Community Experience Distilled

Android UI Design

Plan, design, and build engaging user interfaces for your Android applications

Jessica Thornsby

[PACKT] open source
PUBLISHING
community experience distilled

Community Experience Distilled

Android UI Design

Plan, design, and build engaging user interfaces for your Android applications

Jessica Thornsby

[PACKT] open source* PUBLISHING

Table of Contents

Android UI Design
Credits
About the Author
About the Reviewer
www.PacktPub.com
    eBooks, discount offers, and more
        Why subscribe?
        Free access for Packt account holders
Preface
    What this book covers
    What you need for this book
    Who this book is for
    Conventions
    Reader feedback
    Customer support
        Downloading the example code
        Downloading the color images of this book
        Errata
        Piracy
        Questions
1. Introducing the Android UI
    What is a user interface anyway?
    Is developing UIs for Android really that different from other platforms?
    What are the characteristics of an effective UI?
    Why is UI essential for the success of your app?
        Instant familiarity
        Easy and enjoyable to use
        Consistency
        Preventing user frustration
        Helping users fix their mistakes
        Providing a better overall Android experience
    The UI case study – Google+

- The action bar
  - Navigational controls
  - Action buttons
  - The action overflow
- Floating action button
- Menus
- Settings
- Dialogues
- Toasts
- Search
- Input controls
- Styles and themes

Summary

2. What Goes into an Effective UI?

What is a view?

What is a layout?

Building your UI – XML or Java?

- Declaring your UI with XML
- Declaring your UI programmatically
- Using both programmatic and XML layouts

Deep dive – exploring layouts

- Defining the size of your layouts
  - A supported keyword
  - A dimension value
  - Setting the layout size programmatically

Exploring different layouts

- Everything you need to know about LinearLayout
- Everything you need to know about RelativeLayout
  - Relative to the parent container
  - Relative to other elements
  - Aligning with other elements

Creating views

- Assigning the ID attribute
- Setting a view's size
- Android gravity and layout gravity
- Setting the background – working with color
- Assigning a weight value

- Adding and customizing view objects
  - TextView
    - Brightening up your text
    - Setting the size of your text
    - Emphasizing your text
    - Setting the typeface
    - How many lines?
  - EditText
    - Controlling keyboard behavior - setting the inputType
    - android:imeOptions
    - Giving the user a hint
  - ImageView
    - Supporting multiple screens
      - Supporting different screen densities
        - Creating density-specific images
    - Adding ImageView
  - Buttons and ImageButtons
    - Creating buttons with text labels
    - Creating buttons with image labels
    - Creating buttons with text and image labels
  - State list resources
  - Summary
- 3. Expanding your UI – Fragments, Resources, and Gathering User Input
  - More resource types
    - Creating and styling string resources
    - Creating string arrays
    - Defining dimensions in dimens.xml
    - Color state lists
  - Working with 9-patch images
    - How do I create 9-patch images?
  - Registering the user input
    - Handling click events
      - Handling onClick Via Java
      - Handling onClick Via XML
    - Registering the EditText input
      - The example app

- Working with fragments
  - Why do we need fragments?
  - The fragment life cycle
  - Creating a fragment
    - Fragments and backwards compatibility
    - Creating your fragment class
    - Adding a fragment to your activity declaratively
  - Adding a fragment to an activity at runtime
    - Fragment transactions and the back stack
      - Adding a fragment
      - Removing a fragment
      - Replacing a fragment
- The multi-window support in Android N
  - How does the multi-window mode work?
  - Getting your app ready for multi-window mode
    - Testing your app's multi-window support
  - Picture-by-picture mode
- Summary

4. Getting Started with Material Design
- The Material Design ethos
  - Case study – Hangouts
  - Case study – Google Calendar
  - Case study – Google Maps
- Getting started with Material Design
  - Applying the Material theme to your app
    - Choosing your color scheme
    - Backwards compatibility
  - Creating a sense of depth
    - Creating a Material Design structure
  - Floating action buttons
  - Bottom sheets
  - CardView
  - Lists and RecyclerView
  - Animations and transitions
    - Reinforcing the Material Design illusion
  - Providing the user with a visual feedback
    - Animations – a word of warning

The finishing touches
Designing your product icon
System icons
Typography and writing
Typefaces
Text opacity
Writing guidelines
Summary
5. Turning Your Bright Idea into a Detailed Sketch
Brainstorming - taking advantage of mobile hardware
Touch and gestures
GPS
Vibration
Audio input/output
Interacting with other devices
The difference between UX and UI
Brainstorming your app
Writing a concept
Creating an ultimate features list
Identifying your app's primary task
Is this application right for mobile?
Do you have the budget?
Planning your app
Identifying your target audience
Creating a persona
The mobile persona
Creating use cases
Deciding on a feature list
Taking a critical look at your target audience
Does your target audience need a mobile app?
Is your target audience on Android?
Okay, so your target audience is on Android, but does it own a suitable Android device?
Do they use this "type" of application?
Would your target audience be willing to pay for this application?

Are there enough people in your target audience for your project to be profitable?
Identifying user and product goals
Product goals
User goals
Creating a roadmap
Some final things to think about
What devices should you support?
How should you market your app?
Designing your application
The high-level flow
Creating a screen list
Creating a screen map
Grouping screens into multi-pane layouts
Navigation
What are your app's most important tasks?
Are there any tasks you can group together?
Is my navigation consistent?
Common navigation patterns
The embedded navigation
Buttons and simple targets
Lists
Grids, cards, and carousels
Tabs
Horizontal paging (swipe views)
The social layer
Summary
6. Turning Your Sketches into Wireframes
What is wireframing?
What are the benefits of wireframing?
Creating a Wireframe
Creating your first wireframe
Exploring more wireframes
Wireframing a details screen
Wireframing search screens
The search screen as a fragment
Digital wireframes

Adding content to your wireframes
What is paper prototyping?
Usability testing
Rapid prototyping
Summary
7. Building a Prototype
Creating a prototype in Android Studio
Creating your first prototype
Creating your second prototype
Finalizing your design
Visuals
Background music and sound effects
Your text
Your app's personality
Creating themes and styles
Defining styles
Inheritance
Working with themes
Get ready for errors!
User input errors
Summary
8. Reaching a Wider Audience – Supporting Multiple Devices
Supporting different versions of Android
Specifying minimum and target API levels
minSdkVersion
targetSdkVersion
compileSdkVersion
Check version at runtime
Supporting different screens
Configuration qualifiers
How Android selects the perfect resource
Creating alias resources
Screen density
Converting dpi into pixels and vice versa
Providing different layouts for different screen sizes
smallestWidth – sw<N>dp
Available screen width – w<N>dp

Available screen height – h<number>dp
Designing for different screen orientations
Reacting to orientation changes
Testing across multiple screens
Showing off your screen support
Attracting an international audience
Identifying the target languages and regions
Providing alternate text
What other resources might you need localizing?
Why default resources are important
Which configuration qualifiers are the most important?
Getting your app translated
Getting the most out of your translations
Putting your strings in context
Using terms consistently
Getting your strings in order
Not creating unnecessary strings
Always marking non-translatable text
Other things to consider
Right to left support
Formatting values
Localizing your app – best practices
Design a single set of flexible layouts
Create alternate languages only when needed
Testing your app across different locales
Testing for different locales
Look for common localization issues
Testing for default resources
Plan a beta release in key countries
Get ready for lift off!
Localize your Google Play store listing
After launching your project
Following a successful launch – supporting international users
Monitor your app's international performance
Summary
9. Optimizing Your UI

- Timing your code
- Identifying overdraw
- Simplifying your Hierarchy View
  - Tree View
  - Tree overview
  - Layout View
- Spotting memory leaks
  - Memory monitor
  - Heap tab
  - Object allocation – understanding memory churn
- Debugging your project
  - Working with breakpoints
    - Configuring your breakpoints
- Examining your code with Lint
- Optimizing your code with ProGuard
- Scrutinize each pixel
  - Pixel Perfect pane
  - Pixel Perfect tree
  - Pixel Perfect Loupe pane
- Processes and threads
  - Terminating processes
    - Foreground processes
    - Visible processes
    - Service processes
    - Background processes
    - Empty processes
- Re-using layouts with > and <merge/>
  - Loading views only when needed
- Summary

10. Best Practices and Securing Your Application
- Keeping user data secure
  - Connecting to a network
- Requesting permissions
  - The new permissions model – backwards compatibility
  - Permission groups
  - Declaring permissions
    - Verifying permissions

- Handling the permissions request response
- Permissions and <uses-feature>
- Best practices for app permissions
  - Making as few permission requests as possible
  - Requesting critical permissions up front
  - Providing extra information where necessary
  - Paying attention to permissions required by libraries
  - Being transparent about accessing the camera and microphone
  - Considering alternatives
- Notifications
  - Notification best practices
    - Providing the right content
    - Using notifications sparingly
    - Giving users a choice
    - Categorising notifications
    - Making use of actions
    - Using expanded layouts
      - Big text style
      - Big picture style
      - Inbox style
  - Direct reply notifications
  - Bundled notifications
- Application widgets
  - Declaring an AppWidgetProvider class in your project's Manifest
  - Creating an AppWidgetProviderInfo file
  - Creating the widget's layout
    - App widget best practices
      - Including margins for earlier versions of Android
      - Providing flexible graphics and layouts
      - Not updating too often
- Accessibility best practices
  - Adding descriptive text to your UI controls
  - Designing for focus navigation
  - Custom view controls
    - Providing alternatives to audio prompts

Testing various font sizes
Using recommended touch target sizes
Providing alternatives to affordances that time out
Testing your application's accessibility features
Summary

Android UI Design

Android UI Design

Copyright © 2016 Packt Publishing

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

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

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

First published: May 2016

Production reference: 1180516

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham B3 2PB, UK.

ISBN 978-1-78588-742-0

www.packtpub.com

Credits

Author Jessica Thornsby	**Copy Editor** Yesha Gangani
Reviewer Leonardo Risuleo	**Project Coordinator** Sanchita Mandal
Commissioning Editor Edward Gordon	**Proofreader** Safis Editing
Acquisition Editor Larissa Pinto Pratik Shah	**Indexer** Rekha Nair
Content Development Editor Samantha Gonsalves	**Production Coordinator** Aparna Bhagat
Technical Editor Shivani K. Mistry	

About the Author

Jessica Thornsby studied poetry, prose, and scriptwriting at Bolton University before discovering the world of open source and technical writing, and has never looked back since. Today, she is a technical writer and full-time Android enthusiast residing in sunny Sheffield, England.

She enjoys writing about rooting and flashing mobile devices, Java, Eclipse, and all things Android and Google. She is also the co-author of *iWork: The Missing Manual*.

When not wordsmithing about technology and obsessing over the latest Android developments, she keeps a blog about her local food scene and writes about looking after exotic pets. On the rare occasions that she's dragged away from her computer, she enjoys beer gardens, curry houses, the great British seaside, scary movies, and spending lots of time with her house rabbits and chinchillas.

I'd like to thank the entire team at Packt Publishing for their support and encouragement on this project. I also have to thank all my friends and family for putting up with me during the writing process (and in general!), with special thanks to my parents, Peter and Pauline, and my fiancé, Toby. Last, but not least, I'd like to thank my menagerie for keeping me company even when deadlines keep me shackled to my keyboard: my chinchillas Taco, Buca, Churro, and house bunnies Peanut and Stewart.

About the Reviewer

Leonardo Risuleo is the owner and creative director at Small Screen Design. He is a designer and developer with several years of experience in mobile, new media, and user experience. Leonardo is a highly dedicated professional, and he's passionate about what he does. He started his career back in 2003, and during past few years, he has worked on a variety of different mobile and embedded platforms for a number of well-known brands and studios. Leonardo designs, prototypes, and develops mobile applications, games, widgets, and web sites.

He has had the honor of being Nokia Developer Champion, a recognition and reward program for top mobile developers worldwide, for 3 years in a row (2008 to 2010). In 2008, Leonardo founded Small Screen Design (www.smallscreendesign.com), a design and development studio focused on mobile design and user experience. In 2015, he became Digital Champion for Squillace, an ambassador for Digital Agenda, to help every European become digital.

www.PacktPub.com

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

eBooks, discount offers, and more

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

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

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

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

Why subscribe?

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

Free access for Packt account holders

Get notified! Find out when new books are published by following `@PacktEnterprise` on Twitter or the Packt Enterprise Facebook page.

Preface

Your UI is your most direct communication with your users, but all too often in app development, design is an afterthought, or something that "just happens" along the way.

In order to develop a great app, you need to take your UI design seriously. Even if your app delivers great functionality, if its user interface is clunky, laggy, difficult to navigate, or just an all-around eyesore, then no one is going to want to use it.

Android isn't the easiest platform to design for. Creating an app that looks great across countless different devices, all with different combinations of hardware, software, and screens, is no mean feat. But if you put some thought into the design process, you can create a UI that delivers a great experience across the *entire* Android ecosystem.

This book will help you become a design-minded developer. Over the course of 10 chapters, you'll learn how to design, refine, test, and develop an effective, engaging UI that people will *love* using and that delivers the best possible experience regardless of device specifics.

From the fundamentals of layouts and views right through to creating responsive, multi-pane layouts, and from sketching your screens, through to using Android Studio's tools to scrutinise your view hierarchy and hunt out memory leaks, this book covers everything you need to create the perfect Android UI.

What this book covers

Chapter 1, *Introducing the Android UI*, explains why designing and developing an effective UI is essential to the success of your app.

Chapter 2 , *What Goes into an Effective UI?*, teaches you how to master the building blocks of every Android UI: layouts and views. Learn how to create and style common UI components including text, images, and buttons.

Chapter 3 , *Expanding your UI – Fragments, Resources, and Gathering User Input*, helps you take your UI to the next level using arrays, dimensions, state lists, and 9-patch images. Using fragments, you'll learn how to create a flexible UI that reacts to the user's specific screen size.

Chapter 4 , *Getting Started with Material Design*, teaches you how to master Google's new design principles and delight your users with a UI that feels like a seamless extension of the Android operating system.

Chapter 5 , *Turning Your Bright Idea into a Detailed Sketch*, helps you become a design-minded developer by introducing roadmaps, flowcharts, screen lists, and screen maps to your Android app "To Do" list.

Chapter 6 , *Turning Your Sketches into Wireframes*, shows you how to transform the high-level plans from the previous chapter into detailed screen designs using paper prototypes and wireframes.

Chapter 7 , *Building a Prototype*, put your plan to the test! By the end of this chapter, you'll have created a complete digital prototype.

Chapter 8 , *Reach a Wider Audience – Supporting Multiple Devices*, teaches you how to attract a wider audience with an app that supports a wide range of hardware, software, screen sizes, screen

densities, orientations, versions of the Android platform, and even different languages.

Chapter 9 , Optimizing Your UI, shows you how to create a smooth and responsive UI that people will love using. If your app is laggy, prone to crashing, gobbles up data and memory, or drains the user's battery, then no one is going to want to use it!

Chapter 10, *Best Practices and Securing Your Application*, guides you through putting the finishing touches to your UI, including using notifications from the upcoming Android N release.

What you need for this book

You will need the Android SDK and Android Studio (preferred) or Eclipse.

Who this book is for

If you are a Java developer with a keen interest in building stunning Android UIs for your applications in order to retain customers and create great experiences for them, then this book is for you. A good knowledge of XML and some grounding in Android development is assumed.

Conventions

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

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "We can include other contexts through the use of the `include` directive."

A block of code is set as follows:

```
<LinearLayout

  xmlns:android="http://schemas.android.com/apk/re
s/android"
   android:orientation="vertical"
   android:layout_width="match_parent"
   android:layout_height="match_parent">
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
<LinearLayout

  xmlns:android="http://schemas.android.com/apk/re
s/android"
   android:orientation="vertical"
   android:layout_width="match_parent"
   android:layout_height="match_parent">
```

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

```
cd /Users/username/Downloads/android/sdk/platform-
tools
```

New terms and **important words** are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: "Clicking the **Next** button moves you to the next screen."

> ## Note
>
> Warnings or important notes appear in a box like this.

> ## Tip
>
> Tips and tricks appear like this.

For this book we have outlined the shortcuts for the Mac OX platform if you are using the Windows version you can find the relevant shortcuts on the WebStorm help page https://www.jetbrains.com/webstorm/help/keyboard-shortcuts-by-category.html.

Reader feedback

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

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

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

Customer support

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

Downloading the example code

You can download the example code files for this book from your account at http://www.packtpub.com. If you purchased this book elsewhere, you can visit http://www.packtpub.com/support and register to have the files e-mailed directly to you.

You can download the code files by following these steps:

1. Log in or register to our website using your e-mail address and password.
2. Hover the mouse pointer on the **SUPPORT** tab at the top.
3. Click on **Code Downloads & Errata**.
4. Enter the name of the book in the **Search** box.
5. Select the book for which you're looking to download the code files.
6. Choose from the drop-down menu where you purchased this book from.
7. Click on **Code Download**.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR / 7-Zip for Windows
- Zipeg / iZip / UnRarX for Mac
- 7-Zip / PeaZip for Linux

The code bundle for the book is also hosted on GitHub at https://github.com/PacktPublishing/Android-UI-Design. We also have other code bundles from our rich catalog of books and videos available at https://github.com/PacktPublishing/. Check them out!

Downloading the color images of this book

We also provide you with a PDF file that has color images of the screenshots/diagrams used in this book. The color images will help you better understand the changes in the output. You can download this file from https://www.packtpub.com/sites/default/files/downloads/AndroidUIDesign_ColoredImages.pdf.

Errata

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

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

Piracy

Piracy of copyrighted material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the Internet, please provide us with the

location address or website name immediately so that we can pursue a remedy.

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

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

Questions

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

Chapter 1. Introducing the Android UI

If you've picked up this book, then it's because you want to develop user interfaces that people will love to use.

Developing for the Android platform is a mix of exciting opportunities and headache-inducing challenges; and designing and building your app's UI is no different.

Some of the challenges that you'll face when building your UI are very precise and technical (such as creating a UI that'll correctly display across Android 5.0 and Android 5.1), whereas others are more personal, such as resisting the temptation to do crazy and unusual things with your UI, simply because you can.

That's where this book comes in.

Over the course of 10 chapters, you'll gain some design skills and the technical know-how that you'll need to seize all the opportunities and overcome all the challenges of developing UIs for the Android platform.

We're going to be covering the *entire* design and development process from using plans, sketches, and wireframes, to turning that initial spark of inspiration into a step-by-step blueprint of how we're going to build the perfect user interface, before finally putting this plan into action by building, testing and refining your UI. Along the way, we'll cover all the latest best practices, including Material Design principles and some of the new UI features coming up in Android N.

Since creating a UI that your users will like isn't quite good enough, we'll go one step further and use a range of tools and techniques to analyze and optimize every part of our user interface. By the end of this book, you'll know how to develop a UI that your users will *love*.

Let's start with the basics.

What is a user interface anyway?

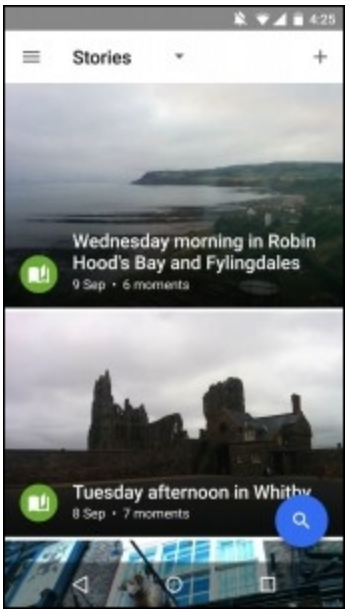
This may seem like an obvious question. After all, we interact with UIs every day, whether it's on our computer, mobiles, tablets, or other electronic devices. But sometimes, the simplest questions are the hardest to answer.

The technical definition of a user interface is the junction between a user and an app or a computer program. A user interface is everything the user can see and interact with, and unless you're developing a very special (or very unusual) kind of Android app, then every app you develop will have some form of user interface.

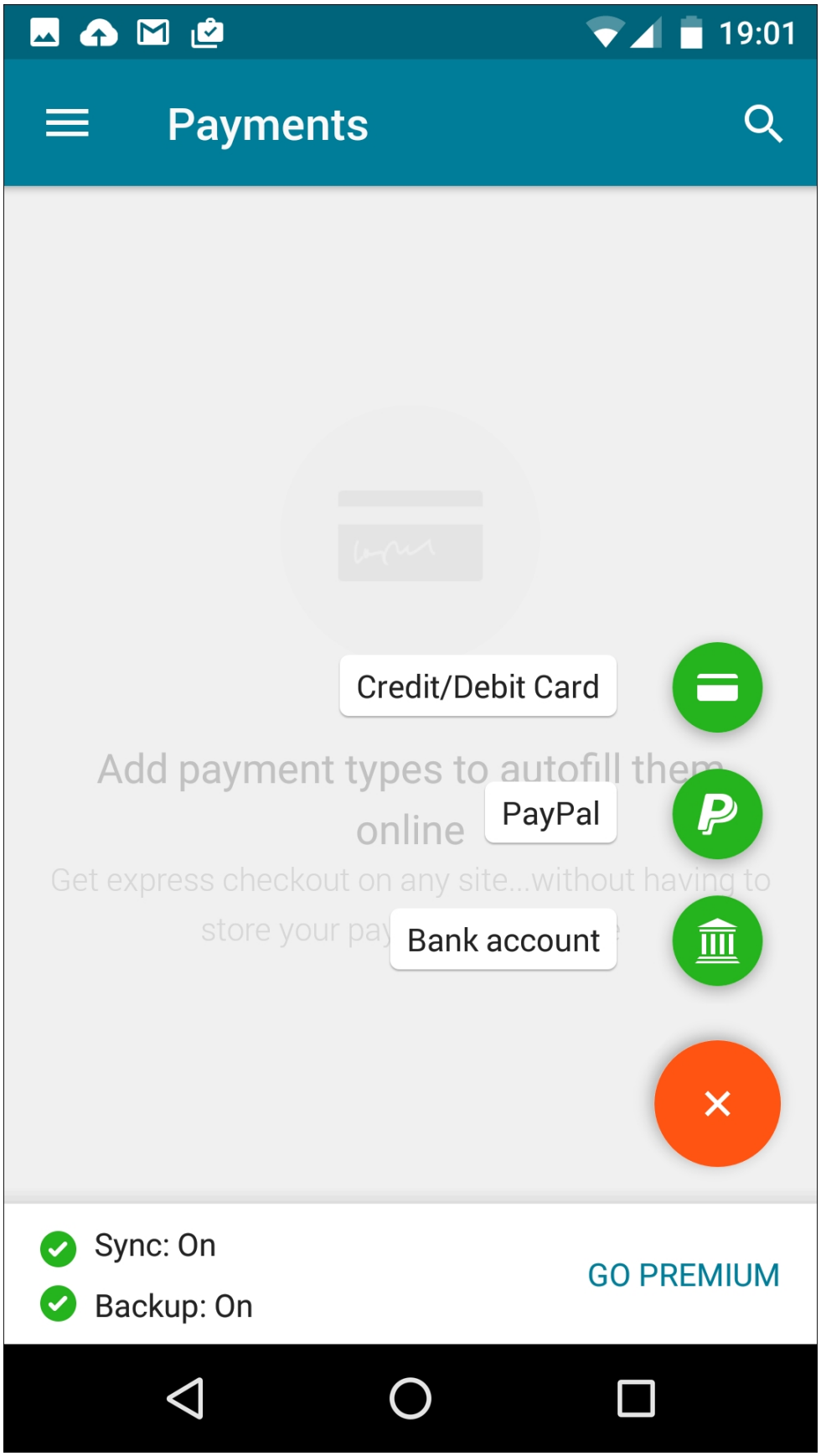
When it comes to creating your app's UI, the Android platform gives you the freedom to bring your vision to life. Just flick through a few of the apps installed on your Android device, and chances are that you'll encounter UIs that look very different from one another.

Although these UIs may look different on the surface, they do share lots of common elements, whether they're the layouts working quietly behind the scenes, or visible elements such as buttons, menus, and action bars.

Here are a few UIs from different Android apps. Although they each have their own look and feel, they also have a lot of UI elements in common:

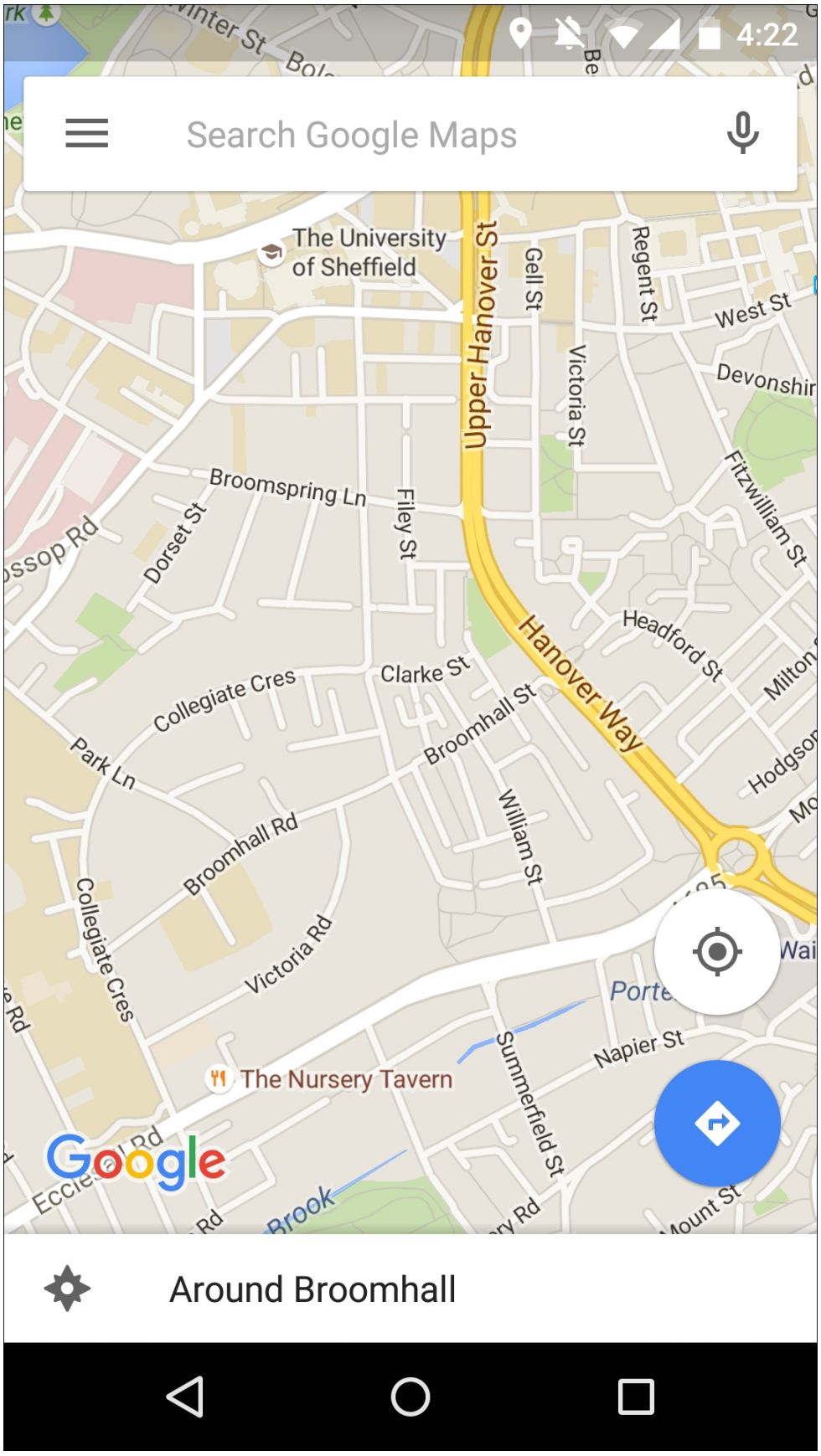


Google Photos isn't just a gallery, it also gives snap-happy Android users new ways to enjoy their media by organizing photos and videos based on factors such as location, date, and subject matter. Since the Photos app wants to encourage you to spend time exploring and enjoying its photo and video content, it's no surprise that the most prominent UI element is the floating **Search** button in the bottom-right corner, which allows users to search their media based on factors such as location and subject.


19:01
Payments
Credit/Debit Card
Add payment types to autofill them
online
PayPal
Get express checkout on any site...without having to
store your pay
Bank account
Sync: On
Backup: On
GO PREMIUM

Announced at the 2014 Google I/O conference, and later making an appearance in Android 5.0, Material Design is a new design language that provides a more consistent user experience across Google products-including Android. Taking its cues from paper and ink, Material Design uses shadows, edges, dimensions, and the concept of **sheets of material** to create a striking, minimal experience.

Many apps now implement Material Design. This helps to provide a more seamless UI experience, even when the user is switching between apps created by entirely different developers. The previous screen shows the Dashlane password manager app, which has a Material Design theme.


4:22
Search Google Maps
The University of Sheffield
Upper Hanover St
Gell St
Regent St
West St
Victoria St
Broomspring Ln
Filey St
Dorset St
Fitzwilliam St
Headford St
Hanover Way
Collegiate Cres
Clarke St
Broomhall St
Park Ln
Broomhall Rd
William St
Victoria Rd
Napier St
Summerfield St
The Nursery Tavern
Google
Around Broomhall

The UI elements you'll find in the Maps app are carefully designed to stay out of the way, so the main body of content (that is, the *map* part of Maps) is clearly visible and uninterrupted. This is a good example of a UI that strikes a tricky balance between being unobtrusive, while also ensuring all the UI elements you could possibly need are always within easy reach. Another example of this kind of UI is the YouTube app.

Is developing UIs for Android really that different from other platforms?

If you have experience developing for platforms besides Android, then you may feel confident that you can turn out a pretty good UI, whether it's the UI for an iPhone app or a piece of software that's destined to run on the latest version of Windows.

There are two factors that make developing Android UIs different to some of the other platforms you may be used to:

- It's an open platform. As already mentioned, Android gives you the freedom to express yourself and create whatever UI you want. Even the latest Material Design principles are merely recommendations and not hard and fast rules that you need to follow if Google are going to allow your app into the Play Store (although it's highly recommended that you do follow these design principles!).

  This means you have the freedom to make some really innovative UI decisions that your users will *love*, but it also means you're free to make some really innovative design decisions that your users will *hate* (sometimes, the reason why you've never seen something done before is because it's a *bad* idea). Restraining yourself is a big part of designing and developing effective UIs for the Android platform.

- It's a varied platform. When you release an Android app, there's no telling where it might end up! Your app could potentially wind up on all sorts of different devices consisting of different hardware, software, and screens, including different versions of the Android operating system itself.

Design a UI that's flexible enough to handle all these variables, and your app has the potential to wow a wide and varied audience. Design a UI that isn't quite up to the challenge, and your users will have an inconsistent experience, where your UI looks great and functions perfectly on some devices and runs less-than-perfectly on others. A poorly designed UI might even make your app appear completely broken on certain devices.

What are the characteristics of an effective UI?

Over the course of this book, you'll learn how to create an effective UI, but before we get stuck into this, it's helpful to have the end goal in sight.

A successful Android UI should be as follows:

- Clear...
- Your UI is how your app communicates with your users. Make sure it's clear what your app is trying to say!
- If the user can't glance at every single screen within your app and immediately know what the screen's asking them to do, then your UI isn't clear enough.
- ....But not too clear!
- How can a user interface be *too* clear?
- Imagine a screen that feels the need to explain each and every UI element. Sure, this kind of UI would be clear, but would you persevere with such an app? Or would you look for an alternative app, one that isn't crammed full of unnecessary and, most likely, irritating text, such as **Tap the Submit Results button to submit your results?**
- An effective UI strikes a balance between being clear and concise. If you design your UI well, the user will immediately know that the **+** icon in your music-playing app means *add new track to playlist*, without you cluttering up your UI with additional text.
- Responsive
- No matter how great your app's content, if the UI is laggy and prone to freezing, no one is going to want to use it.
- An effective UI is responsive, smooth, and plays out like a conversation between the user and your application. For example, imagine you've just downloaded a new app and your first task is to create a new account. You fill in a form and tap

Submit. Then, suddenly you're taken to an entirely new section of the app with no explanation of what's happened. Have you just created an account? Are you already logged into your new account? What's going on?

- In this scenario, an effective UI would respond to you tapping the **Submit** button even if it's just a simple **Thank you for registering your details - you're now logged into your account!** popup.
- Easy on the eye

- In an ideal world, it wouldn't matter how attractive your UI is (or isn't). After all, if your content is good, what does it matter if your buttons aren't perfectly aligned? Or if your images aren't quite as high-definition as they could be?
- The simple truth is that it does matter. A lot.
- It's not enough for your UI to be clear, consistent, and responsive—it has to look nice, too.

It's also worth noting that creating an appealing, professional-looking UI may mean forgoing your own tastes and sticking with the safe option. Maybe you have a fondness for neon colors and have always thought that orange and green is a criminally underrated color combination, but remember that you're trying to attract as wide an audience as possible; is this aesthetic *really* going to appeal to the masses?

Why is UI essential for the success of your app?

To fully understand why mastering the art of the effective UI is so important, let's look at some of the things your app's UI can help you achieve, if you get it right.

Instant familiarity

If you follow best practices and Material Design guidelines, your app will reflect the UI principles the user has encountered many times before in other Android apps. Your users will feel instantly at home and also understand how to interact with many elements of your app, even if they're launching it for the very first time. For example, once the user has encountered a floating action button in one app, they'll know that tapping this button will give them access to important actions (also known as **promoted actions**).

Easy and enjoyable to use

Your app's UI determines how easily users can make your app do what they want. Create a UI that helps users get the value out of your app quickly and with the minimum amount of effort, and you're well on your way to racking up those 5 star Google Play reviews.

Although providing valuable content is still a crucial part of developing an effective app, remember that your app's UI is equally important. If your app's useful features and great content is hidden behind a clunky and generally unpleasant UI, then no one is going to hang around long enough to discover just how much your app actually has to offer.

Make sure you put as much effort into designing and developing your UI as you put into crafting your app's content and features.

Consistency

A good UI establishes the rules of your app early on and sticks with them throughout. Once users feel comfortable interacting with a single screen in your app, they should be able to find their way around your *entire* app.

Preventing user frustration

Your UI should ensure users never feel confused or frustrated by your app, by gently guiding them toward the tasks they need to complete next in order to get value from your application.

Whether your UI takes the subtle approach (such as using size and color to make certain UI elements stand out) or a more obvious approach (for example, highlighting the text field the user needs to complete next), you should make sure the user never has to sit back and ask, "*So what am I supposed to do next?*"

Helping users fix their mistakes

A good UI is like a helpful, non-judgmental friend, gently pointing out where you've gone wrong and giving you advice on how to fix it.

Imagine your app contains a form that users need to complete before tapping **Submit**. The user completes this form and taps the **Submit** button, but nothing happens. At this point, your app can either leave them scratching their head and wondering whether the **Submit** button is broken, or your UI can step in and show them what's wrong by underlining the one text field they forgot to fill in.

Providing a better overall Android experience

Because it isn't *all* about your app.

If you design your UI well and follow Material Design principles, your app will feel like a seamless extension of the Android platform and an extension of the other apps users have installed on their device.

By putting the effort into designing your UI, you can actually improve the user's overall Android experience. No pressure, then!

The UI case study – Google+

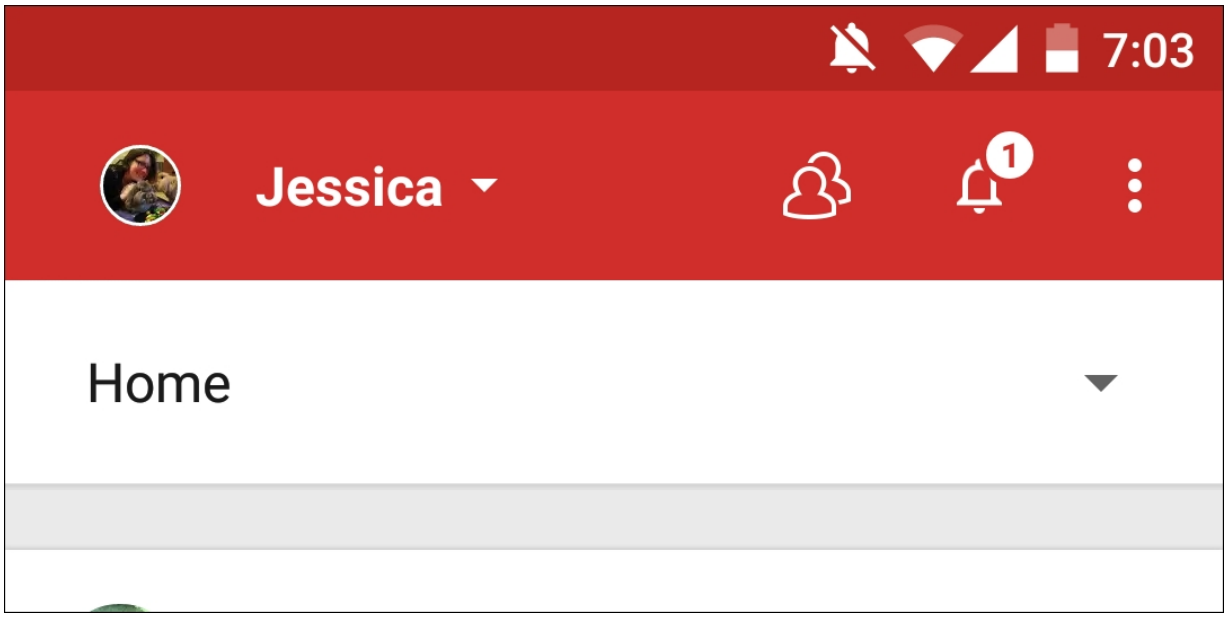
When developing apps for Android, you never have to start completely from scratch, as the Android platform does a really good job of supplying a wide range of prebuilt UI components that you can simply drop into your UI.

This means that the same UI elements crop up time and again in all sorts of Android apps. So, it's well worth spending some time seeing what we can learn from other Android applications—particularly apps that do UI well.

In this section, we'll analyze the Google+ app, which comes preinstalled on many Android devices, and is also available to download for free from Google Play. We'll work our way through all the core UI components that Google uses in this app and look at what makes them so effective.

The action bar

The action bar runs along the top of the screen and typically contains action and navigation elements:



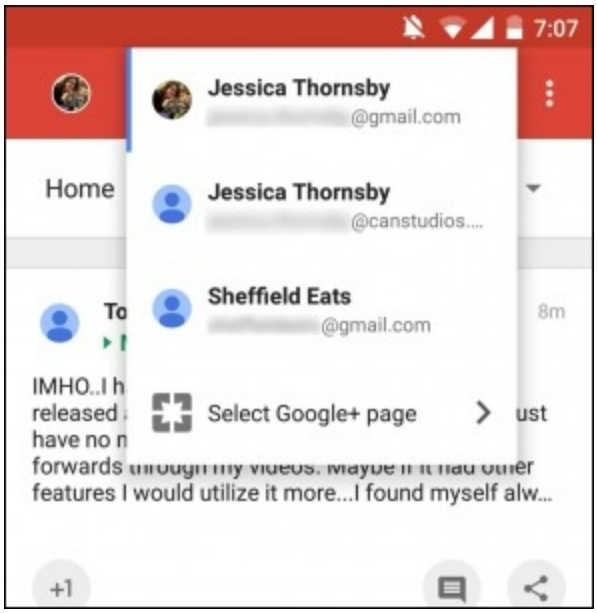
Google+'s action bar contains three important UI elements that you'll typically want to include in your own Android UIs.

Navigational controls

Action bars tend to persist throughout an app, which makes them perfect for providing your users with a consistent, easily accessible way of navigating your UI.

Navigational controls can either be straightforward, such as the standard **Back** button, or they can be more complicated, such as view-switching controls.

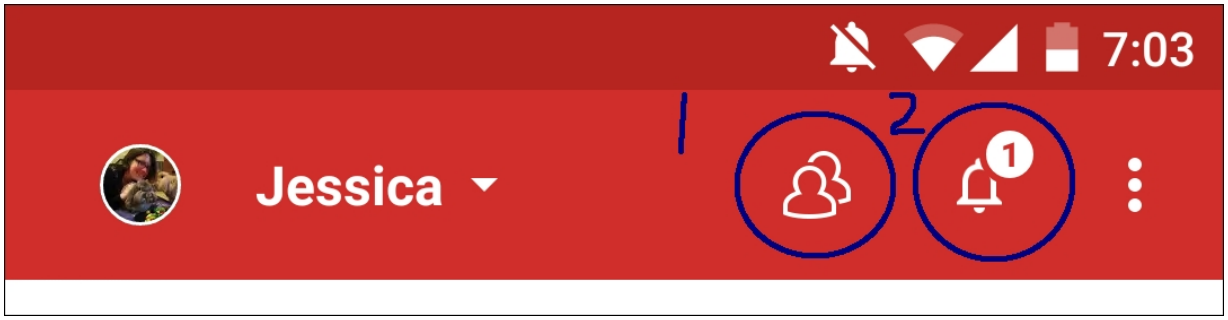
The main Google+ action bar contains a navigational control that gives users the ability to switch between their Google+ accounts, which is handy for anyone who has multiple accounts, such as a work Google+ account and a personal account.



Action buttons

The action bar is also ideal for providing easy access to your app's most important actions, based on the user's current context. The actions that appear in the action bar are known as **action buttons**.

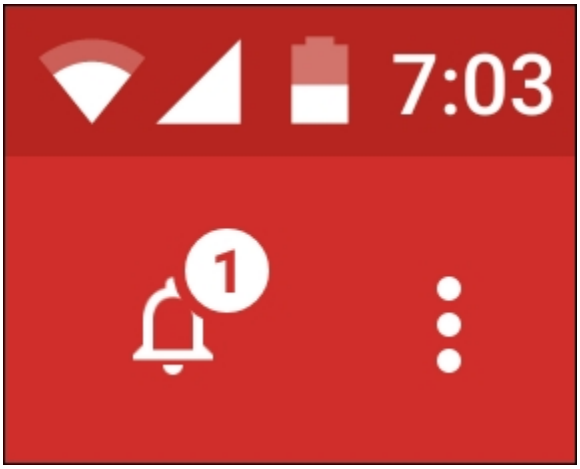
In the following screenshot, the Google+ action bar contains two action buttons:



You'll typically represent an action button with an icon and/or text. Android provides a range of ready-made action buttons that you should use wherever possible, as they ensure that the average user will be familiar with at least some of the action buttons that you use in your UI.

The action overflow

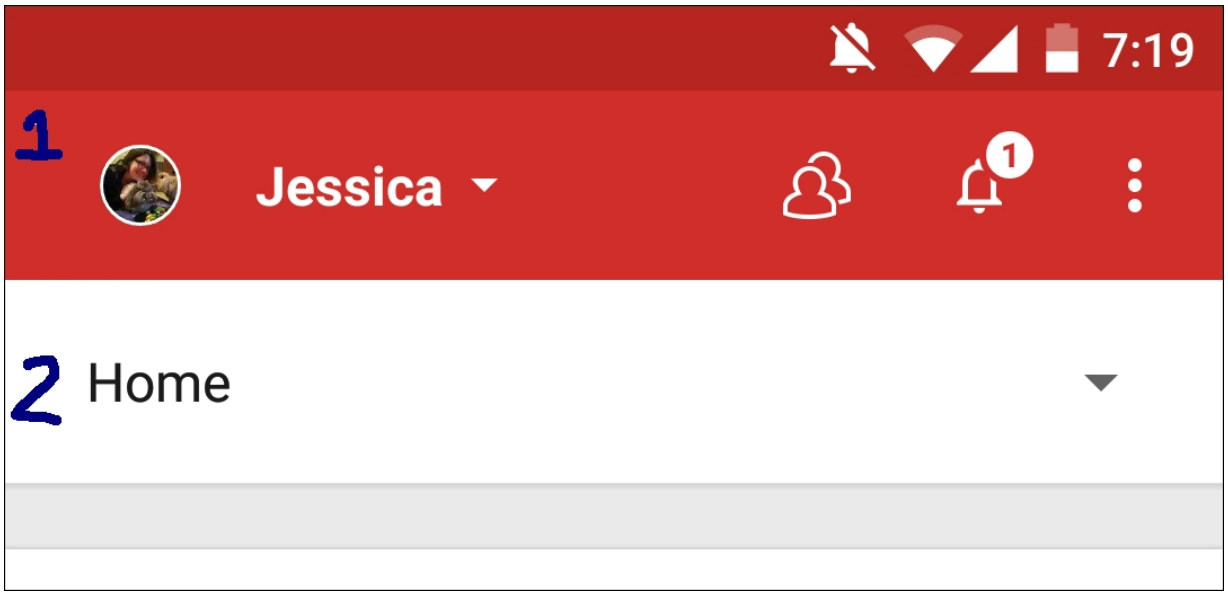
Depending on the app and the available screen space, it's possible that all your action buttons won't always fit into the action bar. If this is the case, the Android system will automatically move some of these action buttons into the action overflow, which is the dotted icon that appears in the upper-right corner throughout the Google+ app:



How many action buttons wind up hidden in the action overflow depends on the user's screen. At the extreme ends of the scale, a tablet held in the landscape mode will have much more space available for the action bar, compared to a smartphone held in portrait mode.

When designing your action bar, make sure you place the most important action buttons toward the left of the screen, so they're less likely to wind up in the action overflow. Actions that your users are less likely to need can be placed toward the end of the action bar where they have a greater risk of winding up in the action overflow.

Although the action overflow does help to reduce action bar clutter, in some scenarios, you may want to display a large number of actions without risking *any* of them ending up in the action overflow on smaller screens. If this is the case, you may want to use a split action bar:

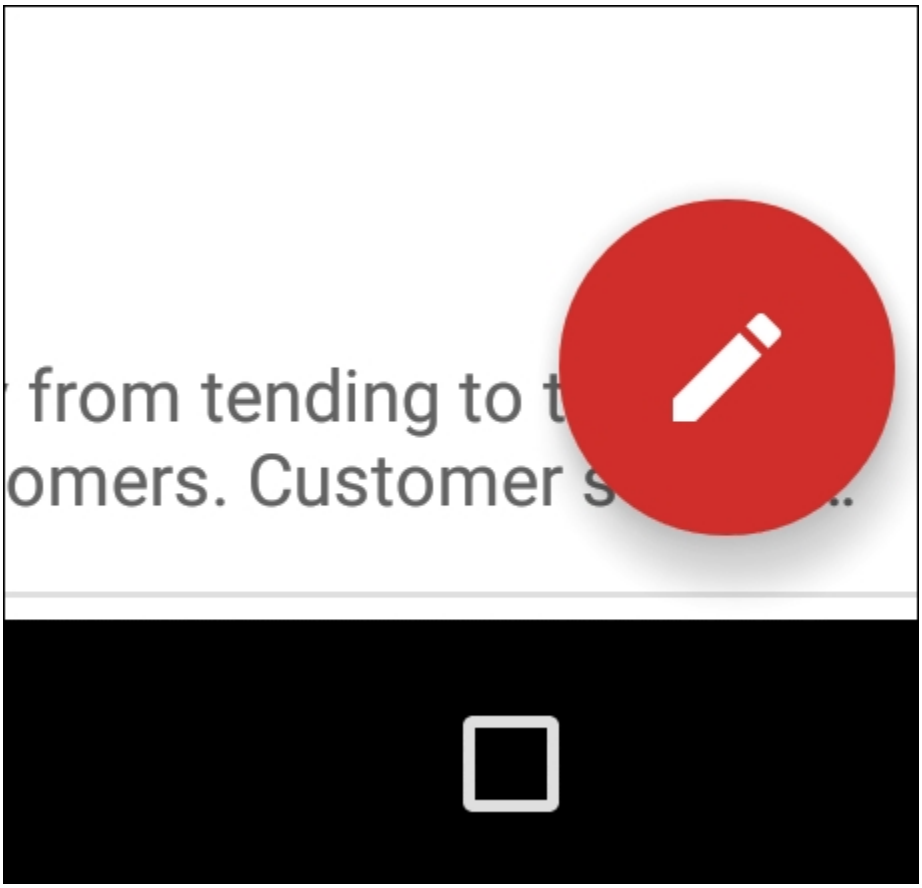


In the previous screenshot, the Google+ app uses a split action bar to reduce action bar clutter without relegating any actions to the action overflow. A split action bar is also useful to prevent the action overflow from growing out of control and becoming unmanageable.

Floating action button

The main Google+ screen is the home screen, which happens to contain a special kind of action button that was introduced as part of Google's Material Design overhaul.

This is known as a **floating action button** (**FAB**), and it's displayed prominently in the bottom-right corner of the Google+ home page:



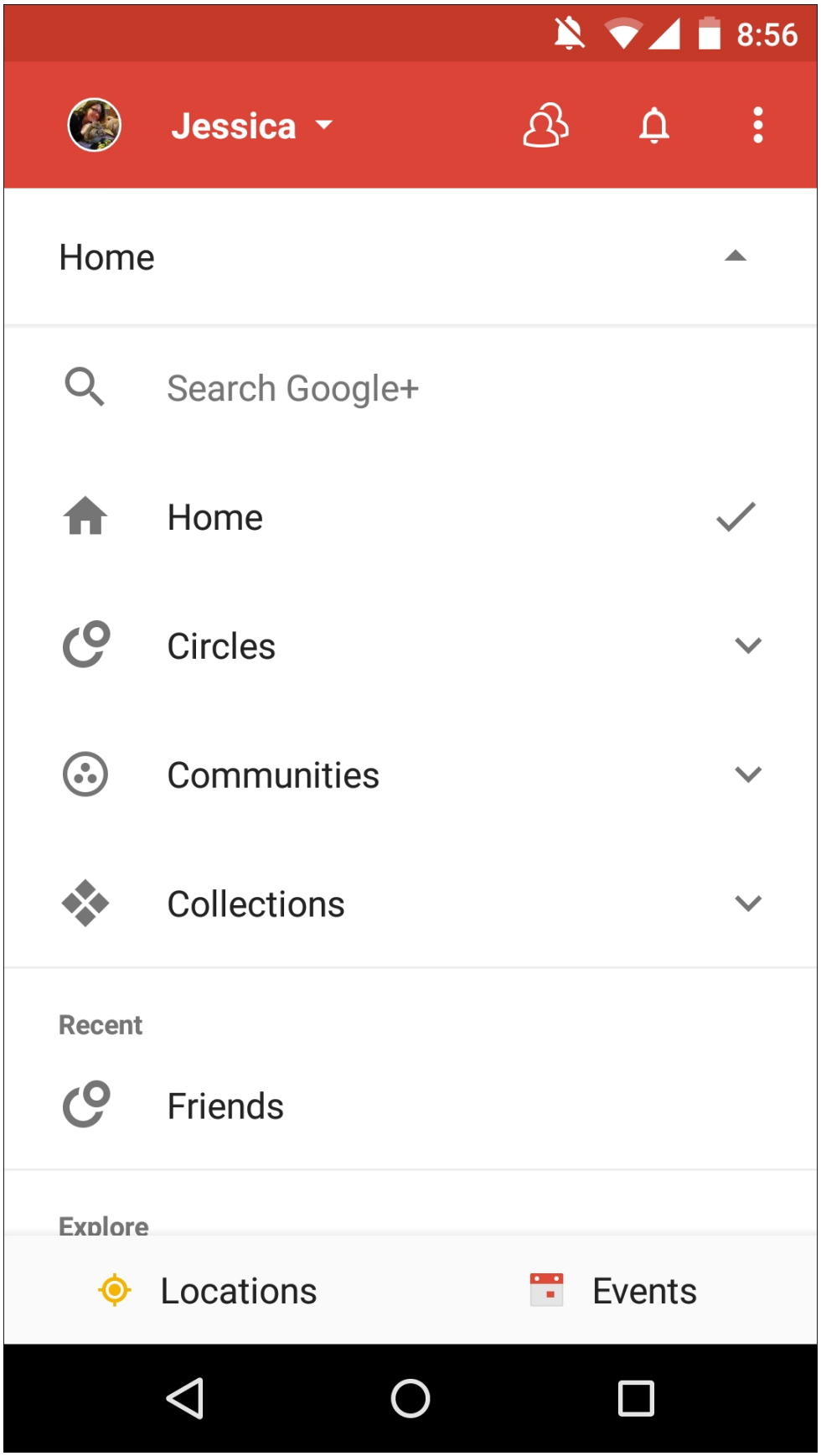
As the name suggests, FABs are circular buttons that *float* above the main user interface. These floating buttons give you a handy way of highlighting the primary action within the current context.

Google+ uses its prominent floating action button as a way of encouraging the user to join in the conversation by writing a status update.

