, 624, 627–628
- SQL (Structured Query Language), 631, 675
- src attribute, ImageButton, 214
- src attribute, ImageView, 212
- src folder, 17
- StaggeredGridLayoutManager class, 556–557
- Starbuzz Coffee app, 248–288, 622–656, 658–692, 694–738
 - activities, 248–249, 252–255, 262–264, 267–272
 - adapters, 269–271
 - database, 626–656, 659–692, 694–718
 - DrinkActivity. *See* DrinkActivity
 - image resources, 257
 - intents, 277–285
 - Java classes, 256
 - layout, 258–260
 - listeners, 261–262, 276
 - navigation, 250–251
 - threads, 720–738
 - top level activity
 - adding favorites to. *See* top level activity
 - startActivity() method, activity, 86, 112, 117, 121
 - started services
 - about, 740–741, 748
 - class hierarchy for, 752
 - in combination with bound services, 786
 - compared to bound services, 786
 - creating, 741–742, 744
 - declaring in AndroidManifest.xml, 745
 - lifecycle of, 750–751
 - methods associated with, 750–752
 - starting, 746–747
 - startService() method, Intent, 747, 751
 - states, of activities. *See* activity lifecycle
- Stopwatch app, 122–168, 434–480
 - activities, 125–127, 130–133
 - activity lifecycle, 138–139, 146–163
 - activity states, 134–144
 - dynamic fragments, 434–436, 444–460
 - fragment lifecycle, 439
 - fragment transactions, 463–469, 472–475
 - handlers, 128–129
 - layout, 123–124, 471
 - project, 122
 - String resources, 123
- string-array element, 56
- string element, 54
- @string reference, 52, 81
- String resources
 - about, 38, 50, 54, 55
 - action titles in, 320
 - adding, 512, 517

- arrays of, 56–57, 210, 259
- creating, 51, 81
- getting value of, 115
- location of, 54, 55
- referencing strings in, 52, 81
- updated in R.java file, 69
- strings.xml file. *See* Array resources; String resources
- Structured Query Language. *See* SQL
- style element, 301
- @style reference, 300
- style resources, 300–301
- styles
 - applying themes using, 300–301
 - customizing themes using, 303
- Support Libraries. *See also* v7 AppCompat Library
 - adding to project, 296
 - list of, 294
- supportsRtl attribute, application, 172
- Swing, 4
- swiping through fragments, 489–493
- Switch element, 205
- sync adapters, 864
- syncState() method, ActionBarDrawerToggle, 607
- system image. *See* Android SDK (API level)

T

- TabLayout element, 499, 501
- tab navigation
 - about, 482–484, 493
 - adding tabs to layout, 498–504
 - compared to navigation drawers, 580
 - fragments for, 483–488
 - swiping between tabs, 489–493
- tasks, 78
- testing
 - automated testing, 870
 - emulator compared to device, 117
 - on-device tests, 871
 - unit tests, 870
- text attribute
 - Button, 44, 203
 - CheckBox, 206
 - TextView, 33, 34, 50, 201
- text field. *See* EditText element

- textOff attribute
 - Switch, 205
 - ToggleButton, 204
- textOn attribute
 - Switch, 205
 - ToggleButton, 204
- textSize attribute, TextView, 201
- TextView element
 - about, 33, 44, 201
 - code for, 33–34, 44–47, 91
 - setting text in, 64
- theme attribute
 - AppBarLayout, 519
 - application, 299, 300
- themes
 - about, 84
 - app bar requiring, 293
 - applying to project, 299–300
 - built-in, list of, 302
 - customizing, 303
 - for navigation drawers, 589–590
 - removing app bar using, 308
 - v7 AppCompat Library for, 294, 296–298
- threads
 - about, 720
 - background thread, 720–731, 736
 - main thread, 127, 720
 - render thread, 720
- title attribute, 322
- toasts, 216
- ToggleButton element, 204
- toolbar
 - adding as layout, 311–313, 316
 - collapsing, 507, 517–525
 - for navigation drawers, 589
 - replacing app bar with, 292, 306–313
 - scrolling, 507, 508–515
- Toolbar class, 306, 309, 314
- top-level activities, 249–250, 252–255, 290
- transactions, for fragments. *See* fragment transactions

U

- unbindService() method, ServiceConnection, 779
- unit tests, 870

Up button, 292, 327–330
 update() method, SQLiteDatabase, 646, 698
 USB debugging, 109
 USB driver, installing, 109
 uses-permission element, 791

V

v4 Support Library, 294. *See also* Design Support Library
 v7 AppCompat Library
 about, 294
 adding to project, 296, 307
 AppCompatActivity class in, 297–298
 fragments using, 345
 Location Services using, 790
 navigation drawers using, 584
 notifications in, 756
 v7 CardView Library, 294, 542
 v7 RecyclerView Library, 294
 values resource folders. *See* string resources; dimension resources
 variables, setting, 126, 127
 vertical constraints, 228
 view animations, 868
 ViewGroup class, 198, 200
 ViewHolder class, 546, 548
 ViewPager class, 489–493, 501
 views. *See also* specific GUI components
 about, 44, 198–199
 aligning, 238
 biases for, 231
 centering, 230
 getting and setting properties, 199
 gravity for view contents, 182–183
 gravity for view placement, 184–185, 190
 height, 175, 180, 232–233
 ID, 44, 175, 199
 margins, 176, 228–229
 methods associated with, 199
 weight, 179–181
 width, 175, 232–233

W

website resources
 activity actions, types of, 101
 source code for examples, xxxvi
 USB drivers, 109
 WebView class, 866
 widgets. *See also* GUI components
 about, 869
 adding, in blueprint tool, 226
 constraints for. *See* constraint layout
 Workout app, 340–392, 394–432, 434–480
 activities, 346–347, 359–363, 381, 389, 421–423, 470
 adapters, 375–376
 back button, 413–415
 device sizes, supporting, 340–341, 394–398
 dynamic fragments, 434–436, 444–460
 fragment lifecycle, 439
 fragments, 342–344, 348–356, 359–363, 365–369, 416–421
 fragment state, 427–431
 fragment transactions, 463–469, 472–475
 Java classes, 360
 layout, 471
 libraries, 345
 listener interface, 384–388
 list fragments, 372–374, 377–378
 screen-specific resources, 402–409
 tablet AVD, 399–401
 wrap_content setting, width and height, 232

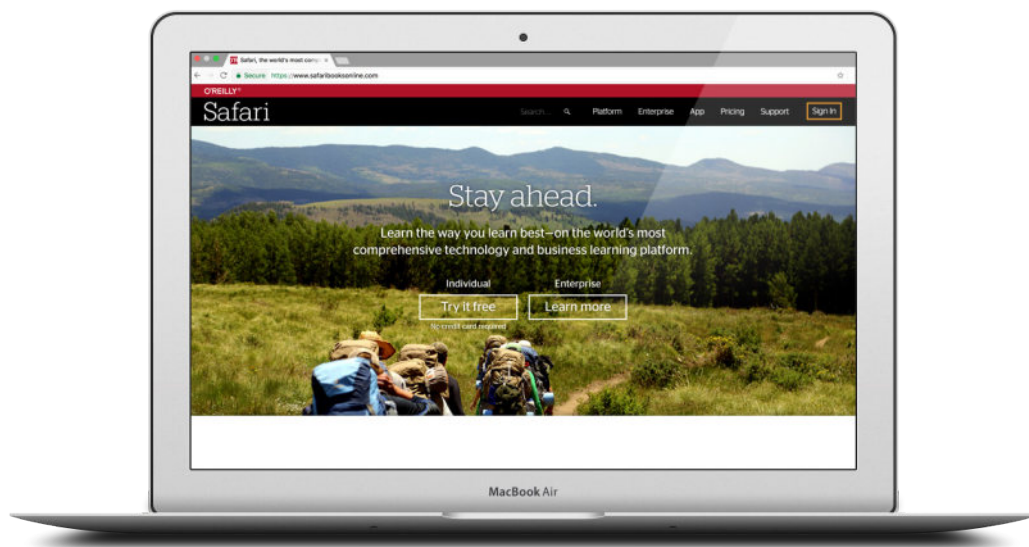
X

xmlns:android attribute, LinearLayout, 171

Z

zipalign tool, 845
 Zygote process, 846

Get up to speed on Agile.



Quickly learn what Agile development is all about on Safari, O'Reilly's online learning platform. Through books, videos, interactive tutorials, and live online training, you'll learn how to design and build products incrementally, test constantly, and release as often as you need to.

Check out *The Agile Sketchpad*, a video course by David and Dawn Griffiths that helps you:

- See what it's like to work on an Agile team.
- Discover how Agile helps you deliver value early, respond to change, and manage risk.
- Master the secrets of accurate estimation.
- Explore how test-driven development, continuous integration, and the 10-minute build help you deliver better software.
- Learn why programming in a pair can be more effective than programming alone.
- Find out how Agile techniques such as Kanban can help you identify and avoid disruptions to your development process.
- See how the five key values—communication, courage, feedback, simplicity, and respect—drive everything you do in an Agile project.

Try Safari for free for 10 days and get up to speed with Agile.

oreilly.com/go/agile-sketchpad

www.EBooksWorld.ir

O'REILLY®
Safari