Menus

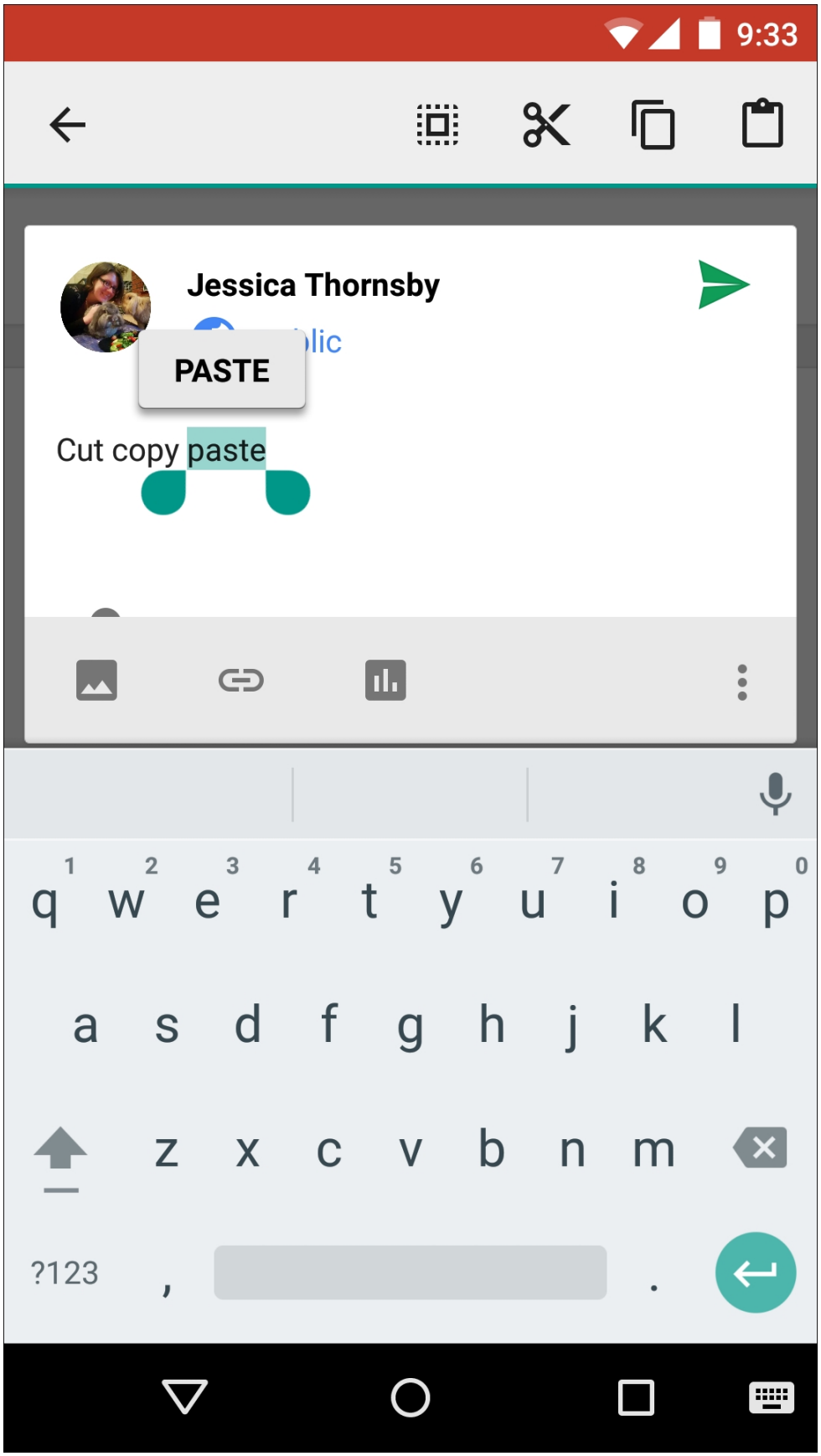
Menus are one of the main ways of navigating an app, and they are an essential component of most Android UIs.

Menus present the user with a list of options, which are usually represented by a single word or one line of text at the most:


8:56
Jessica
Home
Search Google+
Home
Circles
Communities
Collections
Recent
Friends
Explore
Locations
Events

Google+ features lots of different menus, but the one that deserves a special mention is the context menu.

Context menus are special because they dynamically update to include the actions that are relevant to the selected content only. Context menus appear when you long press an onscreen element. The context menu you've probably encountered the most often is the Select All/Cut/Copy/Paste menu, which appears when you long press some text:


9:33
Jessica Thornsby
PASTE
Cut copy paste
?123

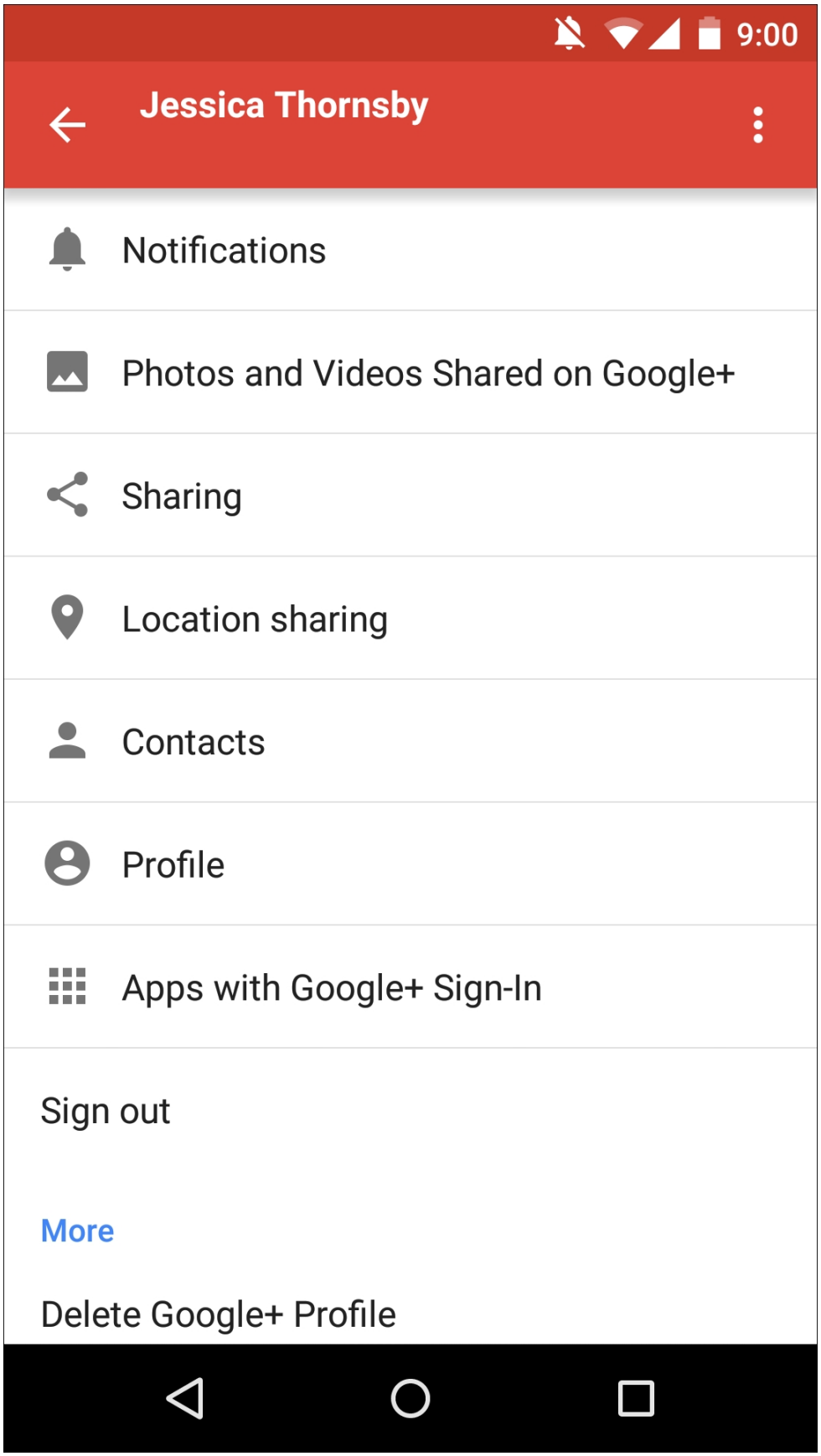
Settings

Settings is a mainstay of many Android applications as it gives users a way to tweak an app's behavior to better suit them. If users can modify your app in any way, you'll need to provide them with some form of settings screen.

The main challenge when creating a settings screen is ensuring your users will be able to understand all the available options and their current values at a glance. This means designing your settings in such a way that each screen only ever contains a manageable number of options (usually less than `10`).

If you have lots of options you want to include in your app's settings, resist the temptation to display them as one long list. Instead, divide them into groups of related options. Then you can either display your settings as a single screen formatted as multiple lists of related items, or you can take a leaf out of Google+'s book and move each group of related options to its own page.

If you opt for the latter, your main settings screen can then serve as an index, linking the user to each subpage, which is exactly the approach Google+ takes with its main **Settings** screen:


9:00
Jessica Thornsby
Notifications
Photos and Videos Shared on Google+
Sharing
Location sharing
Contacts
Profile
Apps with Google+ Sign-In
Sign out
More
Delete Google+ Profile

For ease of use, you should prioritize your settings so the options users are most likely to need are always within easy reach at the top of the settings screen.

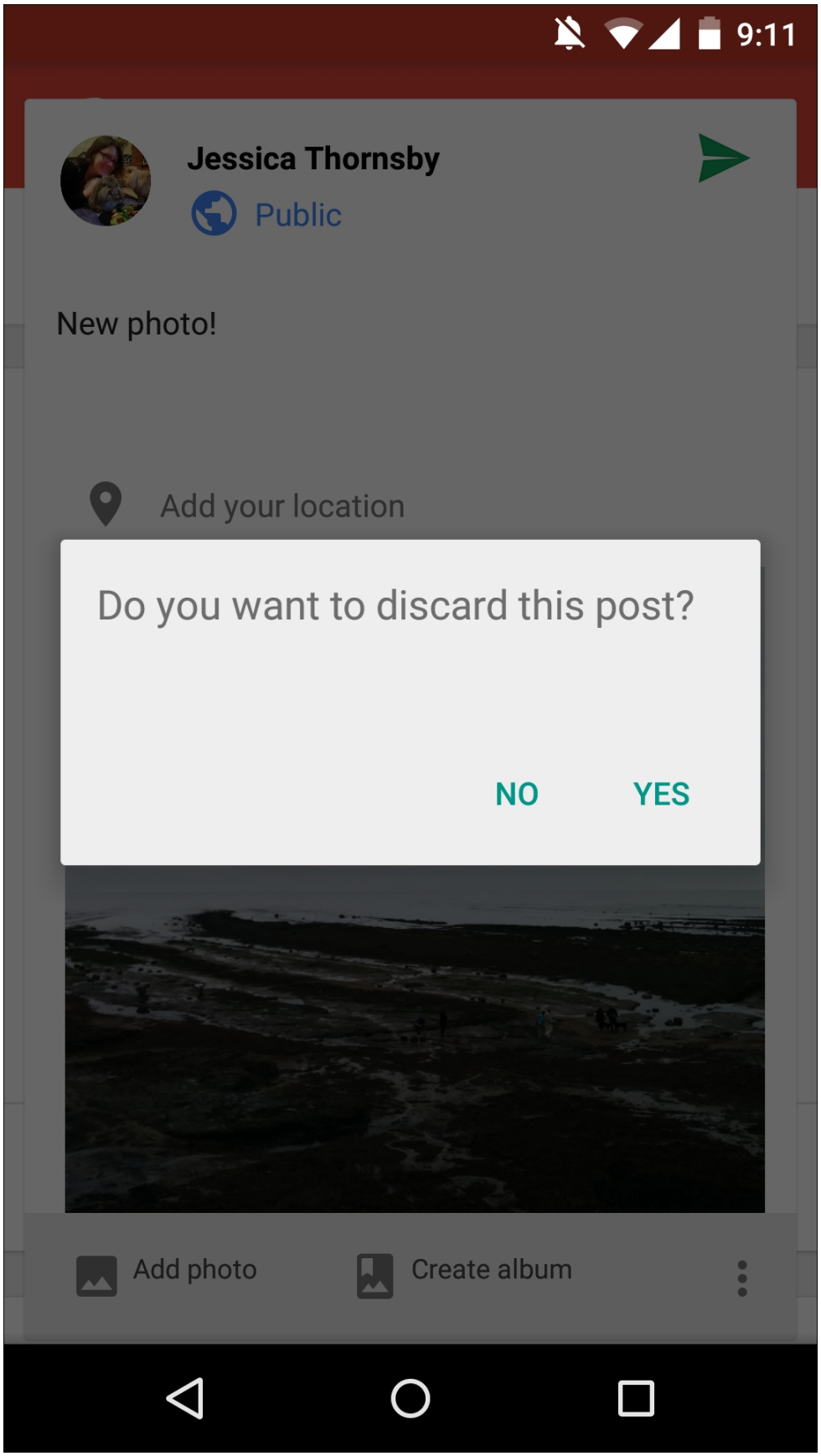
This can also be seen in Google+'s settings where obscure options, such as **Delete Google+ Profile**, are confined to the very bottom of the settings screen.

Dialogues

From time to time, you may need to *really* grab the user's attention, and the UI elements that can help you with this are dialogues.

A dialogue is a small window that can contain text and buttons and is typically used for one of the following purposes:

- Giving the user some important information, such as terms and conditions, or a disclaimer they need to read before they can progress to the next task
- Requesting additional information, for example, a dialogue that prompts users to enter their password
- Asking users to make a decision; for example, if you draft a Google+ update and then try to navigate away without posting that update, Google+ uses a dialogue box to check whether you *really* do want to discard your post


9:11
Jessica Thornsby
Public
New photo!
Add your location
Do you want to discard this post?
NO
YES
Add photo
Create album

Toasts

A toast is a small popup that provides the user with simple feedback.

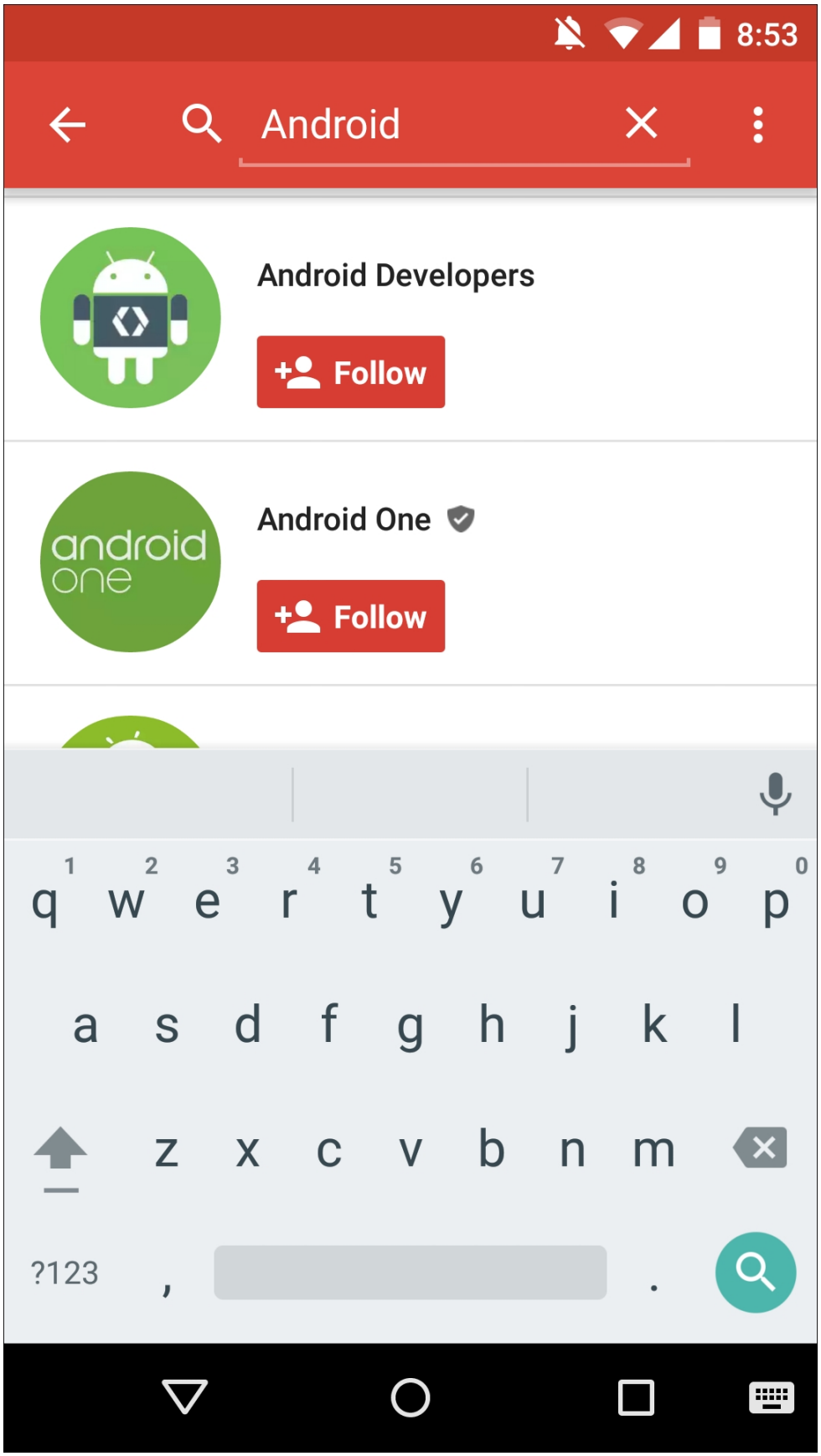
Unlike dialogues, the app underneath the toast remains active the entire time the toast is visible. Opt for a toast rather than a dialogue when you want to convey a short and simple message to the user, and crucially, you don't require any user input—toasts don't support any kind of user input.

In Google+, you'll occasionally see a toast when new posts are available to view on the home screen. This toast will prompt you to swipe the screen to see updates.

Search

Android users typically expect to be able to search all the data that's included in your app, so most UIs include some form of search functionality.

The good news is that the Android platform has a ready-made search framework that you can implement in your UI. Using this standard search framework isn't just easier for you, it's also easier for your users, as most Android users will immediately know how to interact with Android's standard search functionality, since they'll have encountered the underlying framework many times before (even if they weren't aware of it):


8:53
Android
Android Developers
Follow
android one
Android One
Follow
1 2 3 4 5 6 7 8 9 0
q w e r t y u i o p
a s d f g h j k l
z x c v b n m
?123
,
.

You can use Android's standard search framework to implement searching in one of the following ways:

- **A search dialogue**: This dialogue is hidden by default. When activated by the user, it appears as a floating search box at the top of the current activity. The search dialogue component is controlled by the Android system.
- **A search widget**: This is an instance of `SearchView` that you can place anywhere in your app's UI. If you do decide to use the search widget, you'll need to do some additional work to get the Android system to handle search events that occur via this widget.

Input controls

Input controls are a variety of components provided by the Android platform that the user can interact with. Boot up any Android app on your device, and chances are it'll contain some form of input control—whether it's a button, a checkbox, a text field, or something else.

Since your average Android user will have encountered these standard input controls many times before, including them in your UI guarantees that the majority of users will immediately know how to interact with at least *some* of your app's core UI elements.

> ## Note
>
> Input controls are a crucial part of almost any Android UI, so we'll be covering them in much greater detail in the next chapter.

Styles and themes

We've looked at lots of different UI elements you can find in the Google+ app, but what ties all these elements together and ensures your UI feels consistent?

The answer is styles and themes.

While Android does give you the power to customize the look and feel of *every* part of *every* screen of your UI, you'll have to balance this against providing a consistent UI experience for your users. Themes and styles are powerful tools that can help you provide this app-wide consistency.

Styles and themes allow you to define a group of attributes once and then use them throughout your app. So, you could create a style that specifies that text should be blue with the monospace typeface, and then you can apply this style across your entire application. Instantly, all your text will have the same formatting.

Styles and themes can be simple, such as our text example, or you can create long lists of attributes that dictate how every element of your UI should look and feel, from the shade of the action bar, to the background image and the amount of padding used throughout your layout.

Although they're often talked about together, there is a one crucial difference between styles and themes:

- **Style**: This is a collection of properties that you can apply to an entire activity or application, or to an individual view
- **Theme**: This is a style that you can apply to an entire activity or application, but not to an individual view

The Android platform provides plenty of predefined themes and styles that you can use in your Android apps (including some crucial Material themes), or you can create your own.

Although you can create themes and styles completely from scratch, you'll usually want to take Android's ready-made themes and styles as your starting point, and then define any properties that you want to add, change, or remove. This is called **inheritance**, and it's one of the quickest and easiest ways of creating a UI that feels familiar and professionally designed but is still subtly unique to your app.

Summary

In this section, we looked at why the quality of your user interface is crucial to creating an app that'll have no problems winning over a wide audience, before taking a closer look at a real-life example of an effective and well-designed Android UI.

In the next chapter, we'll take a closer look at the building blocks of every Android UI: Views, ViewGroups, and layouts. Although we touched on input controls in this chapter, in the next chapter we'll look at them in more detail, including how to implement some of the most common input controls in your own user interfaces.

Chapter 2. What Goes into an Effective UI?

In this chapter, we'll look at the core components of every Android user interface: layouts and views. You'll learn how to add these elements to your Android projects and all the different ways you can customize these UI building blocks to meet your exact design needs. We'll also look at how to enhance our UI using strings, colors resources, and state lists, and we'll begin to look at how we can create a user interface that looks crisp and clear regardless of the screen it's being displayed on.

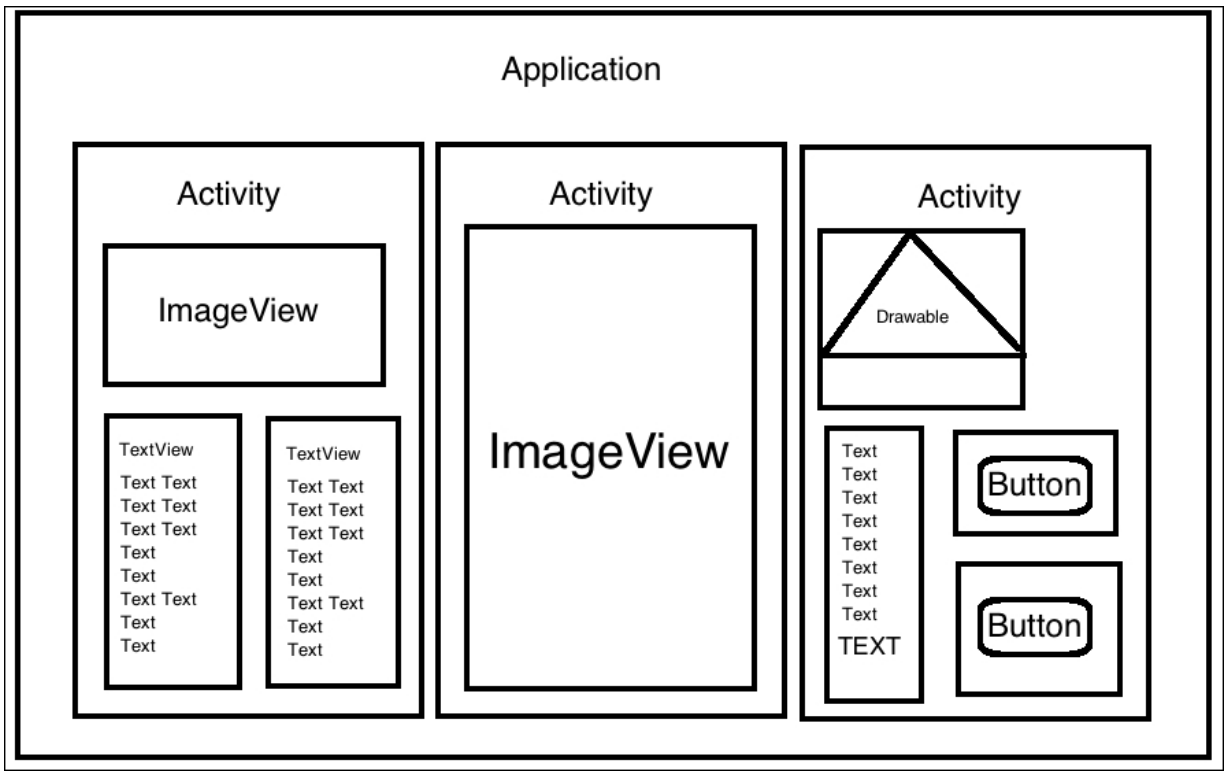
Although we'll be exploring layouts and views in greater detail throughout this chapter, the two are intrinsically linked. Before you can add a view to your app, you need a layout, and a layout without any views isn't likely to win your app any fans.

So, before we dive into the finer details of UI design, let's get an overview of how views and layouts come together to create an effective Android UI.

What is a view?

As you're already aware, Android apps are made up of Activities. Typically, one **Activity** is displayed at a time and this Activity occupies the entire screen.

Each Activity is made up of views, which are the most basic component of a user interface. Views always occupy a rectangular area, although a view can display content of any shape:

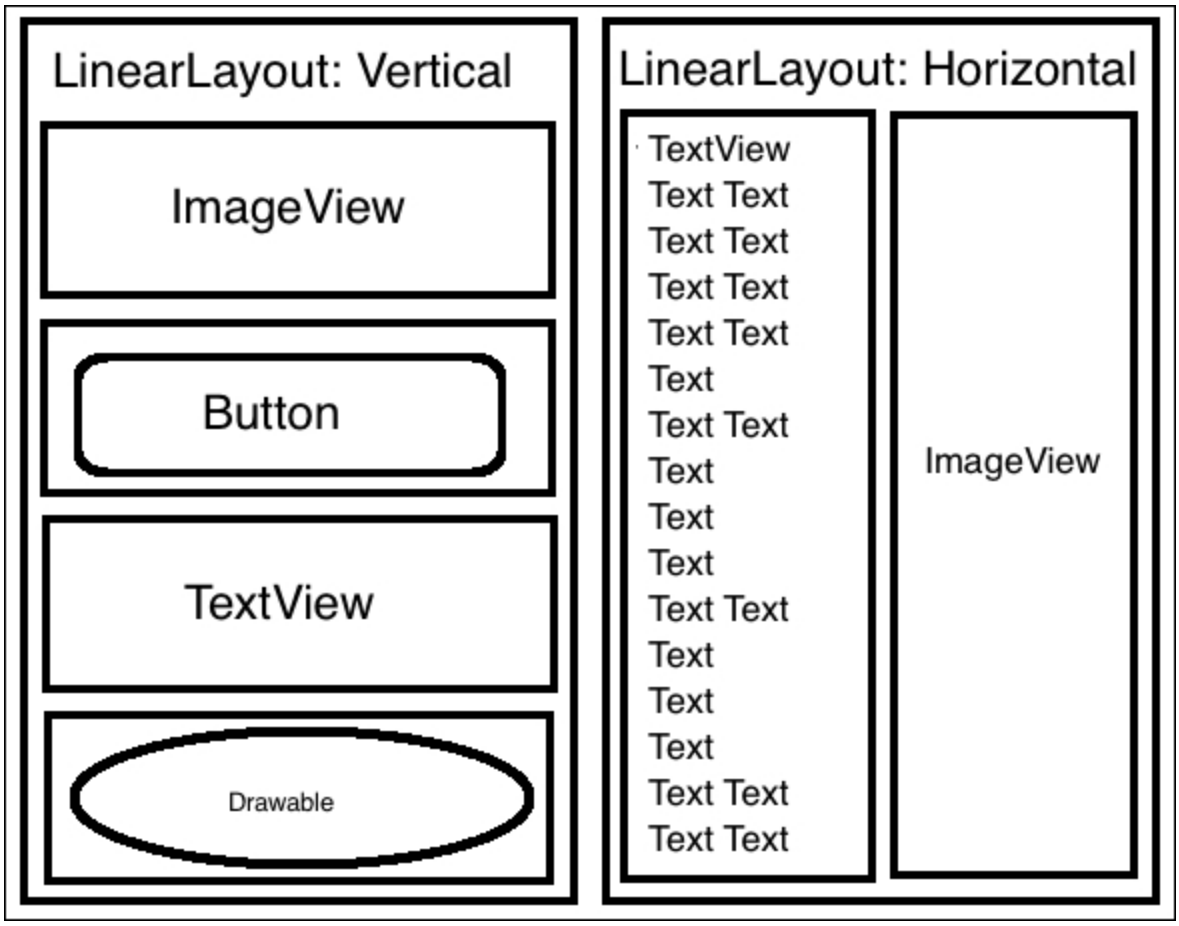


Some examples of the most commonly used views are `TextView`, `EditText`, `ImageView`, `Button`, and `ImageButton`.

What is a layout?

A **ViewGroup** is a container that groups other child views and ViewGroup objects together.

One of the most common examples of a ViewGroup is a layout, which is an invisible container that's responsible for positioning the child elements on the screen. For example, **LinearLayout** is a ViewGroup (also sometimes known as a **layout manager**) that arranges its child elements (views or ViewGroups) into vertical or horizontal rows:



In this chapter, I'll mostly be focusing on layout managers, as these are the ViewGroup you'll typically use most often; but just be aware that other kinds of ViewGroup also exist.

Building your UI – XML or Java?

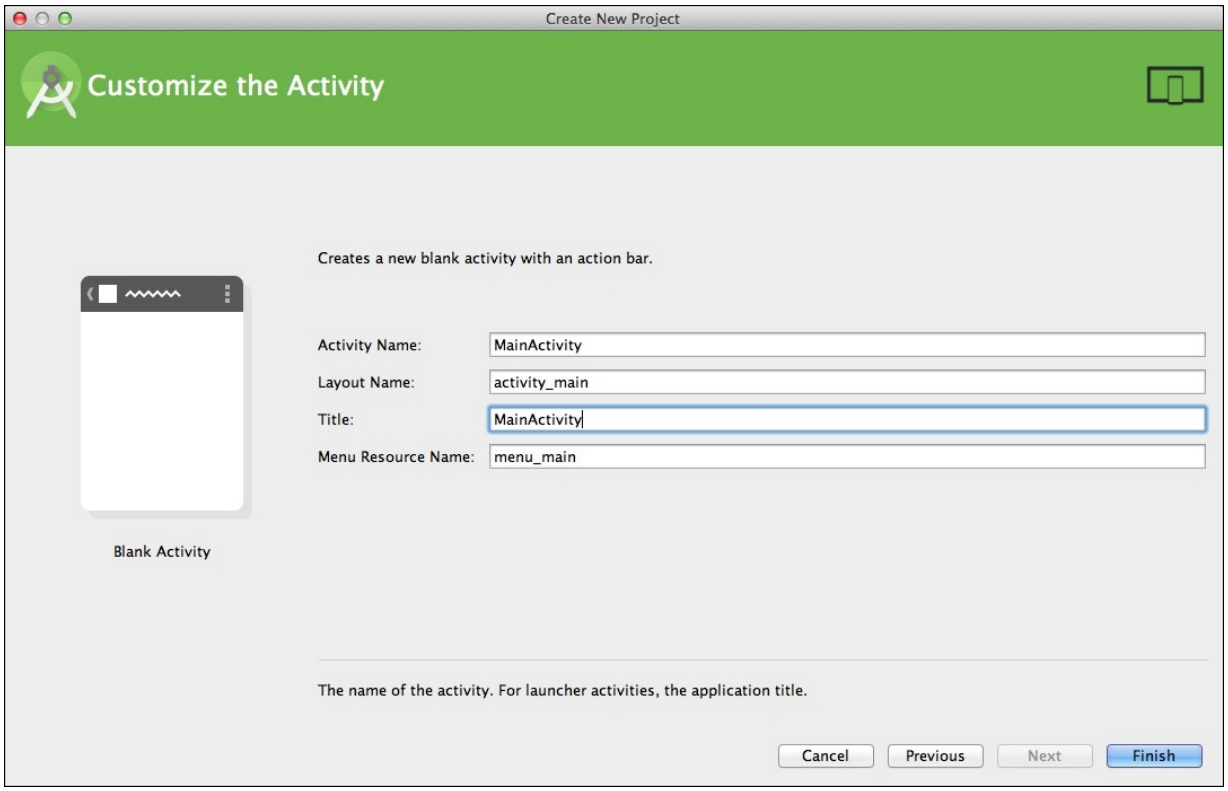
The easiest way of defining your user interface (and the views, ViewGroups, and layout elements that it contains) is via your project's XML file.

Declaring your UI with XML

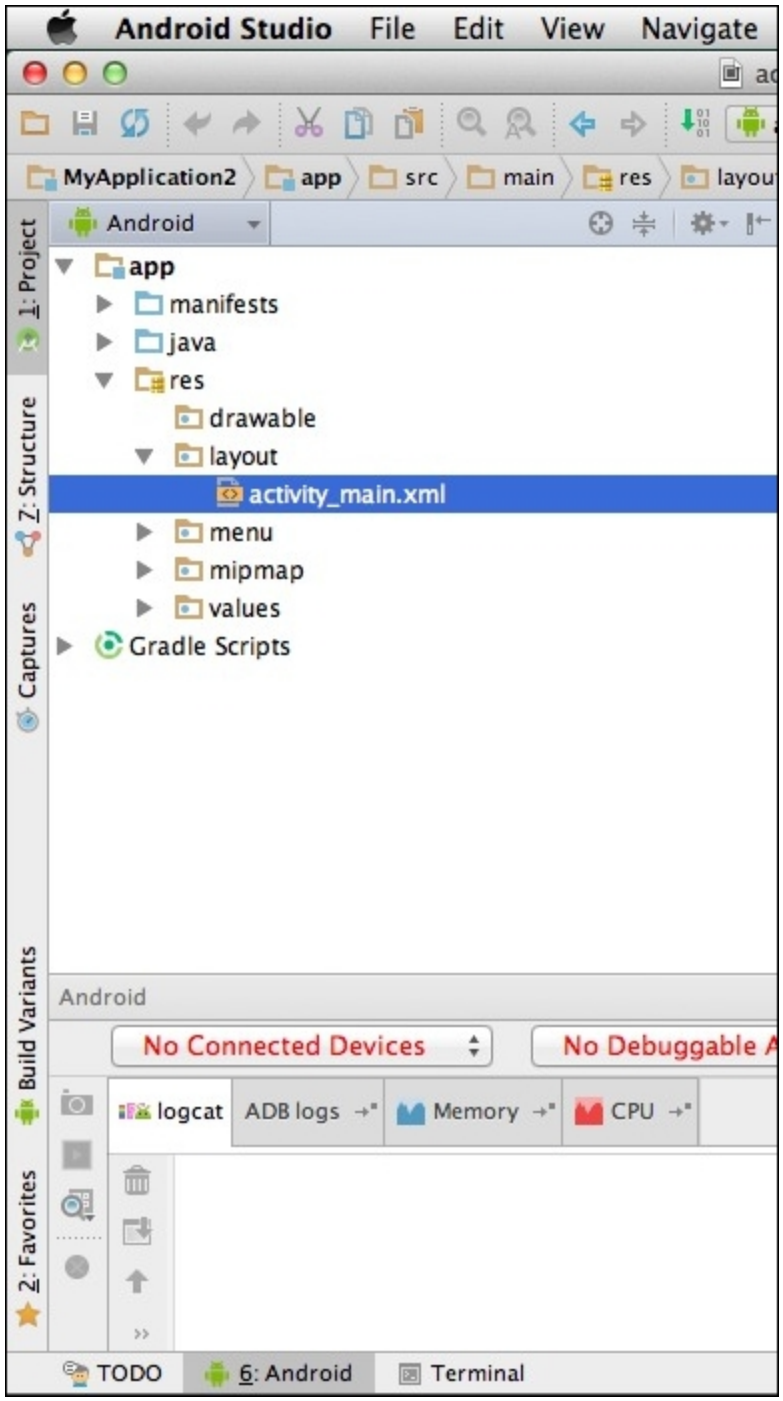
Android provides a straightforward XML vocabulary that gives your user interface a human-readable structure, and creates a separation between the code that defines your UI and the code that controls your app's behavior. You define your layouts in XML in a dedicated layout resource file. This helps to keep both sets of code cleaner, and it gives you the ability to tweak and refine your UI without having to touch your app's underlying code. For example, you can update your layout to support an additional language *without* having to touch the previously-tested code.

Declaring your UI in XML also makes it easier to provide alternate layouts; for example, at some point, you may want to create an alternative version of your layout that's optimized for landscape mode. If you declare your original layout in XML, providing a landscape-optimized layout is as easy as creating a `res/layout-land` directory, and then filling this folder with XML files that define your app's landscape-optimized layouts.

When you create an Android project in Eclipse or Android Studio, the IDE's project creation wizard automatically generates a layout resource file for your application's main activity:



You'll find this layout resource file in your project's `res/layout` folder:



Each layout resource file must contain exactly one root element, which can either be a view or a ViewGroup. For example, you can use the vertical `LinearLayout` element as your layout's root element:

```
<?xml version="1.0" encoding="utf-8"?>
```

```
<LinearLayout
xmlns:android="http://schemas.android.com/apk/res/
android"    android:orientation="vertical"
android:layout_width="fill_parent"
android:layout_height="fill_parent">

</LinearLayout>
```

> **Note**
>
> You can also use the `<merge>` element as your root element. We'll cover merging in Chapter 9, *Optimizing your UI*.

Once you've defined your layout resource file's root element, you're ready to build a view hierarchy by adding objects such as `TextViews`, `Buttons`, and `ImageViews`.

To load a layout resource file, you need to reference it from your application's `onCreate()` callback implementation. For example, open your project's `MainActivity.java` file and you will see something like this:

```
public void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  setContentView(R.layout.main_layout);
}
```

Here, your project calls `setContentView()` and passes it the reference to your project's automatically-generated resource file; in this instance, it is `R.layout.main_layout`. When the user loads your application, `MainActivity` will read this referenced layout resource file and display its content to the user.

Declaring your UI programmatically

The second option is to create your UI programmatically at runtime. This approach isn't generally recommended, as it means that your app's underlying code and the UI code get mixed up. So, tweaking

an element of your app's UI becomes much more difficult than it needs to be.

However, sometimes you will need to define certain aspects of your user interface programmatically, and occasionally you may even need to define the whole thing in Java.

As we've already seen, when you define your layout resource in XML, you load it from your application code:

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

Here, you're telling your Activity to load the `main_activity.xml` layout resource file, but if you're creating your layout programmatically, you need to remove this bit of code so that your Activity doesn't go looking for a layout resource file.

For example, take a look at the following:

```
@Override
protected void onCreate(Bundle savedInstanceState)
{
super.onCreate(savedInstanceState);

  // Create a LinearLayout object that will
contain all the views in this Activity//

    LinearLayout linearLayout = new
LinearLayout(this);

//Set the LinearLayout's orientation. In this
instance, it's set to horizontal//


linearLayout.setOrientation(LinearLayout.HORIZONTA
L);

  // Create a LayoutParams object to be used by
the LinearLayout//

    LayoutParams linearLayoutParam = new
LayoutParams(LayoutParams.MATCH_PARENT,
```

```
LayoutParams.MATCH_PARENT);


    // Set LinearLayout as the screen's root
element//
    setContentView(linearLayout,
linearLayoutParam);
  }
```

Using both programmatic and XML layouts

Sometimes the best solution maybe to use both programmatic and XML layouts. This approach helps you to keep some separation between your UI and application code by defining the bulk of your UI in XML, while also allowing you to create a more dynamic user interface by modifying the state of some onscreen elements at runtime.

For example, your app may contain a slideshow that's controlled by a single button. When the user taps the button, a new image appears. To create this effect, you can define the button in XML, and then add a new image programmatically whenever the user taps the button.

Since XML is the easiest and most efficient way of defining your UI, this chapter mainly focuses on creating and customizing views, ViewGroups, and layouts in XML. However, occasionally you may need to define parts of your UI programmatically, so I'll also include some snippets of Java along the way.

Deep dive – exploring layouts

Now that you have an idea of how views, ViewGroups, and Layouts come together to create a user interface, and how to create your user interface programmatically and in XML, it's time to look at each of Android's most common UI components in more detail. We'll start with the component that pretty much lays the foundation of any user interface: a layout container.

The Android platform supports a range of layouts, so your first task is deciding which layout best meets your design needs. If you're struggling to make this decision, keep in mind that you can nest layouts within one another to create your perfect layout container.

> **Note**
>
> Don't go overboard with the nesting as this can have a negative impact on your app's performance. If you find yourself nesting multiple layouts, then this could be a sign that you're using the wrong kind of layout!

Before you create any kind of layout, here are a few rules, regulations, and attributes that you need to get to grips with.

Defining the size of your layouts

Whenever you create a layout, you need to tell the Android system how big this layout should be.

The XML attributes you use to define the size of your layouts are `android:layout_height` and `android:layout_width`. As their names suggest, these attributes set the height and width of your layouts respectively.

Both accept the following values:

A supported keyword

Android screens come in lots of different sizes. One of the easiest ways of ensuring your user interface is flexible enough to cope with all these differently-sized screens is to set your layout's width and height to one of the following supported keywords:

- `match_parent`: This makes the height or width expand to completely fill the available onscreen space
- `wrap_content`: This sets the height or width to the minimum size required to fit the element's content and no larger

> **Note**
>
> You can also use `match_parent` and `wrap_content` to set the size of other onscreen elements, including views and ViewGroups.

A dimension value

Alternatively, you can set the size of your layout using one of the units of measure supported by the Android system:

- **Density-independent pixels (dp)**: This is an abstract unit which is based on the screen's physical density. The dp unit is relative to 1 physical pixel on a 160 dots per inch screen. At runtime, Android automatically adjusts the number of pixels used to draw 1 dp by a factor that's appropriate for the current screen's dp. Using density-independent measurements is a straightforward solution to creating a UI that can adjust automatically across a range of screen sizes.
- **An absolute unit**: Android supports a number of absolute units of measure (specifically pixels, millimeters, and inches), but you should avoid defining your layout using absolute sizes as this

makes your UI very rigid, and can prevent it from resizing itself to suit a range of screens. Unless you have a *very good reason* not to, stick to relative measurements, such as dp, `match_parent`, and `wrap_content`.

Setting the layout size programmatically

You can also set the size of your layout programmatically by creating the `LayoutParams` object:

```
LayoutParams linearLayoutParam = new LayoutParams
```

You can then set its width and height as follows:

```
LayoutParams linearLayoutParam = new LayoutParams
(LayoutParams.MATCH_PARENT,
LayoutParams.MATCH_PARENT);
```

Exploring different layouts

Now that you know how to create layouts in XML and Java, and how to set their height and width, you can take a closer look at two of the most commonly used layouts: the straightforward and easy-to-use `LinearLayout` layout and the incredibly flexible `RelativeLayout` layout.

Everything you need to know about LinearLayout

`LinearLayout` aligns all its children in a single horizontal or vertical row, stacking them one after the other.

You set the direction of your `LinearLayout` layout using either of the following:

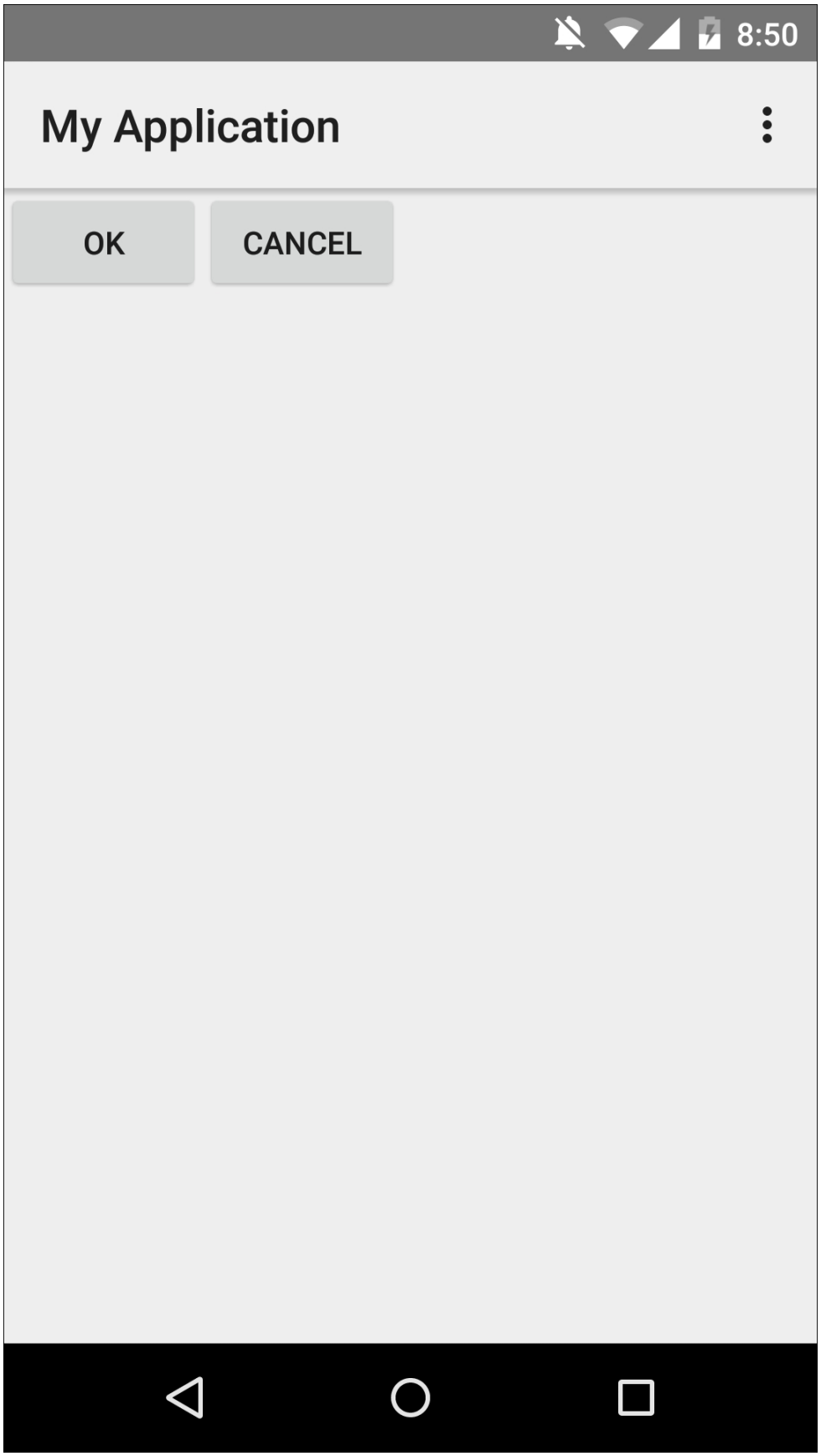
- `android:orientation="horizontal."`: The views are placed next to each other in *rows*. A horizontal `LinearLayout` layout is only ever one row high.
- `android:orientation="vertical."`: The views are placed below each other in *columns*. A vertical `LinearLayout` layout only ever has one child per row.

Here's a simple `LinearLayout` layout with a horizontal orientation:

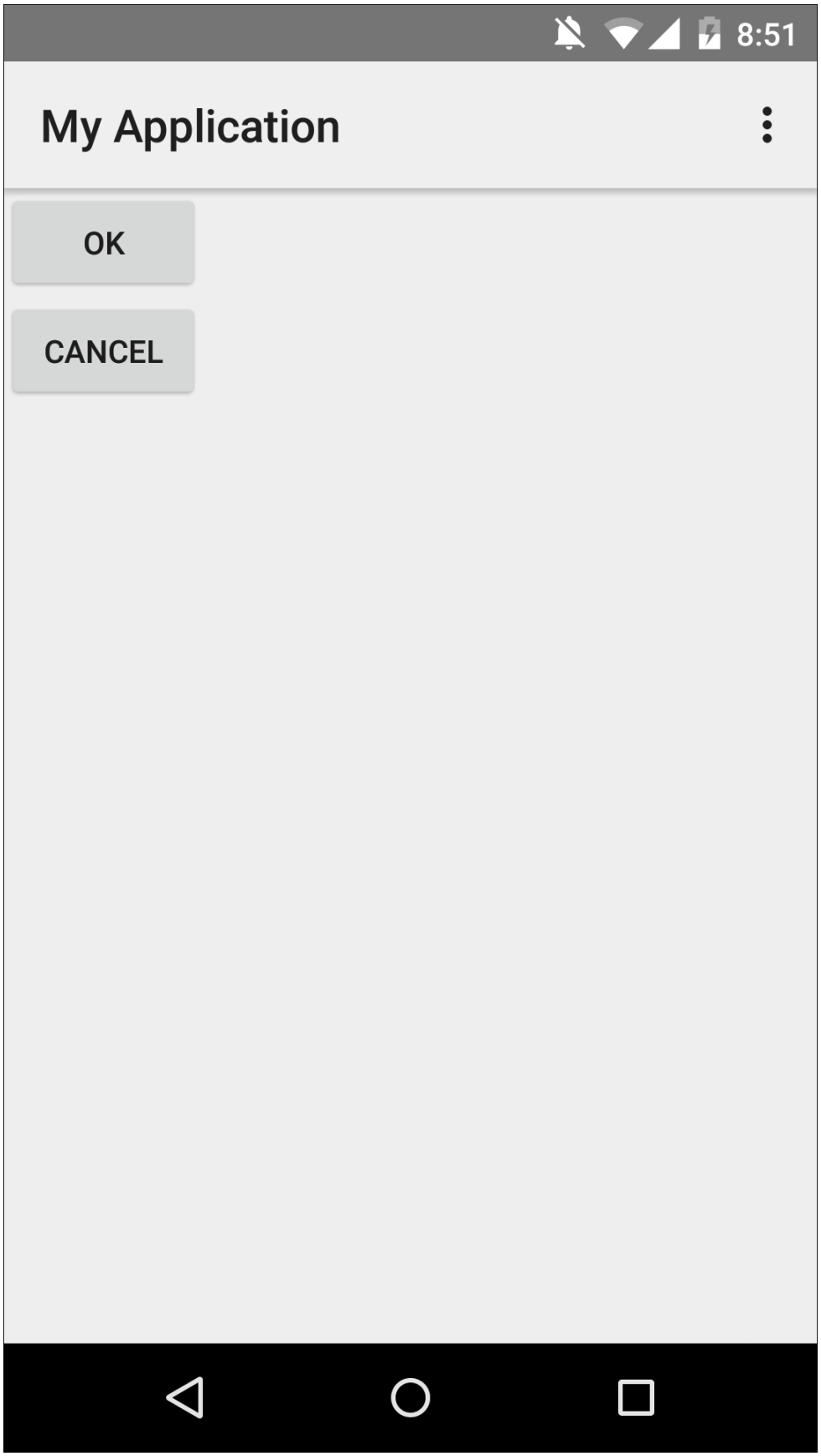
```
<LinearLayout
xmlns:android="http://schemas.android.com/apk/res/
android"
  android:orientation="horizontal"
  android:layout_width="match_parent"
  android:layout_height="match_parent">
  <Button
      android:id="@+id/okButton"
      android:text="Ok"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content" />
  <Button
      android:id="@+id/cancelButton"
```

```
        android:text="Cancel"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" />

</LinearLayout>
```

Here's how this horizontal `LinearLayout` layout appears on an Android device:


8:50
My Application
OK
CANCEL

Here's the same `LinearLayout` based UI with the orientation set to `android:orientation="vertical"` instead:


8:51
My Application
OK
CANCEL

Everything you need to know about RelativeLayout

`RelativeLayout` is one of the most flexible layouts at your disposal, giving you the freedom to position every child element based on its relationship with any other child element, with its parent container.

For example, you have the flexibility to position `TextView` so that it aligns with the edge of the `RelativeLayout` container, and then you can position a `Button` 100 density-independent pixels above `TextView`.

> ## Tip
>
> **Using RelativeLayout to optimize your UI**
>
> In addition to giving you control over where each element appears on the screen, `RelativeLayout` can improve your app's overall performance by reducing nesting. If you find yourself using several nested `LinearLayouts`, you maybe able to flatten your layout hierarchy by replacing them with a single `RelativeLayout`.

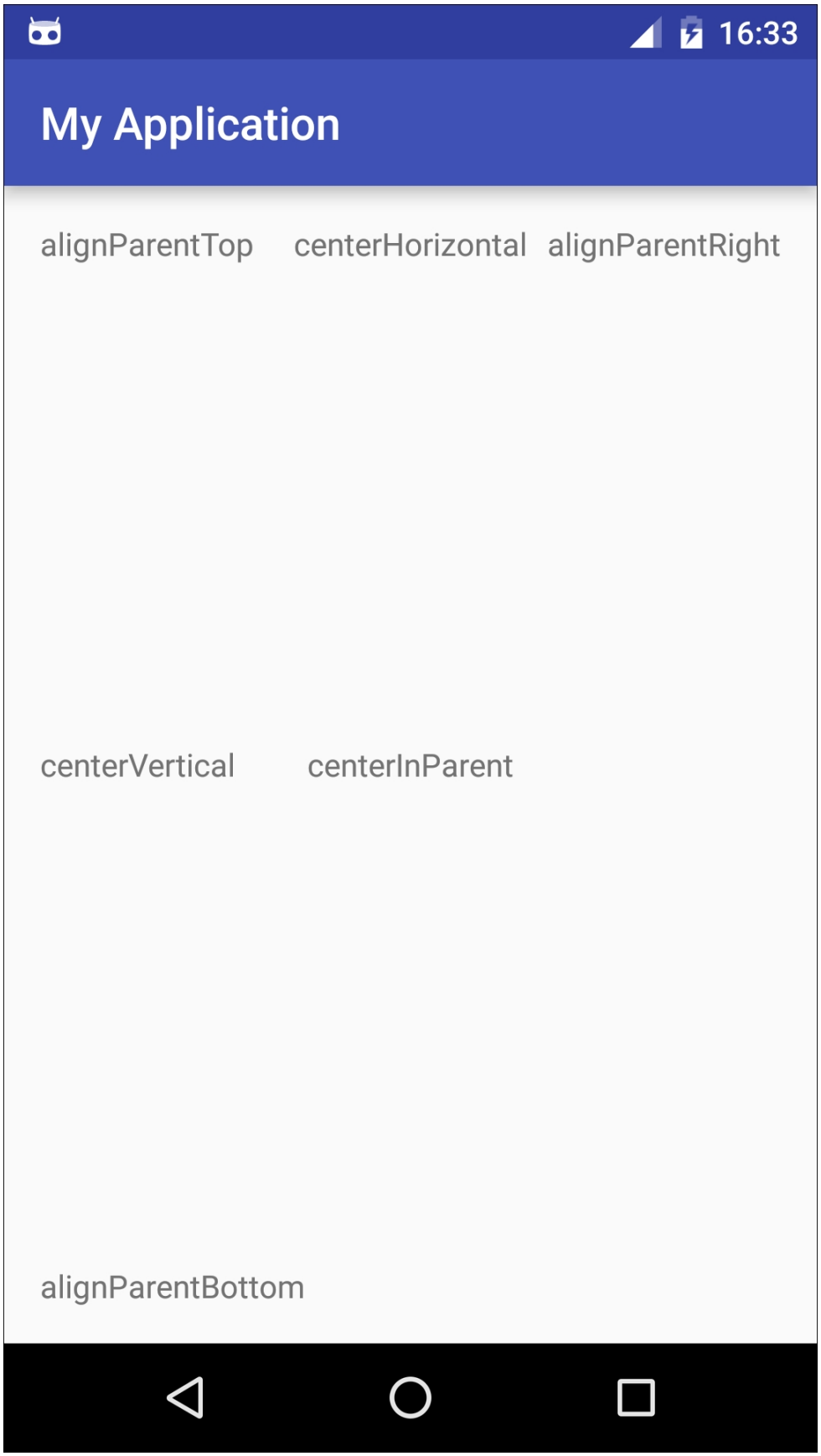
Since `RelativeLayout` is all about giving you the flexibility to position your UI elements wherever you want, it's no surprise that this layout supports a long list of attributes that let you position your UI elements relative to their parent container and relative to one another.

Relative to the parent container

All the following attributes accept the `true` value; for example, `android:layout_alignParentTop="true"` and `android:layout_alignParentStart="true."`:

- `android:layout_alignParentTop`: This aligns the top edge of a view with the top edge of its parent
- `android:layout_alignParentBottom`: This aligns the bottom edge of a view with the bottom edge of its parent
- `android:layout_centerInParent`: This centers a view horizontally and vertically within its parent
- `android:layout_alignParentRight`: This aligns the right edge of a view with the right edge of its parent
- `android:layout_alignParentLeft`: This aligns the left edge of a view with the left edge of its parent

- `android:layout_centerHorizontal`: This centers a view horizontally within its parent
- `android:layout_centerVertical`: This centers a view vertically within its parent
- `android:layout_alignParentStart`: This aligns the start edge of a view with the start edge of its parent
- `android:layout_alignParentEnd`: This aligns the end edge of a view with the end edge of its parent
- `android:layout_alignWithParentIfMissing`: If the view references an element that's missing, this attribute will align the view with the parent instead

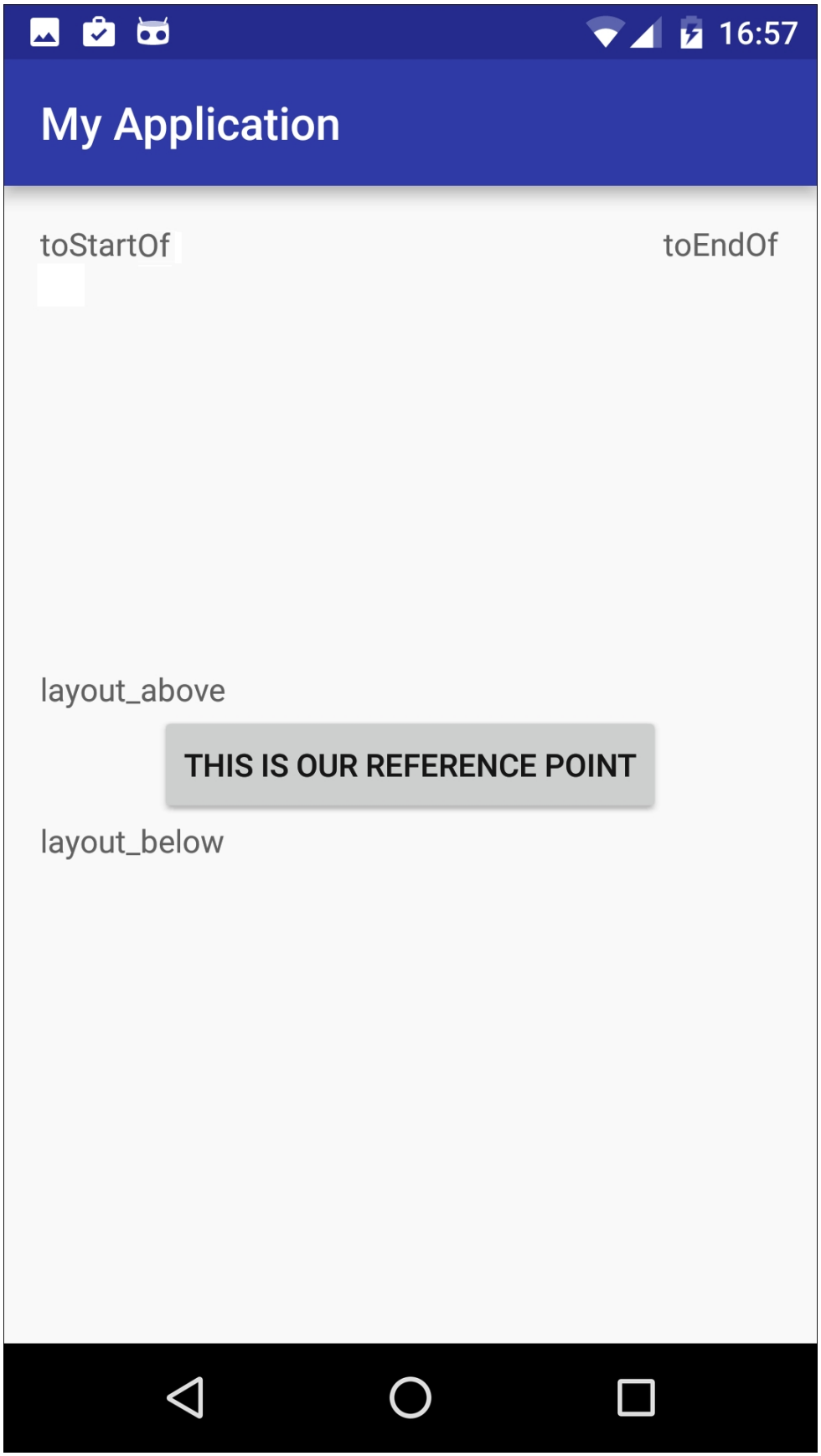

16:33
My Application
alignParentTop
centerHorizontal
alignParentRight
centerVertical
centerInParent
alignParentBottom

Relative to other elements

You can also position UI elements relative to other onscreen elements; for example, you may want to position the `back_button` view to the left of `forward_button` and position `titleTextBox` above `subheadingTextBox`.

All the following attributes should reference the ID of the element you're using as your reference point (we'll look at IDs in more detail shortly, but essentially a view's ID is the value of its `android:id` element in your layout resource file, for example, the ID of `android:id ="@+id/viewName"` is `viewName`):

- `android:layout_above`: This places a view above the specified element; for example, `android:layout_above="@+id/subheadingTextBox"` will place the UI element above `subheadingTextBox`
- `android:layout_below`: This places a view below the specified element
- `android:layout_toLeftOf`: This places a view to the left of the specified element
- `android:layout_toRightOf`: This places a view to the right of the specified element
- `android:layout_toStartOf`: This aligns the end edge of a view with the start of the specified element
- `android:layout_toEndOf`: This aligns the start edge of a view with the end edge of the specified element


16:57
My Application
toStartOf
toEndOf
layout_above
THIS IS OUR REFERENCE POINT
layout_below

Aligning with other elements

You can also position a UI element by specifying how it aligns with other on-screen elements. Again, the value for all the following attributes is the ID of the element you're using as your reference point:

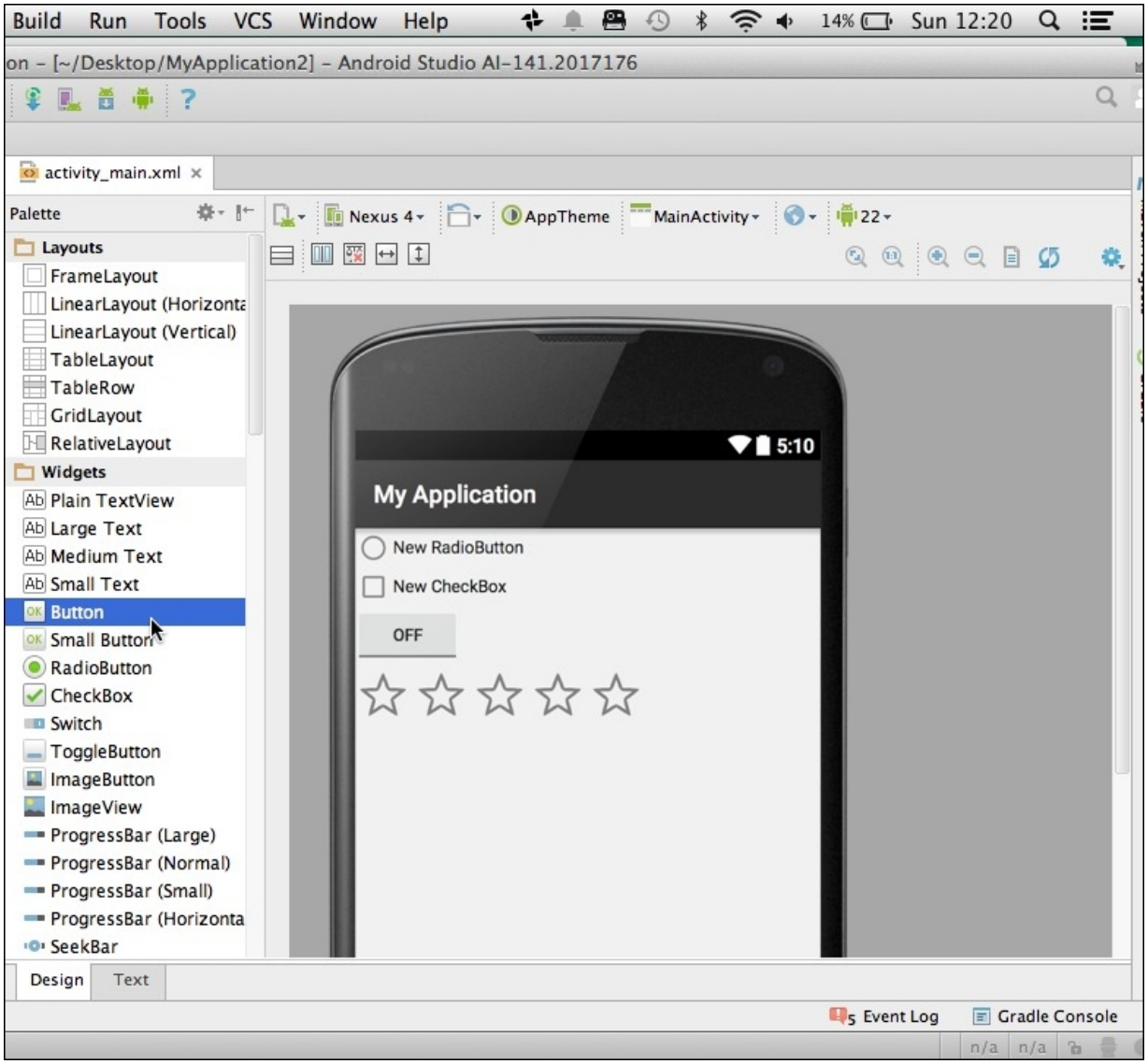
- `android:layout_alignBottom`: This aligns the bottom of a view element with the bottom of the specified onscreen element. For example, `android:layout_alignBottom="@+id/back_button"` aligns the bottom of a UI element with the bottom of `back_button`.
- `android:layout_alignLeft`: This aligns the left edge of a view with the left edge of the specified element.
- `android:layout_alignRight`: This aligns the right edge of a view with the right edge of the specified element.

- `android:layout_alignTop`: This aligns the top edge of a view with the top edge of the specified element.
- `android:layout_alignStart`: This aligns the start edge of a view with the start edge of the specified element.
- `android:layout_alignBaseline`: This attribute is a bit different. **Baseline** is a typography term for the invisible line that text sits on. So, this attribute aligns a view's baseline with the baseline of the specified element. For example, if you have two `TextViews`, you may want to use `alignBaseline` to create the impression that the text in both views is written on the same invisible line.

Creating views

Views are the basic building blocks of a user interface.

Most of the time you'll create view objects by adding them to your Activity's corresponding layout resource file. You can either edit the XML code directly, or you may want to drag UI elements from your IDE's palette and drop them onto the UI preview:



You also have the option of creating views programmatically. For example, if you wanted to instantiate a `TextView` programmatically, you would add `TextView` to your Activity's `onCreate()` method:

```
   @Override    protected void onCreate(Bundle
savedInstanceState) {
super.onCreate(savedInstanceState);
setContentView(R.layout.activity_main);
       LinearLayout linearlayoutLayout =
(LinearLayout) findViewById(R.id.rootlayout);
//Create a new TextView and assign it the ID
txview//
       TextView txView = new TextView(this);

//Set the text programmatically//

       txView.setText("Hello World");

//Add the txView TextView object to your layout//

       linearLayout.addView(txView);
   }
}
```

Assigning the ID attribute

An ID gives you a way of identifying individual views within your layout. For example, if you create two buttons, you can distinguish between them by assigning them the IDs `yesButton` and `noButton`.

You assign an ID to a view using the `android:id` attribute:

```
<Button android:id="@+id/backButton"
```

You can then use the ID to reference this particular view:

```
android:layout_below="@id/backButton"
```

You can also use the ID to locate a view programmatically using `findViewById(id)`:

```
findViewById(R.id.myButton)
```

Once this method returns the desired view, you can interact with the view programmatically.

Setting a view's size

In the same way you need to set the size of your layout container, you need to set the size of all the views you add to your layout resource file.

The good news is that you can use exactly the same attributes and values, which means you can add `android:layout_width` and `android:layout_height` to your layout resource file, and then choose from the following values:

- **wrap_content**: This sets the view's height or width to the minimum size required to accommodate the view's content. For example, if you apply `wrap_content` to a button that contains a text label, the system will size the button so it's *just* big enough to contain the button's text label.
- **match_parent**: This expands the view's height or width to fill all the available space.
- **Density-independent pixels (dp):** When you assign a dp measurement, the Android system scales the view up or down based on the specific density of the user's screen.
- **An absolute unit**: Although this is not recommended, you can also use absolute units of measurements: pixels, millimeters, or inches.

Android gravity and layout gravity

The gravity attribute specifies how an object should be positioned along the *X* and *Y* axis inside its enclosing object. This may sound straightforward, but there's a catch: you'll encounter two different gravity attributes in Android. Although they look similar, they can actually yield very different results:

- `android:gravity`: This positions the content that's inside a view, for example, the text inside a `TextView`
- `android:layout_gravity`: This positions a child view inside its parent container, for example, a `TextView` inside a `LinearLayout`

Both accept a wide range of values, including several values that *won't* change the size of the object you're applying the gravity attribute to:

- `top`: This positions the object at the top of its parent container. For example, `android:gravity="top"` will position the text at the top of `TextView`, while `android:layout_gravity="top"` will position `TextView` at the top of `LinearLayout`.

- `Left`: This positions the object to the left-hand side of its parent container.
- `center_vertical`: This positions the object in the vertical center of its parent container.
- `Start`: This positions the object at the beginning of its parent container.

Both `gravity` attributes also support several values that *do* alter the object's size, including the following:

- `fill_vertical`: This expands the object vertically, so it completely fills its parent container. For example, `android:gravity="fill_vertical"` will expand an image vertically to fill its `ImageView` container, while `android:layout_gravity="fill_vertical"` will expand `ImageView` vertically to fill its `RelativeLayout` container.
- `fill_horizontal`: This expands the object horizontally and vertically so that it completely fills its parent container.

For the full list of supported values, see http://developer.android.com/reference/android/widget/LinearLayout.html#attr_android:gravity.

You assign gravity programmatically using the `setGravity` attribute, for example, `Gravity.CENTER_HORIZONTAL`. You could also use `setHorizontalGravity` and `setVerticalGravity`.

Setting the background – working with color

When it comes to background, some views have a completely transparent background, such as `TextViews`, while others have a standard background color, for example buttons, which are grey by default.

If you're not a fan of a view's default background, you can always change it. Android gives you several options for adding a splash of color to your UI.

Firstly, the Android system does support a number of colors out of the box, so if you want to use any of the following hues, then you're in luck, because the Android system has already done all the hard work for you:

- `black`
- `white`
- `holo_blue_bright`
- `holo_blue_dark`
- `holo_blue_light`
- `holo_green_dark`
- `holo_green_light`
- `holo_orange_light`
- `holo_orange_dark`
- `holo_purple`
- `holo_red_dark`
- `holo_red_light`
- `darker_gray`

To apply any of these ready-made color resources to a view, add the `android:background` attribute, but set its value to `"@android:/color"`

followed by the color of your choice:

```
android:background="@android:color/holo_red_light"
```

Note

Android also supports values such as `primary_text_dark` and `widget_edittext_dark` for specific UI elements. You'll find the complete list of predefined colors in the official Android docs at http://developer.android.com/reference/android/R.color.html.

However, this list is pretty limited! Sooner or later Android's predefined colors just aren't going to cut it, and you'll want to create your own color resources, which means using hex codes.

If you have a particular shade in mind, you can usually find its hex code by performing a quick Google search, for example, by searching for `cyan hex code` or `light pink hex code`. Alternatively, you can go window shopping by browsing the Android style guide, which contains a wide range of colors and their corresponding hex codes. This is available at https://www.google.com/design/spec/style/color.html#color-color-palette.

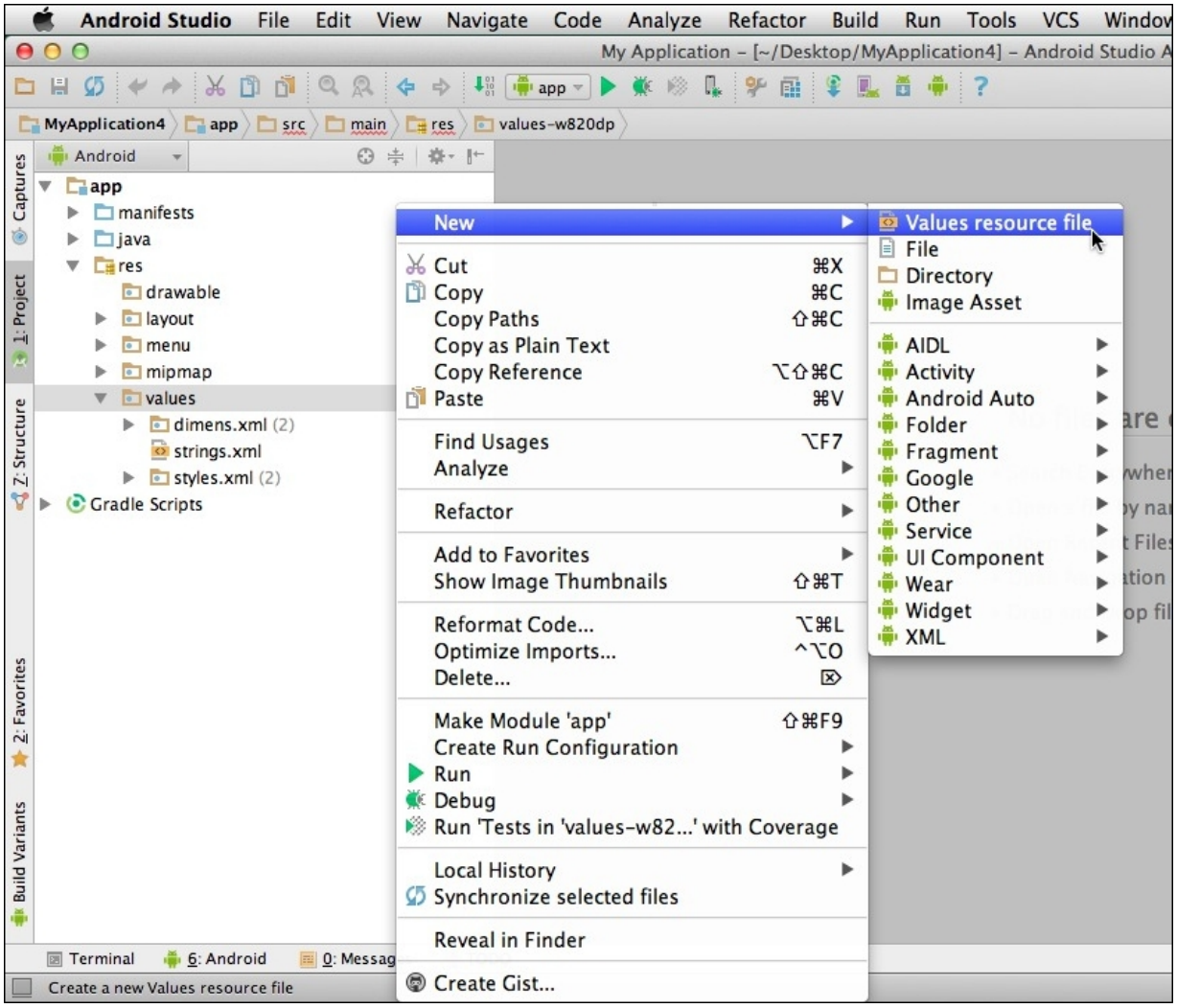
Once you have your hex code, you can take the quick and easy route of entering the code directly into your layout resource file:

```
android:background="#0000ff"
```

This may *seem* quick, but it might just end up costing you more time in the long run. Since consistency is a big part of providing a great user experience, chances are that you'll use the same colors multiple times throughout your application, and typing out the entire hex code every single time you want to use that color can really add up.

Although it may require a bit of initial effort, most of the time it makes sense to define your colors as color resources in your project's `res/values/colors.xml` file. You can then reference these color resources as many times as you want without having to type out entire hex codes.

If your project doesn't contain the `colors.xml` file, you can create one by right-clicking on your project's `values` folder and selecting **New**, followed by **Values resource file**. Give your file the name of `colors.xml`:



Open the `colors.xml` file and define all the colors you want to use in your app using the following format:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <color name="cyan">#00FFFF</color>
</resources>
```

You can then use this color resource anywhere in your app, including in the background of your views:

```
android:background="@color/cyan"
```

You can also change the background of your entire UI by adding the `android:background` attribute to your layout container such as follows:

```
<LinearLayout
xmlns:android="http://schemas.android.com/apk/res/
android"
  android:orientation="vertical"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:background="@color/cyan" />
```

Alternatively, you can use images as the backdrop to your views, whether it's a simple textured background image or a high definition photo. We'll cover images in greater detail when we take a look at `ImageViews`, but as a quick overview, you just need to add the image to your project's `res/drawable` folder and then reference the image in your layout file:

```
android:background="@drawable/imagename"
```

If you want to set your app's background programmatically, use the `setBackgroundResource` method.

Assigning a weight value

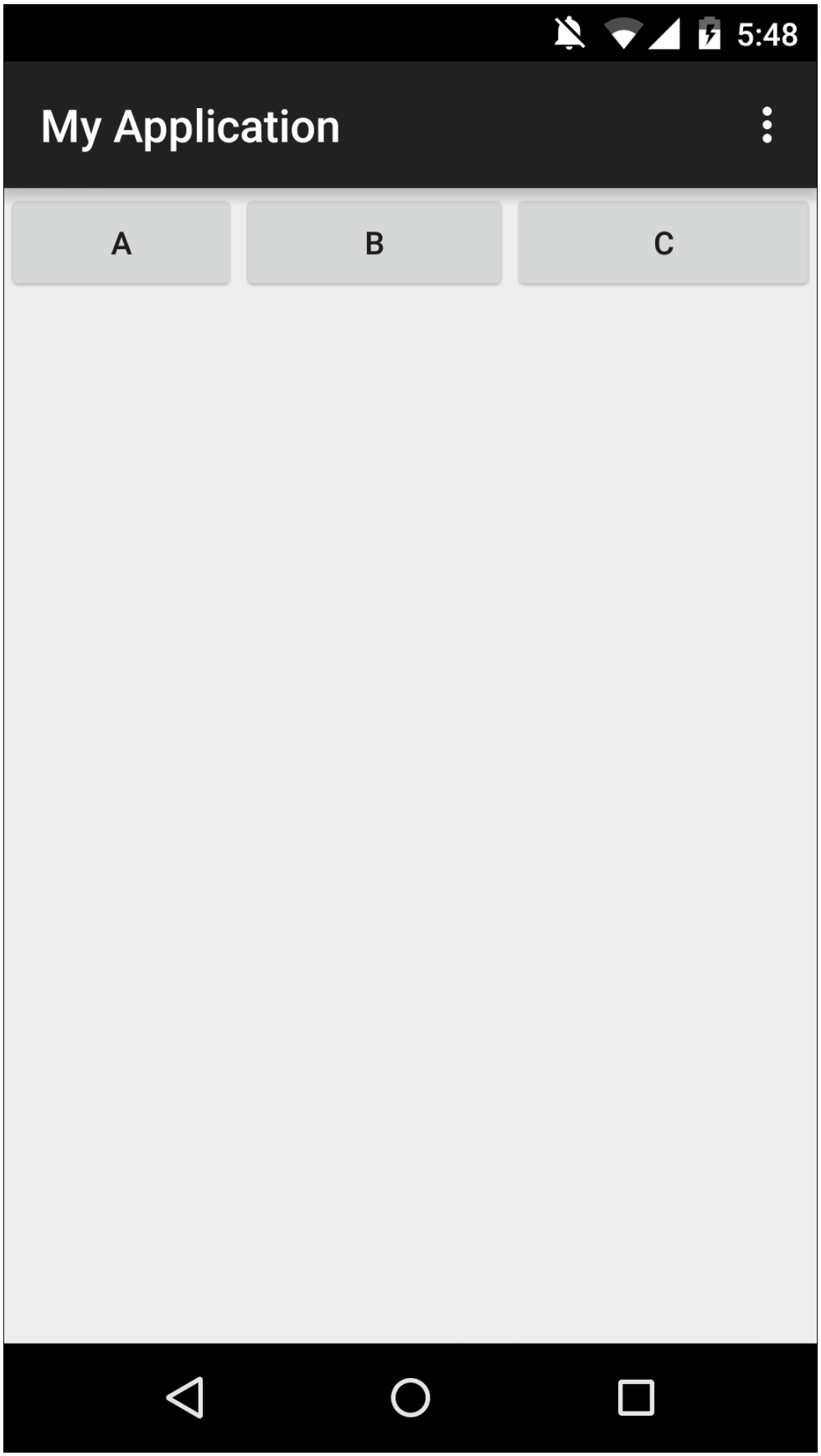
When you're positioning views inside `LinearLayout`, you can control how much space each view occupies onscreen by assigning it a

weight value.

When you assign weight values to your views, any remaining space in the layout is assigned to your views in the proportion of their declared weight. For example, imagine your layout contains three buttons with different weight values:

```
<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=".MainActivity">

  <Button
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="A"
      android:id="@+id/button1"
      android:layout_weight="1"/>

  <Button
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="B"
      android:id="@+id/button2"
      android:layout_weight="1"/>

  <Button
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="C"
      android:id="@+id/button3"
      android:layout_weight="2"/>

</LinearLayout>
```

`Button3` is declaring that it's more important than `Button1` and `Button2`, so it'll be assigned half of any remaining space, while `Button1` and `Button2` have to share the rest of the available space equally:


5:48
My Application
A
B
C

Just be aware that other attributes may interact with your `layout_weight` values. For example, imagine your layout contains three `TextViews` and all of them are set to `android:layout_width="wrap content"`. In this scenario, the Android system calculates how wide each `TextView` needs to be in order to accommodate their text and *only* then does it divide up the remaining space. So, if one `TextView` needs to accommodate much more text than the other two `TextView`, this will have an impact on your weight results.

Note

All views have a default weight of 0, unless you specify otherwise.

Adding and customizing view objects

The easiest way to add view objects to your layout is via the layout resource file in your project's `res/layout` folder, although you can also add views programmatically as and when required.

Over the next few sections, I'll show you how to create some of Android's most commonly used views, specifically `TextViews`, `EditText`, `ImageViews`, `Buttons`, and `ImageButtons`. Once you've created each view, I'll show you how to configure that view so it looks and functions *exactly* as you want it to.

TextView

It may not be the most exciting part of your user interface, but the vast majority of Android apps feature some kind of text. You display text to your users via `TextView`.

To create `TextView`, add the `<TextView>` tag to your project's layout resource file, and then tell the `TextView` what text it should display either by:

- **Adding the text directly to the layout**: This is pretty straightforward. Just add the `android:text` attribute and your text to the `TextView` XML code, for example, `android:text="Hello world!"`.

- **Referencing a string resource**: Most of the time, if your app needs to display text, then this text belongs in your project's resources and *not* in your actual application code. This separation helps to keep your app's code clean and readable, and it also means that you can tweak and change your app's text at any point without ever having to touch your app's underlying code. To create a string resource, open your project's

`res/values/strings.xml` file and add your text in the following format:

```
<resources>
    <string name="helloWorld">Hello world!
</string>
</resources>
```

You can then reference this string resource from your layout file:

```
<TextView
  android:layout_width="wrap_content"
  android:layout_height="wrap_content"
  android:text="@string/helloWorld"
  android:id="@+id/textView" />
```

This is all you need to know in order to display basic text, and sometimes this may be enough. However, text does have the potential to be a bit dry and boring! If you want to create more visually appealing text, you have several options.

Brightening up your text

You can use the `android:textColor` attribute to change the color of the text inside a `TextView`. To reference one of the default colors supported by the Android system, use the following:

```
android:textColor="@android:color/"holo_green_dark
"
```

If you're referencing a color that you defined yourself in your project's `res/values/colors.xml` file, the value is laid out slightly differently:

```
android:textColor="@color/mycustomcolor"
```

If you want to set the color of your `TextView` programmatically, use the `setTextColor()` method.

Setting the size of your text

You can make your text larger or smaller using `android:textSize`.

Once again you need to remember that Android screens come in all sorts of different sizes, and your text needs to be easily readable regardless of the screen it's being displayed on.

To further complicate things, Android users can actually change the size of the font that's displayed on their device by opening their device's **Settings**, tapping **Display,** and selecting **Font Size**. This is a really useful feature for people who have vision problems.

The easiest way of ensuring that your text is flexible enough to adapt to the user's font preferences and screen size is to use scale-independent pixels (`sp`) units:

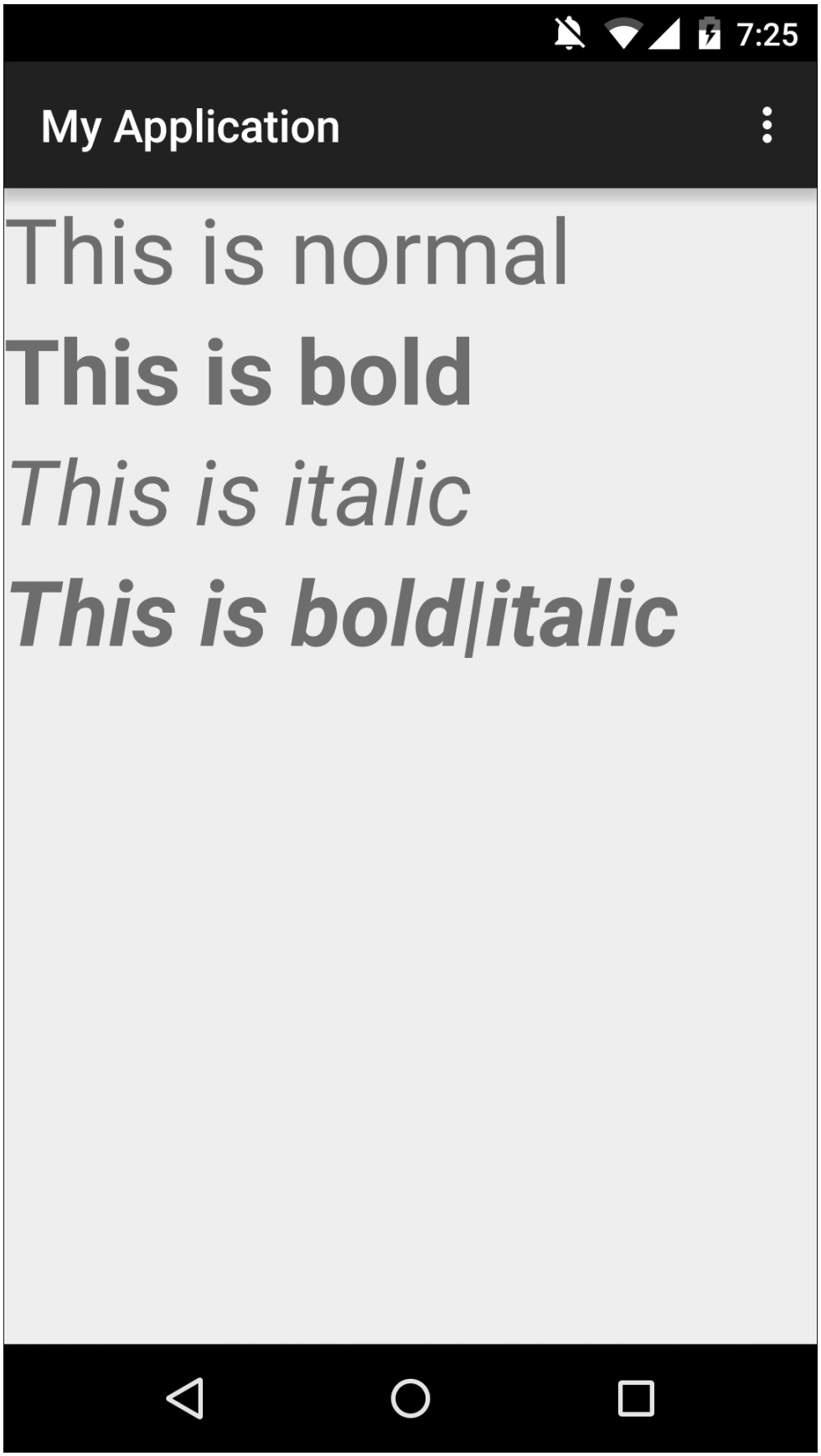
```
android:textSize="30sp"
```

Android also supports three relative font size styles that you may want to use in your application:

- `TextAppearance.Small`, such as `style="@android:style/TextAppearance.Small."`
- `TextAppearance.Medium`
- `TextAppearance.Large`

Emphasizing your text

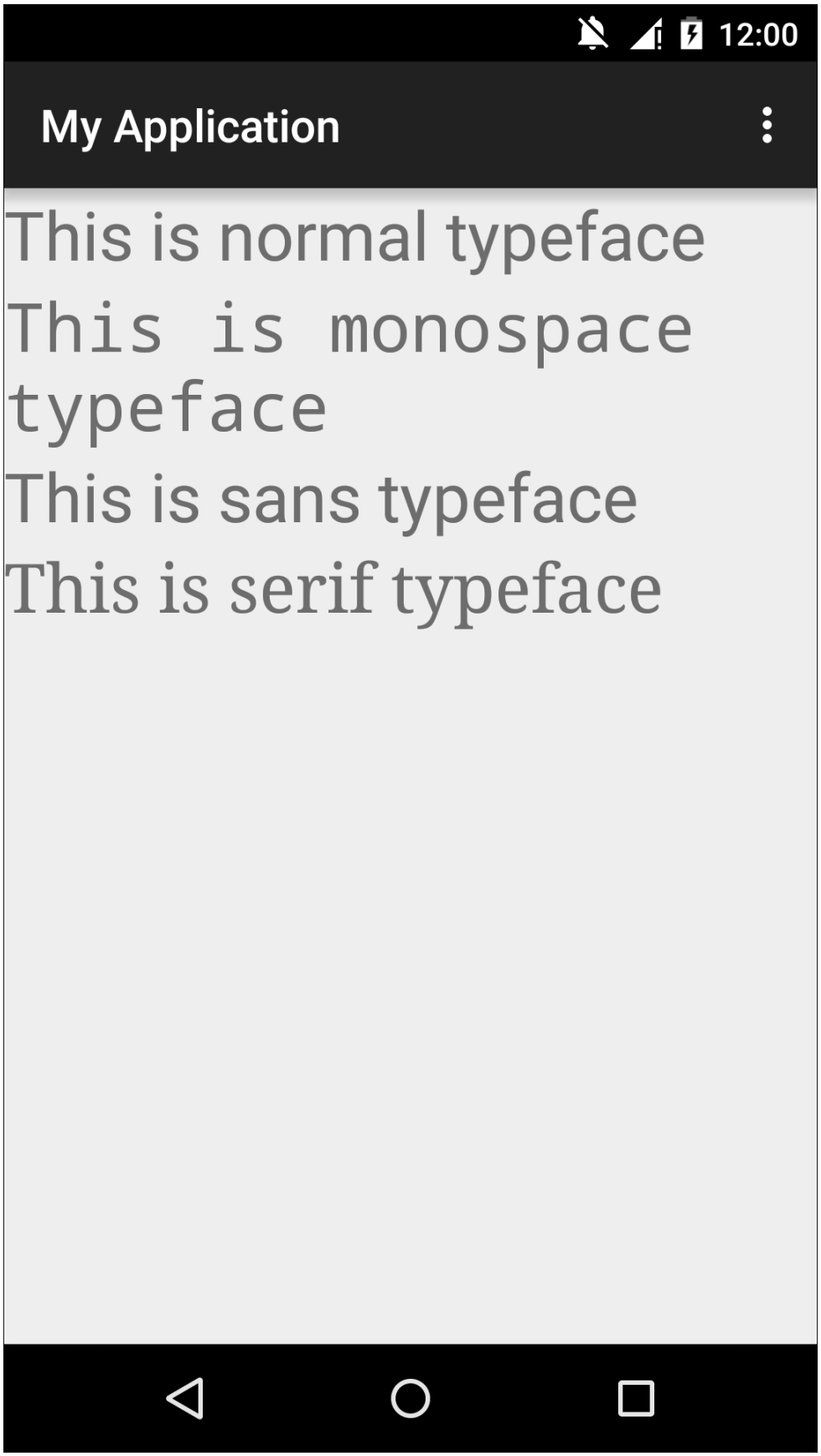
You can add bold or italic emphasis to your text using the `android:textStyle` attribute. The possible values are normal, bold, and italic, or you can combine bold and italic by separating the two values with a pipe character (`android:textStyle="bold|italic"`):


7:25
My Application
This is normal
This is bold
This is italic
This is bold|italic

Setting the typeface

By default, Android applies the normal typeface to your text, but the system also supports `sans`, `monospace`, and `serif` typefaces, which you set using `android:typeface`:

```
android:typeface="monospace"
```


12:00
My Application
This is normal typeface
This is monospace typeface
This is sans typeface
This is serif typeface

To set the typeface programmatically, you need to use the `setTypeFace` method.

How many lines?

By default, the content of your `TextView` will run across multiple lines depending on how much text it has to display. If you want more control over the span of your `TextView`, you have a few options, as follows:

- `android:lines`: This makes `TextView` exactly *X* number of lines tall, for example, `android:lines="2"`
- `android:minLines`. At a minimum, `TextView` will be this many lines tall
- `android:maxLines`: This limits the `TextView` to this many lines tall

EditText

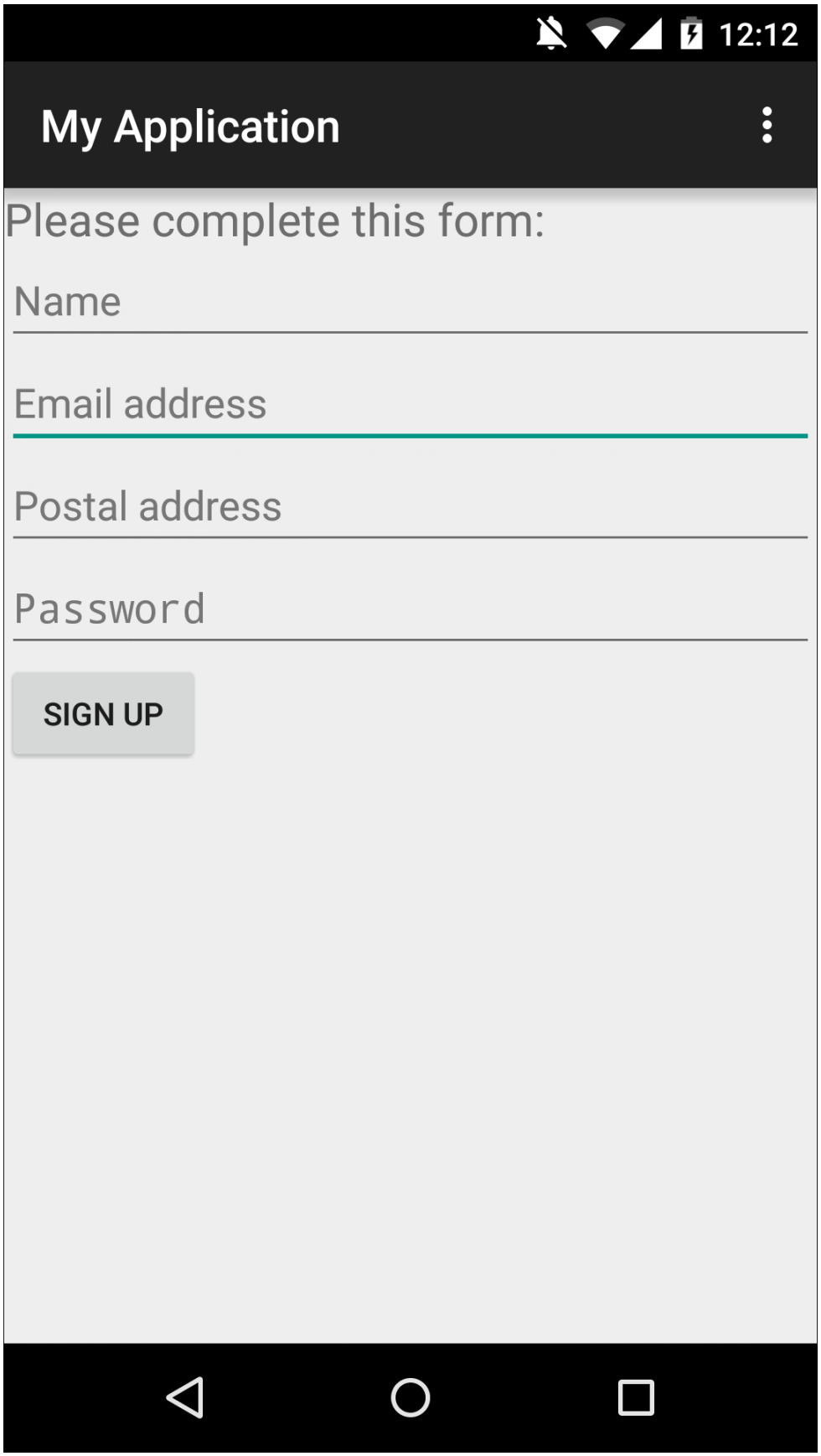
While `TextViews` are great for displaying text, if you want the user to be able to input text, you should use `EditText` instead.

One of the most common examples of `EditTexts` in action is a form that requests user data. In this example, each input field is a separate `EditText`:

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

  <TextView
      android:layout_width="match_parent"
      android:layout_height="wrap_content"
      android:text="Please complete this form:"
      android:textSize="20sp"
      android:id="@+id/TextView" />

  <EditText
      android:layout_width="match_parent"
      android:layout_height="wrap_content"
      android:inputType="textPersonName"
      android:hint="Name"
      android:id="@+id/editText" />

  <EditText
      android:layout_width="match_parent"
      android:layout_height="wrap_content"
      android:inputType="textEmailAddress"
      android:hint="Email address"
      android:id="@+id/editText2" />

  <EditText
      android:layout_width="match_parent"
      android:layout_height="wrap_content"
      android:inputType="textPostalAddress"
```

```
        android:hint="Postal address"
        android:id="@+id/editText3" />

    <EditText
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:inputType="textPassword"
        android:hint="Password"
        android:id="@+id/editText4"
        android:layout_gravity="center_horizontal"
/>

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

Here's how this simple form looks on the user's device:


12:12
My Application
Please complete this form:
Name
Email address
Postal address
Password
SIGN UP

Since the `EditText` class is derived from the `TextView` class, many of the `TextView` attributes are also applicable to `EditText`, including `textColor`, `textSize`, `textStyle`, and `typeface`. However, `EditText` also supports a range of XML attributes that are specific to the `EditText` class.

Controlling keyboard behavior - setting the inputType

By default, when the user taps the `EditText` field, the standard keyboard appears and they can enter whatever characters they like. However, you may want to restrict the kind of data users can enter into your `EditText` fields using the `android:inputType attribute`.

This restriction prevents the user from entering invalid data, but some `inputType` values also prompt the Android system to optimize the virtual keyboard for a particular kind of content. For example, if you specify that the `EditText inputType` value is a phone number (`android:inputType="phone"`) the Android system will display the numerical keyboard, which makes it easier for the user to input the required data.

Some `inputType` values also prompt the keyboard into other useful behavior; for example, if you set `inputType` to `textCapWords`, the keyboard will capitalize every new word automatically. This is handy when you're asking the user to enter data that should always be capitalized, such as their name or address.

Here are the different `inputType` values you can use:

- `text`: This displays the standard text keyboard
- `textEmailAddress`: This displays the standard text keyboard with the addition of the `@` character
- `textUri`: This displays the standard text keyboard with the addition of the `/` character
- `number`: This displays the basic numerical keyboard

- `phone`: This displays the phone-style keyboard

All the following values display the standard text keyboard, but modify other keyboard behavior:

- `textCapSentences`: This automatically capitalizes the first letter of every new sentence.
- `textCapWords`: This automatically capitalizes every word.
- `textAutoCorrect`: This automatically corrects commonly misspelled words.
- `textPassword`: This masks the user's password by transforming every inputted character into a dot. This is the behavior you typically encounter when entering passwords on your Android device. Another way of creating this masking effect is via the XML attribute `android:password="true."`.
- `textMultiLine`: By default, the `EditText` field is constrained to a single line. This attribute allows the user to enter multiple lines of text.

You can also combine multiple values using the pipe character. For example, if you're asking the user to create a password, you could automatically capitalize every new word while also masking their input:

```
android:inputType="textCapWords| textPassword"
```

To specify keyboard behavior programmatically, use the `setRawInputType` method.

> **Note**
>
> While we're on the subject of controlling user input, you can restrict what numbers the user can enter into an `EditText` using `android:digits` (`android:digits="12345."`).

android:imeOptions

Once the user has entered information into the `EditText` field, they'll usually confirm their input by tapping the action key that appears in place of the keyboard's usual carriage return key.

If you *don't* specify what action key the system should display, the Android system defaults to the following:

- `actionNext`: If there's at least one focusable field to move on to, the system displays the **Next** action key.
- `actionDone`: If there's no subsequent focusable fields, the system displays the **Done** key. In our `form` example, the **Done** key would appear once the user has completed the final `EditText` field.

Sometimes, you may want to override this default behavior and specify which action key the keyboard should display. You can do this using the `android:imeOptions` attribute and one of the following values:

- `actionGo`: This displays the **Go** key (`android:imeOptions="actionGo"`)
- `actionNext`: This displays the **Nex**t key
- `actionDone`: This displays the **Done** key
- `actionSearch`: This displays the **Search** key
- `actionSend`: This displays the **Send** key

Giving the user a hint

Even though `EditTexts` are designed to collect user input, you may want to prompt the user for a specific input by displaying temporary, greyed-out text in your `EditText` fields. These hints are useful when it isn't immediately obvious what information the user is supposed to enter.

You display hints using the `android:hint` attribute:

- Enter the hint into your layout resource file directly (`android:hint="Please enter your password"`)

- Create and reference a string resource (`android:hint="@string/messageHint."`)

ImageView

Images are a handy way of conveying information to your users without forcing them to read lots of on-screen text. Although you can add images to lots of different areas within your app, (such as your layout's background and the background of onscreen elements, such as buttons), the Android SDK provides a dedicated view for displaying images, called `ImageView`.

We've already discussed how you can use density-independent and other relative units of measure to create a user interface that displays correctly across a range of different screens. However, ensuring your images look crisp and clear across a range of different screen sizes isn't quite so straightforward.

This makes `ImageView` one of the more complicated views, but since images are such an integral part of most Android UIs, it's well worth taking the time to properly master Android's `ImageView`.

In this section, I'll show you how to add **drawables** to your UI and the steps you should take to ensure these images display properly across the full range of possible screen sizes.

> ## Note
>
> A drawable just means "something that can be drawn on the screen," and is often used to describe your app's graphical content.

Supporting multiple screens

Let's get the tricky stuff out of the way first: how to create images that display correctly across a wide range of different screens.

Although the Android system scales your content automatically to fit the current screen configuration, you shouldn't rely on the system to do all the hard work for you, *especially* when it comes to `ImageViews` as this can result in blurry or pixelated images.

To provide the best possible user experience, you'll need to provide alternate versions of all the images you use in your app. These versions should be optimized for different screen densities.

The good news is that you don't have to provide images for *every* screen density imaginable, as the Android system groups all possible screen densities into generalized density spans. As long as you provide a version for each density span, the Android system will choose the version that's the best match for the current screen configuration.

Android supports five main generalized screen densities:

- **Low**: `ldpi 120dpi`
- **Medium**: `mdpi 60dpi`
- **High**: `hdpi 240dpi`
- **Extra-high**: `xhdpi 320dpi`
- **Extra-extra-high**: `xxhdpi 480dpi`

Note

Android actually supports a sixth screen density: extra-extra-extra-high, also known as `xxxhdpi`. This `640dpi` density span is a bit different from the others as it *only* applies to your application's launcher icon. Some devices, such as tablets, may display extra-large app icons in their launcher. To make sure your app's icon doesn't end up looking fuzzy on a large screen with an `xxxhdpi` display, you should supply an extra-extra-extra high density version of your app's icon. You don't need to provide `xxxhdi` versions of any other UI elements.

Supporting different screen densities

So, how do you let Android know which image is optimized for `hdpi` displays, and which image is optimized for `xhdpi` displays? The answer is to create directories that are tagged with the `ldpi`, `mdpi`, `hdpi`, `xhdpi`, `xxhdpi`, and `xxxhdpi` qualifiers. Android will recognize that these directories contain resources that target a specific screen density, and then select an image from the appropriate directory based on the current screen configuration.

When you create an Android project in Eclipse or Android Studio, the project typically contains the default `res/drawable` directory only, so you'll need to create the following directories manually:

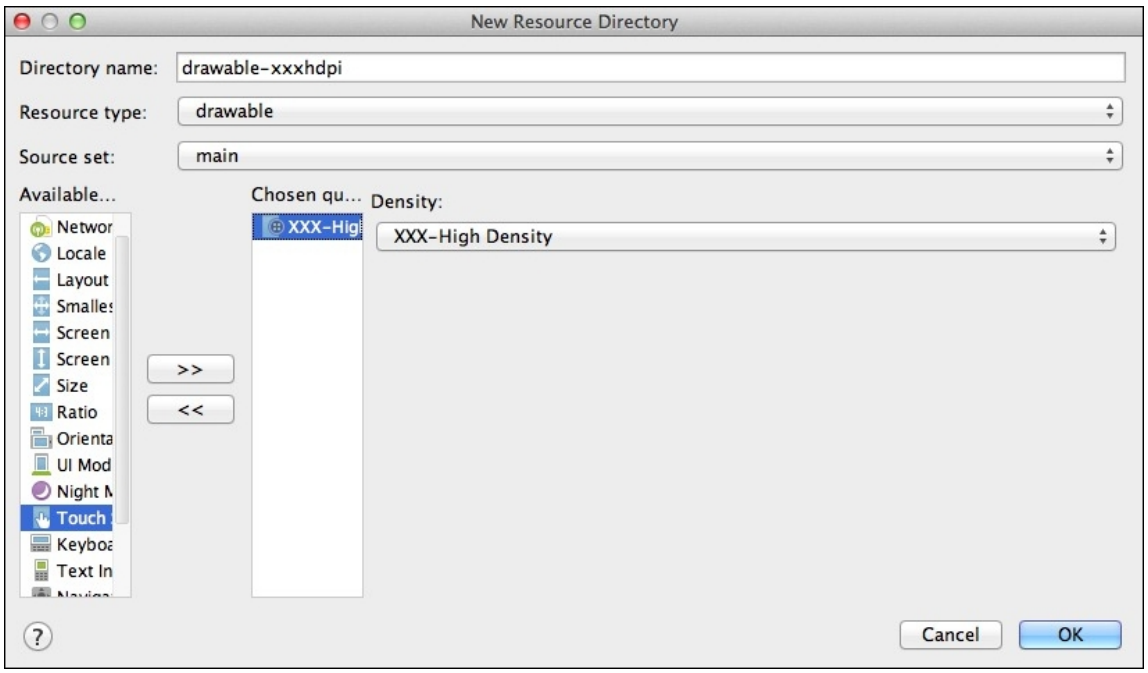
- `drawable-ldpi`
- `drawable-mdpi`
- `drawable-hdpi`
- `drawable-xhdpi`
- `drawable- xxhdp`
- `drawable-xxxhdpi`

Remember, this directory should contain the extra-extra-extra-high density version of your app's launcher icon only.

To create these density-specific directories, perform the following steps:

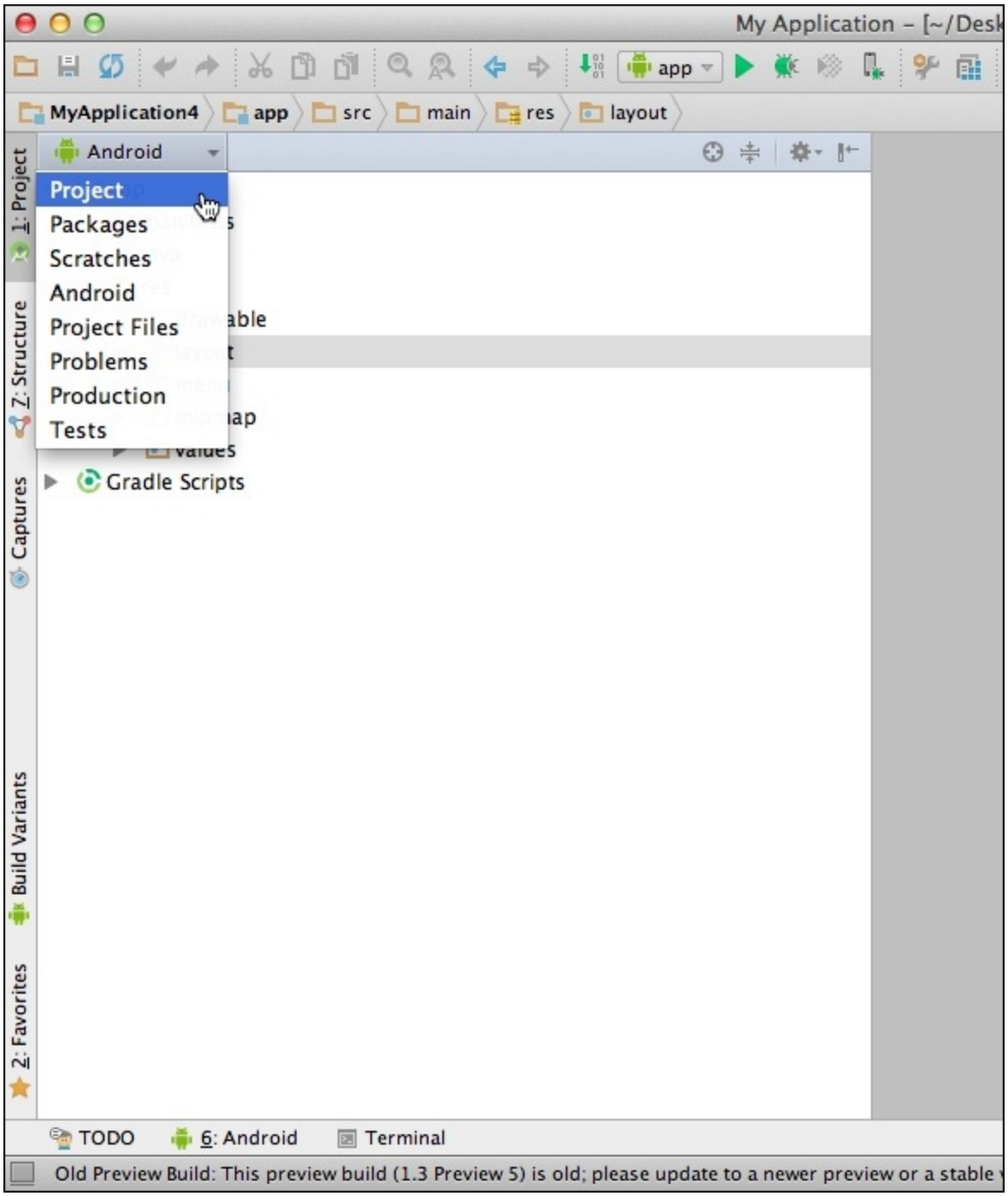
1. Right-click on your project's `res` folder and select **New**, followed by **Android resource directory**.
2. In the window that appears, open the **Resource type** dropdown and select **Drawable**.
3. In the **Available qualifiers** section, add **Density** as **Chosen qualifier**.
4. Open the **Density** dropdown and select the desired density from the list, for example, **Low Density** if you're creating the `drawable-ldpi` directory, or **XXX-High density** if you're creating

a `drawable-xxxhdpi` directory. You'll notice that when you select the density, **Directory name** gets updated automatically:



5. When you're happy with the information you've entered, click **OK**. Your IDE will then create the new directory.
6. Repeat! You'll typically want to create a directory for each of the generalized screen densities.

If you're working with a project in Android Studio, sometimes you may create all your density-specific directories and then realize that none of them are appearing in Android Studio's **Project** view. If this happens, the problem may be that you have the **Android** view selected instead of **Project** view. To switch views, click on the **Android** label in Android Studio's **Project** view, and then select **Project** from the drop-down menu:



You'll notice your project structure has changed. Open the `app/src/main/res` folder, and you will see all the density-specific directories you created earlier.

Creating density-specific images

Now that you've updated your project structure, it's time to actually create those optimized images and add them to your project.

Android supports several different image types, but you'll typically use one of the following:

- **Bitmaps**: Android supports bitmap files in three formats: `.png` (preferred), `.jpg` (acceptable), or if you really *must*, `.gif` (discouraged).
- **Nine-patch file**: This is a `.png` file with a difference! Nine-patch files allow you to define stretchable regions that help your image resize more smoothly. We'll explore nine-patch images in more detail in the later chapters.

The key to creating alternate bitmaps and nine-patch files is to adhere to a 3:4:6:8:12:16 scaling ratio. For example, if you have an image that's 68 x 68 pixels and targets medium-density screens, you'd need to create the following alternatives:

- **LDPI**: 51 x 51 pixels (0.75% of the original size)
- **MDPI**: 68 x 68 pixels (original size)
- **HDPI**: 102 x 102 pixels (150% of the original size)
- **XHDPI**: 136 x 136 (200% of the original size)
- **XXHDPI**: 204 x 204 (300% of the original size)
- **XXXHDPI**: 272 x 272 (400% of the original size)

When you're creating your alternate images, you *must* use the same filename for each version of the image. This is essential if the Android system is going to recognize these files as alternate versions of the same image.

The final step is placing each file inside the appropriate directory.

Adding ImageView

Creating the correct `drawable` directory structure and providing multiple versions of the same image may feel like a lot of work, but

once you've done all this groundwork, displaying `drawable` content in `ImageView` is a fairly straightforward task.

Most of the time, you'll create your `ImageView` by adding an `<ImageView>` element to your Activity's layout resource file:

```
<ImageView
     android:id="@+id/imageView1"
android:layout_width="wrap_content"
android:layout_height="wrap_content"
     android:src="@drawable/myImage"
/ >
```

Note the `android:src` attribute. This is how you tell your `ImageView` what `drawable` to display. Assuming you've provided several versions of the `myImage` file, the Android system checks each `drawable` directory for the most suitable version, and then it displays this image.

If you'd prefer to set your `ImageView` content programmatically, you can use the `setImageResource()` method instead.

Buttons and ImageButtons

`Buttons` (and by extension, `ImageButtons)` are UI components that react to the user tapping the screen. Whenever you add a button to your UI, it should be immediately clear to the user what this button will do when they touch it.

While you *could* inform the user about a button's purpose with an accompanying `TextView` (something along the lines of *tap the button below to move onto the next screen*) this is wordy and inefficient. Most of the time, you'll want to communicate a button's purpose by adding a label to this button.

Android provides you with several labeling options:

- A text label, for example **Next**, **Submit**, or **Cancel**.
- An image icon, such as a checkmark or a cross icon.
- Both! If the button represents an unusual, unexpected, or complicated action, you may want to clear up any potential confusion by labeling the button with both text and an image:

12:40
My Application
OK
CANCEL
SHARE

Depending on whether you want your button to display text, an icon, or both, you can add a button to your layout resource file in one of three ways.

Creating buttons with text labels

To create a basic button with a text label, insert a `<Button>` element into your layout resource file:

```
<Button
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:text="@string/button_text" />
```

You set the button's text label using the `android:text` attribute. As with `TextViews`, you can either insert the text into your layout directly (`android:text="Submit"`) or you can create a string resource in your project's `res/values/strings.xml` file, and then reference the string resource (`android:text="@string/submitText"`).

To set the button's text programmatically, use `setText`:

```
button.setText("Submit!");
```

Creating buttons with image labels

As the name suggests, `ImageButton` is a button with an image label.

To insert `ImageButton` into your UI, you need to add the `<ImageButton>` tag to your layout resource file:

```
<ImageButton
  android:layout_width="wrap_content"
  android:layout_height="wrap_content"
  android:id="@+id/imageButton"
  android:src="@android:drawable/send" />
```

Recognize the `android:src` attribute? It's referencing a `drawable` resource, in exactly the same way an `ImageView` references a `drawable` reasource.

Creating buttons with text and image labels

If you want to leave the user in no doubt about what'll happen when they touch a button, you can label that button with both text and an image. This involves utilizing the `Button` class with the addition of `android:drawableLeft`:

```
<Button
  android:layout_width="wrap_content"
  android:layout_height="wrap_content"
  android:id="@+id/imageButton3"
  android:text="Send"
  android:drawableLeft="@android:drawable/send" />
```

Tip

Change your button's background

If you feel like these three options aren't enough, you can also change a button's background using `android:background` and then referencing a color value or an image.

State list resources

Your typical button has three states:

- `Default`: This button is neither pressed nor focused
- `Pressed`: This button is in a pressed state
- `Focused`: This button is currently in focus

All the button attributes we've discussed so far (text labels, image labels, and backgrounds) apply to a button element regardless of its current state. But sometimes you'll want your button to give the user visual clues about its current state. For example, when the user taps a button, you may want it to briefly display a darker color while it's in its pressed state, so users know that their devices have successfully registered the touch event.

If you want your button to react to its current state, you need to create a state list resource, which is an XML file that defines three different images or colors to use for each of the button's states.

To create this state list resource, do the following:

1. Create three drawables to use as your button's background. These drawables represent each of the button's states.
2. Give each `drawable` a name that reflects the state it represents, for example, `button_pressed`, or `button_default`.
3. Add these drawables to the appropriate `res/drawable` directory.
4. Create a new XML file in your project's `res/drawable` directory, by right-clicking on the `drawable` directory and selecting **New**, followed by **Drawable resource file**. Give the XML file a descriptive name, such as `button_states`, and then click on **OK**.

Open your new `drawable` resource file and define all the drawables you want to use for each state by adding them as separate `<list>` elements inside a single `<selector>` element:

```
<?xml version="1.0" encoding="utf-8"?>
<selector
xmlns:android="http://schemas.android.com/apk/res/
android">
<item android:drawable="@drawable/button_pressed"
        android:state_pressed="true" />
   <item
android:drawable="@drawable/button_focused"
        android:state_focused="true" />
   <item
android:drawable="@drawable/button_default" />
</selector>
```

In the preceding code, we're defining which `drawable` the button should use when it's in each state:

- When `pressed = true`, the button should use the `button_pressed` drawable.
- When `focused = true`, the button should use the `button_focused` drawable.
- If the button isn't pressed or focused, it should use the `button_default` drawable.

The order of the `<item>` elements in your state list resource is important, as when you reference this state list, the system moves through the `<item>` elements in order and uses the first `<item>` element that's applicable to the button's current state. Since the default state is *always* applicable, you must always place the default `drawable` at the end of the list, to ensure it's only used if the system has checked and discarded `android:state_pressed` and `android:state_focused` first.

To apply a state list resource to a button, you need to reference it as a single `drawable` in your Activity's layout resource file:

```
<Button
   android:id="@+id/button_send"
   android:layout_width="wrap_content"
   android:layout_height="wrap_content"
```

```
    android:text="@string/button_send"
    android:background="@drawable/button_custom" />
```

You can also create state list resources that use colors instead of images. These kinds of state lists are known as **color state list resources**. You use the same `<selector>` and `<item>` elements; the only difference is that each `<item>` element references a color rather than a `drawable`:

```
<item
android:drawable="@android:color/holo_red_light"
        android:state_pressed="true" />
```

Or

```
<item android:drawable=""#0000ff"
        android:state_pressed="true" />
```

Summary

In this chapter, we covered some of the most commonly used views and layouts, and we looked at all the different ways to customize them to suit our specific design needs.

Even though there are views and layouts we haven't covered yet, many of the attributes we've explored in this chapter are applicable to the views and layouts that we'll encounter in later chapters, for example, `layout_width`, `android:src`, `android:id` and the relative units of measure.

In the next chapter, we'll expand on using resources such as strings, nine-patch images, colors, and state lists, and also take a look at some new resources, including arrays and dimensions.

Finally, you'll learn all about an important aspect of UI design that we haven't touched on yet: *fragments*.

Chapter 3. Expanding your UI – Fragments, Resources, and Gathering User Input

In the previous chapter, we concentrated on creating a strong foundation for your app's user interface. In this chapter, we'll build on these foundations using additional resources, such as arrays, dimensions, and 9-patch images.

Once we've perfected the *appearance* of your UI, we'll explore how to make the UI react to user input, before taking a look at one UI component that can help us make the most out of devices with larger screens: *fragments*. And since fragments aren't the *only* way to take advantage of the extra screen space, we'll also take a look at the multi-window and picture-in-picture modes coming up in Android N.

More resource types

If your app features any text, then as a general rule, the text belongs in your project's `res/strings.xml` file as a string resource and *not* in your application code. Although we've touched on string resources in the previous chapter, they're such an integral part of the vast majority of Android applications that it makes sense to look at them in more detail, particularly in regards to some of the more complex tasks you can perform with string resources, such as creating string arrays.

Creating and styling string resources

A string is a simple resource that you define once in your project's `res/values/strings.xml` file, and then use it multiple times throughout your project.

You define string resources using the following syntax:

```
<resources>

<string name="string_name">This is the text that'll
appear whenever you reference this string resource.
</string>

</resources>
```

You also have the option to add style attributes to your string resources, so every time you use the string resource in your application, it has exactly the same styling. Android supports the following HTML markup:

- `<b>` for bold text
- `<i>` for italic text
- `<u>` for underlined text

Wrap your chosen HTML markup around the text you want to style:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>

<string name="Hello World">Hello world,
<b>welcome</b> to my app!</string>

<string name="click">Please click the <i>button</i>
to continue</string>

</resources>
```

Although you'll typically reference string resources from your application's layout resource files, you can also reference a string via your Java code:

```
String string = getString(R.string.string_name);
```

Creating string arrays

A string array is exactly what it sounds like: an array of string resources.

String arrays are useful when you have multiple related strings that always appear together, such as a list of options in a recurring menu.

While you *could* define every item as a separate string resource and then reference each string individually, this is pretty time consuming, plus you'll need to remember a long list of different string IDs! It usually makes more sense to add all your string resources to a single array, so you can display all these strings just by referencing a single string array.

You typically create string arrays in a dedicated `res/values/arrays.xml` file. For example, the following XML defines a string array called **ingredients** that consists of five string resources:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>

   <string-array

//Create your string array//

name="ingredients">

//Give your array a descriptive name//

      <item>Self-raising flour</item>

//Add each string to your array//

      <item>Butter</item>
      <item>Caster sugar</item>
      <item>Eggs</item>
      <item>Baking powder</item>
   </string-array>

</resources>
```

To load this string array, use the `getStringArray()` method of the `Resources` class:

```
Resources res = getResources();
String[] ingredients = res.getStringArray
(R.array.ingredients);
```

String arrays are also useful for rapidly populating spinner controls. Assuming the spinner control's options are static and predetermined, you can define all these options in a string array and then load the array inside the spinner control.

Note

A **spinner** is a UI element where the user selects one value from a list of options. When the user touches the spinner control, a drop down appears displaying all the available options. The user can then select an option from the list, and the spinner control then displays this selected option in its default, unopened state.

To populate a spinner using a string array, add the spinner control to your activity's layout resource file, and then reference the array using `android:entries:`

```
<Spinner
    android:layout_height="wrap_content"
    android:layout_width="match_parent"
    android:id="@+id/spinnerOfIngredients"
    android:entries="@array/ingredients">
</Spinner>
```

Defining dimensions in dimens.xml

Android supports several different units of measurement that you can hardcode into your project using XML or Java, such as `android:textSize="20sp"`. However, you can also define the dimensions you want to use in advance in your project's `res/values/dimens.xml` file:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <dimen name="textview_width">26dp</dimen>
  <dimen name="textview_height">35dp</dimen>
  <dimen name="headline_size">41sp</dimen>
```

```
  <dimen name="bodytext_size">20sp</dimen>
</resources>
```

Then, you can set the size of your UI components by referencing the corresponding value within your `dimens.xml` file:

```
<TextView
   android:layout_height="@dimen/textview_height"
   android:layout_width="@dimen/textview_width"
   android:textSize="@dimen/headline_size"/>
```

You can also apply values from your `dimens.xml` file using Java:

```
textElement.setWidth(getResources().getDimensionPixe
lSize(R.dimen.headline_size));
```

But why go to the extra effort of using the `dimens.xml` file, when you can just add dimension information to your layout directly?

Though this may seem like the quickest option, it isn't good practice to mix your dimensions with your layout and application code. Similar to string resources, the `dimens.xml` file provides a single, dedicated place where you can change your project's dimensions *without* having to touch the rest of your code. It also helps you create a consistent user interface, as it encourages you to define a set of values once and then use these same values throughout your app.

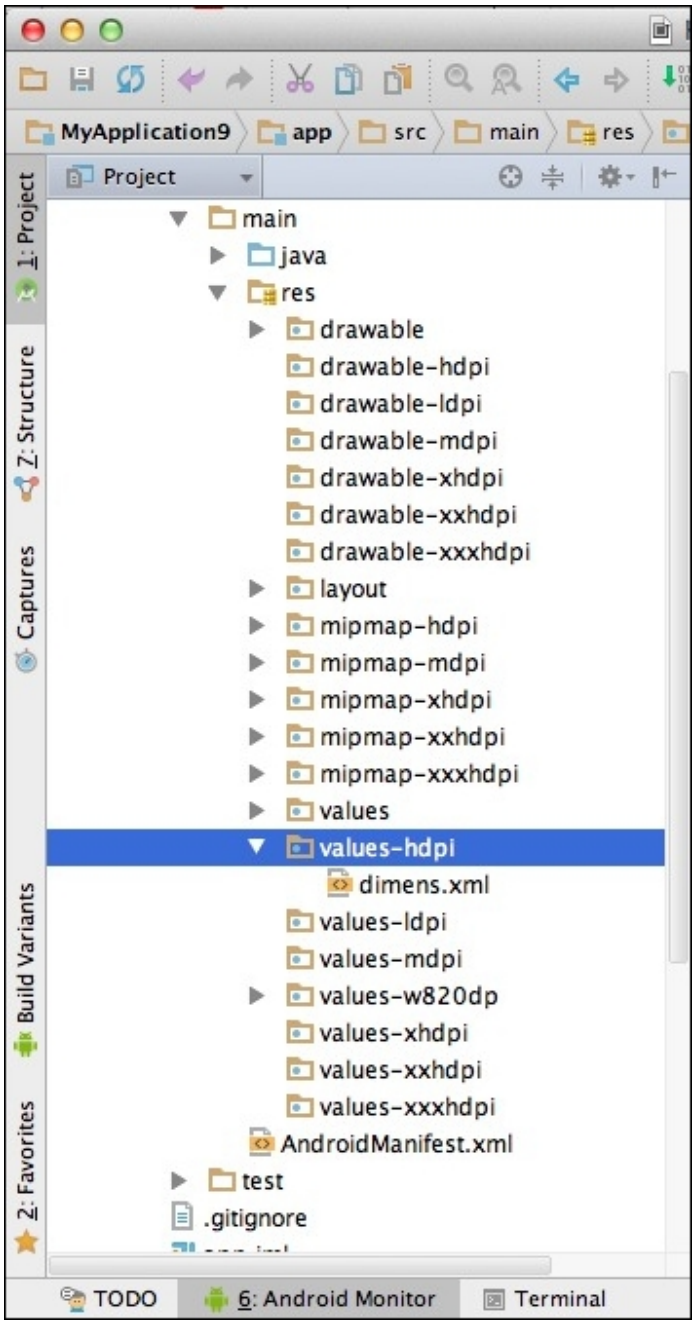
Secondly, you can use multiple `dimens.xml` files to create a more flexible user interface. Using the `dimens.xml` files in this way does require a bit of prep work, as you'll need to create multiple `res/values` folders that target Android's different generalized densities. In the same way you, create multiple `drawable` folders to manage your project's images.

Open your project's `res` folder and create the following folders:

- `values-ldpi`: Targets 120dpi devices
- `values-mdpi`: Targets 60dpi devices
- `values-hdpi`: Targets 240dpi devices

- `values-xhdpi`: Targets 320dpi devices
- `values-xxhdpi`: Targets 480dpi devices

Then, create a `dimens.xml` file inside each folder:



This may seem like a lot of work, but once you have this structure in place, you can use your `dimens`. files to define the sets of values that are optimized for each screen density category. Then, when the

Android system loads your layout, it'll select the most appropriate `dimens.xml` file for the current screen configuration and apply the file's dimensions to your layout.

Tip

Downloading the example code

You can download the example code files for this book from your account at http://www.packtpub.com. If you purchased this book elsewhere, you can visit http://www.packtpub.com/support and register to have the files e-mailed directly to you. You can download the code files by following these steps:

1. Log in or register to our website using your e-mail address and password.
2. Hover the mouse pointer on the **SUPPORT** tab at the top.
3. Click on **Code Downloads & Errata**.
4. Enter the name of the book in the **Search** box.
5. Select the book for which you're looking to download the code file
6. Choose from the drop-down menu where you purchased this bool from.
7. Click on **Code Download**.

You can also download the code files by clicking on the **Code Files** button on the book's webpage at the Packt Publishing website. This page can be accessed by entering the book's name in the **Search** box. Please note that you need to be logged in to your Packt account.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR / 7-Zip for Windows
- Zipeg / iZip / UnRarX for Mac
- 7-Zip / PeaZip for Linux

Color state lists

Each time the user interacts with a UI element, your app will signal that it's registered this interaction. Sometimes, this signal will be built-in: a new screen loads up, a pop up opens, or a *tick* appears in a checkbox. However, if it isn't immediately obvious that a UI element has registered the user's input, you should give them a visual clue. One possible method is to use the **color state list**.

A color state list defines a series of states and assigns a color to each of these states. When you apply a color state list to a view, the view displays different colors from the list based on its current state.

Color state lists are most commonly applied to buttons. For example, a grey button may briefly turn a darker shade of grey when it's in a depressed state, leaving the user in no doubt about whether the button has registered their interaction.

> ## Note
>
> Color state lists are similar to the state list resources that we looked at in the previous chapter—we're just signaling changes of state using colors rather than images.

To create a color state list, open your project's `res/drawable` folder and create a new XML file; give the file a name that indicates its purpose, such as `res/drawable/button_background.xml`. Then, fill this file with all the different states that you want to trigger a color change and the colors you'd like to use.

The potential states include the following:

- `android:state_pressed="true/false."`
- `android:state_focused="true/false."`
- `android:state_selected="true/false."`
- `android:state_checkable="true/false."`

- `android:state_checked="true/false."`
- `android:state_enabled="true/false."`
- `android:state_window_focused= "true/false."`

In this example, we're going to add two states to a color state list:

```
<selector
xmlns:android="http://schemas.android.com/apk/res/an
droid" >

//A color state list must be inside a single
selector element//

<item

//Add each state and color as a separate <item>//

android:state_pressed="true"

//If the current state is pressed...//

android:color="@color/green" />

//....Apply the color green to this view//

<item

android:color="@color/blue/>

//If the View is in its default state, apply the
color blue instead. We're using blue as the default
that'll be applied to the view when none of the
above states are relevant//
```

Note

The order you place your `<item>` element within the selector element is crucial, as the system works its way through the color state list in order and selects the first item that applies to the view's current state. As you saw in the preceding example, you can create a default color that'll be applied to the view when none of the other states are applicable. If you create a default state, then you must

always place it at the very end of the color state list as a final resort.

The only thing left to do is apply the color state list resource to your view:

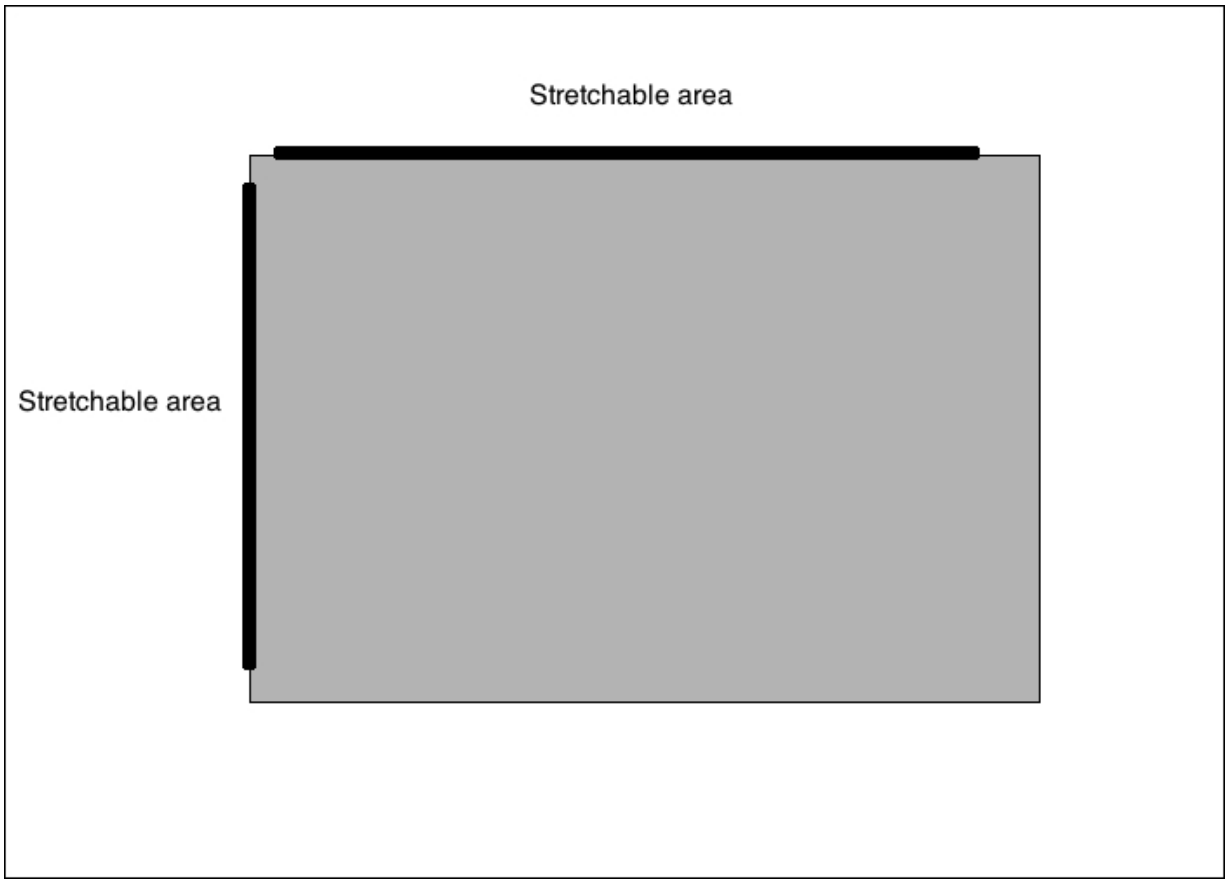
```
<Button
   android:layout_width="match_parent"
   android:layout_height="wrap_content"
   android:text="@string/button_text"
   android:background:="@drawable/button_background"
/>
```

You can also refer to your color state list resource in Java using `R.drawable.button_text`.

Working with 9-patch images

A 9-patch graphic is a stretchable bitmap that allows you to define which areas the system can and can't stretch when it needs to resize your image to fit the current screen.

When you convert a regular image into a 9-patch image, you add an extra 1-pixel-wide border around your image's top and left areas. This 1-pixel border pinpoints exactly which pixels the system should replicate if it needs to create a stretched effect rather than simply resizing the entire image:



Your border doesn't have to be a continuous line; if there's any area you don't want the system to stretch, just leave the area as a gap in your border. In the previous example, the system can stretch the drawable horizontally and vertically by replicating the pixels marked

with the black line. However, the corners aren't marked, so these will remain the same size, creating a sharper, more defined corner.

> **Note**
>
> Converting an image into 9-patch format adds an extra pixel to your image's perimeter. Keep this in mind when you're creating images with the intention of converting them into 9-patches!

Although your image can include multiple stretchable sections, your lines must be exactly 1 pixel wide for the Android system to correctly recognize these lines; stretch your image accordingly (if appropriate), and then remove the lines from the finished image. If you add any lines that are thicker than one pixel, the system will treat these lines as just another part of your image.

If you do use 9-patches, you still need to provide alternate versions of these images for each of Android's generalized screen densities (`ldpi`, `mdpi`, `hdpi`, `xhdpi`, and `xxhdpi`). When the system loads your app, it'll select the 9-patch image that's the best fit for the current screen density, and then it will stretch the image's *stretchable* sections if required.

> **Note**
>
> 9-patch images can stretch, but they can't shrink. For the best results when creating 9-patch images that target the different density folders, you should target the lowest common resolution for each density category.

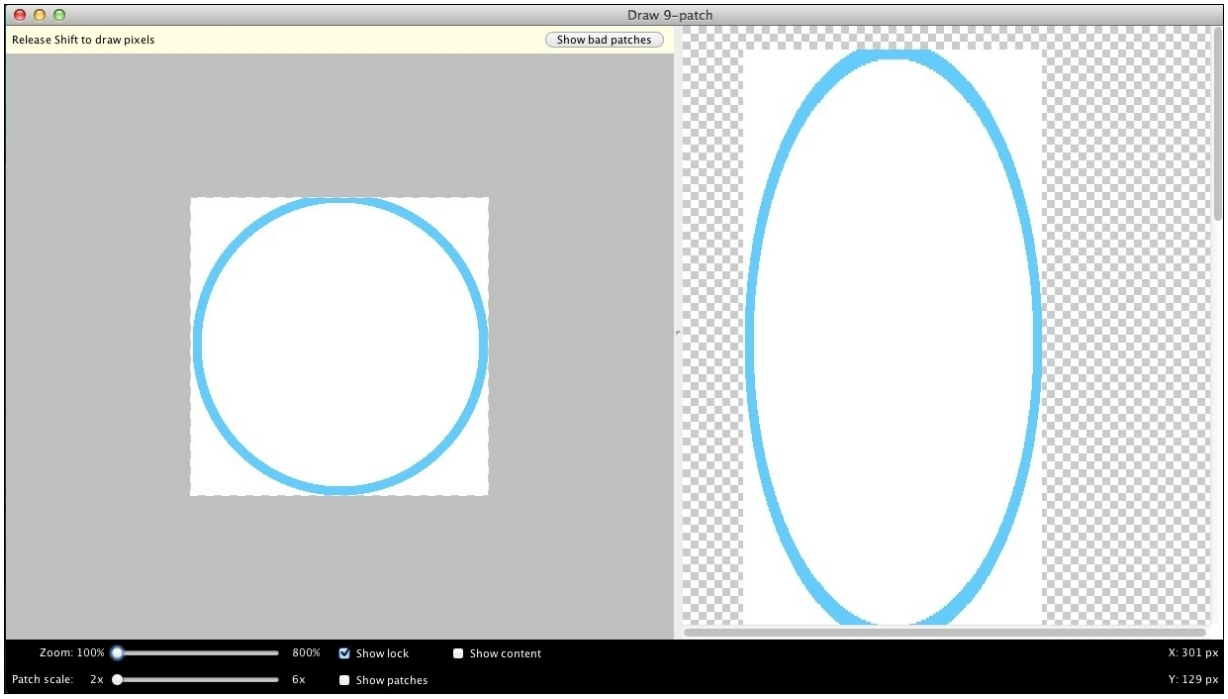
How do I create 9-patch images?

There are lots of different PNG editors out there, but I'll be using Draw 9-patch as this editor is included in the Android SDK, so chances are you already have it installed on your computer.

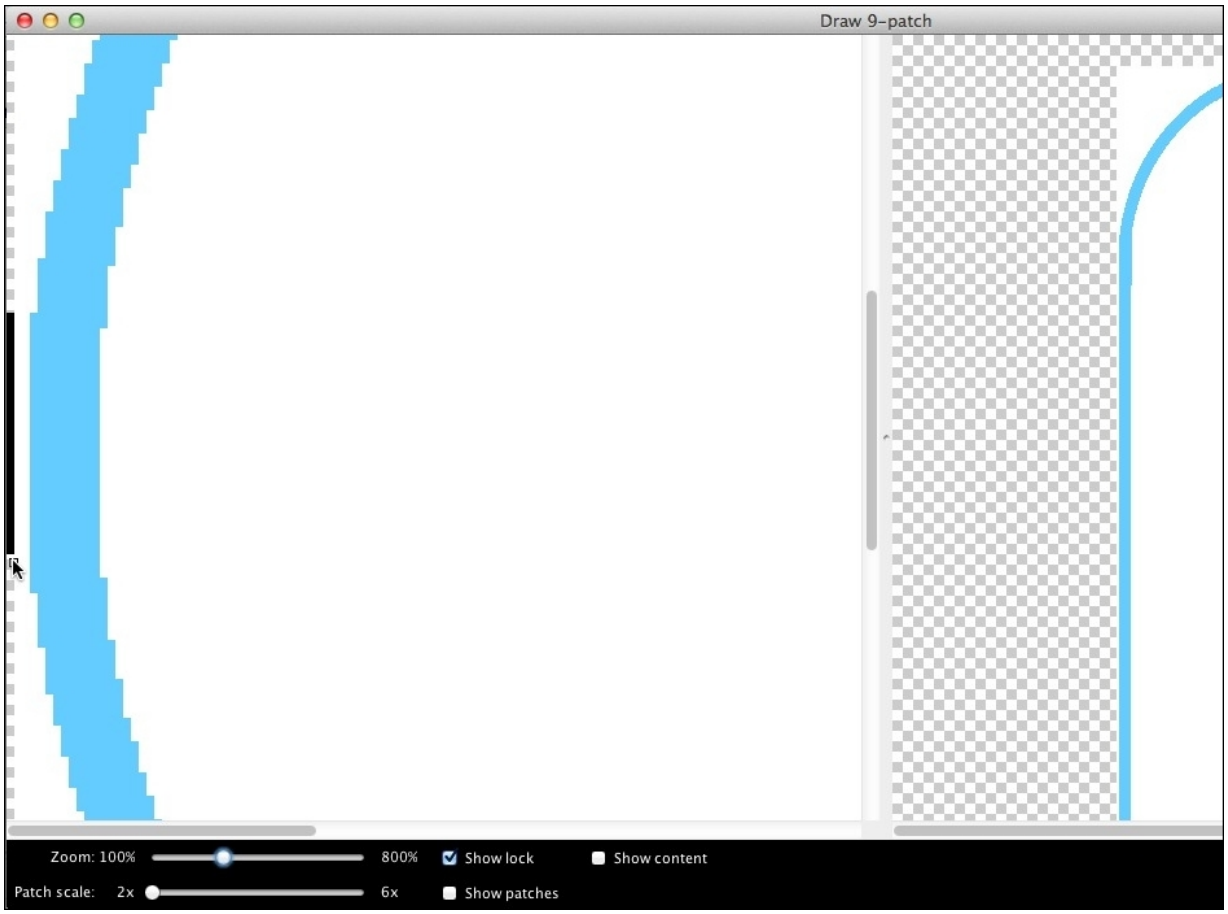
You'll find the `Draw9patch.bat` program in your computer's `sdk/tools` folder. Launch the editor and import your image by dragging it into the Draw 9-patch window.

The 9-patch workspace consists of the following:

- **Left pane**: This is the drawing area where you define your image's stretchable sections.
- **Right pane**: This is the preview area, which displays a preview of how your graphic will appear when stretched. As you're editing your image in the left pane, make sure you keep an eye on this preview.



To define the area that can be stretched horizontally, draw a line along the top of your image by clicking. Every time you click, a new pixel will be added to your line.



If you make a mistake, you can remove pixels by holding the ***Shift*** key while clicking each pixel you want to remove.

To define the area that can be stretched vertically, click to draw a line along your image's left edge.

Once you're happy with the results, save your 9-patch image by selecting **File** | **Save 9-patch**. This saves your image with the `.9.png` extension. You can then add this graphical resource to your project as normal.

Note

If you're not a fan of Draw 9-patch, or you just want to try an alternative, there are several free online tools that you can use to create 9-patch images, including http://draw9patch.com/.

Registering the user input

Input controls are your UI's interactive components, such as buttons, `EditText` fields, and checkboxes.

We've already seen how you can drop input controls into your UI, but input controls don't register user input out of the box.

To turn UI components, such as buttons and `EditText` fields, into fully-functioning interactive components, you'll need to write some extra code.

Handling click events

Click events are one of the most common input events your UI will have to handle. A click event is simply where the user touches an onscreen element, such as tapping a button or a checkbox.

A button cannot process clicks on its own; you'll need to create a listener and then assign it to the button. When the user taps the button, the listener registers the tap and executes the code from your `onClick` method.

Let's imagine that your layout contains a single button, and you want that button to register click events. As with many things in Android, you can create the same effect via XML or via your application code.

Handling onClick Via Java

Here's how you can handle these events using `Java`.

```
Button button = (Button) findViewById(R.id.button1

//Get a reference to the view you want to assign
the listener to//
button.setOnClickListener(new
View.OnClickListener() {
```

```
//Assign the setOnClickListener listener to this
view//

                @Override
                public void onClick(View view) {

//Override the onClick method//

....
....

// This is where you'd implement the code that
tells your app what action it needs to perform
whenever it registers a click event, for example
you might want your app to launch a new activity,
open a menu, or start playing a video//

}
  });
```

Handling onClick Via XML

You can create the same functionality via your layout resource file by adding the `android:onClick` attribute to your view, and then adding the `onClick` method to the corresponding Java file:

```
<Button
      android:id="@+id/button1"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:onClick="buttonClicked"
      android:text="Click me" />
```

Whenever the user clicks this button, the `buttonClicked` method will be executed, so the next step is to add this method to your Java file:

```
public void buttonClicked(View v) {
Toast.makeText(this, "The button has been
clicked!", Toast.LENGTH_LONG).show();

//So you can test whether the buttonClicked method
is working correctly, we'll tell the app to
```

```
        display a "The button has been clicked" message
        whenever buttonClicked is executed//

          }
```

Registering the EditText input

When you drop `EditText` into your UI, the user can type text into the field, but by default, `EditText` cannot read or use this information.

For `EditText` to acquire user input, you need to do the following:

1. Get a reference to the `EditText` field using `findViewById`.
2. Get the text from the `EditText` field using `getText()`.

You may then want to display this text elsewhere in your UI using `setText()`.

Now that you know how to register click events and retrieve the user input from `EditText`, let's look at an example app that combines all this functionality.

The example app

We'll create a simple app that asks users to enter their name into `EditText` and then tap a **Submit** button. The app will then retrieve the user's name from the `EditText` field and display it in `TextView` as the part of a welcome message.

Create a simple layout that contains these three onscreen elements:

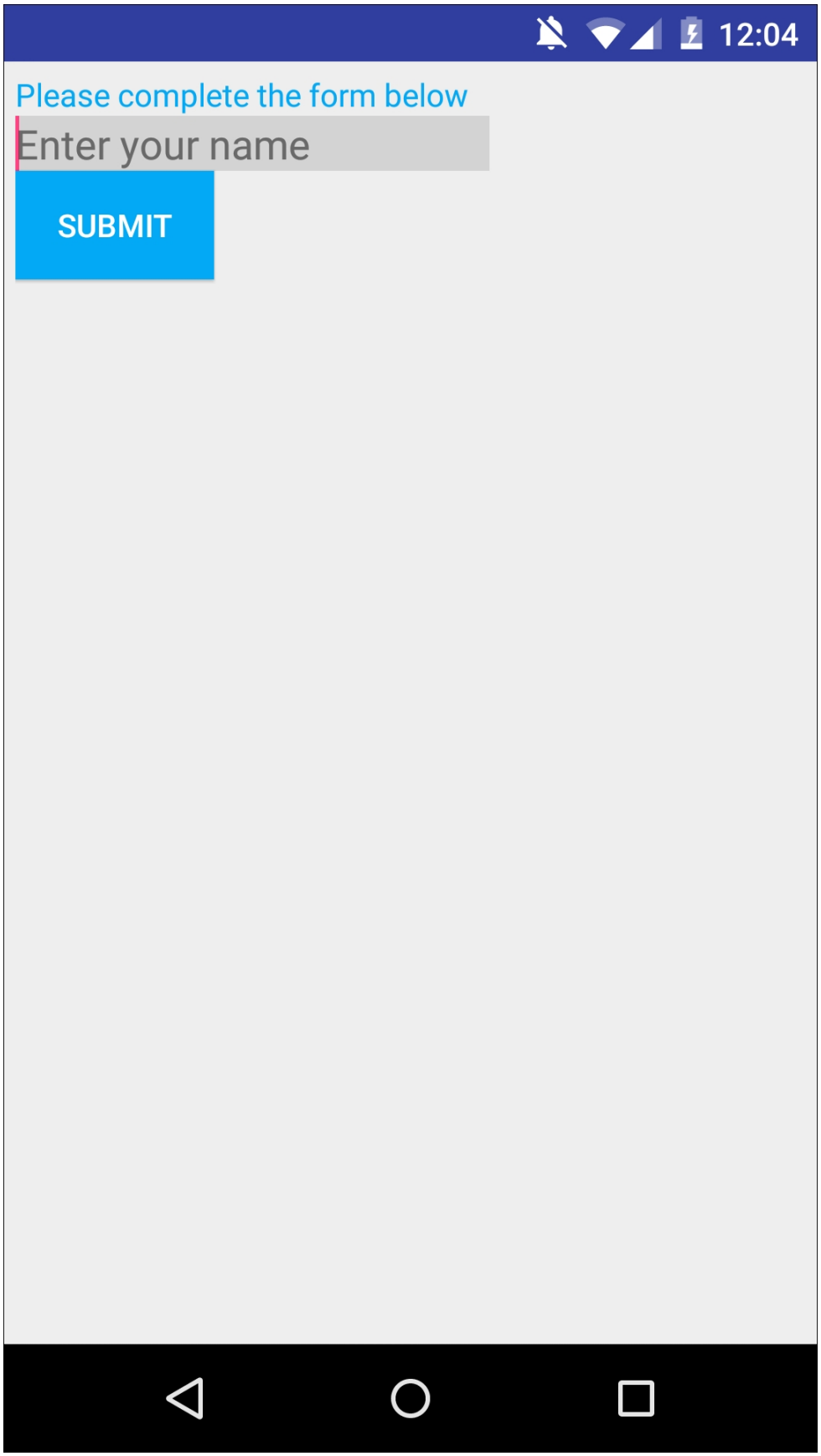
```
        <LinearLayout

        xmlns:android="http://schemas.android.com/apk/res/
        android"
          xmlns:tools="http://schemas.android.com/tools"
          android:orientation="vertical"
          android:layout_width="match_parent"
          android:layout_height="match_parent" >
```

```
      <TextView
          android:id="@+id/textView1"
          android:layout_width="wrap_content"
          android:layout_height="wrap_content"
          android:text="@string/form"
          android:textColor="@color/blue" />

//Add the TextView. Initially this view will
display instructions, but once the user has
submitted their name it'll update to display our
welcome message instead//


       <EditText
          android:id="@+id/editText1"
          android:layout_width="wrap_content"
          android:layout_height="wrap_content"

//Add the EditText//

          android:background="@color/grey"

//Depending on your app's color scheme, an empty
EditText may blend into the background, so you may
want to give the EditText its own background
color//


          android:hint="@string/yourName"
          android:ems="10" >

//Make it clear what information the user needs to
enter, using android:hint//

          <requestFocus />
      </EditText>

      <Button
          android:id="@+id/button1"
          android:layout_width="wrap_content"
          android:layout_height="wrap_content"
          android:background="@color/blue"
          android:text="@string/submit"
          android:textColor="@color/white" />

//Add the Submit button//
```

```
</LinearLayout>
```

Next, open `res/values/strings.xml` and create your string resources:

```
<resources>
<string name="app_name">Form</string>
  <string name="form">Please complete the form
below</string>
  <string name="yourName">Enter your name</string>
  <string name="submit">Submit</string>
</resources>
```


12:04
Please complete the form below
Enter your name
SUBMIT

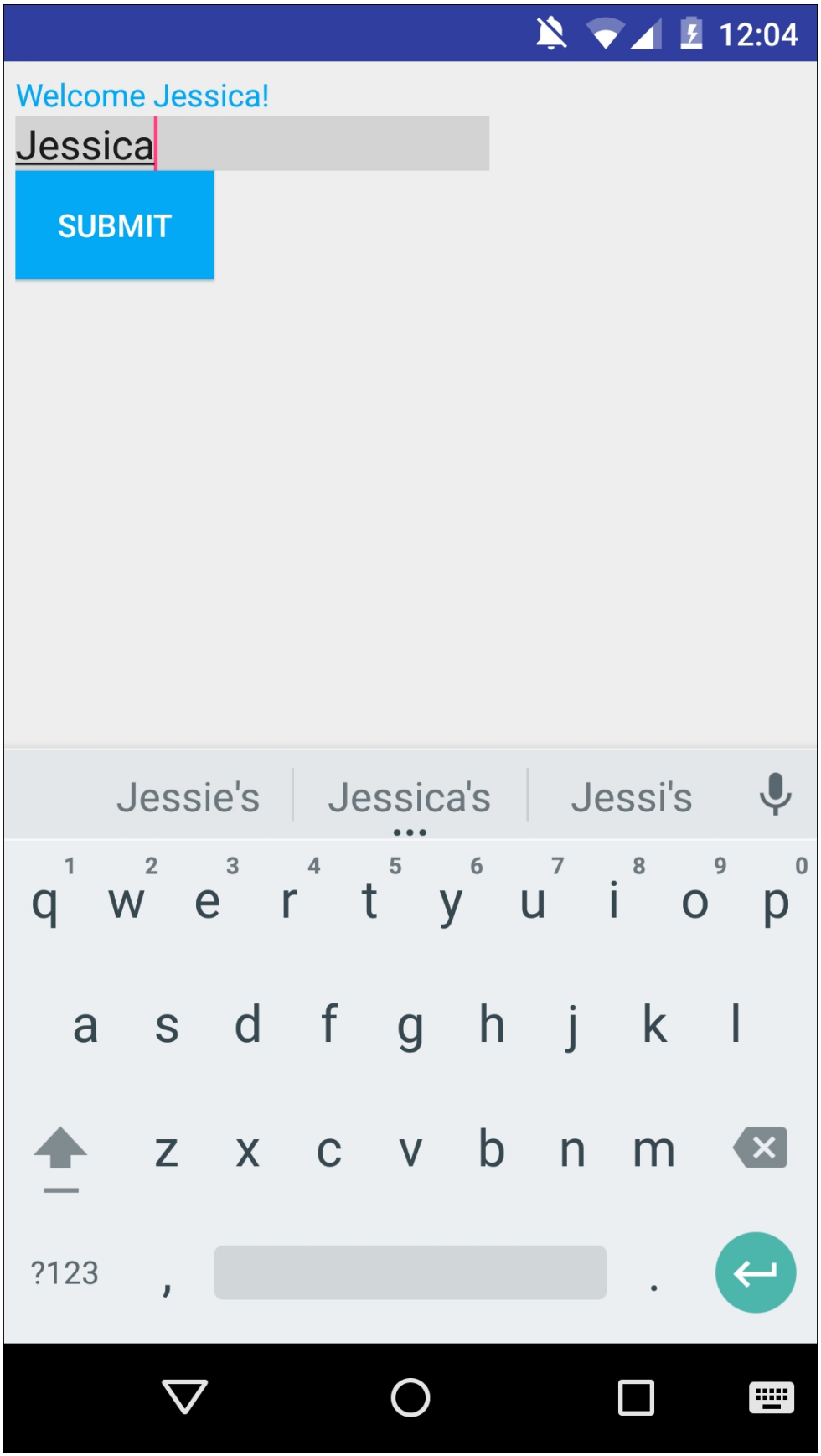
Now that you have your UI, it's time to give these on-screen elements the ability to register and process the user input.

In this example, we create an event listener and assign it to our `submitButton`. When the user taps the **Submit** button, the app registers this interaction and retrieves whatever text is currently in the `EditText` field. It then sets this value to `TextView`, replacing the default **Please complete the form below** text with the `EditText` value, plus two other bits of text, to create our complete welcome message:

```
package com.example.jessica.myapplication;

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

public class MainActivity extends Activity {

  Button submitButton;
  EditText nameEdit;
  TextView welcomeText;

  @Override
  public void onCreate(Bundle savedInstanceState)
{
      super.onCreate(savedInstanceState);
      setContentView(R.layout.activity_main);
      submitButton =
(Button)findViewById(R.id.button1);

  submitButton.setOnClickListener(new
View.OnClickListener() {
          public void onClick(View view) {

//Assign the setOnClickListener to your
submitButton//

              nameEdit = (EditText)
findViewById(R.id.editText1);
```

```
//Get a reference to the EditText//

              welcomeText = (TextView)
findViewById(R.id.textView1);

//Get the text from nameText and set it to the
welcomeText TextView. At this point, I'm adding a
bit of extra text (Welcome, and !) to create a
nicer greeting//

              welcomeText.setText("Welcome " +
nameEdit.getText().toString() + "!");
           }
       });
  }

}
```

This is it—boot up your app and try interacting with the different UI elements.


12:04
Welcome Jessica!
Jessica
SUBMIT
Jessie's
Jessica's
Jessi's
q w e r t y u i o p
a s d f g h j k l
z x c v b n m
?123
,
.

Working with fragments

When you're developing your UI for pre-Android N devices, one of the major restrictions you'll encounter is that you can only ever display a single activity on the screen at any one time. Fragments give you a way to overcome this restriction; although *technically* you can still only display one activity at a time; each activity can consist of multiple fragments.

> **Note**
>
> Android N introduced multi-window mode, which gives users the ability to display more than one app at a time, allowing them to see multiple activities at once, albeit from different applications! We'll take a closer look at multi-window mode later in this chapter.

A fragment is a self-contained, modular section of your app's user interface that you embed inside an activity. You *cannot* instantiate a fragment as a standalone application element. Think of a fragment as a kind of *sub activity* that has its own life cycle, behavior, and (usually) its own user interface.

Why do we need fragments?

The Android team introduced fragments in Android version 3.0, also known as **Honeycomb**, mainly to help developers make better use of all the extra screen space available on larger devices, such as tablets. Using fragments, you can divide each activity into distinct components and then control each part separately.

You can also create multiple layouts that combine your project's fragments in different ways depending on the current screen configuration. For example, you could create a multi pane layout that

combines multiple fragments in a single activity; and you could create a single pane layout that displays each fragment separately, which is better suited to smaller screens. Your app can then select the most appropriate layout (multi pane or single pane) depending on the current device.

In this way, fragments are useful for making the most out of devices with larger screens, while also providing a good user experience for users viewing your app on smaller screens.

Note

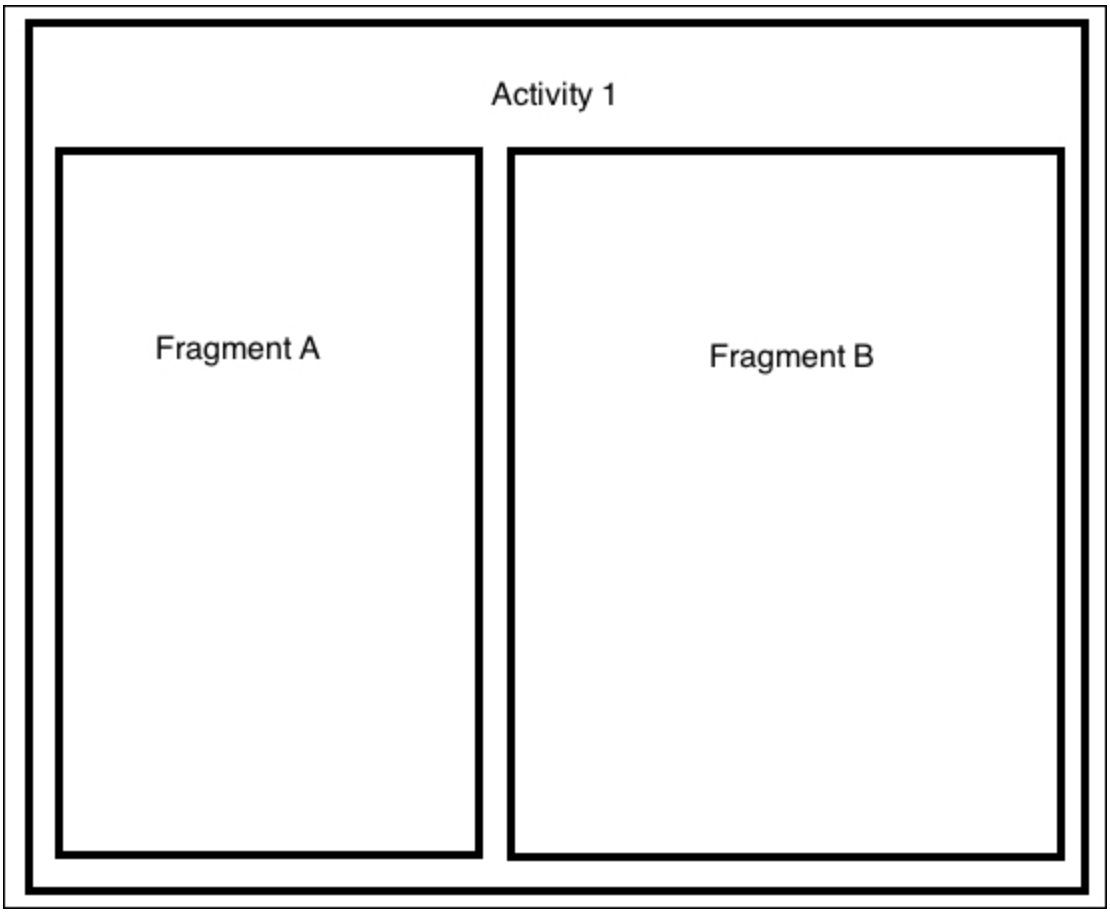
You may also want to use fragments to create layouts that are optimized for devices held in landscape and portrait orientation. For example, you can create a layout for landscape devices that displays multiple fragments in a side-by-side configuration, and you can create a single-pane layout that displays one fragment at a time when the device is in portrait mode.

The final major benefit of using fragments is that when a fragment's host activity is running, you can add, remove, and replace each fragment independently to create a truly dynamic user interface.

Let's take a look at an example of how an app that provides single-pane and multi-pane layouts may work.

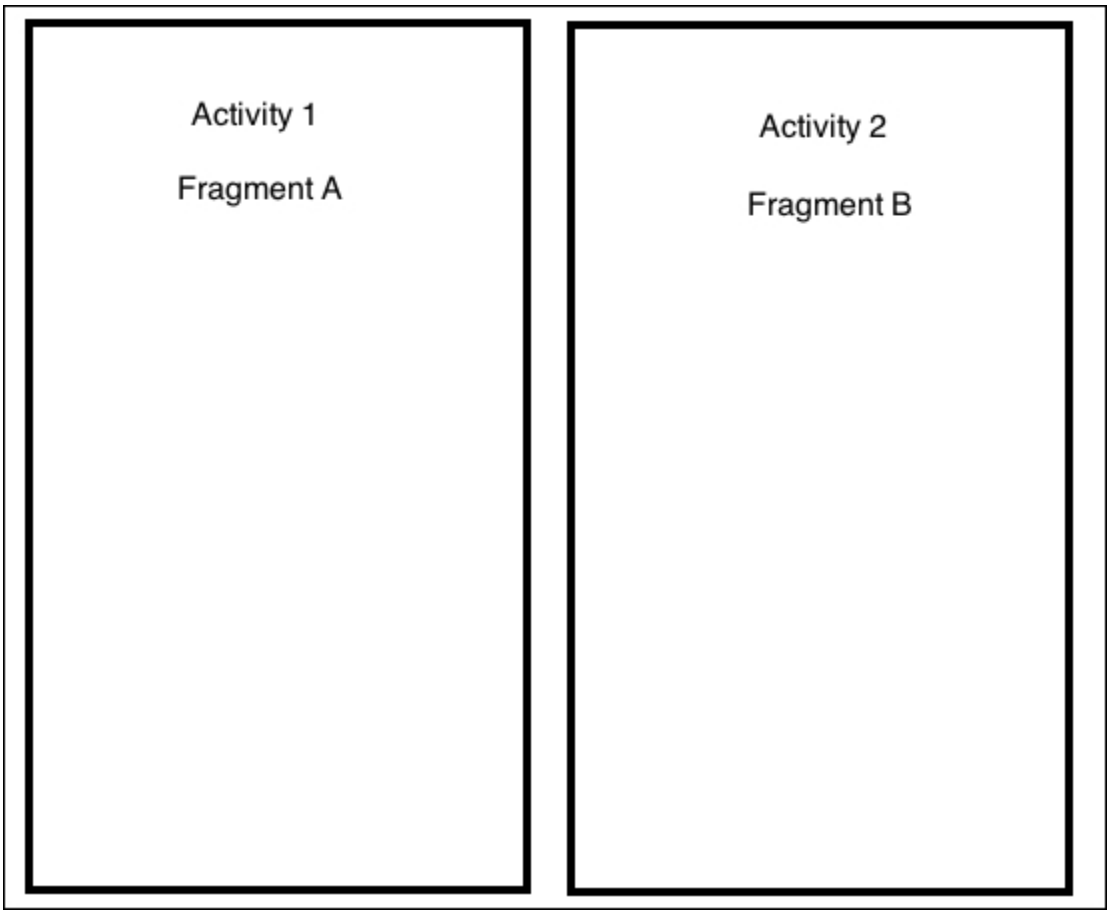
Imagine an activity (**Activity 1**) that contains two fragments: **Fragment A** and **Fragment B**. **Fragment A** displays a list of items. When the user selects an item in **Fragment A**, **Fragment B** updates to display information related to the selected item.

This app includes two different layouts, and it selects which layout to display based on the size of the device's screen. If the screen is large enough to accommodate both **Fragment A** and **Fragment B**, the app displays these fragments side-by-side in **Activity 1**, in a multi-pane layout.



If there isn't enough room to accommodate both fragments, the app will display each fragment separately, as different screens in a single-pane layout.

In this scenario, **Activity 1** displays **Fragment A** *only*. When the user selects an item from **Fragment A**, the screen updates to display **Fragment B**. This means your app creates an entirely new activity (**Activity 2**) that exists simply to host **Fragment B**:



Note

While fragments are important tools that can help you create more flexible layouts, fragments aren't a magical cure. Even if you do include fragments in your user interface, you'll still need to follow all the usual guidelines and best practices *in addition to using fragments*.

The fragment life cycle

Although a fragment has its own life cycle, the life cycle is directly affected by the life cycle of the host activity. Each life cycle callback for the host activity results in a similar callback for all its fragments; for example, when the host activity's `onStop()` method is called, all

the fragments within the activity also receive a call to `onStop()`, and when the activity is destroyed, so are all its fragments.

You can only manipulate a fragment's life cycle independently while the host activity is in its resumed state. At this point, you can add, remove, and replace fragments. However, once the host activity leaves its resumed state, all its fragments lose their independence and are once again dependent on the host activity's life cycle.

Like an activity, a fragment can exist in three states:

- **Resumed**: The fragment is visible in the running activity.
- **Paused**: Another activity is in the foreground and has focus, but the fragment's host activity is still visible.
- **Stopped**: Either the host activity has been stopped, or the fragment has been removed from the activity and added to the back stack. A stopped fragment is still alive, but it's no longer visible to the user.

One major difference between the life cycle of an activity and the life cycle of a fragment is how you restore each one and get its state back:

- **Activity**: When an activity is stopped, it's placed into a back stack of activities that's managed by the system.
- **Fragment**: When a fragment is stopped, it's only placed into a back stack that's managed by the host activity *if* you explicitly request that the instance should be saved. To make this request, call `addToBackStack()` during the transaction that removes the fragment.

Note

What is the back stack? And why is it so important?

The back stack keeps track of all the actions the user can reverse by tapping the device's **Back** button, whether that's a

physical back key or one of Android's illuminated soft keys. If you add a fragment to the back stack, then the user can backtrack to that fragment. If you don't add a fragment to the back stack, the user cannot recover that fragment.

Creating a fragment

In this section, I'll show you how to create a simple fragment. The first step is defining the fragment's UI component:

1. Open your project's `res` folder.
2. Right-click the `layout` folder and select **New**, followed by **Layout resource file**.
3. Give your layout resource file a descriptive name, such as `list_fragment`.
4. Select the **Root Element** option you want to use.
5. Double-check the **Directory name** option is set to **layout**.
6. Click **OK**.

Open your new layout resource file and define your fragment's UI:

```
<?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">

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

</LinearLayout>
```

Before we create the actual fragment, let's take a moment to look at the issue of backwards compatibility; specifically, how you can reap

the benefits of fragments even on devices that run pre-Honeycomb versions of Android.

Fragments and backwards compatibility

When you're developing an app, you should aim to support as many versions of Android as possible, as this will give you the widest potential audience. Since fragments didn't find their way into Android until version 3.0, if you want your app to be compatible with devices running anything earlier than Android Honeycomb, you'll need to add `v4` of **Android Support Library** to your project.

Note

Android Support Library is a handy set of code libraries that enable you to use features and APIs that wouldn't otherwise be available to the earlier versions of the Android platform.

Once you add the v4 library to your project, you can use fragments while remaining backwards compatible with devices running versions as low as Android 1.6.

To add this library to your project, launch Android SDK Manager, open the `Extras` folder and download the **Android Support Library** (if you're using Eclipse) or **Android Support Repository** (if you're using Android Studio).

If you're using Android Studio, add the library to your project by opening its module-level `build.gradle` file and adding the support library to the dependencies section:

```
dependencies {
....
....
....
```

```
    compile 'com.android.support:support-v4:23.1.0'
}
```

If you're developing in Eclipse, perform the following:

1. Create the `libs` directory in the root of your project.
2. Locate the support library called the JAR file in your Android SDK directory (for example, `<sdk>/extras/android/support/v4/android-support-v4.jar`). Copy this JAR into the `libs` directory you created in the previous step.
3. Right-click the JAR file and select **Build Path**, followed by **Add to Build Path**.

You'll also need to import the fragment class from the v4 support library (`import android.support.v4.app.Fragment`) and extend `FragmentActivity` instead of the usual `Activity` class (`public class ListFragment extends FragmentActivity`).

> ## Note
>
> There are no hard and fast rules about which versions of the Android system you should support, though you'll generally want to support as many versions as you feasibly can without compromising on your app's functionality. If you do decide against supporting the earlier versions of the Android platform, then you don't need to use the support library. If you're unsure, it may help you to look at Google's stats about the percentage of Android devices currently running each version of the Android platform, available at http://developer.android.com/about/dashboards/index.html.

Creating your fragment class

In addition to creating a layout, you need to have a class associated with your fragment. This class must extend `Fragment` or

`FragmentActivity`.

To add a new class to your project, perform the following:

1. Open your `Java` folder and right-click on your project's package name.
2. Select **New** followed by **Java Class**.
3. Give your class a descriptive name, for example, `ListFragment`.

Open your new `ListFragment` class; it will look similar to this:

```
package com.example.jessica.myapplication;

public class ListFragment {
}
```

You'll want to make the following changes:

```
package com.example.jessica.myapplication;


import android.os.Bundle;
import android.support.v4.app.Fragment;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;

public class ListFragment extends Fragment {

//Remember, if you want your app to work on
devices running anything earlier than Honeycomb,
you need to extend FragmentActivity rather than
extending the Fragment class//

    @Override

//To inflate your fragment's layout inside the
current activity, you need to override the
onCreateView() method//


    public View onCreateView(LayoutInflater
inflater, ViewGroup container,
```

```
//Your implementation of onCreateView() must
return a View, which is the root of your
fragment's layout//

Bundle savedInstanceState) {

///savedInstanceState is a Bundle that passes data
about the previous instance of the fragment, just
in case this fragment is being resumed//


      View view =
inflater.inflate(R.layout.list_fragment,
                container, false);
      return view;

//Inflate a new view hierarchy from the specified
layout resource file. In this example, that's
list_fragment.xml//

  }
}
```

The `inflate()` method (in the preceding code) takes the following arguments:

- The layout file that's being inflated (`R.layout.list_fragment`).
- The parent `ViewGroup` where the inflated fragment layout should be inserted (`container`).
- The `false` Boolean that indicates the inflated layout should be attached to the `ViewGroup` during inflation.

The next step is to add the fragment to your activity. You have two options:

- Embed the fragment in your activity's corresponding XML layout file
- Add the fragment at runtime via your application code

The most straightforward option is embedding your fragmentinside a layout file; however, this does have one big drawback—when you

add a fragment declaratively, the fragment is static and will remain on the host activity until it's destroyed. You *won't* be able to add or remove this fragment during the host activity's life cycle.

Adding a fragment to an activity via your application code gives you more freedom and flexibility, but it is more difficult to implement a fragment programmatically.

Since it's the most straightforward approach, let's start by looking at how to add a fragment via your layout resource file.

Adding a fragment to your activity declaratively

You can add a fragment to an activity using `<fragment>` in the same way you declare a view:

```
<?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:id="@+id/listFragment"
      android:layout_width="match_parent"
      android:layout_height="match_parent"

//Add the fragment//


class="com.example.jessica.myapplication.ListFragm
ent" />

//Identify the fragment you want to instantiate,
using the class attribute//

</LinearLayout>
```

Alternatively, you can identify the fragment using `android:name`:

```
<fragment
android:name="com.example.jessica.myapplication.Li
stFragment"
android:id="@+id/listFragment"
android:layout_width="match_parent"
android:layout_height="match_parent" />
```

When it's time to create this activity's layout, the system will instantiate the specified fragment, retrieve its layout, and then display it in place of the original `<fragment>` tags.

Adding a fragment to an activity at runtime

If you want to add, remove, or replace your fragments during the life cycle of the host activity, things get a bit more complicated, as you'll need to place these fragments in your activity at runtime, which means delving into your application code.

In this example, we'll be using the same `list_fragment.xml` file and the `ListFragment.java` class we created earlier. However, instead of the `<fragment>` placeholder, we'll be using `FrameLayout`, a special container view that indicates where the fragment will eventually be displayed in the layout file:

```
<?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">

  <FrameLayout

xmlns:android="http://schemas.android.com/apk/res/
android"
      android:id="@+id/fragment_container"
```

```
        android:layout_width="match_parent"
        android:layout_height="match_parent" />

</LinearLayout>
```

You then need to tell your activity to replace the `FrameLayout` container with your fragment at runtime:

```
package com.example.jessica.myapplication;

import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.support.v4.app.FragmentActivity;

public class MainActivity extends FragmentActivity
{

  @Override
  public void onCreate(Bundle savedInstanceState)
{
      super.onCreate(savedInstanceState);
      setContentView(R.layout.activity_main);

       if (findViewById(R.id.list_fragment) !=
null) {


          if (savedInstanceState != null) {
              return;
          }

           // Create a new Fragment//
          ListFragment firstFragment = new
ListFragment();


firstFragment.setArguments(getIntent().getExtras()
);

// If this activity was started with special
instructions from an Intent, pass the Intent's
extras to the fragment as arguments//


getSupportFragmentManager().beginTransaction()
```

```
// Call the beginTransaction() method on the
fragment manager instance//

                  .add(R.id.list_fragment,

//Call the add() method of the fragment
transaction instance, and pass it the resource ID
of the view that'll contain the fragment
(R.id.list_fragment) and the fragment class
instance(firstFragment)//

firstFragment).commit();

//The final piece of the above code calls the
commit() method of the fragment transaction//

        }
    }
}
```

When you add a fragment at runtime, you're free to add, remove, and replace this fragment as and when required. These changes are known as **fragment transactions**.

Fragment transactions and the back stack

Fragment transactions are changes you commit to an activity in response to user interaction.

Whenever you perform a fragment transaction, you have the option to save this transaction to the back stack. If you do add a transaction to the back stack, the user can navigate back to this fragment state by pressing the device's physical *back* button or the softkey.

If you perform a transaction that removes or replaces a fragment and *doesn't* add the transaction to the back stack, when you commit that transaction, the fragment is destroyed and the user can't navigate back to it.

If you want the option to add a fragment transaction to the back stack, make sure you add the fragment to the host activity during

that activity's `onCreate()` method. You can then add the fragment to the back stack by calling `transaction.addToBackStack` before you commit a transaction that'll remove the fragment.

In the next section, you'll learn how to add, remove, and replace fragments. Regardless of the kind of fragmenttransaction you're performing, you'll need to get an instance of `FragmentTransaction` from the `FragmentManager` class:

```
import android.support.v4.app.FragmentTransaction;
import android.support.v4.app.FragmentManager;

//Add the FragmentTransaction and FragmentManager
import statements//

FragmentManager fragmentManager =
getSupportFragmentManager();
FragmentTransaction fragmentTransaction =
fragmentManager.beginTransaction();

//Get the FragmentTransaction instance//
```

You can then perform the following fragment transactions.

Adding a fragment

You can add a fragment to an activity using the `add()` method.

Pass the `add()` method to `ViewGroup` where you want to place the fragment, identify the fragment you want to add, and then commit the transaction. For example, take a look here:

```
fragmentTransaction.add(R.id.fragment_container,
firstFragment).commit();
```

Removing a fragment

To remove a fragment from an activity, you need to use the `remove()` method. This method takes a reference to the fragment instance you want to remove, plus the aforementioned `commit()` method.

In this example, we're removing a fragment called `previousFragment`:

```
fragmentTransaction.remove(previousFragment).commi
t();
```

Replacing a fragment

To replace one fragment with another fragment at runtime, you need to call the `replace()` method of the fragmenttransaction instance.

The following example shows you how to replace one fragment with another fragment (`newFragment`) so that the user has the option to navigate back to the previous fragment. We'll also add the replaced fragment to the back stack:

```
Fragment newFragment = new Fragment();

// Create a new fragment//

FragmentTransaction transaction =
getSupportFragmentManager().beginTransaction();

//Create a new transaction//

transaction.replace(R.id.list_fragment,
newFragment);

//The replace() method takes two arguments: the id
of the view containing the fragment that's being
replaced, and an instance of the new fragment//

transaction.addToBackStack(null);

//Give the user a way of reversing the transaction
by adding the replaced fragment to the back stack.
Note that addToBackStack() takes an optional
String parameter that identifies this fragment
state on the back stack. If you don't need this
parameter, you can just pass null, similar to what
we're doing in this example//

transaction.commit();

// Commit the transaction//
```

Note

If you add multiple changes to the transaction and then call `addToBackStack()`, all changes applied *before* you called `commit()` are added to the back stack as a single transaction.

The multi-window support in Android N

Beginning with Android N, the Android operating system supports multi-window natively on both tablets and smartphones.

This new multi-window mode gives users the option to display more than one app at a time in a split-screen environment, either side-by-side or arranged one above the other. The user can resize these split-screen apps by dragging the dividing line that separates them, making one app larger and the other one smaller.

Giving users the ability to view multiple apps simultaneously is good news for productivity, paving the way for multi-app multitasking, such as bringing up a restaurant's address in Google Chrome and then typing the address directly into Google Maps, or replying to an incoming SMS without having to abandon the video you were watching on YouTube.

Another major benefit of the multi-window support is that users can drag data from one activity and drop it into another activity directly, whenever these activities are sharing the same screen. Since this direct drag and drop can come in handy for all sorts of everyday tasks, if your app doesn't already support drag and drop, then you should enable it for Android N.

How does the multi-window mode work?

Android smartphone and tablet users can switch to multi-window mode in the following ways:

- This can happen by opening the Overview screen (also known as the **recent apps screen** or **task list**) and long pressing an

activity title. The user can then drag the activity to a highlighted portion of the screen to open that activity in the multi-window mode.
- This can happen by opening the activity they want to view in the multi-window mode and then pressing the **Overview** button. The device will then put the current activity in multi-window mode and open the Overview screen ready for the user to select another activity to share the screen.

In the multi-window mode, only one activity is active at any given time; that's the activity the user has most recently interacted with, also known as the topmost activity. All other activities are placed in a paused state even though they're still visible to the user. This means that certain activities may need to continue running even when they're paused; for example, video-playing activities should continue playing video content even when they're not the topmost activity.

> ## Note
>
> If you're developing a video-playing app, the solution is to pause the video in `onStop` and resume playback in `onStart`, rather than in your app's `onPause` handlers.

When the user interacts with a paused activity, the activity is resumed and it's the *other* app's turn to be placed in a paused state.

Getting your app ready for multi-window mode

If your app targets Android N or higher and you *don't* specify whether your app supports multi-window mode, the Android system assumes that your app does include multi-window support.

However, it's good practice to explicitly state whether your app or activity supports multi-window mode by adding the new

`android:resizeableActivity` attribute to the `<activity>` or `<application>` sections of your project's `Manifest` file:

- `android:resizeableActivity="true."`: This app or activity can be launched in multi-window mode on phones and tablets.

- `android:resizeableActivity="false."`: This app or activity cannot be launched in multi-window mode. If the user attempts to launch this activity in multi-window mode, the app will take over the entire screen instead. If you want to ensure that the system only ever displays your app in full-screen mode, you'll need to use `android:resizeableActivity="false"` to *explicitly* disable multi-window support.

You can also set minimum allowable dimensions so that users cannot shrink your UI further than a specified size; you can do so using the `android:minimalSize` attribute. If the user tries to resize the activity so it's smaller than `android:minimalSize`, the system crops the activity to the size the user requests instead of shrinking your content.

Since your app may need to behave differently when it's in multi-window mode, Android N extends the `Activity` class so you can query an activity to find out whether it's in multi-window mode:

- `Activity.inMultiWindow`: This is called to find out whether the current activity is in multi-window mode. The `Fragment` version of this method is `Fragment.inMultiWindow`.
- `Activity.onMultiWindowChanged`: This is called whenever the activity switches into or out of multi-window mode. The fragment version of this method is `Fragment.onMultiWindowChanged`.

Testing your app's multi-window support

If your app targets Android N or higher and you *haven't* explicitly disabled multi-window support, then you'll need to test your app in multi-window mode to ensure you're providing the best possible user experience.

In particular, you should check the following:

- *Your app switches between full-screen and multi-window mode smoothly*. Launch your app in a full-screen mode, and then switch to multi-window mode. Check that this action happens quickly, smoothly, and doesn't cause your app to lag.

- *Your app resizes properly in multi-window mode*. Launch your app in multi-window mode, open another app, and then drag the divider line to test your app across a range of sizes. In particular, check that all UI elements remain visible and reachable, check that touch targets never shrink to the point where they become difficult to interact with, and check that your app's text remains readable. Test how your app handles resizing when it's sharing space with another app in both side-by-side and one-above-the-other configurations. You should also check that performing multiple resizing operations in quick succession doesn't result in lag or cause your app to crash.
- *The system respects your app's minimum dimensions*. If you have specified a minimum dimension, check that the system will prevent users from shrinking your app beyond this `android:minimalSize` value, by dragging the divider line.
- *Your app behaves as expected when it's visible but not active*. For example, if you've developed a video-playing app, you should verify that your app continues to play video as expected when it's not the topmost activity.

If you explicitly disable multi-window support (by including `android:resizableActivity="false"` in `Manifest`), then you should also install your app on an Android N device and verify that it's not possible to view your app in multi-window mode.

Picture-by-picture mode

Android N doesn't restrict the multitasking fun to smartphones and tablets! Android 7.0 also introduces a multitasking feature especially for Android TV users.

This new **picture-in-picture** (**PIP**) mode gives Android TV users the ability to watch a pinned window in a corner of the screen, while another activity runs in the background. The user can then toggle between the PIP window and the full-screen mode. If the user tries to play another video on the main screen, the PIP window will automatically close.

To use this feature in your Android TV apps, you need to register your app's video activity by adding `android:resizeableActivity="true"` and `android:supportsPictureInPictur="true"` to `Manifest`. You can then decide what events trigger the PIP mode in your app by calling `getActivity().enterPictureInPicture`.

When your activity switches to PIP, the system considers the activity to be in a paused state and calls your activity's `onPause` method. However, the whole point of PIP is that your app continues to play video in the corner of the screen. Therefore, it's crucial that your app checks whether an activity is paused because it's in the PIP mode. If it is, your app will continue to play its video content:

```
@Override
public void onPause() {

   // If onPause is called due to PIP, do not
pause playback//

if (inPictureInPicture()) {
       // Continue playback//

       ...
   }

   // If the activity isn't paused due to PIP
mode, then pause playback if necessary//

   ...
}
```

Bear in mind that in PIP mode, your video content is displayed in a small overlay window. This means users won't be able to clearly see

small details or interact with any UI elements. So, if your video activity features either of these, you should remove them when your activity enters PIP. You can then restore these UI elements when your activity switches back to the full-screen mode.

For example, look at this:

```
@Override
public void onPictureInPictureChanged(boolean
inPictureInPicture) {
   if (inPictureInPicture) {

       //This is where you'd hide the controls in
PIP mode//

       ...
   } else {

       //This is where you'd restore any controls
that are hidden when the activity enters PIP
mode//

       ...
   }
}
```

Summary

In this chapter, we saw how to take a basic UI to the next level using arrays, dimensions, 9-patch images, and color state lists. We also looked at how to create a more flexible user interface by incorporating fragments and multi-window mode into your design.

Now that we've spent a few chapters looking at the mechanics of how to build an effective UI, it's time to switch things up a bit and look at the *theory* behind creating a great Android user interface.

There's no shortage of best practices and guidelines that have come and gone with the different versions of the Android platform, but in version 5.0, the Android team announced a completely new direction for the Android UI.

It's called Material Design, and it's the subject of our next chapter.

Chapter 4. Getting Started with Material Design

Announced at the 2014 Google I/O conference and making its first appearance in Android Lollipop, Material Design is the new design language from Google.

This is particularly big news for Android, as Material Design's whole purpose is to provide a more consistent user experience; and as an open platform Android is particularly vulnerable to inconsistencies. Open your Android device's app drawer and spend some time flicking through your apps, and chances are you'll encounter at least a few apps that will look and feel *very* different from one another.

Material Design sets out to change all this by providing the tools and guidelines that you need to deliver a more unified user experience.

By adhering to Material Design principles, you can create apps that look good, run smoothly, and feel like a seamless extension of the Android platform. And who wouldn't want that?

However, Material Design is more than just a bunch of technical specifications about how much shading you should apply to a button and how opaque your UI's primary text should be. So, before we get into *how* to create a UI that perfectly compliments Google's new design direction, let's get a better understanding of what Material Design is by looking at the theory behind it.

The Material Design ethos

Material Design is based on the idea of translating the physical properties of real-world materials into the virtual screen, and it takes much of its inspiration from paper, ink, and print techniques.

Material Design encourages designers and developers to create on-screen objects that seem to possess the same qualities as real-world objects. This means using techniques such as shadows, light, and elevation to create a sense of depth and edges.

The way Material Design objects move also mimics how objects move in the physical world; for example, two real-world objects cannot occupy the same space simultaneously or pass through one another, so your on-screen objects shouldn't either.

To help you create this illusion, Material Design introduces the concept of a simulated 3D space where all UI objects possess *X*, *Y*, and *Z* coordinates. The *Z* coordinate is particularly important, as the positive *Z* axis extends outward toward the user, creating that sense of depth that's so very crucial to Material Design.

Note

In Material Design, every object occupies a position on the *Z* axis, and iteach object has a standard 1dp thickness.

Everything happens within Material Design's simulated 3D environment; objects appear, disappear, and transform into new objects without ever breaking the illusion of a continuous 3D environment.

When objects move through the Material Design space, they mimic how paper can be shuffled and bound together. For example, you can bind two sheets of material together along a common edge or *seam*, so they move together. By contrast, when two sheets of material overlap but occupy different positions along the *Z* axis, they're *not* bound together, and so can move independent of one another.

These design principles may give your user interface a distinct look and feel, but Material Design isn't just about how your UI looks. You

can use Material Design elements such as depth and shadow to give your users visual clues about your interface's hierarchy, subtly guiding them toward the UI elements that they need to interact with next. If used correctly, Material Design can help to ensure that your users instinctively know how to navigate and interact with your app's user interface.

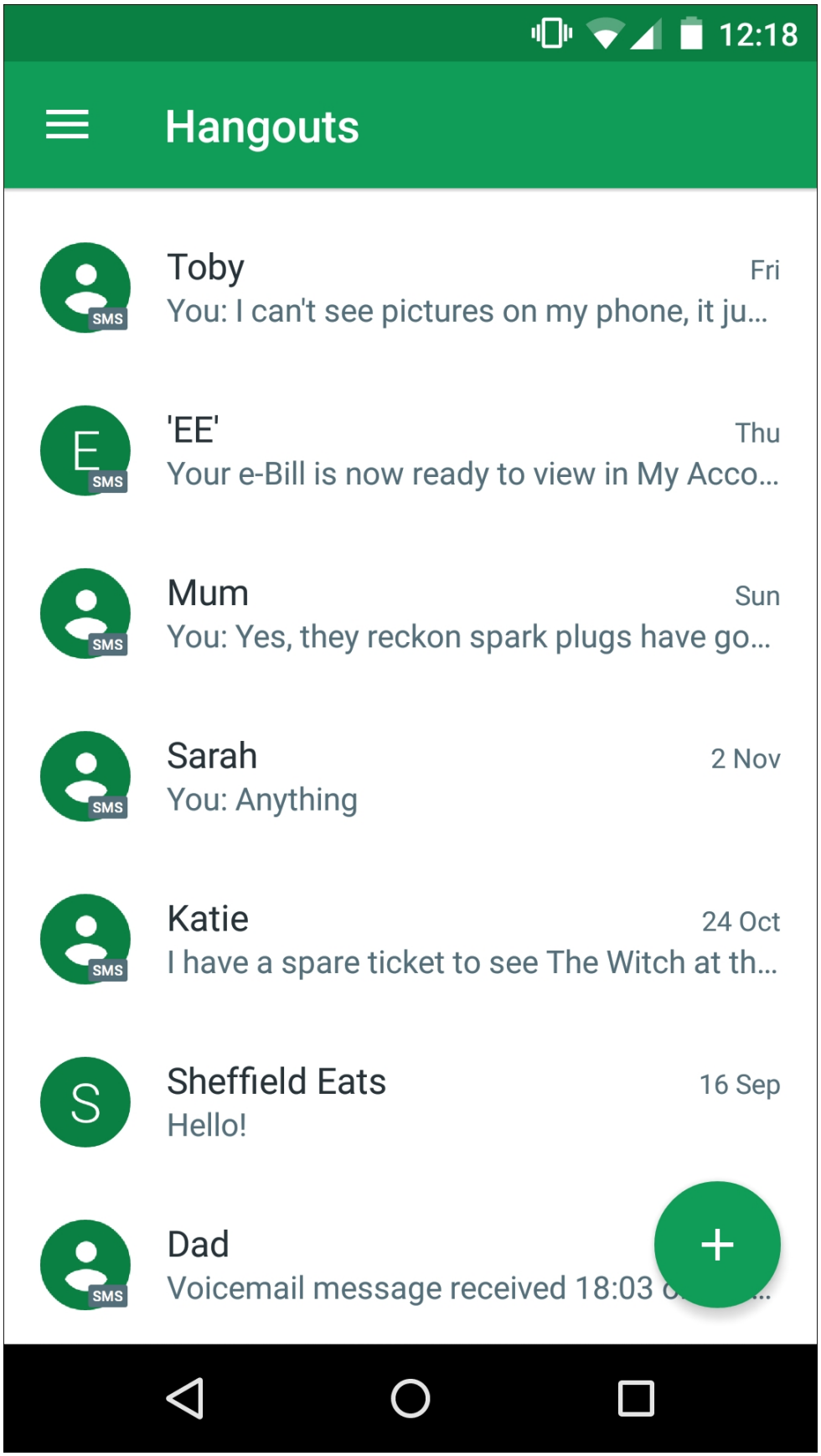
One of the most effective ways of getting to grips with Material Design is to look at some examples of Material Design that are done well, so you know what you're aiming for.

Since Material Design is Google's design language, what better place to look for pointers than Google's own apps?

Case study – Hangouts

The Hangouts app has undergone a major overhaul to bring it into line with Material Design.

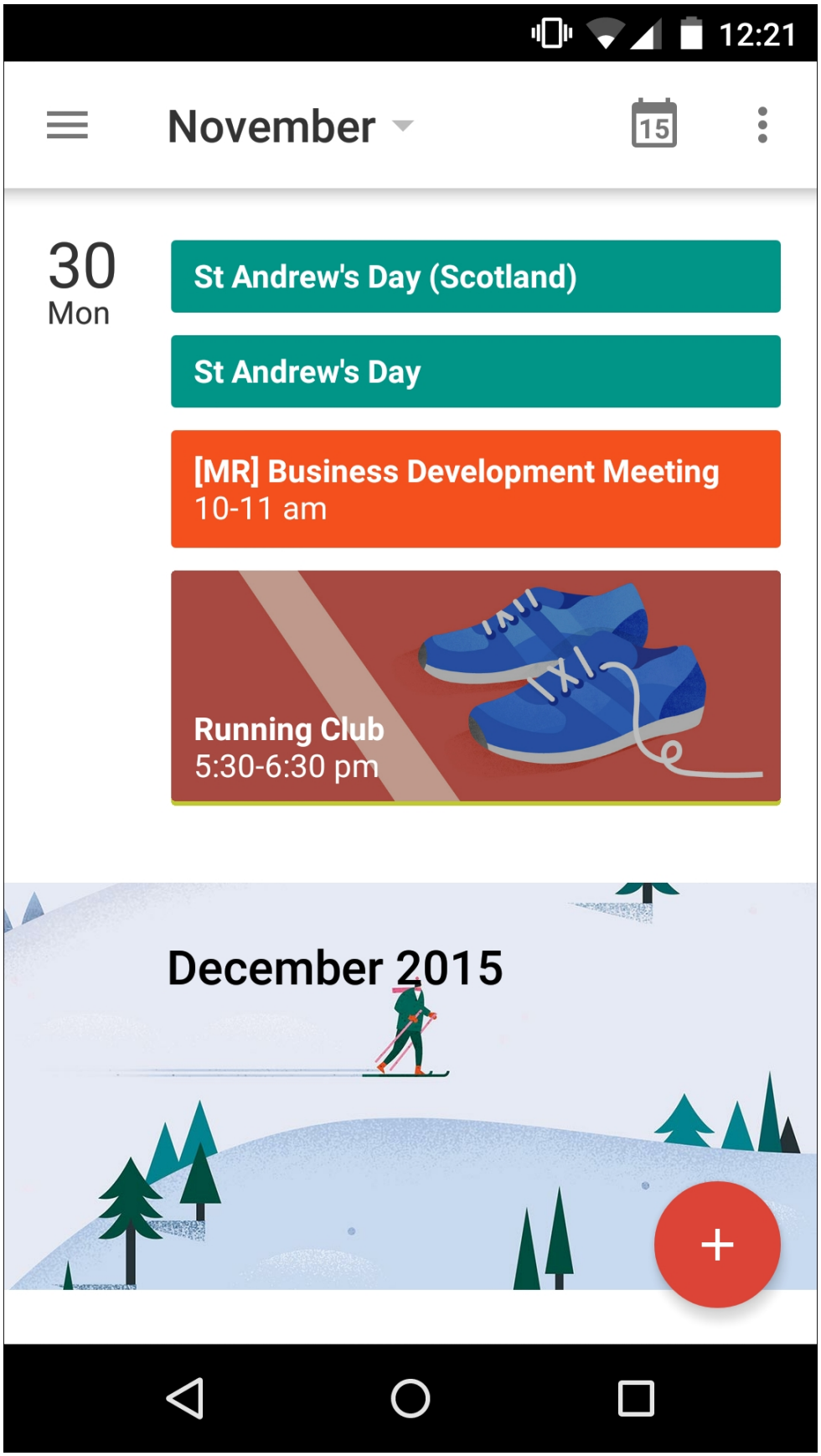
One of the most notable UI changes is the **Create New Message** button, which now appears as a floating action button, also known as a FAB. As the most important action, this FAB is prominently and conveniently located in the bottom-right part of the main Hangouts screen, so it's always within easy reach when the user needs to create a new message.


12:18
Hangouts
Toby
Fri
You: I can't see pictures on my phone, it ju...
'EE'
Thu
Your e-Bill is now ready to view in My Acco...
Mum
Sun
You: Yes, they reckon spark plugs have go...
Sarah
2 Nov
You: Anything
Katie
24 Oct
I have a spare ticket to see The Witch at th...
Sheffield Eats
16 Sep
Hello!
Dad
Voicemail message received 18:03
SMS

The Hangouts app uses two Material Design staples—elevation and shadows—to create the sense that the FAB is floating above all the other UI elements. This naturally draws the user's attention toward the screen's most important task; in this instance, the most important task is creating a new message.

Case study – Google Calendar

Material Design encourages the use of bold colors and large images. You'll find both of these in the updated Google Calendar app.


12:21
November
15
30
Mon
St Andrew's Day (Scotland)
St Andrew's Day
[MR] Business Development Meeting
10-11 am
Running Club
5:30-6:30 pm
December 2015

Google Calendar is a great example of a UI where colors and images not only make the app fun to look at, but also serve a purpose by helping the user pick up important information about their schedule at a glance. For example, a quick glance at the previous calendar image is all that's needed to see I have running club coming up on Monday night, thanks to the large, colorful picture of the running shoes.

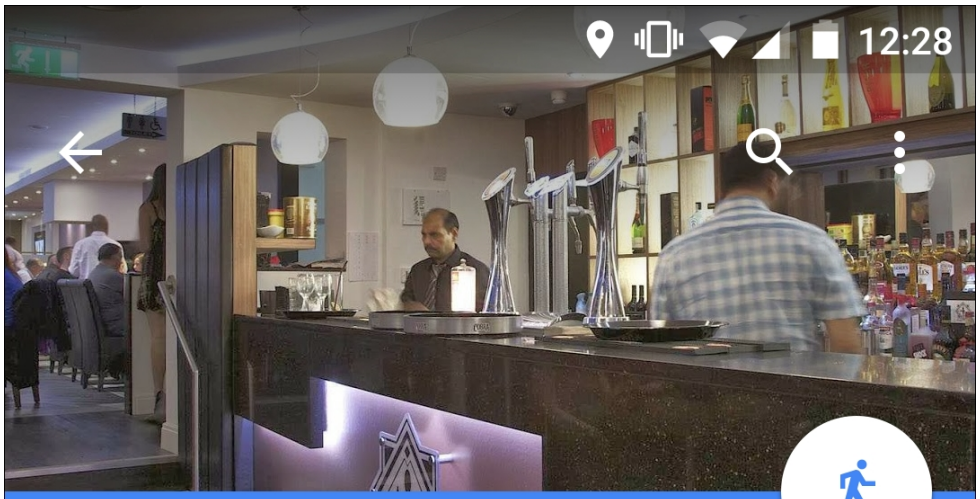
Calendar is also a great example of Material Design animations. Spend some time moving around the Calendar app, and you'll encounter different animated flourishes, such as the on-screen elements moving on and off the screen, assembling into new views.

These short animations make navigating through the Google Calendar app a more fluid, natural, and generally much more enjoyable experience.

Case study – Google Maps

Google Maps uses Material Design's concept of **bottom sheets** to create an immersive experience, where the user selects a location and then explores all the information related to this location without leaving the Maps environment.

Open the Google Maps app and select a location (whether it's a huge tourist attraction, a famous landmark, or simply your local pub), and you'll notice a bottom sheet peeking up from the bottom of your screen. In its default state, this bottom sheet displays a few facts about your chosen location, but when you drag the sheet upward, it expands to fill the entire screen. This expanded sheet contains much more information, including opening times, contact details, photos, and user reviews of your chosen location:



Butlers Balti House

4.0 ★★★★☆ 21 reviews 32 min

Indian restaurant

CALL SAVE WEBSITE

Aromatic Northern Indian dishes in a stylish, spotlit, contemporary space with upholstered seats.

44-46 Broad Ln, Sheffield, Sheffeld, South Yorkshire S1 4BT

Closed. Reopens at 5:30 pm

More info

Bottom sheets use shadows and elevation to create the impression that some components are positioned higher than others. In our Google Maps screenshot, the photo component is styled, so it appears to be lower than the rest of the bottom sheet.

Once you've read everything the bottom sheet has to offer, you can dismiss it by dragging the sheet off the screen. The bottom sheet will fold away, revealing the main Google Maps screen, leaving you with the impression that this main screen was hiding behind the bottom sheet the entire time.

Getting started with Material Design

Now that we've explored the major concepts behind Material Design and seen a few examples of Material Design done well, it's time to look at how you can apply some of the core Material Design principles to your own Android apps.

The next few sections will show you how to give your app a Material Design makeover using visual techniques such as shadows and elevation. You will also learn how to make some more fundamental changes to the way your app functions, by adding things such as FABs, cards, and `RecyclerView`. Let's start by making sure your app *looks* the part.

Applying the Material theme to your app

Applying the Material theme is the quickest and easiest way to get a consistent Material Design look across your entire app.

Android provides light and dark variations for you to choose from:

- `Theme.Material`: This is the dark version of the Material theme. This is considered the default Material theme.
- `Theme.Material.Light`: This is the light variation of the Material theme.
- `Theme.Material.Light.DarkActionBar`: This is the light version of the theme but with a dark action bar.

To apply the Material theme to your app, you need to create a new style that inherits from the version of the theme that you want to use (`Theme.Material`, `Theme.Material.Light`, or `Theme.Material.Light.DarkActionBar`). Open your

project's `res/valus/styles.xml` file, and create a new style that inherits from the theme of your choice:

```
<resources>
  <style name="AppTheme"
parent="android:Theme.Material">

//Inherits from the standard Theme.Material//

  </style>
</resources>
```

To give your app its own identity while maintaining the look and feel of Material Design, you may want to add your own customizations to your inherited Material Design style. One of the most common customizations is changing the theme's base colors; for example, you may want to change the color of the action bar to match your app's *primary color*.

Material Design uses two kinds of colors: primary and accent. As the name suggests, the primary color is the main color that's used throughout your app—in Google Hangouts the primary color is green.

The accent color is a brighter shade that you use to draw attention to your app's most important elements, such as the floating action button or title. By using a consistent primary color with the occasional splash of bolder accent color, you can create a user interface that's colorful and vibrant but doesn't distract the user from your application's content.

When it's time to customize the colors used in your inherited Material Design theme, there are several available attributes you can use:

- `colorPrimary`: This sets the color of the action bar's background. This is your app's primary color.
- `colorAccent`: This is your app's accent color. This should compliment your app's primary color, and is a good way of drawing the user's attention toward important UI elements.

- `colorControlNormal`: This sets the color of your app's framework controls when they're in their default, non-activated state.
- `colorControlActivated`: This sets the color of your framework controls when they're in their activated state. This attribute overrides `colorAccent`.
- `android:textColorPrimary`: This sets the color of the text on your controls. On devices running a pre-Lollipop version of Android, this attribute sets the color of the overflow menu and the action bar title.

The following attributes only work on devices running Android 5.0 or higher:

- `colorPrimaryDark`: This is a dark variant of your app's primary color. This attribute sets the color of your navigation bar (via `navigationBarColor`) and status bar (via `statusBarColor`).

- `colorControlHighlight`: This is the color applied to your app's framework control highlights, such as ripple animations. You can use this attribute to provide visual feedback that compliments your app's color scheme. Just don't get carried away—too much visual feedback and you're running the risk of overwhelming the user.
- `colorSwitchThumbNormal`: The user interacts with a toggle switch by dragging the switch's `Thumb` section back and forth. This attribute sets the color of the `Thumb` element when it's in the *off* position.
- `android:colorButtonNormal`: This sets the color of a button when it's in its default, non-pressed state.
- `android:colorEdgeEffect`: This sets the color of your app's overscroll effect, which occurs when the user tries to scroll beyond your content's boundaries.
- `android:navigationBarColor`: This sets the color of the navigation bar, which is the bar that appears at the bottom of your device and contains the ***Back***, ***Home***, and ***Recent*** softkeys.

To create your own variation of the Material theme, add any of the preceding attributes to the style we created earlier:

```
<resources>

  <style name="AppTheme"
parent="android:Theme.Material">

    <item
name="android:colorPrimary">@color/blue</item>

//Specifies that the theme's primary color should
be blue//

    <item
name="colorPrimaryDark">@color/darkblue</item>

//Specifies that the navigation bar and status bar
should be dark blue//

    <item name="colorAccent">@color/white</item>

//Specifies that the app's accent color should be
white//

<item
name="colorSwitchThumbNormal">@color/white</item>

//All the switches should have a white thumb
element//

  </style>

</resources>
```

Now, you know how to customize the colors in your Material-inspired theme, but what colors should you use? Once again, Material Design has the answers.

Choosing your color scheme

Choosing your color scheme is one of the most important UI decisions you need to make, as the colors you pick will affect every

single part of your user interface.

To help you make this crucial design decision, the Android team released a complete palette of primary and accent colors that are designed to compliment one another. You can find the complete Material Design palette at https://www.google.com/design/spec/style/color.html# .

When choosing your app's color scheme, you should select three hues from the primary palette (that's any color marked 500) and one accent color from the secondary palette (that's any color *except* the 500 colors).

Tip

Need some help selecting your palette?

Struggling to choose from the massive selection of colors and shades on offer? You may want to check out one of the many websites that can generate a complete Material Design palette for you. Simply select two colors, and the website generates a complete palette of primary and accent colors based on your selection. There's plenty of palette generators available online, but one of the simplest and easiest to use is Material Palette at http://www.materialpalette.com/ .

Backwards compatibility

The Material theme is only available in Android 5.0 (API level 21) and upwards, so in its default state you cannot use the Material theme or any other custom theme derived from it on devices running earlier versions of Android.

However, you can make the Material theme available to users running API 7 and higher using the **AppCompat library**.

To add this library to your project, make sure you've downloaded the latest version of **Android Support Library** (if you're using Eclipse) or **Android Support Repository** (if you're using Android Studio). AppCompat relies on the v4 support library so make sure you've also added this library to your project.

Android Studio users need to add AppCompat as a dependency in their module-level `build.gradle` file:

```
dependencies {
    ...
    compile 'com.android.support:appcompat-
v7:23.1.0'
}
```

Eclipse users will need to locate the AppCompat library in their Android SDK directory, copy the library into their project's `libs` directory, right-click on the JAR file and select **Build Path**, followed by **Add to Build Path**.

To use AppCompat, make sure all your project's activities extend `AppCompatActivity`:

```
import android.support.v7.app.AppCompatActivity;

public class MainActivity extends
AppCompatActivity {
```

Your theme must also inherit from `Theme.AppCompat`:

```
<style name="AppTheme" parent="Theme.AppCompat">
```

After this, you're free to customize the Material theme as normal.

Creating a sense of depth

Material Design combines three visual techniques to create a sense of depth:

- **Lights**: In Material Design's simulated 3D environment, virtual lights illuminate the on-screen objects and allow them to cast shadows. Lighting takes the form of either key lights (which cast directional shadows) or ambient lighting (which create soft shadows from all angles).
- **Shadows**: These give your users important visual clues about each object's depth. When an object moves, shadows continue to supply important information including the direction the object is moving toward, and whether the distance between this object and other on-screen objects is increasing or decreasing. The shape an object casts is defined by the object's background and *not* its content, so a circular button will cast a circular shadow, regardless of the shape of the button's icon.
- **Elevation**: Each UI element has its own elevation, which is the object's elevation along the *Z* axis. Elevation can help you communicate the importance of each screen's different UI elements, naturally drawing the user's eye toward the most important on-screen elements.

You can create shadows by specifying an object's elevation. When you add an elevation, the framework automatically casts a shadow across the items behind the object. An object's elevation determines the appearance of its shadow; a view with a higher *Z* value will cast a larger, softer shadow.

To set a view's elevation, use the `android:elevation` attribute:

```
<TextView
  android:layout_width="wrap_content"
  android:layout_height="wrap_content"
  android:elevation="20dp"
  android:text="Hello down there!" />
```

Tip

Just keep in mind that all material elements have a thickness of 1dp, so elevation is the distance from the top of one surface to

the top of another.

If you want to set a view's elevation programmatically, use the `View.setElevation` and `View.getElevation` methods.

Every object has a default *resting* elevation that should be consistent across your app. For example, if you position a floating action button at 10dp on one screen, it should be positioned at 10dp on *every* screen.

Objects can also have a responsive elevation, which is where they temporarily change their elevation in response to a user action. For example, if the user selects a picture in a gallery app, this picture may temporarily increase its elevation to indicate its selected status.

Responsive elevations should also be consistent across your app, so if the picture in a gallery app changes elevation by 5dp, all other images that have responsive elevations must display the exact same 5dp behavior.

If an object changes elevation, it should return to its resting elevation as soon as possible—typically as soon as the input event is completed or cancelled.

When adding components with responsive elevations, check that there's no possibility of one component encountering another component while it is changing elevation. Remember that in Material Design, objects cannot pass through one another! If the space is tight, then one solution is to use animations to temporarily move objects out of the way; for example, you could animate one object a few pixels to the right in order to clear the way for an object that's changing its elevation.

Creating a Material Design structure

At this point, you've selected your color palette, created a customized version of the Material theme, and know how to add

elevation to your user interface.

The next step is to look at some of the new structural elements you can add to your app, so it doesn't *just* look like a Material app, it acts like a Material app, too.

Let's start with one of the most familiar Material Design features: floating action buttons.

Floating action buttons

A FAB is a circular sheet of material that appears to float above the user interface (hence the name). If you have a persistent action that needs to be readily available to the user, you should consider displaying it as a FAB.

Floating icon buttons represent a single promoted action, and use the familiar system icons.

> **Note**
>
> You can find all the system icons at https://www.google.com/design/icons/ .

Adding a FAB to your project via XML is fairly straightforward, as it uses many attributes you're already familiar with, such as `android:id` and `layout_width`. However, in our FAB example, we're going to use a new element called `CoordinatorLayout`. This attribute lets you control the way your UI elements interact, and is particularly useful for telling FABs how they should react when the user scrolls the screen; should they move or remain anchored in the same place?

In this example, we're placing our FAB inside `CoordinatorLayout` and telling it to remain anchored to the bottom of the toolbar:

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.design.widget.CoordinatorLayout
.........
.........
..........

<android.support.design.widget.FloatingActionButto
n
  android:id="@+id/myfab"
  android:layout_width="wrap_content"
  android:layout_height="wrap_content"
  app:layout_anchor="@id/actionbar"

//The FAB should stay anchored to the action bar//

  app:layout_anchorGravity="bottom|right|end"
  android:layout_margin="@dimen/fab_margin"
  app:elevation="20dp"

//Set the button's elevation so it appears to
hover above the rest of the UI, and casts a shadow
across the items behind it//

  android:src="@android:drawable/ic_dialog_email"
/>

//This references the icon that the FAB should
display. In this example, I'm using the create new
email icon//

</android.support.design.widget.CoordinatorLayout>
```

You may notice that we haven't specified the FAB's background color; that's because the FAB defaults to your theme's `colorAccent` property unless you specify otherwise.

You can add a click event in the usual way. So, to make things interesting, I'll throw in another new element from Material Design: the **Snackbar**.

Snackbars are similar to Toasts, but the key difference is that users can interact with them. The user dismisses a Snackbar by swiping it off the screen. Snackbars appear at the bottom of the screen. So, they are perfect for displaying a message that relates to our FAB,

which also happens to be positioned towards the bottom of the screen:

```
fab.setOnClickListener(new View.OnClickListener()
{
    @Override public void onClick(View v) {
        Snackbar.make(content, "The FAB has been
clicked!",          Snackbar.LENGTH_SHORT).show();

//Create a snackbar and display the message "The
FAB has been clicked!"//

    }
});
```

When creating FABs there are a few guidelines you should keep in mind:

- **Be positive**: Only use FABs for positive actions such as **Create**, **Like**, **Share**, or **Navigate,** and never for destructive actions such as **Archive** or **Trash**.
- **Use consistent spacing**: On mobile devices, you should place your FABs 16dp or more from the edge. On tablet-sized devices, floating action buttons should be a minimum of 24dp away from the edge.

- **Avoid customized FABs**: Always use the standard circular icon, and don't be tempted to give your action button extra dimensions. If you do want to put your own spin on a FAB, you can always animate the icon inside the button.
- **Don't include overflow actions**: Overflow menus belong in toolbars, not in FABs.

Tip

Impress with FAB animations

As a prominent UI element, FABs are the perfect opportunity to surprise and delight your users with animated flourishes. For

example, you could design your `Create new email` FAB, so it transforms into a new e-mail when tapped. Experiment with different animations and transitions, but don't get carried away! Animations should be subtle, finishing touches and never get in the user's way or run the risk of distracting them from your app's actual content.

Bottom sheets

A bottom sheet is a sheet of material that slides up from the bottom of the screen in response to user action, for example the sheet that appears when you select a point of interest in Google Maps.

A bottom sheet's initial height is relative to the height of the list items it contains, but a bottom sheet's initial height shouldn't be more than its 16:9 ratio.

Bottom sheets initially only cover a portion of the screen, but they do expand to fill the entire screen when the user swipes upward. When a bottom sheet is expanded to its full height, the user can scroll through its content; again, the Google Maps app is a perfect example of this functionality.

Bottom sheets are best suited to displaying three or more actions that don't require a description. If you want to display fewer than three actions, or you want to include detailed descriptions, then you should consider using a dialogue or menu instead.

There are two types of bottom sheets:

- Persistent bottom sheets:

  These are persistent structural elements that appear throughout your application. They display in-app content that supplements the main view. Persistent bottom sheets remain visible even when they're not actively in use, and they rest at the same elevation as the rest of your app. Persistent bottom sheets are

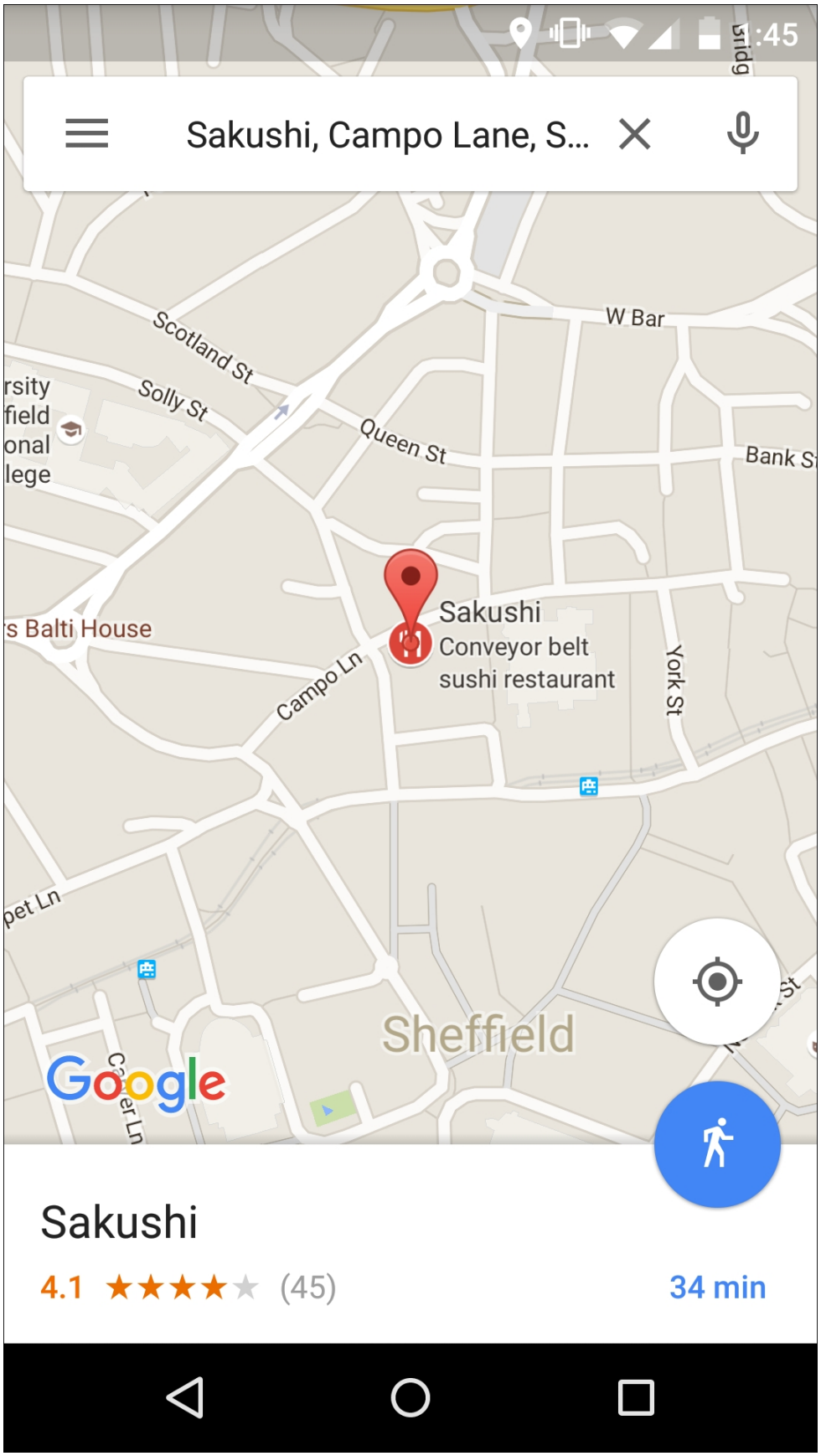
useful for drawing the user's attention toward important content by presenting it in a unique way.

- Modal bottom sheets:

  This is a temporary sheet of material that rests at a higher elevation than the rest of your content.

  You can use modal bottom sheets to present actions in a list or grid format as an alternative to menus and simple dialogues. It's impossible for the user to overlook a modal sheet; when an active modal bottom sheet slides onto the screen, the rest of the screen dims. Users have to dismiss the modal sheet before they can interact with the underlying content. Modal bottom sheets are handy if you need to present the user with a list of actions, and there's no suitable place in your user interface where you can insert a menu button.

However, on larger screens where space is less restricted, components such as dialogues and menus may be more appropriate than modal bottom sheets. This is because bottom sheets, as the name suggests, always appear at the bottom of the screen. For a user interacting with your app on a larger device, such as a tablet held in portrait mode, a modal sheet may appear at a significant distance from where the user triggered the sheet. This might not sound like a big deal, but little annoyances like this can really add up, chipping away at the user's overall experience of your app.


:45
Sakushi, Campo Lane, S...
Scotland St
W Bar
Solly St
Queen St
Bank St
Sakushi
Conveyor belt
sushi restaurant
Campo Ln
York St
Sheffield
Google
Sakushi
4.1
(45)
34 min

The preceding screenshot shows an example of a modal bottom sheet in its default state before it's expanded.

CardView

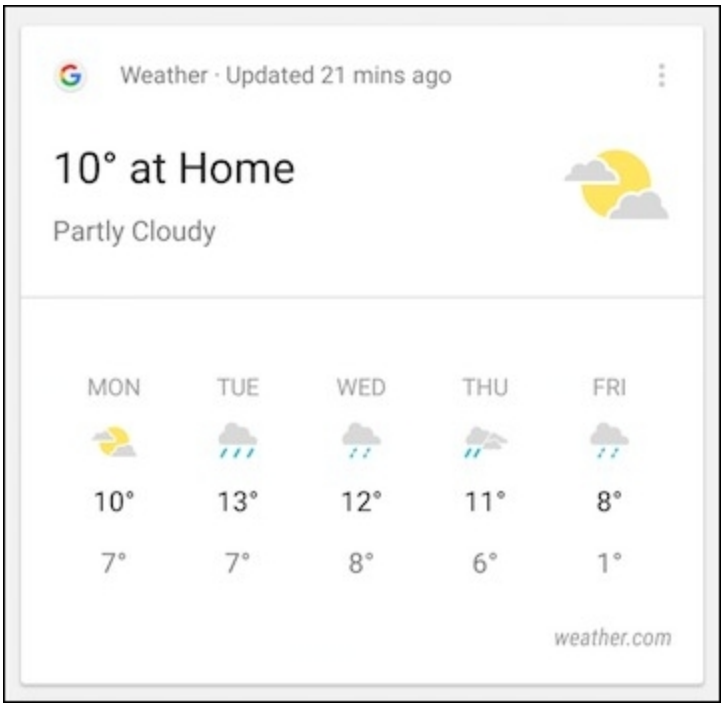
Cards give you a convenient and consistent way of displaying related content, particularly content that comprises of multiple data types. For example, you could create a card that contains images, links, text, or video about a specific subject.

Cards are also handy when you want to display data alongside interactive features, such as **+1**, comments, and user reviews. Just be wary of overloading your cards with too much information.

Cards have a constant width and a variable height that can expand temporarily depending on the available space.

Each card is made up of blocks of content. A typical card consists of the following:

- *A header or primary title*. This should indicate what the card is all about.
- *Rich media* such as images or video. Including rich media helps the user get valuable information from your card at a glance; for example, if you're designing a weather app, featuring a picture on every card means your users get an idea of what the weather is going to be like just by glancing at that card.



- *Supporting text*. Text that provides important information about the card.

- *A primary action*. This is the most important action the user can perform within the context of this card. Think of this as the card's equivalent of a FAB.
- *Optional supplemental actions*. These can be icons, text, or even UI controls that give the user the ability to change the card's content. In our weather example, you may include a slider that allows the user to scroll through the weather forecast for each hour.

When you're designing your card's content hierarchy, you should place the most important content at the top, whereas supplemental icons usually belong at the very bottom of the card.

You add a card to your layout using `CardView`. The following code shows you how to add an empty card to your layout via XML:

```
<android.support.v7.widget.CardView
```

```
xmlns:card_view="http://schemas.android.com/apk/re
s-auto"
     android:layout_width="match_parent"
     android:layout_height="wrap_content">

</android.support.v7.widget.CardView>
```

You add content to `Cardview` in exactly the same way you add content to a regular layout. The following example demonstrates how to create a `LinearLayout` that contains a `Contacts` card. This `Contacts CardView` displays each person's name and avatar:

```
<?xml version="1.0" encoding="utf-8"?>

<LinearLayout

xmlns:android="http://schemas.android.com/apk/res/
android"

xmlns:cardView="http://schemas.android.com/apk/res
-auto"
  android:layout_width="match_parent"

  android:layout_height="match_parent"
  android:padding="20dp"  >

  <android.support.v7.widget.CardView
      android:layout_width="match_parent"
      android:layout_height="wrap_content"
      android:id="@+id/contacts" >

   <LinearLayout

xmlns:android="http://schemas.android.com/apk/res/
android"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

<TextView
     android:id="@+id/contactName"
     android:layout_width="wrap_content"
     android:layout_height="wrap_content"
     android:text="@string/contactName" />
```

```
 <ImageView
     android:layout_width="wrap_content"
     android:layout_height="wrap_content"
     android:id="@+id/contactphoto"
     android:src="@drawable/avatar" />

</LinearLayout>

  </android.support.v7.widget.CardView>

</LinearLayout>
```

You can use `CardViews` on devices that are running Android 2.1 (API level 7) and higher by adding the `v7 cardview` library to your project. If you're using Eclipse, you'll need to add this library to your project's `libs` directory, then select **Build Path**, followed by **Add to Build Path**. Android Studio users need to add this library as a dependency in their module-level `build.gradle` file:

```
dependencies {
     compile
     'com.android.support:cardview-v7:21.0.+'
}
```

Adding a standard `CardView` to your layout is pretty straightforward, but if you want to customize Android's standard `CardView`, you can make the following changes:

- Change a card's corner radius using `cardView:cardCornerRadius`, for example, `card_view:cardCornerRadius="10dp"`. Alternatively, you can set the corner radius via your application code using the `cardView.setRadius` method.
- Change a card's background color using `card_view:cardBackgroundColor`.
- Give your card an elevation and create a shadow using `card_view:cardElevation`.

Lists and RecyclerView

Lists give you a way of presenting related data types in a consistent, easy-to-read format.

A list consists of continuous columns and rows that serve as containers for tiles. You should prioritize your most important content within each tile; imagine you're designing an e-mail app where each e-mail is represented by a tile. Typically, each tile would display the sender's name and the subject heading in a larger font, as this is the most important information, and then you'd provide a preview of the e-mail's text in a smaller font.

A typical list tile contains the following:

- **Text**: In a single list, each tile contains a single line of text. In a two-line list, each tile contains a maximum of two lines of text. If you need to display more than two lines of text, consider using a card instead. The amount of text can vary between tiles within the same list.
- **Primary actions**: These are consistent throughout every tile in the list. In our e-mail example, the primary action might be *open e-mail*, and this primary action would appear on every tile within the list.
- **Optional supplemental actions**: These usually take the form of icons or secondary text and should be placed on the right side of each tile.

You create a list using the `RecylerView`, which is a container for displaying large data sets. `RecylerViews` offers improved performance over `ListViews`, as it recycles views directly. When item views are no longer visible to the user, `RecylerView` automatically replaces their contents with a different element from the data set, resulting in smoother scrolling.

`RecylerView` also offers default animations for common operations, such as removing items from the list, and it provides layout managers to help you position the items within those lists.

`RecyclerView` provides three built-in layout managers for you to choose from:

- `LinearLayoutManager` displays items in a horizontal or vertical scrolling list
- `GridLayoutManager` displays items in a grid
- `StaggeredGridLayoutManager` displays items in a staggered grid

To use `RecylerView` in your project, you need to perform the following steps:

1. Add the `RecyclerView` support library to your project's Gradle build file (`com.android.support:recyclerview-v7:23.1.0`).
2. Define your data source.
3. Add `RecylerView` to your layout file, like you'd add a regular view:

```
<android.support.v7.widget.RecyclerView
        android:id="@+id/recylerview"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:scrollbars="vertical" />
```

4. Specify the layout manager you want to use:

```
        mLayoutManager = new
LinearLayoutManager(this);

mRecyclerView.setLayoutManager(mLayoutManager)
;
```

5. Create an adapter. To use `RecylerView`, you need to create an adapter that extends the `RecylerView.Adapter` class:

```
        public class MyRecyclerAdapter extends

RecyclerView.Adapter<MyRecyclerAdapter.ViewHol
der> {
```

6. Create `ViewHolder`. The `RecyclerView` adapter relies on the `ViewHolder` object that stores references to all your views, so you don't need to use multiple `thefindViewById()` methods.

```
public class ViewHolder extends
RecyclerView.ViewHolder {
```

7. Assign the adapter to your `RecylerView`, via the `setAdapter` method.

Animations and transitions

Two more key features of Material Design are new animations and transitions. When used correctly, these visual effects don't just look nice, they also serve two major purposes.

Reinforcing the Material Design illusion

A crucial aspect of Material Design is the sense that on-screen elements possess the same characteristics as physical objects, and animation is a powerful tool that can help you really drive this point home.

In the real world, the way objects move varies depending on their physical characteristics. By varying your animations, you can create the sense that different on-screen objects possess different physical characteristics. For example, you can suggest that an object is heavier than others by making it move more slowly. And if the user sees an object moving quickly and accelerating rapidly, they'll assume that the object is lighter.

Another fundamental aspect of Material Design is the illusion that all on-screen objects appear, disappear, and transform inside a continuous 3D space. One of the most powerful ways of making your app feel like a real, 3D environment is to create a visual continuity between your app's activities.

Traditionally, Android apps were designed as a sequence of screens, where each activity was a separate screen. Material Design seeks to blur these boundaries by using transitionsto ease the user from one activity to the next.

For example, when a new activity starts, the previous activity's elements might fade away, while elements from the new activity animate their way on screen. And if these two activities share common elements, you could animate these elements so that they appear to rearrange themselves into a new layout, which is actually an entirely new activity. In this way, you can create a more fluid and immersive user experience where users feel as though they're moving around inside your app's environment rather than switching from one screen to the next.

Tip

Entrances and exits

Put some thought into the way your objects enter and exit the screen, as these are ideal opportunities to strengthen the illusion of Material Design's continuous 3D environment. If an object enters the screen at a considerable pace, then the user will assume that this object has been traveling for some distance off screen, picking up speed along the way. If an object slows down as it exits the screen, the user will assume this object will come to a halt just off screen.

You should also look for opportunities to use these assumptions to your advantage. For example, if you have an object that you know will be making a reappearance, you could animate it so that it exits the screen at a crawl, and then edges its way back on screen when the time is right, giving the user the impression that this object has been hovering just off screen the entire time.

Providing the user with a visual feedback

Whenever the user interacts with a UI element, your app should provide them with some form of visual confirmation that it's

successfully registered their interaction.

> **Note**
>
> The most common user interaction is a touch event, so this is the type of event I'll be focusing on throughout this section. Just be aware that your app may need to handle other forms of user input, such as the user typing on a virtual keyboard or speaking commands into their device's microphone.

In Material Design, a touch ripple is the main mechanism that you'll use to provide the user with this visual confirmation.

A touch ripple effect is a particularly useful animation, as you can use it to communicate additional information about the touch event, such as the duration of the event, the amount of pressure applied, and where the touch event occurred (the touch ripple moves outward from the point of input).

You may also want to animate material so that it responds to user inputs, such as animating a photo so that it appears to lift slightly in response to the user long-pressing it.

Animations – a word of warning

Animations have the potential to be one of the most powerful or destructive tools at your disposal.

Our eyes are experts at detecting and tracking movement, but if multiple elements are moving at once, or moving in random patterns, then the user won't be able to keep up! Bad animation is far worse than no animation at all.

You should also avoid using time-consuming animations that serve no purpose other than to look pretty. Ideally, you should aim to catch your users off-guard with subtle, unexpected uses of animation that

enhance their experience in some way, rather than simply using animation for the sake of it.

The finishing touches

Now that you've created a user interface with a Material Design look and feel and added some structural elements, such as floating action buttons and cards, it's time to put the finishing touches to your Material Design application.

Designing your product icon

Your product icon should communicate your app's identity while also indicating your app's purpose and core functionality.

> ### Tip
>
> **Building your brand**
>
> If you're creating multiple apps, each product icon should be distinct, but you should also use each icon as an opportunity to create and reinforce a wider brand that spans all your Android offerings. Aim for some consistency across related product icons.

Product icons should reflect the same Material Design principles that we've explored throughout this chapter, so once again, you should take the physical qualities of paper and ink as your main inspiration. Your product icon should appear to be cut, folded, and illuminated exactly like other Material Design objects, but the real challenge is communicating all this information while also creating a product icon that's clean and simple.

Two particularly effective examples of Material Design icons are the Google Calendar and Gmail product icons. Both of these icons use simple shapes to create a sense of shadow, depth, and edges, while clearly communicating the app's main purpose and function.

Here is the Gmail icon:

And here's the Google Calendar icon:

To make designing your product icon easier, Material Design introduces the concept of **product icon anatomy**. These are standardized shapes that you can incorporate into your design to help promote a consistent look and feel across product icons. You can view these standardized shapes at https://www.google.com/design/spec/style/icons.html#icons-product-icons .

Each product icon should contain the same structural elements. These elements are always viewed straight from above, and each component is positioned at the top of the previous component:

- **Finish**: This is a soft tint that you should use to highlight the top edge of all your product icon elements. You shouldn't apply this tint to your icon's left, right, or bottom elements.

- **Material background**: This is a sheet of material that serves as your icon's background.
- **Material foreground**: This is an element that's raised above the material background, and casts a shadow across that background.
- **Shadows**: This is a soft shadow that appears around all the edges of a raised material element. This shadow should be slightly heavier along the product icon's right and bottom lines.

Note

Creating the perfect product icon

Although layering paper elements is effective for creating a sense of depth, be careful not to overcomplicate your app by adding lots of layers. You should aim for a product icon that has all the shadows, depth, and edges you'd expect from a physical object, but it should *still* deliver a simple and streamlined look.

You should supply your icon at 48dp with edges of 1dp.

System icons

In the previous section, we took an in-depth look at product icons, which are unique to your application, but what about system icons?

The good news is that the Android team has already given all the system icons a Material Design makeover. You should use these standard system icons unless you have a very, *very* good reason not to.

You can grab the entire pack from https://www.google.com/design/spec/style/icons.html#icons-system-icons .

Typography and writing

In this section, we'll look at everything you need to know about creating text that complies with Material Design principles, from general writing advice through to specific guidelines about how opaque your text should be.

Typefaces

As an Android developer, there are two main typefaces you need to know about:

- **Roboto**: Since Android 4.0, Roboto has been the standard typeface for Android
- **Noto**: Since the release of Android 2.2, Noto has been the standard typeface for all languages that are not covered by Roboto

Material Design uses both of these typefaces.

Text opacity

You can give the user visual clues about how important each piece of text is using varying degrees of opacity.

The opacity levels you should use will vary depending on the color of your text and the color of your app's background. When you're adding light text to a dark background, you should use the following:

- **Primary text**: 100% opacity
- **Secondary text**: 70% opacity
- **Hint text, disabled text, and icons**: 30% opacity

When you're adding dark text to a light background, you should use the following:

- **Primary text**: 87% opacity
- **Secondary text**: 54% opacity
- **Hint text, disabled text, and icons**: 38% opacity

When adding text to your UI, check that the color of your background doesn't make your text difficult to read. Also, be aware that too much contrast can make your text equally tricky to read; ideally, your text should maintain a contrast ratio of 7:1.

Writing guidelines

Now that you know how your text should look on the screen, it's time to turn our attention to what your text is actually *saying*.

To make sure your user interface is as user-friendly as possible, you should create text that's as follows:

- **Clear**: Your users will appreciate simple, direct language that doesn't require repeat reading. Choose your words carefully, and look for the simplest way of conveying your message; for example, it's better to say *move onto question 2* rather than *navigate to question 2*.
- **Accessible**: When writing your text, bear in mind that some people may be using your app in their second language. Even if you translate your app into other languages, culturally-specific phrases and slang may not survive the translation process. Your aim should be to write text that's accessible to *everyone*. On a related note, when you create string resources, it's generally a good idea to include detailed descriptions and perhaps even additional comments, so if your app does wind up getting translated, you stand a much greater chance of it being translated accurately.
- **Necessary**: The whole idea of text is to help the user navigate and get value from your app, so whenever you're tempted to add text to your UI, ask yourself—*does the user really need to know this?* For example, if you create a screen that consists of a form and a **Submit** button, your users will probably know what's expected of them, without you having to add a **Please complete the form and then press the Submit button** disclaimer.
- **Concise**: The text you do decide to include should be short, sweet, and to the point. In the interests of keeping text to a

minimum, use contractions wherever possible (so that's *can't* instead of *can not*).

- **Lacks punctuation**: Punctuation adds visual clutter, so you should omit punctuation wherever possible. In particular you should avoid exclamation marks. Every time you're tempted to add an exclamation point, ask yourself—*do I really want to shout this at the user?*
- **In the present tense**: Most UI events happen in the here and now, so you should write in the present tense unless you have a *really* good reason not to.

- **Use active verbs**: Make your writing more engaging by opting for active verbs over passive ones. The only exception is if the passive version is much shorter and simpler than the active version.
- **Write in the right tone:** The *right* tone is friendly, respectful, and focused firmly on the user. You should address the user directly as *you*, and avoid the temptation of lumping yourself and the user together as *we*.

Summary

In this chapter, we looked at the core principles of Material Design, including creating a UI that has that distinctive Material Design look and feel, and we covered how to incorporate some of the new structural features into our application.

Over the course of the past few chapters, we've mainly focused on the technical aspects of creating a great user interface. In the next few chapters, we'll shift focus and look at how to capture the initial spark of inspiration via sketches, wireframes, and prototypes.

Chapter 5. Turning Your Bright Idea into a Detailed Sketch

There are lots of apps in the Google Play store, and not all of them have five star ratings.

Although there are all sorts of reasons why someone might give an app a low rating and a bad review (maybe it was buggy, difficult to navigate, it was missing some functionality the user was expecting, or the UI was cluttered and confusing), there's usually one universal truth lurking behind all the negative Google Play reviews—the application delivered a bad user experience.

Mobile users are an unforgiving bunch, and in today's competitive market, it isn't enough to have an interesting idea, unusual concept, or well-thought-out features, your app has to be enjoyable to use too.

This is why taking the time to plan your app in advance is *crucial*.

Strangely, even though designers and developers have the same goal (to create amazing apps that people will love using), we tend to think of design and development as separate from one another, but if you're going to create an amazing app, then you need to know a little bit about *both*.

Regardless of whether you have a great idea for a productivity app that'll save your users a ton of time, or a puzzle app that people will happily waste hours playing, design is *everything*. Your app's design will have a massive impact on its usability, usefulness, and ultimately, its popularity; badly-designed apps are difficult to understand and not much fun to use!

Designing for mobile can be tough, but it's also one of the most exciting platforms to develop for, as it's a fast-moving industry where new hardware is being released all the time. This means there are

lots of opportunities to come up something totally unique that'll blow your users away.

This chapter will show you how to become a design-minded developer who knows how to design *and* develop amazing apps that'll have no problems racking up rave reviews on the Google Play store.

Brainstorming - taking advantage of mobile hardware

When you're designing a mobile application, don't forget that your typical Android device has lots of interesting hardware that's unique to the mobile platform. Just think of all the things your mobile app can react to because of this unique hardware: touch gestures, tilting, audio input and output, geolocation, and much more.

If you want your app to stand out from the crowd, you need to look for ways of using these hardware capabilities in new and engaging ways.

Before we start planning our app, let's take a look at some of the most interesting Android hardware capabilities that you may want to incorporate into your mobile applications.

Touch and gestures

The most common method of interacting with an app is via the device's touchscreen, so you'll definitely want to integrate gestures into your design.

One thing to keep in mind is that users do have expectations about the gestures they can use to interact with Android apps. They'll expect to be able to perform certain actions by swiping, dragging, pinching, and tapping on-screen elements. And although you want

your app to stand out for its unique use of touch and gestures, you'll need to balance this against ensuring your users intuitively know how to interact with your app.

Don't try to reinvent common gestures just for the sake of it. For example, most users will expect to be able to move a slider by tapping or dragging it—so what's the point of reinventing something that already works perfectly well?

As you'll find throughout this chapter, designing the perfect Android app is a balancing act between creating something that's unique and innovative and something that feels familiar.

GPS

Location awareness isn't just for Google Maps-style apps!

You can use GPS coordinates in lots of different ways; for example, you may want to create an app that suggests nearby events and tourist attractions based on the user's current location, tags photos with the location where a picture was taken, or broadcasts the user's coordinates to friends who may be wondering if anyone they know is nearby.

> **Note**
>
> Don't fall into the trap of thinking location awareness is only useful for getting directions. There are lots of ways you can use this hardware feature to enhance your Android apps.

Vibration

You should use vibration when audio alerts get distracting or annoying (whether that's annoying for your users or annoying for the people around them!). Although this hardware feature is easy to

overlook, take a moment to think about how your app might use vibration as an additional way of communicating with the user.

One example can be a meditation app that plays soothing music or relaxing ocean noises. Imagine you want to subtly notify the user when they've been meditating for a full 15 minutes, just in case they get carried away and lose track of time. In this scenario, an audio notification would be jarring and would completely ruin the mood. Wouldn't it be better to alert the user with a discrete vibration instead?

Audio input/output

When you think about audio input, chances are you think about talking to your friends or issuing voice commands to apps such as Google Search. When it comes to output, you probably think of audio notifications, listening to said friends, or listening to things such as music and audiobooks.

These are all valid examples, but audio is a feature with endless possibilities, so consider whether your app could use audio input and output in more creative and unusual ways.

In particular, you'll want to make heavy use of audio input and output if your app is going to be used in situations where it might be awkward or inappropriate to interact with an app via other methods. Issuing a quick voice command to our smartphone is often much more socially acceptable than tapping away at a keyboard for ages, which is often be seen as rude.

At the extreme end of the scale, audio gives your users a means of communicating with your app when other methods might be dangerous. If you're creating an app that gives driving directions, then encouraging the driver to take their eyes off the road in order to read text-based directions is *definitely* a bad idea.

Interacting with other devices

No Android device is an island! Android smartphones and tablets can connect over channels such as Bluetooth, Near Field Communication (NFC), Wi-Fi, and mobile networks, which open up a whole new world of possibilities for your app.

Even more excitingly, Android smartphones and tablets can connect with less-traditional Android devices and Google services, such as wearables, Android TVs, and beacons. Would your app provide a better user experience if it could send and receive information from additional devices?

Note

Google's beacon platform allows your app to interact with simple, low-energy devices known as beacons. Nearby beacons are useful for providing an innovative location and proximity experience to your users. They're beyond the scope of this book, but you can read about them at https://developers.google.com/beacons/ .

These are just a few of the hardware-related features you may want to consider when you're in the early stages of brainstorming your app.

If your head is currently spinning with ideas, then it's a good thing! Make a note of all the off-the-wall ideas you come up with, so you can revisit them once you've planned your app a bit more. And even if these ideas don't fit with the app you end up creating, then who knows, they may turn out to be perfect fit for your next Android project!

Tip

Don't get carried away!

When you're brainstorming all the cool and creative features that you can include in your app, the golden rule is not to try to squeeze anything into your app just for the sake of it.

Even if you've come up with an idea that you're convinced is pure genius, if it doesn't fit with your current project, then don't waste a good idea on an app that it just isn't right for. Make a note of your idea, and remember that just because you're not implementing it in your current project doesn't mean it won't find its way into your next one. This is a good rule to remember throughout the entire planning, designing, and developing process.

The difference between UX and UI

Even though UX and UI are often used interchangeably, they're not synonymous.

UX stands for **user experience**. All the code you write contributes to the user experience, whether it's UI code (such as the XML that creates a button and places it on the screen) or non-visual code, such as the code for remembering the user's address and using it to auto-fill all the forms throughout your app.

Non-visual and visual code come together to deliver your app's user experience, but if looked at in isolation, visual code *only* delivers your app's UI.

Tip

Market Research

Throughout this chapter, I'll be encouraging you to examine how other apps deliver an effective UI and UX. But as a general rule, if you ever find yourself struggling to overcome a particular design problem, it may help to flick through a few of the Android apps that you have installed on your smartphone or tablet to see how they tackle similar design challenges. If you're struggling for general inspiration, then looking at a few examples of Android applications that deliver an effective UI and UX is often the best way to kickstart your creativity.

Brainstorming your app

When you have an idea for a new Android project, the first step is to get all your thoughts down on paper.

Writing a concept

A well thought-out app solves a specific (and ideally *unique*) problem, and the best way to capture this information is by writing a clear concept.

A concept should be short and to the point, so try to express your app in a single sentence, for example, "I want to create an easy note-taking app for high school students."

Creating an ultimate features list

Now's the time to let your imagination run wild and write down all the features you'd like your app to have. This list might include features that aren't practical, but that's okay. Think of this as your dream features list of everything you'd include in your app if time, money, and technical limitations weren't an issue.

Identifying your app's primary task

The most effective apps have a clear primary task. A note-taking app might have social features, such as tagging people in notes, the ability to record videos, and support for images, but its primary task is making quick, text-based notes. Everything else is a nice-to-have added extra.

A good trick for getting to the core of what your application is all about is to write a quick explanation of your app's purpose, something that's known as a **product statement**.

Use this template for your product definition statement:

(What makes your application stand out from the crowd) (What your application is)

Some examples of product statements are as follows:

- A streamlined and simple note-taking app
- A secure and encrypted SMS app
- A beautiful lockscreen replacement app

This product statement should be at the forefront of your mind the entire time you're working on your project, so you may want to scribble it on a post-it note.

Congratulations, you've just turned your spark of inspiration into a clear statement about the kind of app you want to create. But before you progress with this idea, it's time to ask yourself some tough questions.

Is this application right for mobile?

Nowadays, we can pretty much do the same things on our mobile devices as we can on our home computers. We can send e-mails, watch YouTube videos, write lengthy text documents, edit photos, and check Facebook all on our smartphones and tablets.

Mobile devices have also taken over the roles of many other electronic and non-electronic items. We can add appointments to the Google Calendar app rather than scribbling them on a paper calendar, store all our contacts in a digital phone book rather than a physical one, and as camera phones become increasingly powerful, we no longer have to cart a bulky digital camera around in order to snap high-quality photos.

Whenever you come up with a software idea, it's tempting to automatically assume that it *has* to take the form of a mobile app. But even the most powerful mobile device has its limitations, so

always ask yourself whether your idea might actually work better on a different platform.

Imagine you had an idea for a piece of software that'd help professional photographers make detailed and complex edits to their photos. You may automatically assume that this has to be a mobile application; after all, we're all taking more photos on our camera phones than ever before. But consider your target audience; are professional photographers likely to be snapping many photos with their camera phone? Aren't they more likely to have specialist photography equipment? Also, consider whether in-depth image editing is a good fit for the small touchscreen you find on your typical mobile device.

It probably makes make more sense to develop this particular idea as a piece of software for desktop computers, perhaps with a companion mobile app that offers a handful of features that are better suited to the mobile environment. This way, your users can snap a photo on their camera phone and make a few simple tweaks on the go. Then, if they decide they want to make more complex edits, they can always boot up their computer when they get home.

Do you have the budget?

Before getting stuck into a new project, it's important to take a long hard look at your budget to see whether this particular project is realistic.

Although, the word *budget* implies money, a budget can also be time-related. If you're a solo Android enthusiast who's creating an app for fun in their spare time, then chances are that your biggest challenge will be finding spare time to devote to your project rather than sourcing funding. Estimate how much time you can realistically dedicate to your project, and then use this information to work out how long it'll take you to complete.

Be honest with yourself about what you can realistically achieve as well as how much time and money you can invest in your project.

Whether your main restriction is money or time, you may realize that this particular project isn't really practical; maybe you're too busy to complete the project within a reasonable amount of time, and you know you'll run out of steam before you reach the finishing line; or maybe your project needs a few extra hands on deck, and your budget won't stretch to employing any more people.

If this is the case, you may want to rethink your project to make it more practical. Maybe you can trim your features list, or free up more time by putting one of your hobbies on hold. Although it may be disheartening to cut back on features because of time or financial restraints, it's far less disheartening than proceeding with your project, and then a few months down the line realizing that, realistically, you're never going to reach that finish line.

Telling yourself some hard truths at this point will save you from investing time, effort, and money in something that isn't realistic.

Planning your app

Planning is a step that's easy to skip, especially if you're flying solo, but taking some time to plan your project properly not only increases your changes of creating a more effective application, it also increases your chances of actually finishing your project, rather than abandoning it halfway through.

Skip this stage at your peril!

Identifying your target audience

When you're in the early stages of planning your app, it's tempting to tell yourself, *"This app is such a good idea, everyone will want to use it!"*

If you buy into this statement, then your app will be perfect—but only for you. Since it doesn't make much sense to develop an app just for yourself (although, if you happen to like the finished product, then that's an added bonus), the first question you'll need to answer when planning your application is, *"Who am I building this thing for?"*

Too often we leave identifying our target audience until too late in the planning process, or even worse, we get carried away and start building an app without really knowing who we're building it *for*. Your target audience should influence every part of your app: from its look and feel to the features you want to include, and even how you market and promote the finished product.

Hopefully, you already have a rough idea of who you're targeting, but to ensure that you create something that people actually want to use, you'll need to develop a deeper understanding of who exactly you're targeting. One way to do that is by developing personas and use cases:

- **User personas**: This is where you create a specific model of the kind of person you're creating your app for. A persona is an individual user who represents your target audience. To encourage you to think of this persona as a real person, you should give your user persona a name.
- **Use cases**: This is a scenario of how, when, and where your persona might use your app.

Creating a persona

To help you create a detailed user persona, ask your persona the following questions:

- How old are you? This could be an exact age such as 18-25 year olds, or it could be more general, such as children or young adults.
- What are your interests and hobbies? Do you like football? Baking? Shopping? Working out?
- Do you identify as male or female (although not all applications will target a specific sex)?
- Where do you live? This could be a specific country, city, or a type of area such as the countryside or by the sea.
- Do you have children?
- Do you have a job? And if so, what do you do?
- Are you currently in school, college, or university?
- Do you have any particular areas of expertise?
- Why type of app do you use most often?
- What other types of apps do you use?
- What kind of application do you *never* use?
- What influences you to make decisions?
- How experienced are you with Android? Are you a pro user, or do you know just enough to get by?
- What resonates with you on an emotional level? What makes you angry, happy, or sad?

Note

This is by no means an exhaustive list, but these questions should be enough to get you thinking about the kind of person you're targeting in more detail.

Let's look at an example of how a user persona can help you plan an application. Imagine you have an idea for a recipe app and have brainstormed a list of all the features you might want to include:

- The ability to filter recipes based on specific dietary requirements (gluten-free, vegan, vegetarian, low fat, non-dairy), how many people they feed, and how easy they are to cook.
- An estimated calorie count for each meal.
- An estimated cost for each meal.
- Social media integration, so users can share their favorite recipes with friends and family.
- The ability to leave a range of feedback on each recipe; whether that's posting a comment, uploading a photo of their attempt, or giving the recipe a star rating.
- An online scrapbook where users can save their favorite recipes.
- The ability to purchase ingredients through the application.
- Video tutorials so the user has the option to read the instructions or follow along with the video.
- A built-in egg timer.

The more ideas you jot down, the more you'll realize that even if all your ideas are well thought out, practical, and would add real value to your app; it simply isn't feasible to try and incorporate every single idea into a single application.

Not only would this be a nightmare for you in terms of the time and effort required to implement so many features, but it'd also be a nightmare for every user who opens your app and is immediately confronted by an exhaustive, incoherent list of features—and probably a cluttered and confusing UI to boot!

You should always aim to design the perfect app for a very specific audience rather than a generic app that tries to appeal to everyone; chances are that this kind of app is going to end up pleasing *no one*.

So, how do you decide which features from your initial list should make the cut? The answer is to create a user persona and then cherry-pick all the features that'd appeal to this particular persona.

Let's create a user persona for our recipe app. Imagine our general target audience are university students who are facing the challenge of cooking for themselves for the first time. To help us bring this target audience to life, let's create a single-named persona—meet Nicole:

- She's 18
- She's a university student
- She lives in student halls with five other students
- She doesn't have a car
- This is the first time she's lived outside of the family home, so she has very little experience of cooking for herself
- She's very money-conscious

The mobile persona

When you're designing a mobile app, you also need to take the **mobile persona** into account. This is a set of additional characteristics that people assume when they're using a mobile app.

If you're one of the many Facebook users who likes to keep track of your friends and family on the go via the Facebook mobile app, then think about how the way you use the mobile app differs from how you use Facebook on your laptop or computer. Chances are that you automatically avoid activities that you know will drain your device's battery or eat into your data plan, such as watching or uploading videos. But this probably isn't something you worry too much about when you're accessing Facebook on your desktop.

When designing your mobile app, bear in mind that your typical mobile persona has the following characteristics:

- A finite data plan.
- A limited amount of battery.
- Usually on the go, so they may have inconsistent Internet access and mobile network signal.

- Rarely gives their Android device their undivided attention. Mobile users are some of the world's best multitaskers, who glance at their device while they're at the pub waiting for their friends, as they're killing time at the bus stop, or while they're laid on the sofa half-watching TV. Your typical mobile user doesn't want to be staring at your app's UI for any length of time; and if they do so, then it is usually a surefire sign that you've done something wrong and the user is trying to work out what they're supposed to do next.
- Viewing your app on the small screen. Even your largest Android tablet has a *much* smaller screen than other electronic devices, such as televisions and laptops.
- Feels an emotional connection to their Android device. The way we relate to our smartphones, in particular, is very different to the way we relate to other electronic devices such as computers. Our smartphones are always by our side—whether they're in our hand, pocket, bag, or on our desk at work, and we use them for very personal tasks, such as keeping in touch with family and friends and snapping photos of our wild nights out and vacations. We store a *lot* of sensitive information on our mobile devices, so it's only natural that we feel protective of them! Not convinced about the special relationship we all have with our phones? Just think about that moment of blind panic you feel when you reach into your pocket or bag for your phone, and it isn't there. This is a feeling we've all experienced at some point (although hopefully, your mobile turned out to be in a different coat pocket, and it wasn't really lost).

If you disregard any of these attributes—for example, you develop a mobile app that gobbles up mobile data, or requests unnecessary access to lots of sensitive information—then your users are going to have a bad time. Your challenge is to create a compelling user experience for your specific target audience while still respecting all the characteristics of the mobile persona.

Creating use cases

When might Nicole use our recipe app? What prompted her to launch your application? And what is she hoping to achieve by interacting with your app?

Here are a few use cases for Nicole:

- It's the weekend, but it's also the last few days before Nicole's student loan comes through, so her usual weekend takeaway treat is off the cards. She needs to cook something that's every bit as tasty as her usual Friday night Chinese takeaway but is within her current budget.
- To save money, Nicole and her flatmates have decided to take it in turns to cook a communal meal. Today, it's Nicole's turn, so she needs a recipe that'll feed six hungry students—bonus points if there are leftovers!
- Nicole's in a rush; she got up late and has just over an hour before she needs to be at her afternoon lecture. She needs a recipe that's super easy and super quick to make.

Deciding on a feature list

Take another look at the initial features list for our recipe app. Now, we have a very specific persona and a few use cases; which features should we cherry-pick to include in our app? What features are going to appeal to Nicole?

I'm going to select the following:

- **The ability to filter recipes based on how easy they are to cook**: This is a top priority for our first-time chef, particularly when she needs to hurry up and eat before her next lecture.
- **The ability to filter recipes based on how many people they feed**: If Nicole is going to be cooking for her flat mates, then she needs easy access to recipes that'll feed six people.
- **An estimated cost for each meal**: As a budget-conscious student, Nicole doesn't want to blow all her student loan on fancy ingredients. She wants to be able to see *exactly* how much it'll cost her to cook each meal—particularly when it's the last few days before her next student loan installment comes through!
- **Video tutorials**: Nicole is an inexperienced cook, so she'd definitely appreciate the option to watch meals being prepared, rather than just reading instructions. She's also 18, which means she's grown up watching video tutorials on YouTube, so the thought of cooking along to a video on her smartphone has definite appeal.
- **A built-in egg timer**: This is the first time Nicole has lived outside of the family home, so she hasn't had the time to build up a stash of cooking utensils yet. She also doesn't want to waste her student loan on boring things such as egg timers, so if your app can provide this kind of functionality, then all the better!
- **The option to purchase ingredients through the app**: As an 18-year old, Nicole is an expert at making online purchases, so getting the ingredients she needs delivered direct to her door is high on her (Amazon) wishlist. Nicole also doesn't have a car, so getting to her supermarket or local farmer's market isn't the easiest thing in the world. Ideally, our recipe app will be able to cross-check several vendors and suggest the place where she can purchase each ingredient at the lowest cost, to save even more of her student loan.

There are other features in our list that might also appeal to Nicole; for example, she could be a vegan or have a few tried-and-tested family recipes she'd like to upload to the app, but be wary of getting *too* specific with your user personas.

While the ability to filter recipes based on dietary requirements, or giving users the option of adding their own recipes to the app would add value for certain users, ask yourself whether they're relevant for your target audience and not *just* your user persona? If you're unsure, then try adding the features to your list and reading it from top to bottom, do these new features *fit*? Or do they stick out like a sore thumb?

> **Note**
>
> Even if you decide against adding certain features to your app now, they may still find their way into subsequent releases. If you feel like a feature has potential but isn't absolutely crucial for your app at this early stage, then make a note of it; you can always return to it at a later date.

Taking a critical look at your target audience

You've identified your target audience, but before we progress any further take a step back and have a critical look at your chosen audience. Does it *really* make sense to target this specific group of people?

Does your target audience need a mobile app?

Is your app offering your target audience anything they're not getting already via a different platform?

Say, you had an idea for an app that keeps track of the user's appointments; might they already be getting this functionality elsewhere? Increasingly, places such as dentists, doctors, and hairdressers are reminding clients of upcoming appointments via SMS messages, or if you're targeting less tech-savvy users, they might prefer scribbling their schedule on a paper calendar.

Even though it's tempting to assume that *everything* is better as a mobile app, your target audience is unlikely to make the effort to download a mobile app if their needs are already being met via a different medium.

Is your target audience on Android?

Gone are the days when serious business users opted for Blackberrys and all the cool kids went with iPhones. Nowadays, the lines are blurred, and it's becoming common practice to release an app across multiple platforms. But before you jump to conclusions, check whether there's evidence to suggest that your target audience favors one mobile platform over another. Perform a quick Google search for any recent research into mobile demographics, and see whether there's any data out there that raises red flags about your planned Android app.

Okay, so your target audience is on Android, but does it own a suitable Android device?

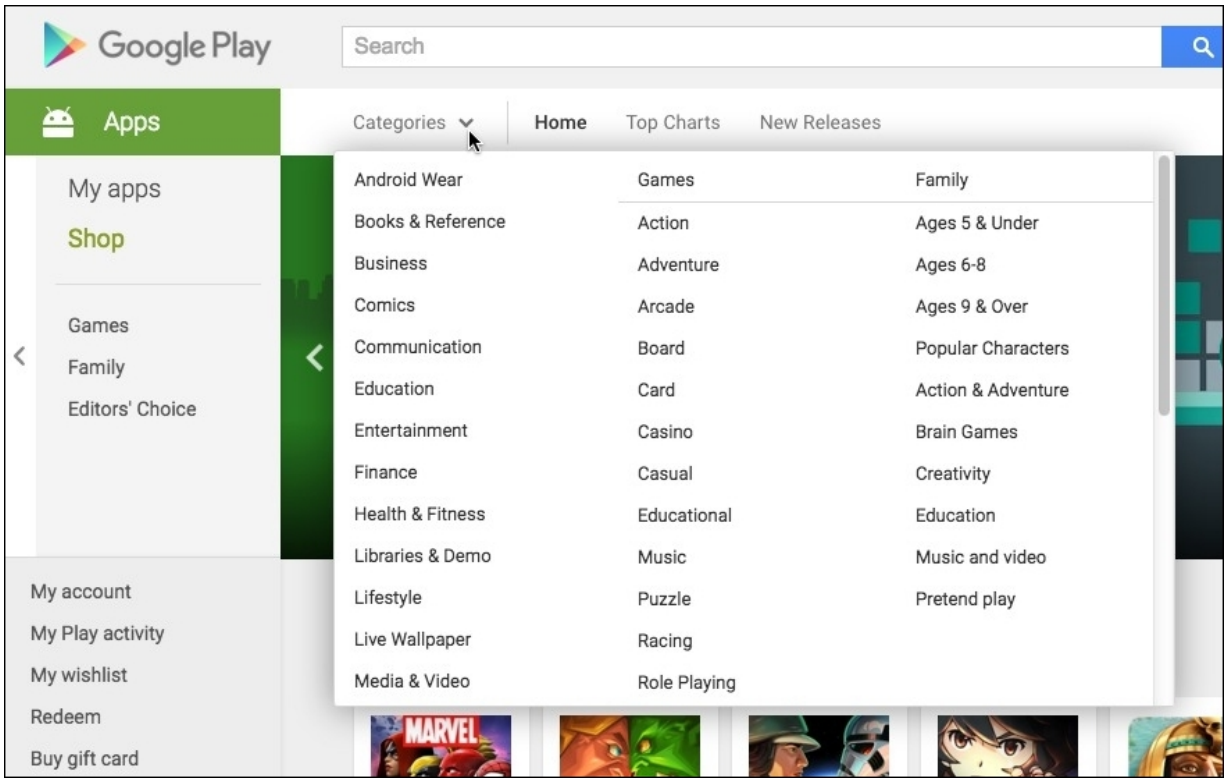
Don't assume that your target audience automatically has access to an Android device that's powerful enough to run your application, particularly since Android is such a fragmented platform.

If you design an app that relies on the user having access to the latest version of the Android operating system, then you're already limiting your audience. This may make sense if you're developing an app for advanced Android users who are more likely to have the latest version of Android installed on their device, but it might not be such a good idea if you're targeting general, non-pro Android users, older users, or people who are on tighter budgets who perhaps haven't invested in the latest Android device, such as money-conscious students.

Do they use this "type" of application?

Even though your app will have a unique combination of target audience, features, UI, and UX, there are still identifiable *types* of

applications. It's worth noting that these types don't correspond to the categories in the Google Play store, which is much more specific.



Application *types* can be grouped into four categories:

- **Useful tools**: These include weather, news, travel, and productivity apps
- **Fun tools**: Examples include wallpapers, social media, and image editing and photography apps
- **Games**: This is pretty self-explanatory and Google Play has a dedicated games category, which makes identifying these kind of apps even easier
- **Entertainment**: This includes music, video, drawing, audio books, podcasts, and comic book apps

There are some crossover and grey areas between categories (for example, is a shopping app a fun tool or a useful tool?), but the more time you spend planning your app and identifying your target audience, the more obvious it'll become which category your app falls into. Our recipe app is a useful tool as it helps students to cook

for themselves, potentially for the first time. However if our app contained lots of Christmas-themed recipes and targeted experienced cooks who love baking festive treats, then it's more likely to fall into the fun tool category.

Identify what type of app you're planning, and identify whether your target audience is likely to use this type of app.
Would your target audience be willing to pay for this application?

You may choose to monetize your app in some way. Popular methods include requiring the user to pay a one-time fee before they can download your app; releasing free and paid-for version of the same app; or what sales folk refer to as *upsell opportunities*, which may include unlocking special *pro* features or in-app extras, such as new levels, characters, in-game currency, or extra lives. You may even choose to offer monthly or annual subscriptions.

If you do choose to monetize your app, then consider what's the most appropriate method for the type of app you're developing for your target audience. For some people on tighter budgets, paying for a mobile app may feel like a waste of money, or a younger target audience may simply not have access to things such as credit and debit cards.

You should also consider your app's purpose. Part of our recipe app's appeal is that it can save students' money by teaching them how to cook for themselves, rather than relying on takeaways and eating out, so asking them to pay to download our app doesn't really make a lot of sense. What student wants to spend money on an app that's supposed to be *saving* him/her money?

Another option is to release your app for free, but ask for an optional donation to support your work as an Android developer. This is a popular choice with developers who create apps for more advanced Android users, or users who may also be developers themselves, as they're more likely to appreciate all the hard work that goes into creating a great Android app.

Another way of monetizing your app is by including adverts, but if you do go down the advertisement route, then make sure they're as unobtrusive as possible. In our recipe app example, adverts may be a viable option, as it doesn't require our cash-conscious students to fork out any money directly.

Note

Including adverts in your app

If you do decide to include adverts, then this opens up an additional revenue stream: offering to remove the adverts if the user updates to the pro version of your app. However, if you're targeting more advanced mobile users, then you should consider how likely your target audience are to use things such as ad blockers. In this scenario, you might get a better response by requesting an optional donation to support your work as an Android developer.

If you do decide to monetize your app, it should always reflect the value your app is offering to your target audience. If your users do decide to invest money in your app, then you should deliver something that's worth their hard-earned cash.

Are there enough people in your target audience for your project to be profitable?

While it's good to be specific, your app has to appeal to enough people to make your project profitable.

Note

A profitable project isn't necessarily one that earns you money (although that's always nice!) What does success mean to you?

It might mean *X* number of people downloading your app, or getting a 3 star+ rating on Google Play.

We've decided that our recipe app will target students who are inexperienced cooks and who want access to recipes that are quick, easy, and cheap to make. We could narrow our demographic further by targeting health-conscious students who are interested in recipes that are quick, easy, and cheap but also low-calorie and full of vitamins and minerals. While health-conscious students definitely do exist, this is more of a niche target audience, and if you do decide to go down this route, you should perform some additional market research to find out whether there's actually a significant market for this very specific kind of app.

Hopefully, after scrutinizing your target audience, you'll come to the conclusion that your target audience *does* desperately need the kind of app you have in mind. If not, then this is the perfect time to go back to the start of this section and explore whether your initial idea might be better suited to a different target audience.

Tip

Time to get real

For the purposes of this chapter, we created a quick and generalized user persona and some possible use cases. However, when you're creating a real-life Android app, you'll typically want to do some form of market research, which means connecting with the people you're creating your app for and getting their opinion on your project.

The good news is that if standing in the middle of your local high street with a clipboard is your idea of hell (ditto), then the Internet is an ideal place for market research. Places such as forums and social media can help you connect with the people who might one day be using your app, or you may want to try reaching out

to relevant bodies to see whether they're willing to put you in touch with your target audience. In our recipe app example, you could try contacting local universities to see whether they'll help you out with your market research—particularly since you're developing a free app that'll help students overcome the challenge of cooking for themselves for the first time.

There's no substitute for actual market research, so you should always make the effort to get in contact with your target audience, as their input will be invaluable for helping you create realistic personas and use cases, and it will also help you identify what features your real-life target audience would like to see in your app.

Identifying user and product goals

Goals are one of those things that are so obvious that they're actually easy to overlook. Why are you releasing this app? What's in it for you? And why would someone download your app? These can be difficult questions to answer, but they can also be crucial for the rest of the planning process.

When you have a set of clearly defined goals, you can refer back to them whenever you're struggling to answer a design-related question. Should you include a *share to Facebook* feature? To find the answer, just consult your goals—how would this feature help your users achieve *their* goals, or help you achieve *your* goals?

There are two types of goal you need to identify.

Product goals

It isn't all about the user! Creating an amazing app takes time, effort, and maybe even money, and even if you're creating an app for fun, you should still get something out of it! So, what do you hope to achieve by creating this amazing app?

Your product goals might include the following:

- The number of downloads you want to reach
- The number of reviews you want to get on Google Play
- A minimum star rating on Google Play
- Spreading the word about your company, products, or services
- A financial reward, which may be generated through users upgrading to the pro version of your app, in-app purchases, adverts, or the additional revenue stream of your choice

User goals

These are the goals of the people who choose to download and use your app. What are they hoping to achieve? In our recipe app example, one of the main user goals is finding pocket-friendly recipes that don't break the bank.

At this point, you don't need to define *how* these goals should be accomplished, as there may be multiple ways of achieving the same goal. For instance, we might add a search box to our app's home screen so that the user can filter recipes based on how much the ingredients cost; or you might decide to divide your recipes into categories and then display these categories as tabs along the top of the user's screen—**Recipes under £1 per head**; **Recipes under £2 per head**, and so on.

For now, concentrate on identifying user goals and leave defining the features needed to accomplish these goals for a later date.

Sometimes, zeroing in on meaningful user goals might not be straightforward. Maybe you can only think of broad, generalized goals that aren't particularly helpful. For example, imagine you have an idea for a document-editing app you're convinced could be the next Google Drive, but the only goal you can come up with is "I want to be able to create and edit documents." That's not very specific! The key is to take this broad, generalized goal and break it down. How will the user create these documents? What edits might they

want to make? And what might they want to do with the finished documents? Brainstorm your answers.

After a brainstorming session, you should have a list of more specific user goals. In our document-editing app, these might be "I want to be able to..."

- Quickly and easily create a new document, so I can jot down notes on the go
- Make my documents more visually appealing using a range of formatting techniques
- Share my documents with others via social media
- Collaborate on documents with friends and colleagues

Creating a roadmap

A roadmap is the key to getting your app from an initial release to the version you had in mind when you created your ultimate features list at the start of this chapter. In fact, now's the perfect time to take a long, hard look at this list and decide what features should make it into the first version of your app.

This initial version should include all the features that are necessary to fulfill your app's primary task (imagine releasing a camera app with a bunch of filters but without the functionality needed for taking photos!). It should also include other high-priority features, which will usually be the features that cropped up time and time again during your market research or when you were creating your user personas and use cases.

If you're unsure whether a feature is important enough to make it into the first version of your app, ask yourself, "Is my app usable without this feature?" The first version of your app should include must-have features *only.*

Note

We're talking about the initial *version* of your app and not the initial *release*. This is because you don't necessarily have to release the initial version to the general public.

After all the work we've done identifying our target audience, some of the items in your ultimate features list may no longer be relevant, so now's the time to take another look at our list and strike out any unnecessary features.

You should be left with a list of must-have features that your app can't function without, plus some nice-to-have extras that aren't absolutely necessary but would add value to your app. The latter are the features that are going to form the basis of your roadmap.

At this point, a roadmap is more of a guideline than a strict schedule, but creating a rough roadmap helps you visualize how your project may evolve over time. It also gives you a rough idea of when you might be ready to release different features.

To create your roadmap, take your list of nice-to-have features and look for relationships between them. For example, you might group together all features that relate to social media or all the features that allow users to upload their own content.

You'll typically want to release related features together, so once you've assembled your groups, you should rank them in their order of importance. This is the order in which you'll release these features, so the next step is assigning these groups to different versions of your app; for example, if you decided that social media features are more important than the ability to upload new content, you may want to release these features in version 2.0, while the latter may form the basis of version 3.0.

Tip

Release little and often

Don't be tempted to try and squeeze lots of features into a single release. It's far better to release little and often, rather than including a ton of new features in each update but leaving your users waiting months and months between each release.

Your typical Android user will quickly forget about an app that's rarely updated. And even if they don't forget about your app, they may still struggle to get to grips with an update that introduces too many changes.

Releasing little and often also gives you more control over the development process and will make it easier to meet those deadlines. Massive releases are much harder to plan correctly and are notoriously stressful, so do yourself and your users a favor and create a roadmap that includes lots of small, frequent releases.

You should also try and add deadlines to your roadmap, even if they're speculative deadlines, as they encourage you to start thinking about how long each feature will take to develop. These deadlines could be vague, such as "in February, I want to release version 1.0," or "I want to release version 2.0 four weeks after 1.0." Alternatively, they might correspond with annual events; for example, if you've developing a shopping app, you may decide to release this app in the run-up to Christmas or during the height of the January sales.

At the very least, you should set a deadline for when you hope to complete the first version of your app.

Some final things to think about

You've nearly finished planning your app! Before we move onto the design stage, there are a few final bits and bobs to think about.

What devices should you support?

What types of devices should you support? And what versions of Android should your app be compatible with? These are tough questions, and unfortunately, you're going to have to tackle them *every time* you start a new Android project.

Considering that an Android app will run on any device unless you specifically set it not to, it may be tempting to think, "I'll just wait and see what devices my apps runs okay on", but this is a surefire way of winding up with an app that works well on your typical Android handset but poorly on everything else.

It's up to you to provide an experience that's optimized for all kinds of tablets, phablets, and Android smartphones held in landscape and portrait mode. Creating an app that functions across a range of devices isn't easy, but the more effort you put into planning how your app will support different devices and screen configurations, the easier this will be to implement. Unless you have a very good reason not to, then now is the time to start thinking about how you're going to support as many different Android devices as possible.

If you do decide not to support certain devices, then your users will appreciate being told upfront rather than waiting until they launch your app for the first time and discover that it's completely broken on their device. At the very least, be sure to give your users a heads-up about unsupported devices in your app's Google Play description.

How should you market your app?

Most of the time, the people you're marketing your app *to* will be the same as your target audience, but this might not *always* be the case. For example, if you're developing an educational app to help young children learn their alphabet, this app might be more appealing to parents rather than your target users (and in this instance, children who are still learning to read are unlikely to be scouring Google Play, looking for apps to download).

Designing your application

At this point, you should have a detailed plan about the application want to create. You've completed the following:

- A written concept
- An ultimate features list
- Identified your app's primary task
- Written a product definition statement
- Identified your target audience by creating a user persona and some use cases, and then incorporated this information into your plan
- Started to think about some tricky subjects including monetization, market research, and marketing your app

Now, it's time to begin exploring how your app should look. The idea of designing your application's UI may seem daunting (particularly if you don't have any design experience), but putting extra effort into UI design can transform a good app into a *great* app; and don't we all want to create great applications?

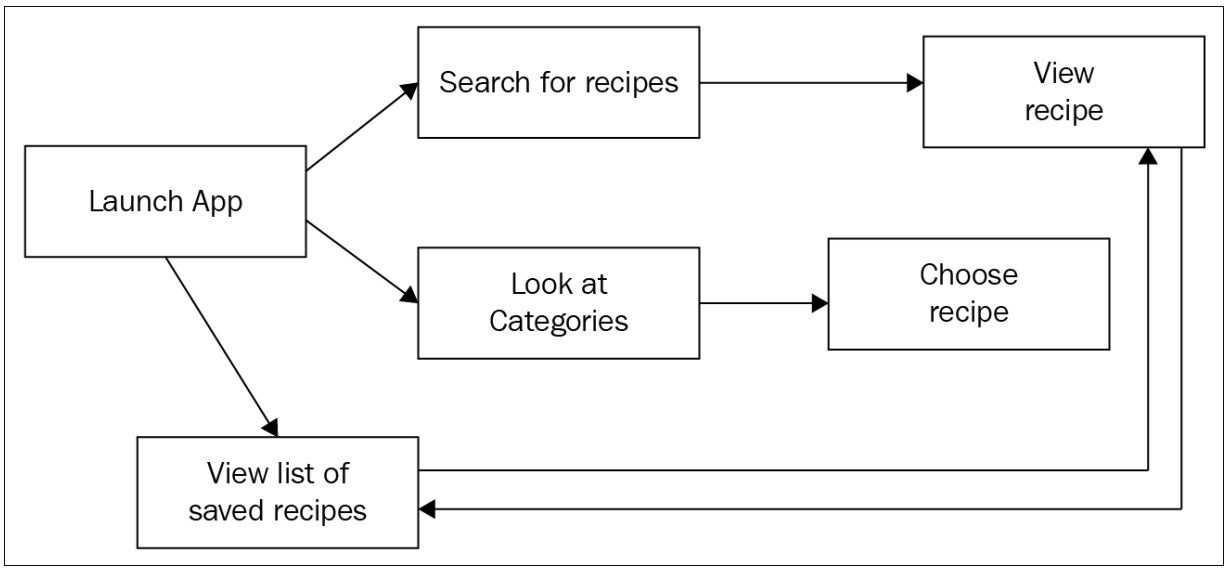
In the final section of this chapter, I'll show you how to take these first steps to designing an effective and appealing UI.

The high-level flow

The first stage of working out your app's structure is creating a high-level flow. This is where you draw a basic plan of the different routes the user will take through your app in order to accomplish different tasks.

As you might have already guessed from the name, the most popular way of expressing an app's high-level flow is with a flowchart. How you create this flowchart is up to you. You may want to create a digital chart using your favorite image-editing software, or

you might go old school with paper and pencil. Pick whichever method works the best for you.



You'll typically represent screens with shapes, and express navigation using lines or arrows. You may also want to number the screens in your flowchart, particularly if you're plotting out a large number of screens. This way, when it's time to create a wireframe, you can number the screens in your wireframe to correspond with the screens in your flowchart.

It's never too early to start looking for ways to improve the user experience, and one of the easiest ways of gauging your app's UX at this early stage is to look at how many interactions the user has to perform in order to complete different tasks.

Your flowchart is *ideal* for identifying any opportunities to minimize or simplify the number of steps required to complete each task, whether that means removing screens, reordering screens, or changing your app's navigation. For example, you may decide to include a menu on Screen A, so the user can jump directly to Screen E and Screen F.

You should also identify which screen should be your app's main or home screen. A home screen should help users complete your app's primary task. Since our recipe app is designed to help users locate

recipes that are quick, cheap, and easy to make, we may decide to select the **Search for recipes** screen as our home screen.

This flowchart is also the perfect opportunity to identify any areas where you might overwhelm the user with too many options. If you're going to include a search box on the home screen, is it wise to display all the user's saved recipes, plus the app's main **Settings** on that home screen too? You may decide to move some of this content to a different screen, or temporarily hide it in a side-menu or separate tab.

Alternatively, you may decide that if you design your app carefully enough, you can get away with displaying all this content on a single screen without overwhelming the user. If you opt for the latter, then make a note that you should devote some extra time to this screen, just to ensure the user experience really is as straightforward as possible.

Tip

Be a critic

While working on your high-level workflow, it may help to take a critical look at how other apps handle the challenge of getting users from the first screen to their desired location with as little friction as possible.

Take a look through the apps installed on your Android device, and if there's an app you particularly like, make a quick flowchart for this app. Try to identify why this flow works so well. It may also help to sketch the flow of an app you're not so keen on, and then try to work out why this flow doesn't work for you. These exercises will help you get into the habit of being critical about application flow, which is essential if you're going to effectively analyze and refine your own app's flowchart.

Creating a screen list

Once you've created a high-level flow, make a list of all the screens you'll need to create if the user is going to be able to accomplish their various tasks and goals.

The screen list for our recipe app will look something like this:

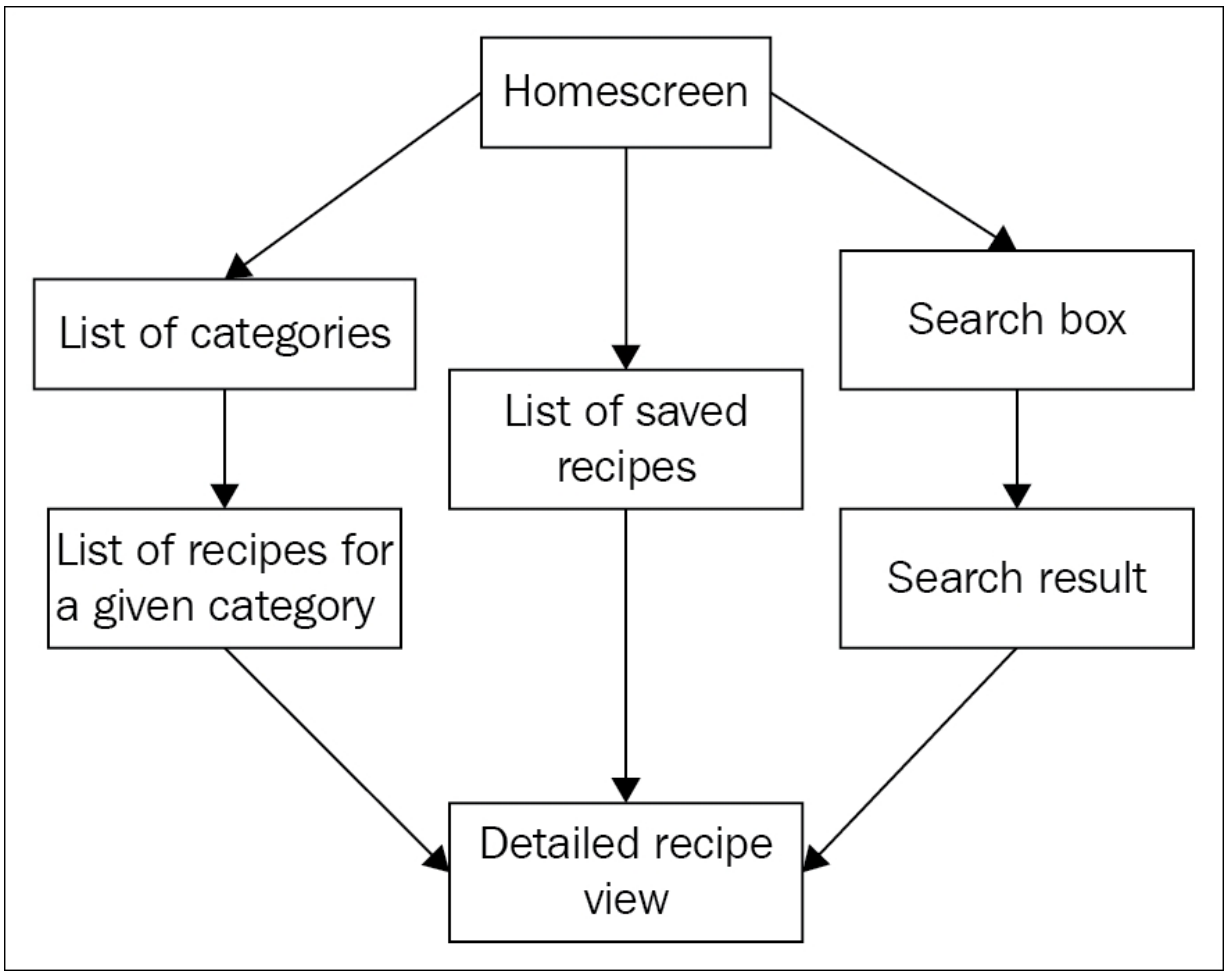
- Home screen
- A list of categories
- A list of recipes for a given category
- A list of saved recipes
- Detailed recipe view
- Search box
- Search results

Creating a screen map

Now it's time to combine our flowchart and screen list, and create a screen map that expresses the navigational relationships between all the different screens that make up our app.

If in doubt, refer back to your flowchart and review the different paths the user might take through your app.

Here is an example of how the screen map for our recipe app might look:



Note

One screen doesn't necessarily equate to one activity. One screen might also equate to one fragment that's displayed as a part of a multi-pane layout.

Grouping screens into multi-pane layouts

We didn't design our screen map with any particular Android device in mind, but Android apps must be flexible enough to adapt to a wide range of different devices, from small handsets right through to large tablets.

To present content to your users in the most effective way, you may want to group related screens together. You can then present these **fragments** in different combinations, depending on the user's device and screen configuration.

Smaller screens are generally only suitable for displaying one pane of content at a time, so each screen in our map will typically equate to one screen on a smaller device.

However, sometimes they'll be more on-screen space available, for example, when a user is viewing your app on a tablet, phablet, or even on an Android smartphone held in landscape mode. When there's more space available, your app should take full advantage, and one way to do this is using fragments.

Fragments help you create an app that can adapt across a range of different device types, screen sizes, and screen orientations. Combining screens into multi-pane layouts also helps to minimize the number of interactions users have to perform in order to complete each task.

How might we group the different screens in our recipe app, so we can effectively display multiple fragments in a multi-pane layout *and* a single fragment in a single-pane layout?

On a smaller screen, we might want to display the recipe app's home screen as a single, standalone fragment. However, if the user is viewing our app on a larger screen, then what other information might it make sense to display alongside the home screen fragment?

Let's run through all the tasks the user might want to perform when they're located on the home screen. They might want to perform the following:

- Access a list of categories
- Access their saved recipes
- Perform a search
- Access a recipe

Allowing the user to perform any of these tasks from the home screen would add value, but I'm going to go with *Perform a search* as I feel this is the most common task users will want to perform in our recipe app. So, on a smaller screen, I want the user to see the home screen; but on a larger device, I want the user to see the home screen fragment *and* the search fragment in a multi-pane layout.

Work your way through your screen map, and look for any additional ways of grouping your screens into multi-pane layouts for users accessing your app on larger devices.

> ### Note
>
> When you're displaying multiple panes of content, the accepted convention is to arrange your content from left to right in increasing detailed order.

Navigation

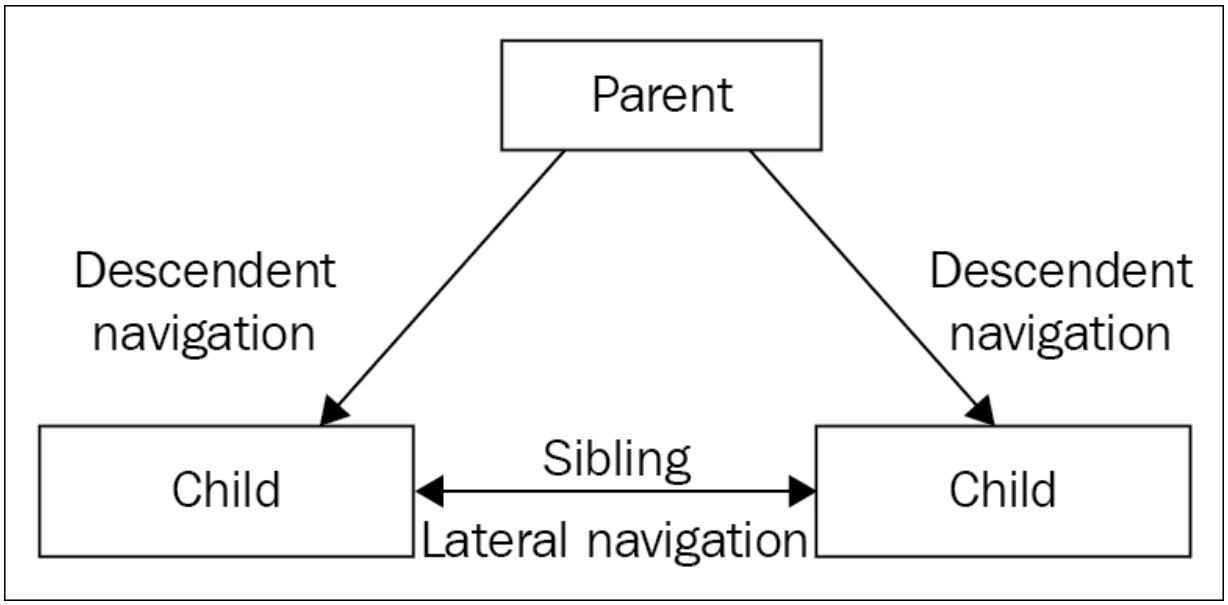
Your app's navigation should feel intuitive and effortless, to the point where even new users should be able to navigate around your app with ease. Since you've already created a screen map, determining your app's navigation should be a doddle.

It should be easy to navigate to the most important destinations within your app; for example, in our recipe app, the home screen will be the most prominent and accessible screen.

When you prioritize a screen by placing it higher in the navigational hierarchy, this screen is called a **parent**. The less important screens, which you typically place beneath a parent screen, are referred to as **children**. Continuing this theme, screens that have the same parent are known as siblings, and they usually have equal priority. You may also hear the term *collections*, which is sometimes used to refer to multiple screens that share the same parent.

Navigation can be broken into the following:

- **Descendent navigation**: When the user navigates deeper into your application, they're descending from a higher level of hierarchy to a lower level of hierarchy. Moving from a parent to a child screen is an example of descending navigation.
- **Lateral navigation**: This is where the user moves between sibling screens.



To help you determine the type of navigation that best suits your app, ask yourself the following.

What are your app's most important tasks?

Identify all the tasks users might want to perform, and then assign a priority level to each of these tasks. You should make it easy for users to accomplish high-priority tasks by featuring them prominently in your app's navigation hierarchy. For instance, if you're designing a music app, you should make it easy for users to play music (a high-priority task) while lower-priority tasks, such as being able to tweet a link to the song you're currently listening to, can be placed lower in the navigational hierarchy.

Are there any tasks you can group together?

You should create some form of navigational relationship between related features; for example, you could include them in the same menu, in the same screen, or in a row of tabs within the same parent screen.

In our music app example, being able to search for more albums by a particular artist, taking a peek at this week's top 10 tracks, browsing new releases, and checking what tracks your friends are currently listening to are all features that help you discover new music, so you might decide to place them together in your app's navigational structure.

Is my navigation consistent?

Consistency is crucial. Even if you design your navigation to be as simple as possible, if the navigation varies from one screen to the next, then you're going to end up with some seriously frustrated and confused users.

Common navigation patterns

When planning your app's navigation, you should focus on how to best meet the needs of your target audience and the tasks they'll want to perform in your application. However, there are some common navigation patterns that can give you a head start in crafting a unique navigation structure that suits your particular application.

The embedded navigation

If your app has a simple structure, you may choose to embed navigation inside your content. The major drawback is that embedding navigation in this way does reduce the space available to display your application's content, so it may not be ideal for devices with smaller screens.

Buttons and simple targets

One of the most straightforward navigation patterns is to include touchable targets, such as buttons in your app's parent screen. When the user taps one of these targets, a child screen opens, which may contain further touchable targets.

This kind of navigation is easy to understand, but be aware that including lots of touchable targets on a single screen can make your app difficult to navigate on smaller devices.

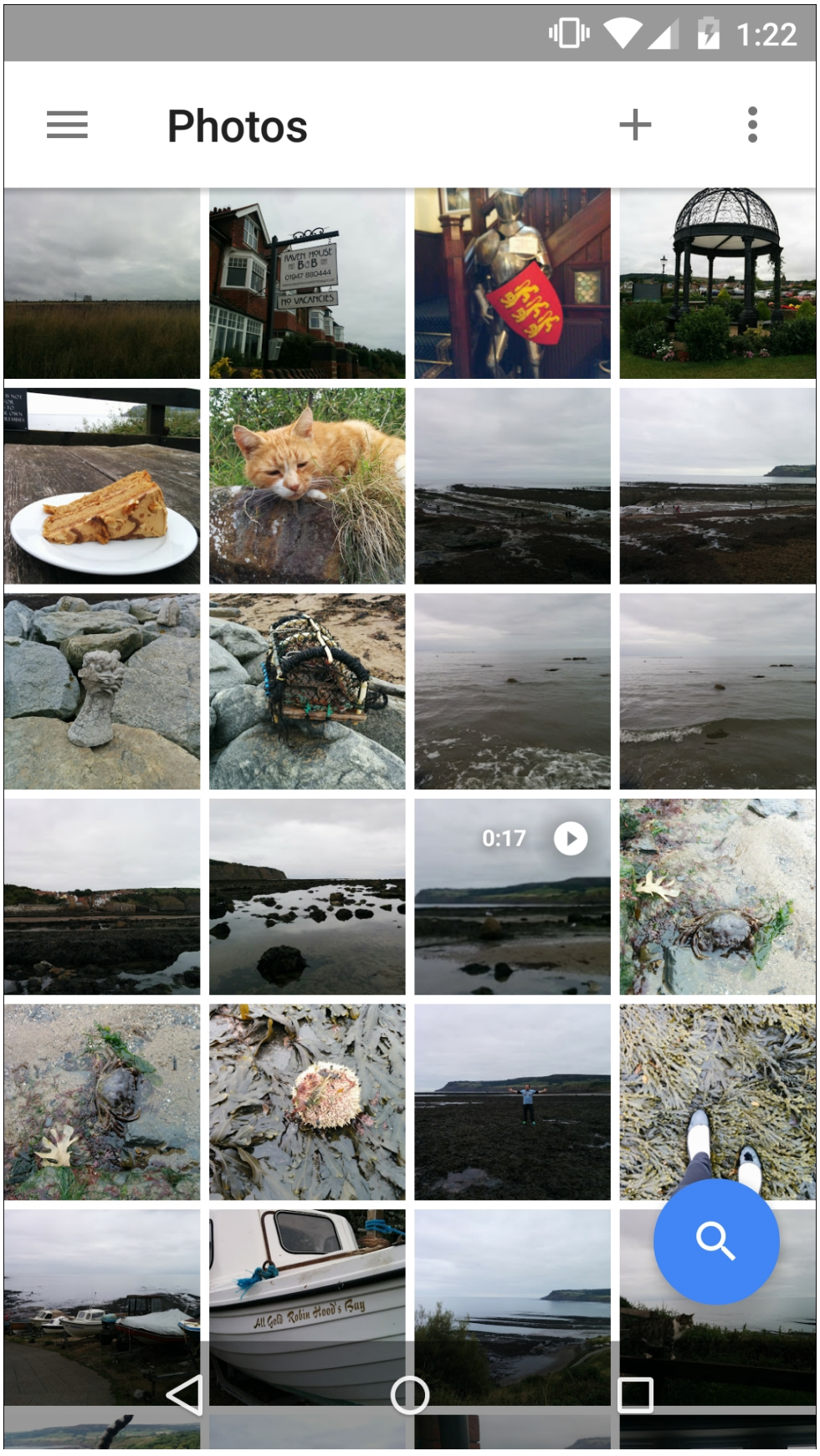
Lists

If you need to display lots of text, then vertically scrolling lists (such as the classic, text-based menu) can provide a straightforward way of navigating your app.

Just be wary of creating a navigational hierarchy where lists lead to more lists, as this type of navigation can cause the number of touches required to complete each task to spiral out of control.

Grids, cards, and carousels

When you need to display lots of visual content, such as videos or photos, you may want to use vertically scrolling grids of items, horizontally scrolling carousels, or cards:

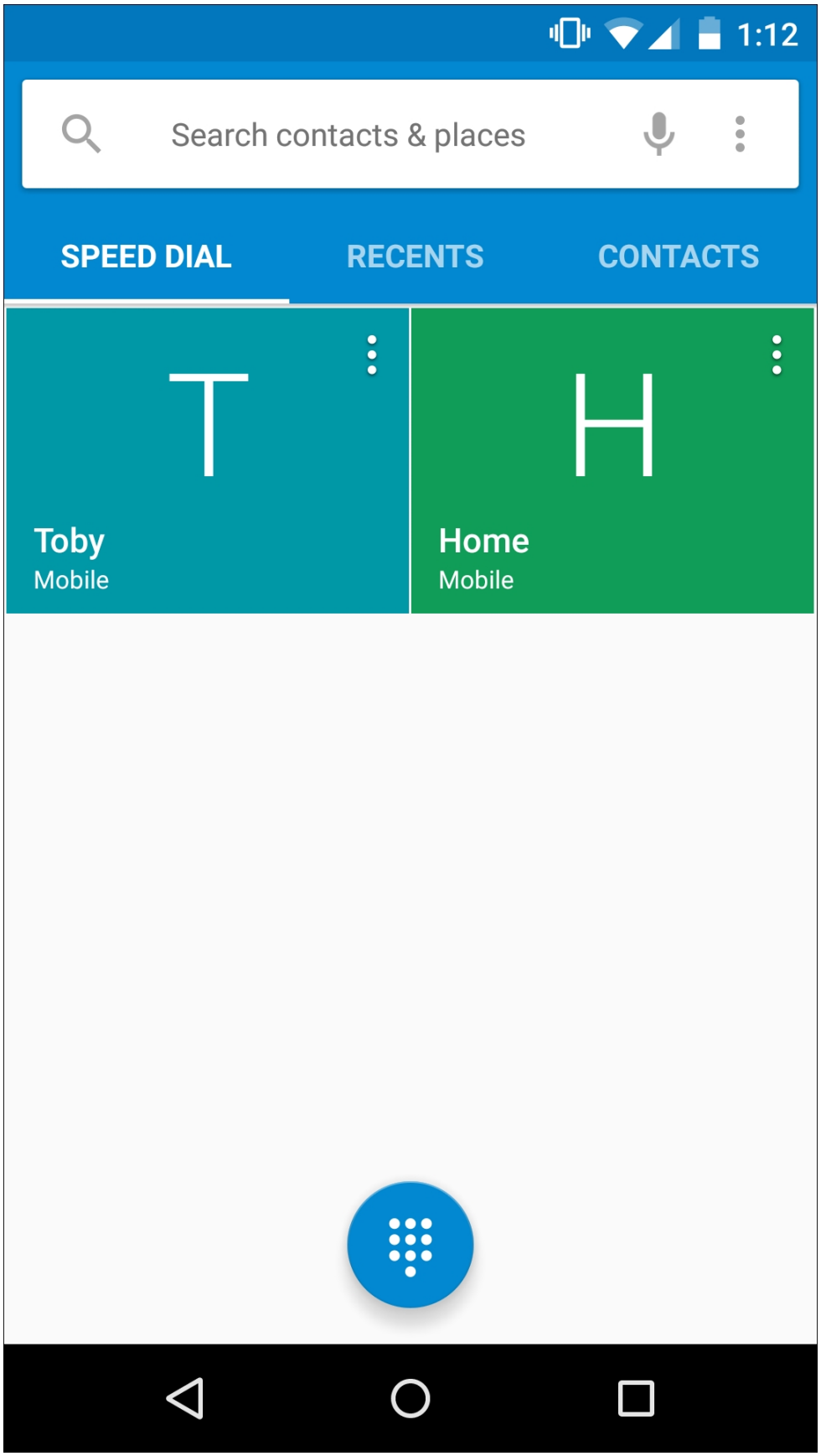

1:22
Photos
RAVEN HOUSE B&B
01947 880444
NO VACANCIES
0:17
All Gold Robin Hood's Bay

An example of grid-based navigation

Tabs

Although Android's traditional navigation model is hierarchical (moving from parent screens to child screens), there are some instances where a horizontal navigation may make more sense, particularly if your app features lots of sibling screens.

Tabs are one common examples of horizontal navigation. If your app includes lots of grouped content, sibling screens, and categorized data sets or content that's equally important, then tabs are an ideal form of navigation as they allow you to embed multiple screens in a single parent container. Tabs give the user a way of accessing all this content directly without them having to navigate back to the parent view every time they want to access a different sibling screen.


1:12
Search contacts & places
SPEED DIAL
RECENTS
CONTACTS
T
Toby
Mobile
H
Home
Mobile

Tabs are also a useful way of ensuring that users are aware of all your app's sibling screens. With tabs, users don't need to go exploring any hidden menus as all your app's sibling screens are lined up along the top of the screen.

If you do opt for tab-based navigation, there are several best practices you should keep in mind:

- Whenever the user selects a tab, new content may appear but the tabs themselves should remain constant.
- Tab labels should consist of either icons or text only. If you do opt for text labels, you should keep them as short as possible.
- Don't treat switching between tabs as history. If the user switches from tab A to tab B, pressing their device's **Back** button *shouldn't* take them back to tab A.
- Arrange tabs horizontally along the top of the screen as a single row. Tabs should never run along the bottom of the screen. If you do want to display actions along the bottom of the screen, use a split action bar instead.
- Reserve swipe gestures for navigating between tabs. Don't include any in-tab content that supports swipe gestures, as your users may end up swiping between tabs by accident.
- Don't include further tabbed content inside a tab. Remember that movie Inception? No one wants a *tab-ception*.
- If you suspect your users are likely to keep switching between tabs in order to compare their content, then this is a sign that you should combine your content into fewer tabs. Or perhaps, tabs aren't the right navigational solution for your app after all?
- Tabs are great for displaying multiple sibling views, but you should limit tabs to no more than four per parent screen. If you do need to display more than four sibling views, then once again, tabs might not be right for your particular app.

Horizontal paging (swipe views)

Horizontal paging is another popular method of navigating between sibling screens, and is sometimes known as *swipe views*. Like tabs,

horizontal paging is useful when your app features multiple sibling screens.

As you might have guessed from the name, swipe views are a form of navigation where one screen is displayed at a time, and the user navigates between the screen's siblings using a *swipe-from-the-right* or *swipe-from-the-left* gesture.

Horizontal paging works the best when your app has a small number of sibling screens, and there's an ordered relationship between those screens; for example, calendar apps typically use horizontal paging for navigating between different months.

The social layer

In our recipe app, we briefly touched on the idea of letting users share their favorite recipes with friends and family via social media. Even simple social features like this can add real value to an application, so it's worth considering whether your app might benefit from a few social features. At the very least, giving your users the option to share your app's content via social media is a quick and easy way of promoting your application.

> ## Note
>
> Adding a social aspect to your app isn't mandatory! You should only include social features if they add value to your app in some way; not *every* application has to have a social aspect.

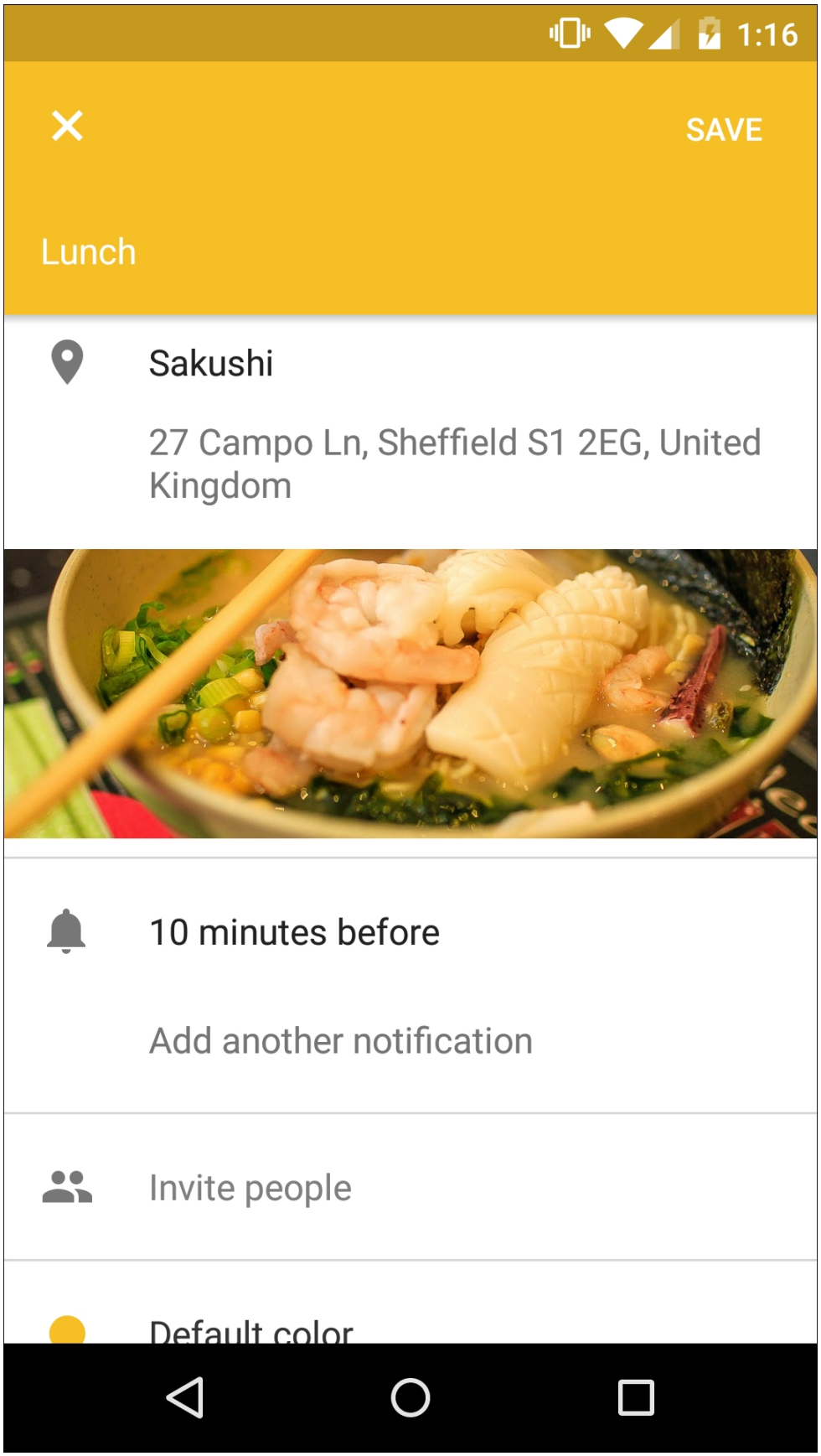
When you're deciding whether to include a social layer ask yourself the following:

- What would your users get out of the proposed social features?
- How would a social layer enhance the user experience?
- What specific social features might add the most value for your target audience?

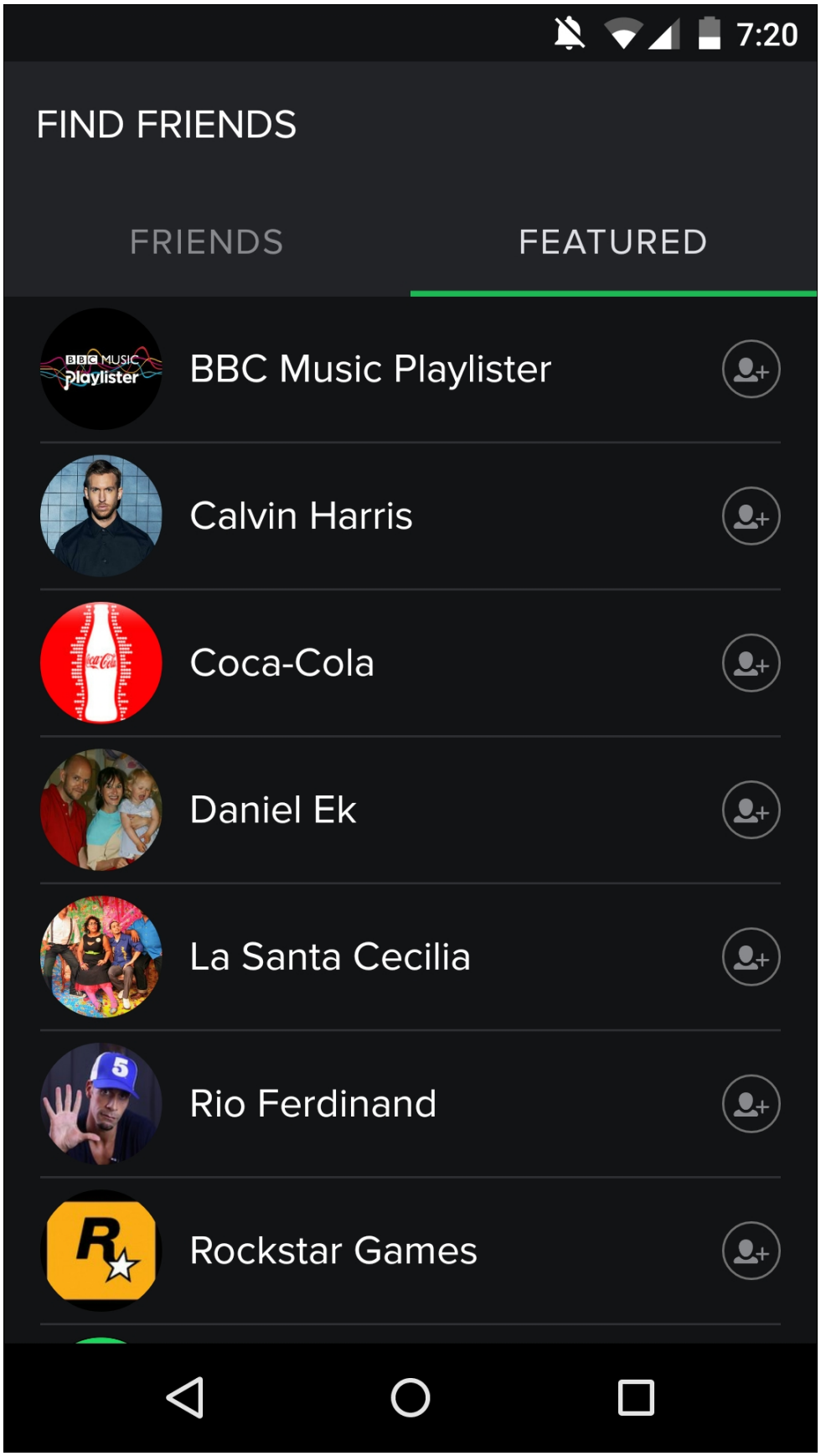
- What social features might help to create a sense of community?
- How much effort would the proposed social features require from the user? What are the potential rewards? Are these features *worth it* from the user's perspective?
- What are the advantages of adding a social layer to your app?
- What are the potential disadvantages?

When you think about social features, chances that are social media apps, such as Facebook and Twitter, immediately spring to mind. However, there are a multitude of ways of subtly integrating social features into your app without turning it into a social *media* app.

Some examples of non-social media apps that have an effective social aspect are Google Calendar and Spotify.


1:16
SAVE
Lunch
Sakushi
27 Campo Ln, Sheffield S1 2EG, United Kingdom
10 minutes before
Add another notification
Invite people
Default color

Whenever you create a new event in Google Calendar, the app prompts you to dip into your device's **Contacts** and invite friends, family, work colleagues, and other contacts to your event. Each contact can then either accept or decline this invitation, and you get the added security of knowing that everyone who accepts your invite will receive automatic Google Calendar reminders as your event draws near.


7:20
FIND FRIENDS
FRIENDS
FEATURED
BBC Music Playlister
Calvin Harris
Coca-Cola
Daniel Ek
La Santa Cecilia
Rio Ferdinand
Rockstar Games

Spotify uses social features to help you find new tracks to listen to. If you open Spotify and navigate to **Activity**, you'll find a list of all the music your Spotify friends have been listening to, and if you don't have any Spotify buddies, the app will suggest a few people for you to follow.

Summary

In this chapter, we created a detailed plan for a new Android project by writing down our user and product goals and identifying our target audience. We created a concept, product definition statement, features list, and considered our options for monetizing our app.

We then started to sketch out our app by creating a roadmap, flowchart, screen list, and screen map with complete navigation. In the next chapter, you'll see how to turn this rough sketch into a detailed paper or digital wireframe and populate this wireframe with dummy content.

Chapter 6. Turning Your Sketches into Wireframes

In the previous chapter, we concentrated on planning our screens at a very high level by creating things such as screen maps and flowcharts. In this chapter, we'll move on from high-level planning to start designing the screens that'll make up our app.

During the screen designing process, you'll usually complete the following steps:

- **First draft wireframing**: This is where you begin to layout your screens by making quick sketches with paper and pencils. A wireframe is a low-fidelity, visual representation of your app's layout; sometimes, it is referred to as a skeleton, outline, page schematic, or screen blueprint. To help you focus on each screen's layout, functionality, and usability, wireframes are usually stripped of all look and feel elements, and at least initially, most content and text.
- **Digital wireframe**: Once you're happy with your pencil-and-paper wireframes, it's time to refine these first drafts using digital software. This step is also where you check whether your emerging designs are suitable for the kind of content you want to display, by starting to add the content and text that each screen will eventually display.
- **Prototyping**: The next step is creating a prototype based on your digital wireframe using paper and pencil.
- **Usability testing**: This is where you test your paper prototypes on an actual user. This type of testing is crucial for highlighting any issues with your navigation and helping you get an idea of how easy your user interface is for users to understand.

Note

In design, low-fidelity refers to a rough representation of concepts, design alternatives, and screen layouts. By creating low-fidelity prototypes and wireframes, you can test various ideas without having to invest too much time and effort in the design process. The high-fidelity design is where you fill in all the details missing from a design's low-fidelity predecessors. In this chapter, we'll create low-fidelity wireframes using paper and pencils, and then develop these initial ideas into more detailed, high-fidelity digital wireframes.

What is wireframing?

Think of a wireframe as a skeleton of a screen's layout that describes how major UI components should be arranged in the screen's hierarchy. The purpose of wireframing is to focus on things such as functionality, usability, behavior, and the positioning and priority of your content. Wireframing *isn't* about graphic design. In fact, at this stage, it's better to leave out all design-related elements. You may even want to make your wireframes look very rough and sketch-like on purpose to help you focus on what each screen *does* and not what it looks like.

Your wireframes typically won't include color, unless you're using color to distinguish certain elements of your UI; for example, you might want to grab a colored pencil and draw a box around related content groups, or to add elements that don't have a physical onscreen presence such as interaction and movement.

You can create wireframes using pencil and paper, by scribbling on a whiteboard or using a wide range of free and commercial software applications. However, for the first drafts, it makes sense to wireframe via quick sketches using a pencil and paper.

What are the benefits of wireframing?

Wireframing may sound straightforward, but these deceptively simple sketches are an important step in designing more effective application screens. Here are a few of the reasons why it's well worth breaking out your pencil case, grabbing a notebook, and wireframing your app:

- Wireframing is ideal for rapidly testing different layouts.

  Creating sketches is quick and easy, which means wireframing is the perfect opportunity to explore lots of different potential layouts.

  Wireframes encourage you to get creative and experiment with different positioning and sizing of UI elements as well as different navigation. They're also an easy ways of mapping out the relationship between your UI elements.

  By wireframing lots of different possibilities, you can discover the optimal layout for each screen that makes up your app.

  While *technically* there's nothing stopping you from getting experimental at every stage of the design process, the more time and effort you invest in your project, the less inclined you'll be to experiment with new ideas and alternate layouts.

  One thing you should be aware of while wireframing lots of different layouts is that wireframes are static representations of your screens. Wireframes *aren't* very good at displaying interactive details or moving elements. You'll need to use your imagination to get a better idea of how any interactive or moving elements work in your layouts, or spend some extra time prototyping and testing your layouts, which is something we'll cover later on in this chapter.

- They help you identify problems early in the design process.

  As with everything in life, sometimes things work better in theory than they do in practice, so what seemed like a good idea on a

screen map or in your head may break down as soon as you try and implement it.

Whether your app goes through multiple extensive design changes or simply a few tweaks, design changes are an inevitable part of creating a mobile app, so isn't it better to discover potential problems as early as possible? Wireframing is one of the quickest and easiest ways of uncovering issues with your designs, and revising a wireframe is approximately 1 billion times easier than revising a high-fidelity mockup or, even worse, a user interface that's already under development.

Discovering issues at the wireframing stage also means you're more likely to make whatever changes are necessary to properly resolve the issue, even if it means drastically changing your design. When you uncover issues further down the line, you're naturally going to be tempted to look for workarounds and quick fixes that'll patch the issue without causing too much disruption to your design. This approach may make your life easier in the short term, but it probably isn't going to make for very happy end users in the long run.

- They encourage you to capture a complete design rather than winging it.

  Do you feel like you have a crystal-clear picture in your mind's eye of how your app should look? Maybe you've had this vision before you even started the planning process.. If this is the case, then great, but you still need to capture this vision on paper!

  Mental pictures have a tendency to be a bit fuzzy around the edges, and sometimes, you don't even quite realize how fuzzy they are until you try to capture them on paper. Once you start trying to recreate what you see in your mind's eye, you may realize that there are still a few question marks hanging over certain parts of your design.

And as you move through the design process, it's far easier to redraft and refine something that's physically there in front of you, rather than an image that's floating around in the ether.

- They give everyone a common design language.

  At their core, wireframes are basically just labeled and annotated sketches, so they're something that everyone can understand. This is good news if your app is a team effort, as wireframes help to bridge the gap between *developing* an app and *designing* an app *without* resorting to any developer or designer lingo.

  If you're developing an app for a client, wireframes are also a great way of making sure your client has a clear picture of how their project is progressing. You can then get the client's input and, hopefully, their sign-off on your design before you move onto the next stage of your project.

  People are naturally more reluctant to pick fault with a design that someone's clearly spent hours, perhaps even days, perfecting. Although it's nice to want to spare a team member's feelings, this isn't going to help you create the best possible app in the long run.

  Make sure your wireframe *looks* like a quick sketch, and even the politest member of your team should feel comfortable pointing out its flaws.

- They encourage you to start thinking about navigation in more detail.

  It's impossible to overestimate the negative impact of poorly-designed navigation. If a user can't navigate through your app with ease, they're going to navigate straight *out* of your app.

  We touched on navigation in the previous chapter, but up until now, navigation has been nothing more than a set of up, down,

and side-to-side arrows on a screen map. Wireframing is the perfect opportunity to start thinking about *how* you're going to implement your app's navigation. How exactly are your users going to move from the home screen to the Settings screen? Via tabs, action bars? How about their device's **Back** button? Wireframing is an essential step in designing effective navigation. In fact, whenever you create a wireframe, the first thing you'll need to do is decide what navigational elements should appear on the screen, as you'll see later on in this chapter.

Creating a Wireframe

Creating a wireframe may seem fairly straightforward. Anyone can slap a bunch of sketches together, right? While wireframing is supposed to be quick and easy, if you're going to get the maximum value out of your wireframing sessions, then you'll want to put as much thought as possible into your wireframes.

For the early drafts, the most effective method of wireframing is to simply sketch out your screens using paper and pencils. Ideally, you should produce several versions of each screen, and paper sketches are the quickest way to test out a few ideas. Once you've finished sketching, simply place your sketches side by side and decide which is the most effective design, or you may even decide to combine elements from multiple sketches to create the perfect layout.

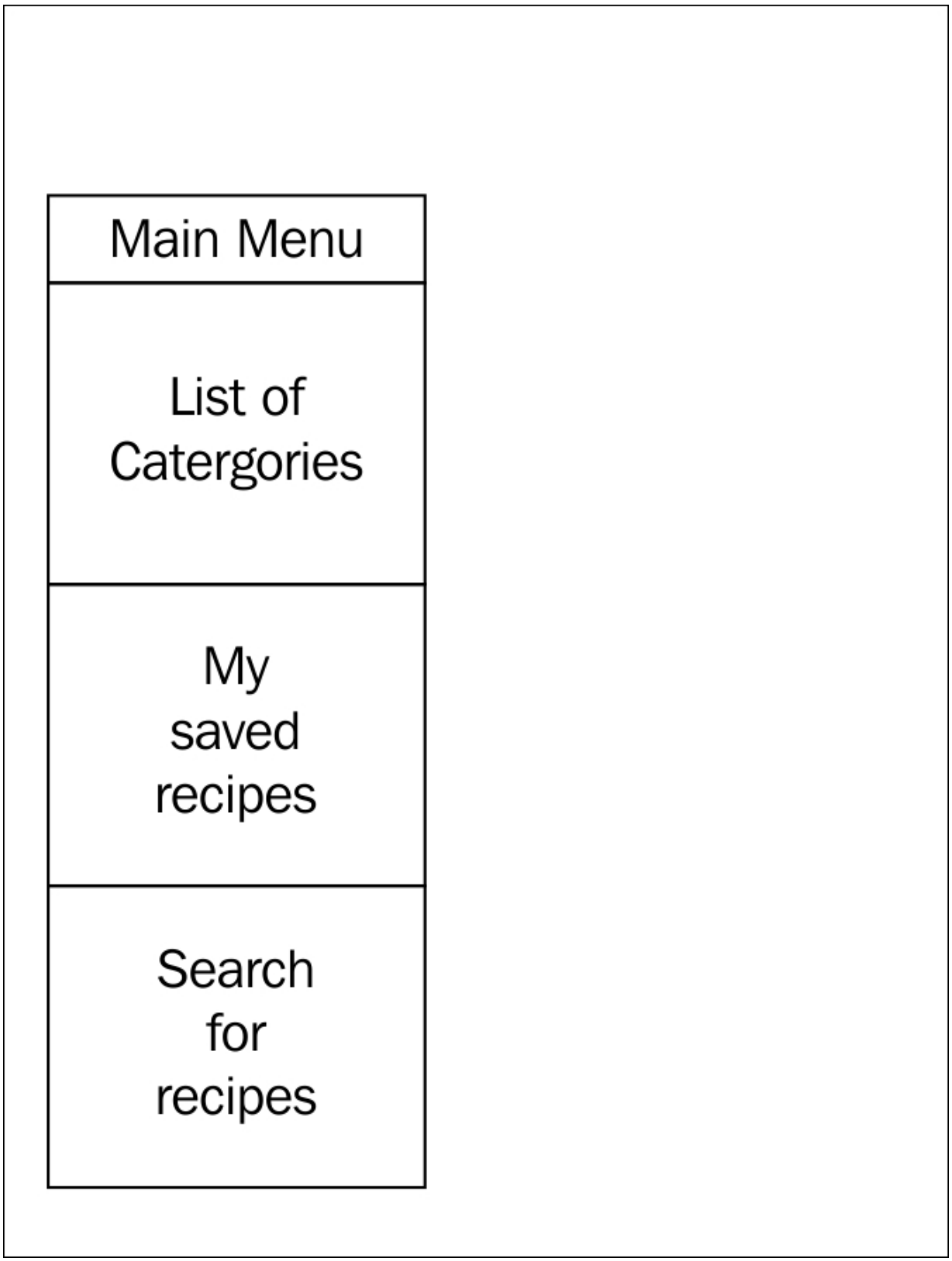
Even if you prefer the thought of digital wireframing and are itching to sit down in front of your computer and boot up your favorite image-editing program, you should still take the time to create rough sketches first, as this is the perfect way to get your ideas flowing, and quickly test out multiple designs before moving onto the more time-consuming process of digital wireframing.

Creating your first wireframe

During your wireframing sessions, it's helpful to have your screen map handy, so you can keep referring to it. When approaching a new project, it usually makes sense to start with the first screen the user will see when they launch your app. In our recipe app, this just so happens to be the home screen.

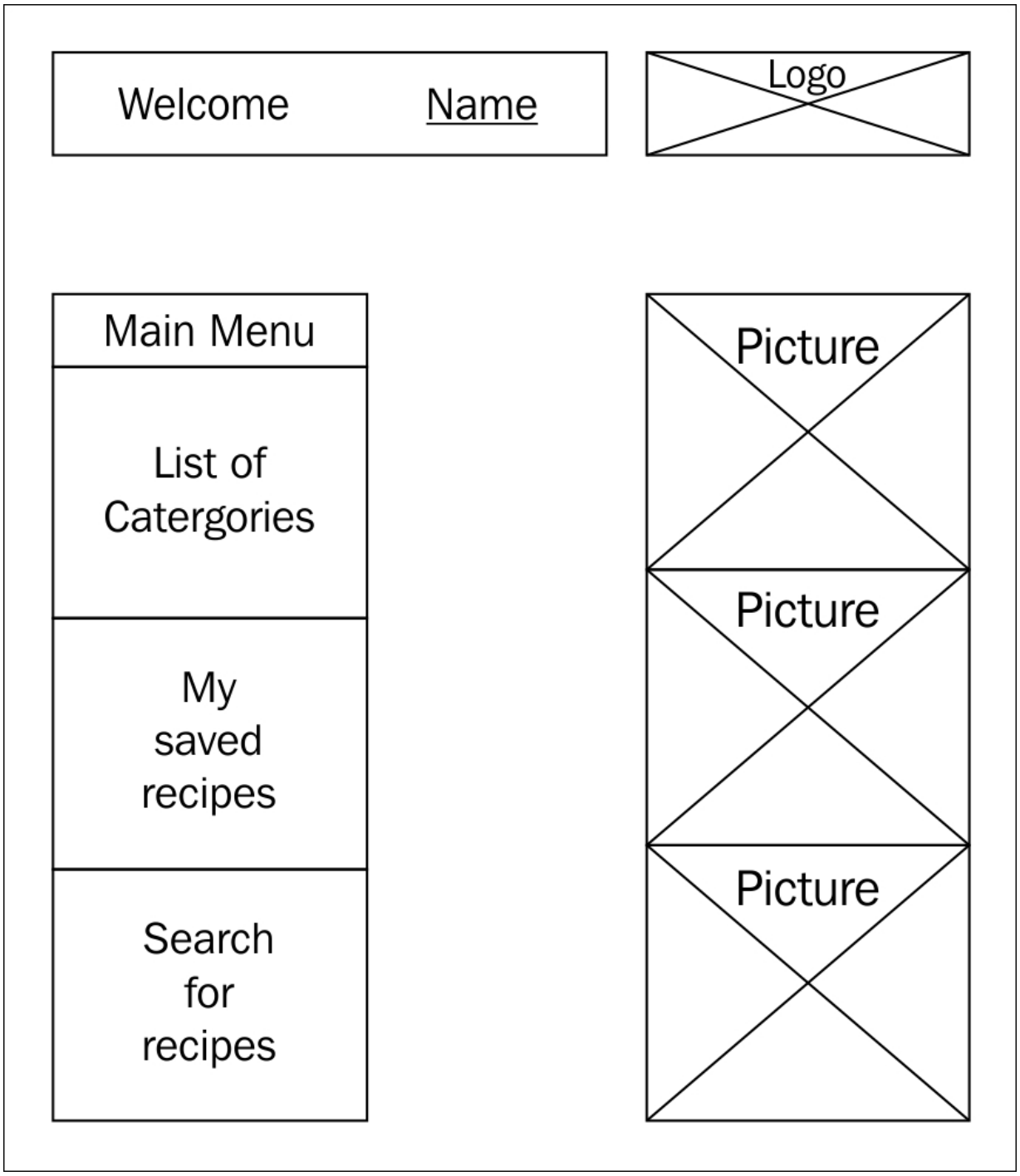
Draw a rough outline of your screen, and then work out what navigational elements you need to include. For our recipe home screen, these are as follows:

- The screens the user can reach the home screen *from*. Essentially, you need to consider how the user will navigate back to the previous screen. A lot of the time, this is handled automatically by the device's **Back** button, but you can also provide backward navigation via the left-point caver that appears in the action bar or a UI element, such as a **Return to previous screen** button. To keep things simple, let's assume that the backward navigation is handled automatically by the device's **Back** softkey.
- The screens the user can navigate *to*. In our recipe example, this is the list of saved recipes, the list of categories, and the search screen. To present all these options in a compact way, I'm going to add them to a menu that'll appear along the left-hand side of the screen:


Main Menu
List of Catergories
My saved recipes
Search for recipes

Next, decide what content you need to display. Start by adding the most important content such as images, headings, subheadings, and other large blocks of content, and then work your way down the content hierarchy.

For my home screen, I want to display the app's title and logo, plus a few tantalizing images of the meals that users can cook by following the recipes included in this app:



Tip

Feeling uninspired?

If you're struggling for inspiration, then taking a critical look at other apps is a great way to kickstart the creative process. The best places to draw inspiration from are high-rated apps on the Google Play store or Google's own suite of apps. The latter are particularly useful as they're usually good examples of the latest best practices and design principles, particularly Material Design.

Spend some time flicking through the different screens and ask yourself what you like about each screen's layout and whether there are any improvements you could make. It may help to create wireframes of any screen you particularly like, and then look for any similarities, elements, or just general design principles that you can use in your own wireframes.

Exploring more wireframes

Every time you create a wireframe, the process will differ slightly depending on factors such as the kind of app you're creating, the content you want to display, the screen's purpose, and your target audience. However, it is possible to identify certain *types* of screens that occur across multiple apps as these often share similar characteristics and UI elements.

Case in point: our home screen wireframe. Many apps have some form of home screen, and these screens often feature things such as headings, logos, and main menus–exactly like our recipe app home screen. In addition to the home screen, this app is a great example of a few other common *types* of screens. Since you're likely to encounter these screens in your own projects at some point, let's look at how you can approach creating wireframes for each of these screens.

I'll start with what I'm going to call a *details* screen. This is a screen that appears when the user selects an item on a previous screen, and the app displays a new screen containing more information about the selected item. In our recipe app, the screen containing each recipe is a details screen; users scroll through a list of recipes,

spots one they like the look of, and then tap the recipe. At this point, they're taken to a separate screen containing all the information they need to recreate this meal.

> **Note**
>
> In our screen map, we have referred to this screen as the *detailed recipe view*.

Wireframing a details screen

As the name suggests, details screens are all about displaying information, which is typically a mix of text and media, such as images, videos, or audio.

Let's follow the same process we used for wireframing our home screen and start by identifying all the navigational elements we need to add. Taking a look at the screen map, I can see that I need to add the following:

- The screens the user can reach the detailed recipe screen *from*. Since we glossed over backward navigation in our home screen wireframe, let's explore it in more detail here. According to the screen map, the user can reach the detailed recipe screen from the search results screen, the list of recipes for a given category, and the list of saved recipes. Although we could potentially rely on the user tapping their device's **Back** softkey to return to the previous screen, wireframing is all about exploring different options, so I'm going to try something different and add a **Back to previous screen** button to this particular wireframe.

- The screens the user can navigate *to*. In our simple screen map, the detailed recipe screen looks like it's the final screen in the app, but in reality, you rarely encounter this kind of *dead end* in a mobile application. When users add this recipe to their list of

saved recipes, I think it'd be helpful if the app takes users to their online scrapbook, so they can view all their saved recipes. To create this effect, I'm going to add an **Add to scrapbook** button that, when pressed, performs this action *and* takes the user to their online scrapbook.

Now, it's time for content. In no particular order, I want this screen to display the following:

- The recipe's title
- How difficult the recipe is to make
- An estimate of how long it takes to cook
- An estimate of how much the ingredients cost
- How many people the recipe feeds
- An enticing, mouth-watering photo of the finished result
- The recipe itself

Tip

Do you really need a details screen?

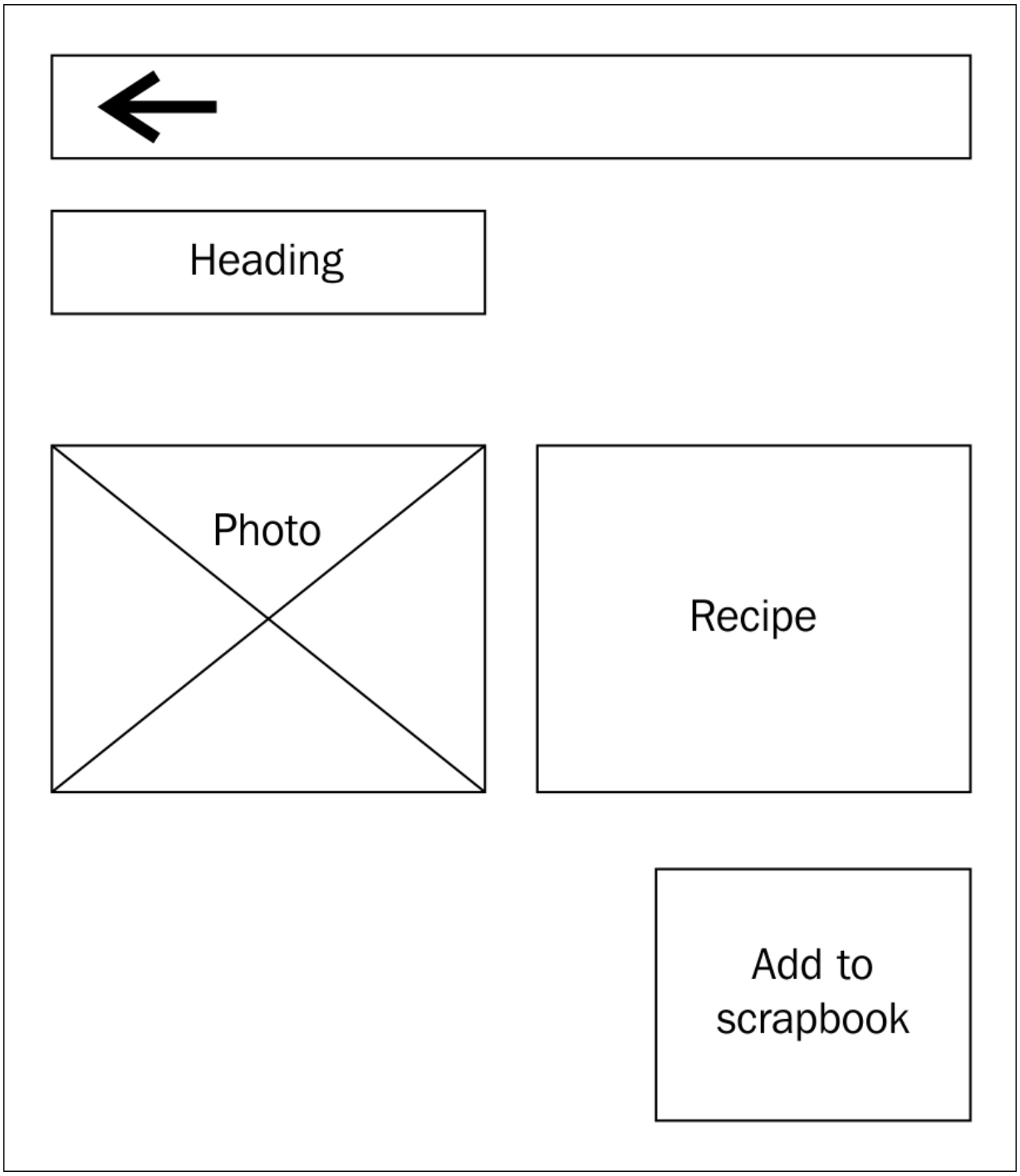
The golden rule of creating a details screen is that it *must* contain enough information to warrant its status as a separate screen. For example, imagine you've just downloaded an app where you can check what films are being shown at your local cinema.

The first screen is a timetable that displays all the films alongside their showing times. You can tap every film in the timetable and the app will load a new screen that's dedicated to the selected film. What new information might you expect to find on this screen? Maybe a synopsis or a list of the lead actresses and actors? How about a star rating or links to reviews, so you can decide whether this is a film that you *need* to see on the big screen with popcorn, or whether you should just wait for it to come out on Netflix instead?

Now, imagine you didn't see any of this additional information, and the details screen simply reiterated the information that you'd already seen on the main screen. Would you feel like this was a well-designed app that was giving you a great user experience? The answer is a resounding *nope*.

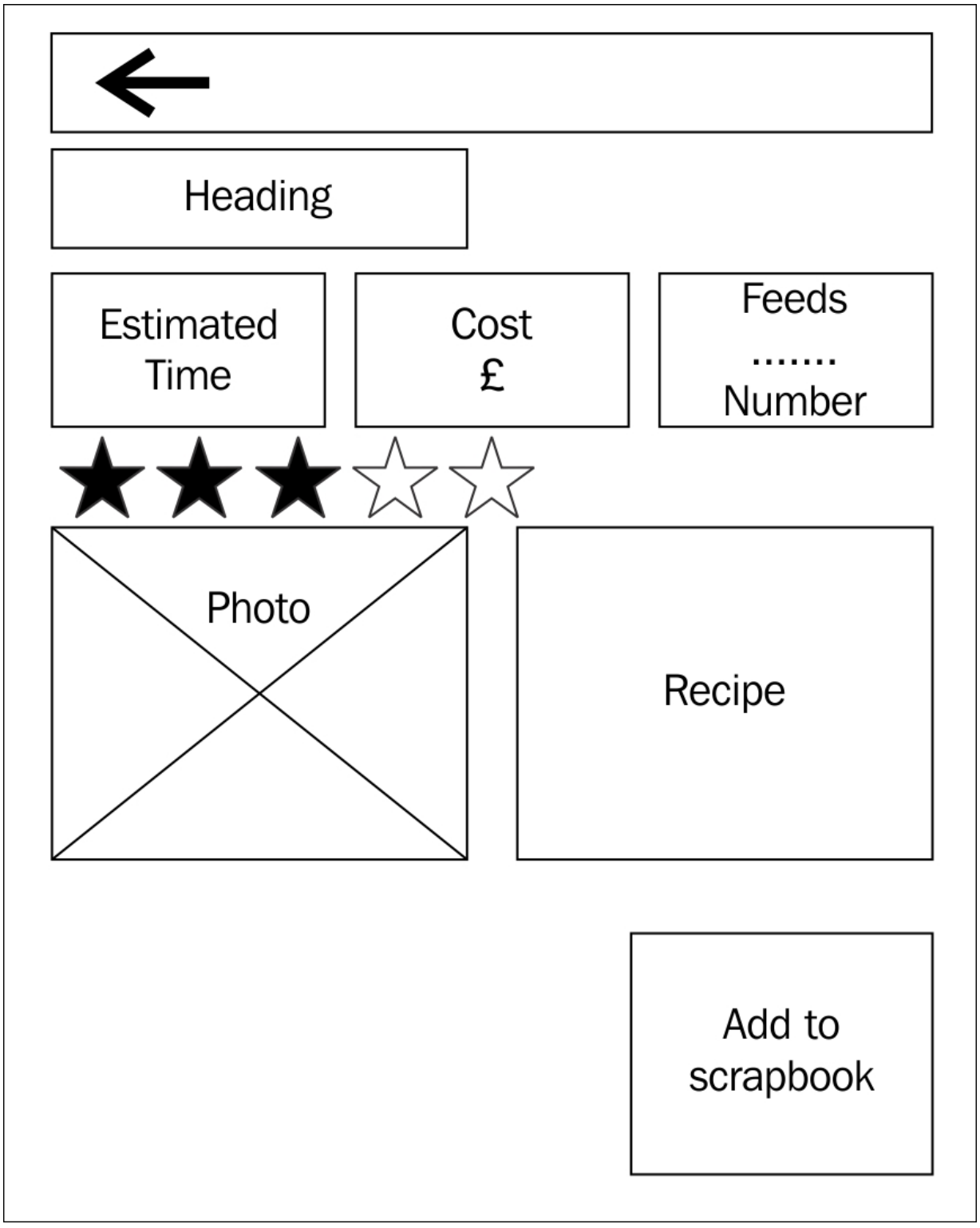
If you *do* decide to include a details screen, then make sure it contains some new information. If it doesn't, then it has no value, and therefore, it has no place in your app.

As always, add the most important content to your wireframe first. In this example, that's the **Heading**, an image of the finished product, and the **Recipe** itself:

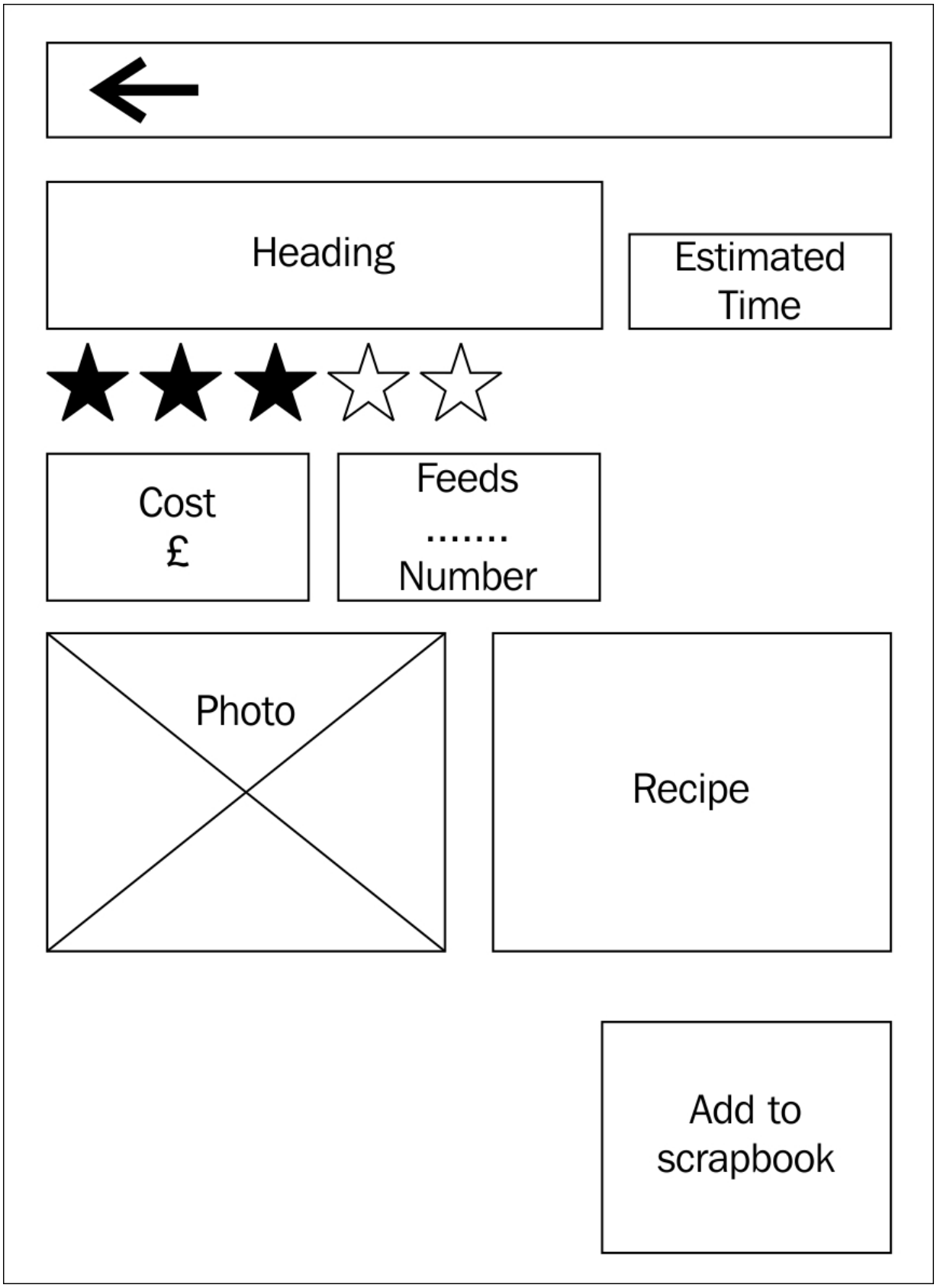


As you work your way down the content hierarchy, you may encounter content that could work in numerous positions in your screen layout. If you do, then this is the ideal time to experiment with different layouts by creating multiple wireframes. You can then place these drawings side by side and decide which layout you prefer, or

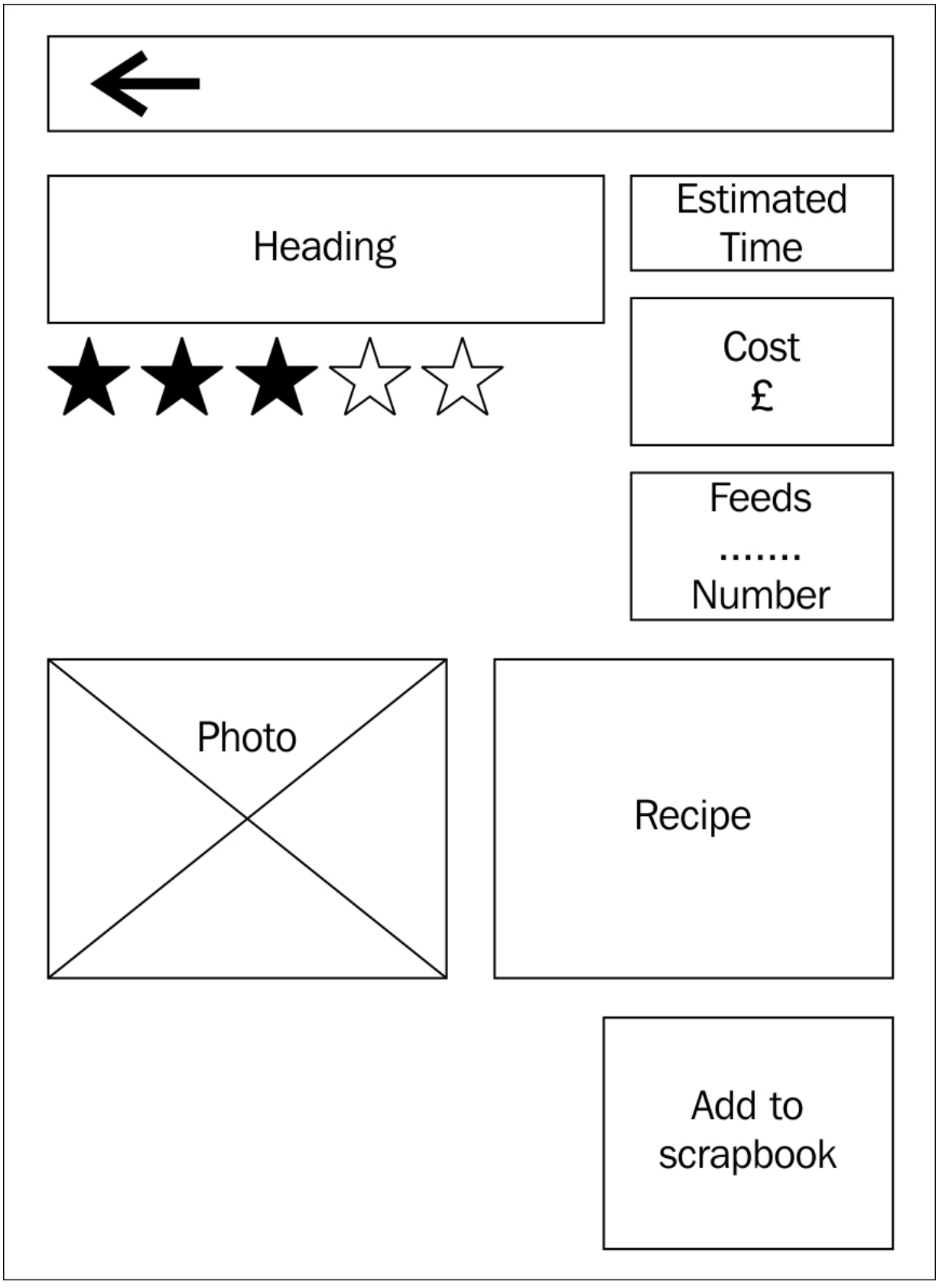
you can combine elements from multiple wireframes to create your perfect layout. Here's a selection of the wireframes I created for the detailed recipe screen:



This wireframe contains several elements that could conceivably work in different locations–specifically, the text boxes that contain the estimated time, estimated cost, and the number of people this recipe feeds, plus the recipe's star rating. Wireframing gives you a quick and easy way of testing out these elements in several different positions.


Heading
Estimated Time
Cost £
Feeds ..... Number
Photo
Recipe
Add to scrapbook

While wireframing these elements in various positions, I've decided to keep the major content areas the same in all these wireframes:


Heading
Estimated Time
Cost £
Feeds ..... Number
Photo
Recipe
Add to scrapbook

Note

There's lots more content we could add to this screen; for example, it might be nice for users to be able to post comments, upload photos, or award each recipe a rating out of 10. You might also want to include multiple photos of the finished results, so the user can scroll through these photos. To keep this chapter short, I'm not going to wireframe all these options, but if you're feeling creative, you might want to grab a piece of paper and a pencil, and create your own wireframes that include all this additional content.

Wireframing search screens

Many apps have some form of search functionality, and our recipe app is no exception, so let's take a look at wireframing the search screen.

You know the drill by now! Look at the screen map and work out all the navigational elements you need to include:

- All the screens the user can reach the search screen *from*. According to the screen map, users can reach the search screen from the home screen only.
- All the screens the user can navigate *to*. The only screen the user can reach from the search screen is the search results screen.

In the previous wireframes, we handled backward navigation by relying on the device's **Back** softkey and by creating a **Return to previous screen** button, so let's try something different this time around! I'm going to use a left-facing caret in the action bar.

Note

When you're creating Android apps in real life, you'll typically select one method of navigation and use it across your app, rather than the mixture we're using in this chapter. Although I'm experimenting with lots of different navigational options, just be aware that in a real-life project, you'd select one form of navigation and apply it across your entire app to avoid confusing and frustrating your users.

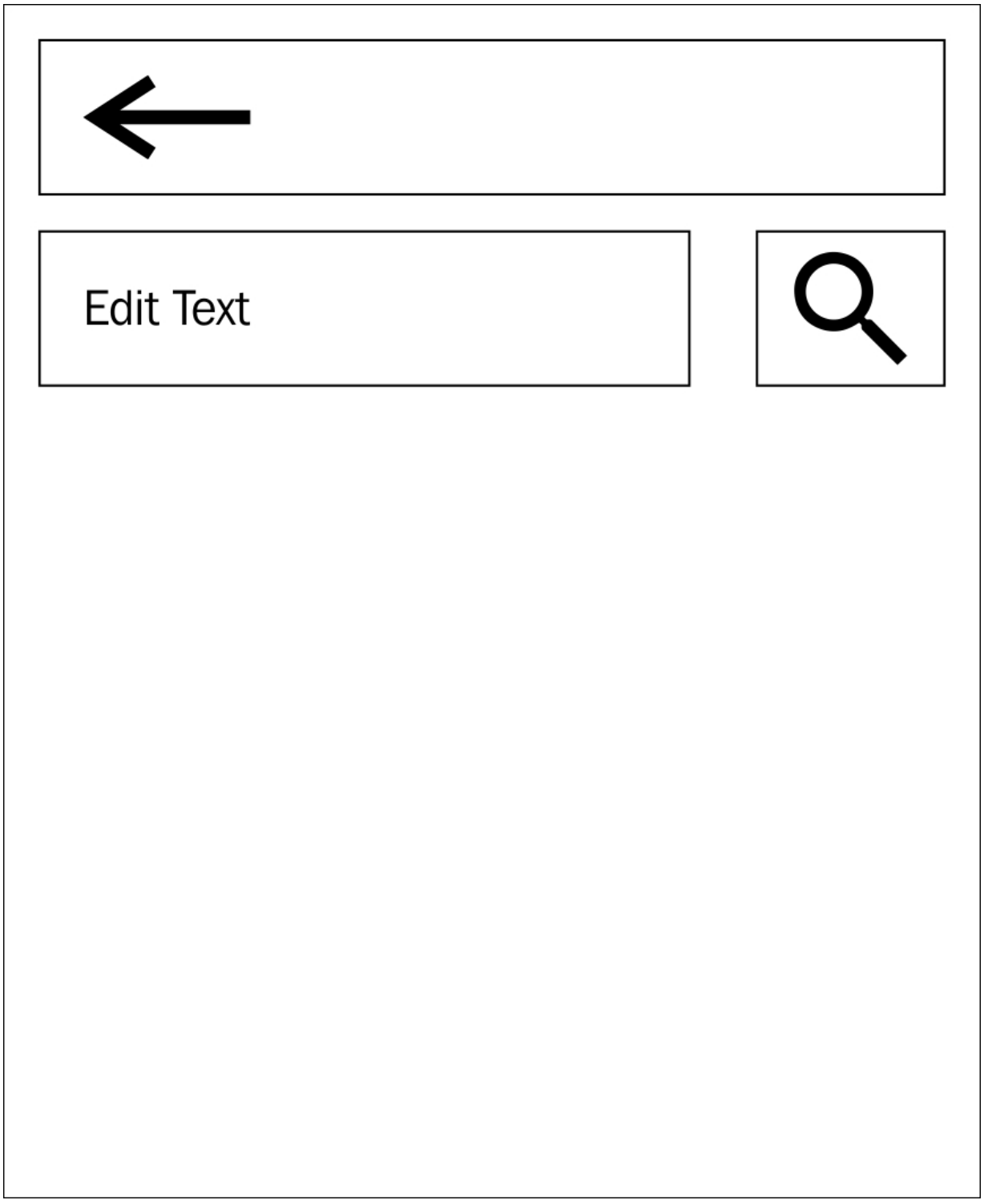
Let's create our search functionality. I'm going to keep things simple and use the following:

- An `EditText` where users can enter the keyword(s) they want to search for.
- A button the user presses to submit their search term(s). I'm going to label this button with the standard search system icon to ensure that the user instantly knows what action this button performs.

You might have noticed that in the previous wireframes, we were talking in general terms, such as heading boxes and images, and now, we're using the technical term *EditText.* This is because there are no strict rules when it comes to wireframing, so if you already have an idea of how you're going to create certain UI elements, and you want to add this information to your wireframes, then go ahead!

Note

The only exception is when you're planning on sharing your wireframes with other people, particularly if these people aren't Android developers themselves! If you're creating wireframes as a collaborative effort with non-Android types, then you should stick to civilian terms such as text boxes and images.



While there's nothing wrong with this layout, there's no denying that it's pretty basic. It would be better if our users could search for recipes based on a few different criteria:

- How easy a recipe is to make
- How much the ingredients cost to purchase
- How many people the recipe feeds

There are lots of ways of implementing this functionality, but the two that immediately spring to my mind are here:

- Give the user the ability to filter recipes based on different criteria by adding checkboxes to the existing search screen wireframe. Users can then enter any keyword they want to search for *and* select the relevant checkboxes; for example, they might search for recipes that contain the words `prawn madras` (yum, yum) and then select the checkboxes marked **Costs £2-£3** and **Takes 15-20 minutes to cook**.
- Stick with our simple search screen, but add filters to the search *results* screen. The user searches for keywords as normal and is taken to the search results screen, which contains additional filters that allow the user to narrow down their search results.

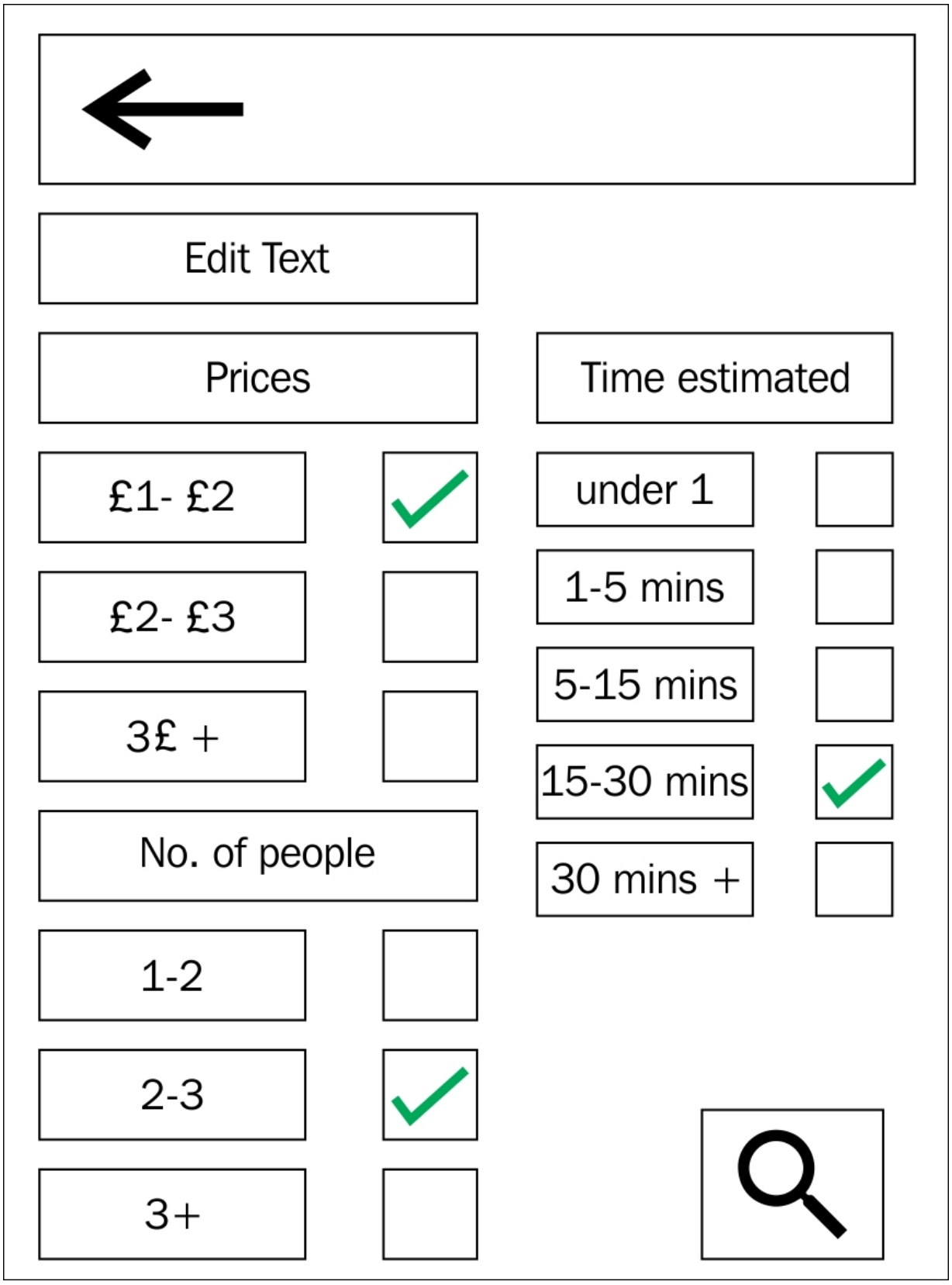
Both of these are viable options, so I've quickly wireframed each approach. This is a great example of how you can use wireframes to quickly test out various layouts:

Price | Time | Feeds...

Photo	Title	Add to scrapbook
Photo	Title	Add to scrapbook
Photo	Title	Add to scrapbook
Photo	Title	Add to scrapbook
Photo	Title	Add to scrapbook
Photo	Title	Add to scrapbook

In this wireframe, the user performs a search as usual, but the search results screen contains three new buttons (**Price**,**Time**, and **Feeds...**), which they can use to filter their search results based on different criteria.

The following wireframe is a new version of the Search screen where users enter their search terms into the `EditText` as normal, but then uses various checkboxes to be more specific about the search results they want to see:


Edit Text
Prices
£1- £2
£2- £3
3£ +
No. of people
1-2
2-3
3+
Time estimated
under 1
1-5 mins
5-15 mins
15-30 mins
30 mins +

The search screen as a fragment

Regardless of where you are in the design and development process, you should always be thinking about how to support as many different Android devices as possible.

When you're busy wireframing, it's easy to fall into the trap of designing with one device in mind. Although this does make wireframing easier, it won't make for a very effective UI in the long run.

Tip

Wireframing for landscape and portrait

Playing around with the sizing and positioning of your content during the wireframing phase not only helps you work out how you might better support multiple screen configurations, but it also helps you to design an app that functions equally well in both landscape and portrait modes.

Planning for multiple screens may mean creating several wireframes that target different screen configurations, or it could mean combining your content in different ways using fragments.

In the previous chapter, we mentioned displaying our search screen as a standalone activity on smaller devices, but displaying it as a fragment alongside the home screen on devices with larger screens. Since we've already wireframed the first possibility, it only seems right to wireframe how the search screen might function as part of a multi-pane layout.

To speed up this process, I'm going to reuse the home screen wireframe that we created earlier as well as the original, basic search screen wireframe.

As always, let's start with navigation. Most of the navigational elements from our home screen wireframe are still valid apart from the **Search for recipes** item in the left-hand menu. Since the search box will appear alongside the home screen in this scenario, we can remove this link from the menu.

Similarly, the original search wireframe included an action bar with a left-facing caret that's no longer necessary; since the fragment appears as part of the home screen, there's nowhere for the user to navigate back *to* apart from exiting the app, and we don't want to help them do that!

With navigation sorted, let's look at the content. All the content from the original home screen and search wireframes is still relevant, and therefore, it has a place in our multi-pane layout.

Let's take a look at the finished result:

Welcome Name

Logo

Edit Text

Picture

Main Menu

List of
Catergories

Picture

My
saved
recipes

Picture

Picture

Picture

Note

Feeling overwhelmed?

Working out the best position for each element while also trying to design for multiple devices may leave you staring at a blank sheet of paper and wondering *where do I start?* If you're feeling overwhelmed, then it may help to think of wireframes as a series of drafts.

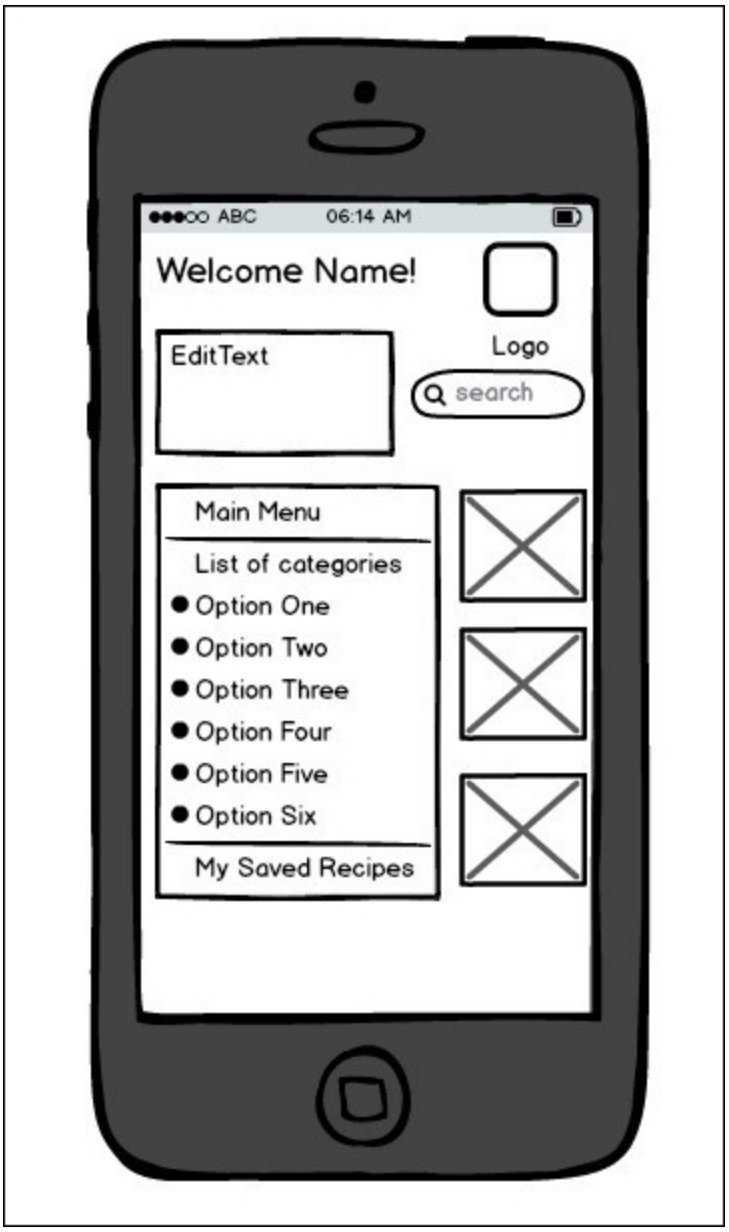
When creating you first draft, you should wireframe without stressing about trying to support multiple devices. Instead, concentrate on defining how your app looks in its *natural* state. How wide would you like that heading to be, if you didn't have to worry about running out of space on smaller handsets? Ideally, how far would that button be placed from the edge of the screen? And do you *really* want to display two images side by side? Are you sure you wouldn't prefer three? Or four? How about five?

Grab a pencil and wireframe without a care in the world. Then, after you've finished patting yourself on the back for an excellent first draft, you can start worrying about how this wireframe might translate across different devices.

Digital wireframes

Once you're satisfied with your paper wireframes, you should develop them further. This may mean creating a more detailed, thought-out sketch using your trusty pencil and paper, or it may mean developing these paper wireframes into *digital* wireframes.

If you do decide to digitize your wireframes, there's plenty of wireframing software out there, from professional image-editing programs such as Adobe Photoshop, Illustrator, and Fireworks, right through to tools created specifically for wireframing, such as Pidoco, Axure, InDesign, Sketch, Omnigraffle, and Balsamiq Mockups.



The previous image is an example of our recipe app home screen, created with the Balsamiq digital wireframing tool.

If you're unsure about what tool to use, then a quick Google search will return lots of potential wireframing software, at which point you'll be faced with a whole new problem–too much choice!

If you're struggling to decide between several potential wireframing tools, then ask yourself the following questions:

- Does this tool have all the features I need? Before taking the plunge, you should make sure your chosen tool has all the features you need to quickly and easily create wireframes for your *specific* project. This is especially important if you're investing cold hard cash in professional wireframing software. If you download free wireframing software, and then realize that it isn't quite right for you, then at least you haven't wasted any of your hard-earned cash! To avoid being swayed by persuasive website copy, jot down all the features your ideal wireframing tool should include. You can then check whether each potential tool meets your list of criteria; and if it doesn't, then you can move onto the next one.

- Is it customizable? Being able to tweak the wireframing program to meet your exact needs is always positive, but this is particularly important if you're developing an unusual or niche UI.
- Can you store reusable elements? You'll typically use many UI elements across multiple screens such as action bars, menu, and buttons. To supercharge the wireframing process, keep an eye out for tools that allow you to easily store and reuse visual elements.
- What's the learning curve? Professional vector illustration tools may sound like just the thing your project needs, but complex tools may leave you scratching your head, particularly if you're not overly familiar with design. The whole purpose of wireframing is to save yourself time by quickly testing out multiple layouts. However, this whole exercise becomes pretty pointless if you end up spending hours with your head buried in the program's user manual rather than actually creating wireframes.
- Are you already familiar with software that has wireframing potential? Even if this program isn't the most modern or feature rich, if you can use it to create effective wireframes in a reasonable amount of time, then it may make sense to use this old favorite, rather than investing time in getting to grips with some new software.

If you're struggling with the sheer variety of wireframing tools out there, then you may want to focus on free wireframing software, rather than jumping headfirst into purchasing a professional image-editing tool. The huge benefit of free tools is that you can see firsthand whether they're right for you by downloading a few and giving them a quick trial *without* racking up a massive bill.

Adding content to your wireframes

Up until now, we've made a point of not adding content to our wireframes; but at some point, you'll need to test whether the way your screens are shaping up is suitable for the kind of content you want to display. Once you have a set of digital wireframes you're happy with, it's time to test whether your content fits into these wireframes, or you need to make some adjustments to said wireframes.

Content, mainly copy, doesn't really look appealing on *any* platform, but this is even more true on mobile devices. The smaller screen of a smartphone or tablet can make large blocks of undifferentiated text seem even larger.

If you're going to present your app's copy in an elegant way, then you need to put some thought into its design, and wireframes are the perfect place to start.

As you move into digital wireframing territory, you can start experimenting with design elements, such as different fonts, text sizes, bullets and numbered lists, as well as headings and subheadings. The end result? A UI that blends your sparkling, engaging copy with appealing presentation and design–basically, text that your users will actually *want* to read.

If you leave content and copy design issues until you have the final screen design in front of you, then chances are that you'll try and convince yourself that it isn't *really* that big a problem; or you'll look

for the quickest and easiest workarounds even if they don't make for the best screen design.

When adding content to your wireframes, it's crucial to keep in mind what *real* content looks like. It's tempting to use text that just so happens to be the right size and length to fit neatly into your wireframe. Unfortunately, unless you're *very* lucky, this simply isn't the way real content works, so it's important to use realistic text in your wireframes.

Although you can use the classic `Lorem ipsum` Latin filler text or some dummy content, the best way to test that your layout is suitable for the kind of content you want to display is to use the same copy that'll eventually appear on the screen.

In the case of our recipe app, this presents a bit of a problem as the text on each detailed recipe screen will be different. For example, each recipe title will be a different length, so how do we ensure all these titles will fit into the available space?

The answer is to look at the full range of titles that are ultimately going to have to fit into the heading box, and then work out the average length. You should also go out of your way to find awkward content, as this'll help you plan how your UI will support these edge cases. In this instance, this means finding the longest and shortest recipe titles, and then working out how to display them effectively.

When the detailed recipe screen has to display a short title, will the size of the heading box remain fixed or will it shrink? And if it does shrink, how can we prevent lots of empty whitespace appearing around the newly shrunken box? Will the box expand to accommodate recipes with unusually long-winded titles? Will the text wrap onto multiple lines?

You don't have to wireframe *every* possibility, but you should try to identify all the major problems varying copy and content could cause your layout. Only then will you be able to create a wireframe that's flexible enough to take unusual copy and content in its stride.

What is paper prototyping?

Once you've created a digital wireframe that you're relatively happy with, it's a good idea to put this design to the test via paper prototyping.

Paper prototyping is where you create rough drawings of the different screens that make up your app, representing each screen as a separate sheet of paper. You then test this paper prototype on a user (or ideally, *users*) and incorporate the user's feedback into your digital wireframes. This method of usability testing may seem simple, but it's actually a valuable way of reviewing and refining your design so it's the best it can possibly be.

Creating a paper prototype is an important step in the design process for two main reasons:

- **They're great for identifying any usability problems that you might have overlooked**: Spotting the faults in your own design isn't always easy. Paper prototyping gives you a way of testing your app on your target audience without you having to write any code. And as we've already discussed, the earlier you can identify flaws in your design, the easier it is to fix those flaws.
- **They help you test your app's navigation up-front**: Designing easy and frictionless navigation is an essential part of creating a successful app, but it's difficult to properly evaluate navigation until you have actually experienced that navigation. Prototypes are a way of taking your planned navigation for a test drive and seeing first-hand how easy or difficult it is for users to navigate your app. If you skip the paper prototyping stage, the danger is that you won't realize there's a problem with your navigation until you've actually started to build your app. And since navigation should be consistent across your app, if you realize that you need to tweak the navigation on one screen, you may find that you need to change the navigation across *all* your screens; and this is not something you want to discover at the

last minute! Paper prototyping increases your chances of getting navigation right the first time around.

Usability testing

Paper prototypes are perfect for usability testing. Ideally, you'll want to test your prototypes on a member of your target audience, but colleagues, friends, and family members make good stand-ins as long as they haven't been involved in the design process.

For the most effective usability testing, you'll want someone's genuine reaction to approaching your app *blind*, just as a regular user would after downloading your app from the Google Play store. If you enlist the help of someone who has insider knowledge of your project, then this isn't really replicating the typical user experience.

In usability testing, your test subject interacts with the paper prototype while you manipulate the prototype to reflect the user's actions. For example, if they tap a **Go to home screen** button, you should switch pages and present them with your home screen prototype. Okay, so this may sound a bit silly, but it's also one of the most effective ways of discovering potential usability issues before you invest any more time in designing and developing your app, so it's well worth feeling a bit silly for.

Ask the user for feedback as they're interacting with your paper prototype, and be sure to prompt them to explain why they're performing each action and what they expect to happen next. You should also encourage them to speak up about anything that confuses them, or if there's anything about your UI that they don't like. Do they understand how to navigate to the next screen? Do they immediately understand what task they need to perform on each new screen, or do they need to stop and think about it?

Ideally, you'll get enough information to warrant making some revisions to your digital wireframes. If you do make any significant changes to your wireframes, then consider creating a new set of paper prototypes and testing them on the same user. Repeat this

process until your test subjects are navigating your paper prototypes with ease and have no further feedback or suggestions.

At this stage, you should have a set of digital wireframes that you feel confident about. We'll be using these wireframes to create more detailed, digital prototypes in the next chapter.

Rapid prototyping

If your app is a team effort, you may want to consider using paper prototyping as part of a rapid prototyping approach.

In rapid prototyping, each team member creates a paper prototype and tests it on a single user. The team then comes together to share their feedback and ideas, at which point each person creates a second prototype. Again, each prototype is tested on a single user before the team meets again to share their feedback. You can then update your digital wireframes based on the collective feedback. This is one of the quickest and easiest ways of collecting feedback from multiple users and incorporating all this valuable information into your design.

Summary

When it comes to developing Android apps, it's impossible to overestimate the value of planning.

In this chapter, we looked at how to translate the high-level plans that we made in the previous chapter into actual screen designs. We created quick paper wireframes, and then used them as the basis for detailed digital wireframes complete with content and copy. Finally, we created paper prototypes and looked at how usability testing can help you refine your screen designs, making them the best they can possibly be. In the next chapter, we'll look at how to create more complex and polished digital prototypes using Android Studio and the Android SDK.

If this chapter was your first taste of wireframing and prototyping, then you may still be feeling overwhelmed, but the important thing to remember is that as long as you're doing *some* form of wireframing and prototyping, then you're already doing much more planning than you were before.

The only word of warning is don't invest too much time and effort perfecting your screen designs early on in the design process. Your designs should start off very rough around the edges and gradually become more refined and detailed as you move through the different planning stages.

If you're still unsure, then take another look at the recipe wireframes included in this chapter and ask yourself "how could I improve these designs?" Then, wireframe the results! You may even want to create prototypes based on these wireframes. Alternatively, boot up one of your favourite Android apps and create wireframe and prototype versions of all the main screens. These are both valuable ways of refining your skills so that by the time you come to plan your own real-life Android projects, you'll be a wireframing and prototyping pro.

Now that we've got a rough design for our application, it's time to dive into some development work. In the next chapter, you'll use Android Studio to create a working digital prototype based on your wireframes and paper prototypes.

Chapter 7. Building a Prototype

After doing much of the initial design work in the previous chapters, it's now time to move past designing and into the early stages of developing. And the first development task is to create a digital prototype.

After creating multiple wireframes and paper prototypes, you may be itching to start some *real* development work, but digital prototyping is a *crucial* step in the design process.

Despite all the time and effort you've already invested in your design, at this point it's still just a plan. Digital prototyping is where you put this plan to the test to see whether it works in the real world. Also, digital prototyping is just plain fun as it allows you to go from no code to a working version of your app in no time at all.

By the end of this chapter, you'll have developed a working digital prototype of your app. You'll have also finalized your design by planning the finer details of your app, which means starting to think about graphic design.

Note

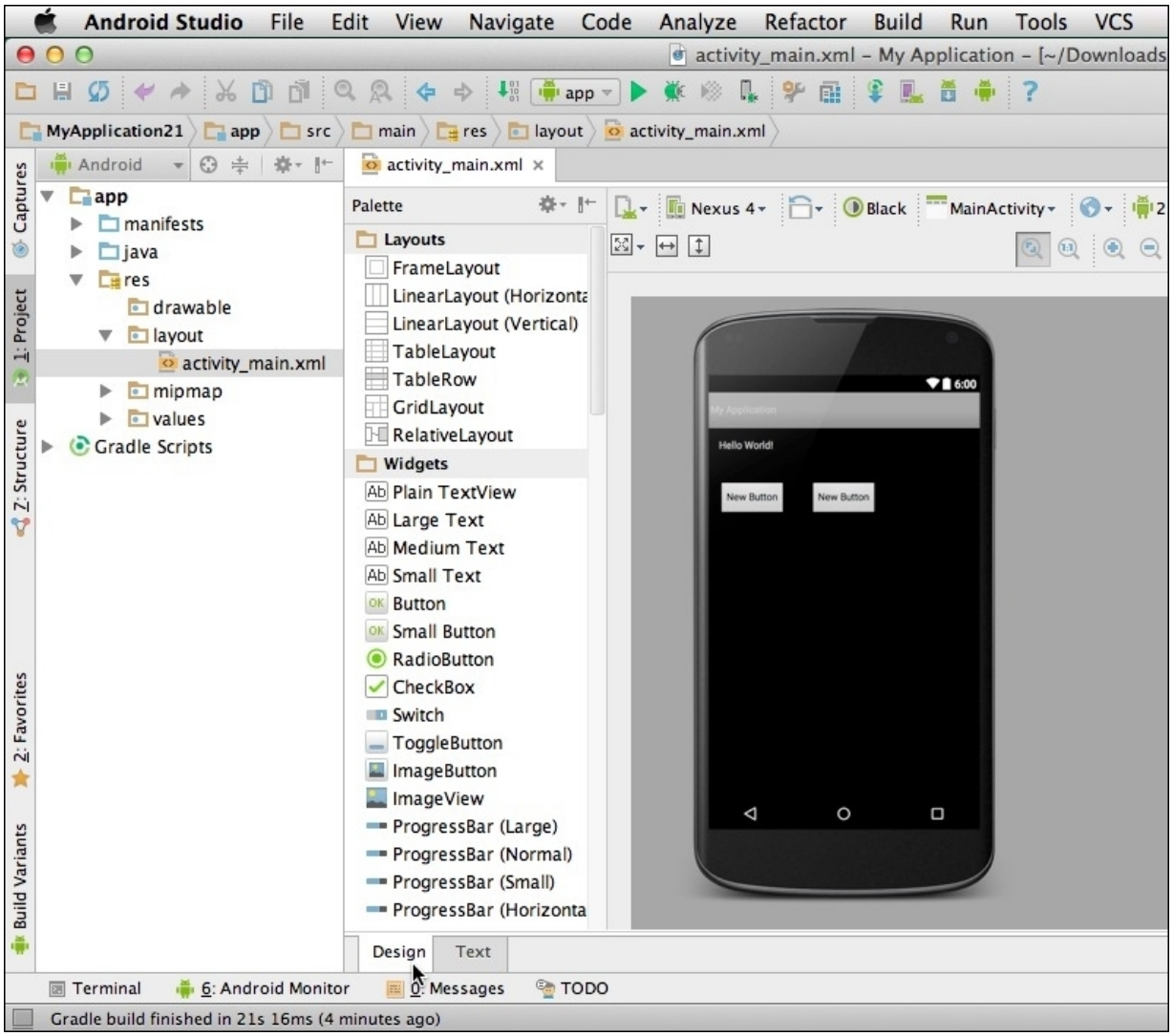
As it's the recommended development environment for Android, this chapter focuses on creating digital prototypes using Android Studio. If you don't have Android Studio installed, you can grab it from http://developer.android.com/sdk.

If you prefer to use a different IDE, such as Eclipse, then the steps involved in creating a digital prototype will be similar, so you will still be able to follow along in the IDE of your choice.

Creating a prototype in Android Studio

Conveniently, one of the best tools for creating Android prototypes is something you probably already have installed—Android Studio.

This IDE features an advanced layout editor that's ideal for rapidly creating digital prototypes. This layout editor automatically opens whenever you open any XML layout resource file in Android Studio.



The quickest way of creating a digital prototype is to select the editor's **Design** tab (where the cursor is positioned in the previous screenshot). A new area will open displaying a **canvas** preview of your app's layout, plus a **Palette** containing lots of ready-made UI elements called **widgets**. You can quickly create a prototype by dragging widgets from the **Palette** and dropping them onto the canvas.

Note

When creating prototypes (and Android apps in general), you should use the default widgets unless you have a very good reason not to. Not only do these ready-made widgets make your life as a developer easier, but most Android users will already be familiar with them. So they'll instantly know how to interact with at least some elements of your app's UI.

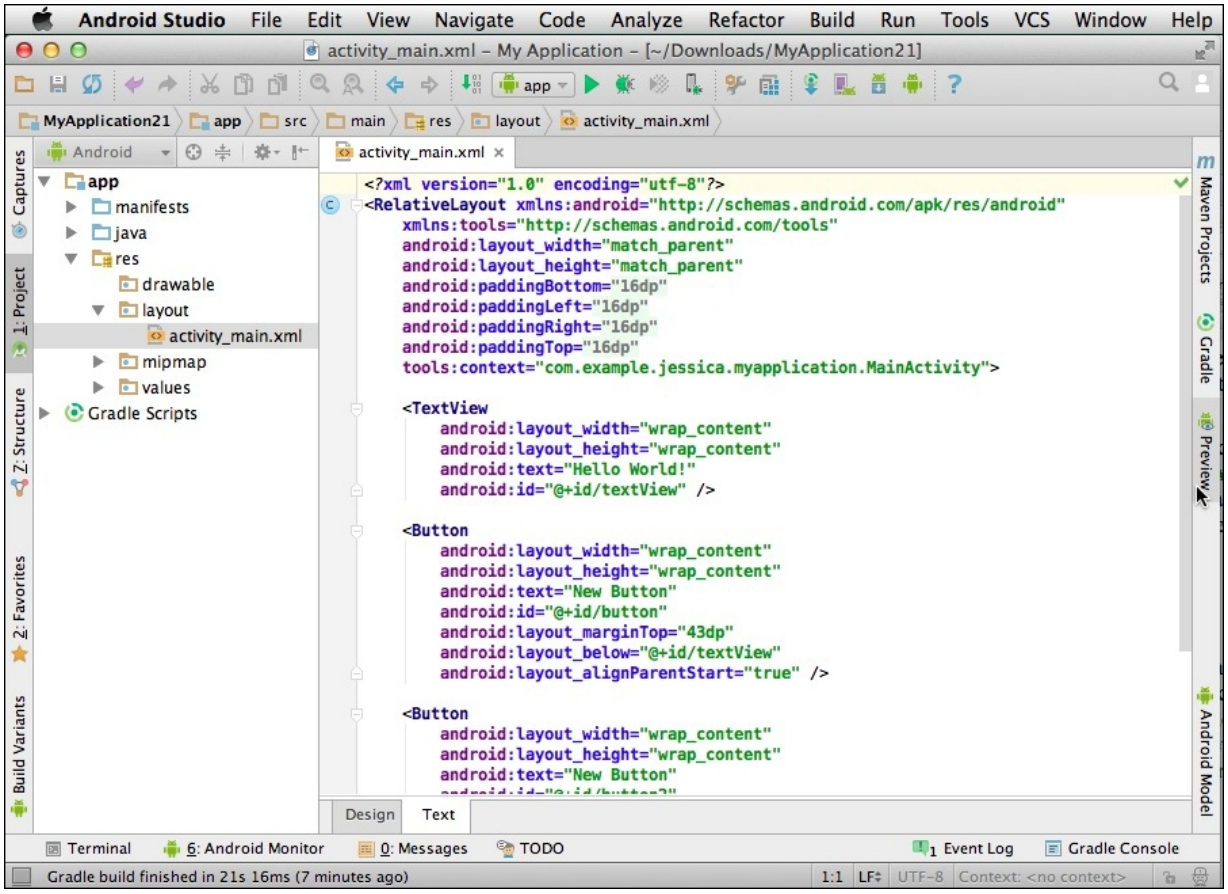
If you need to make more specific changes to your app's layout, you can access its underlying XML by selecting the **Text** tab, making your tweaks at the code level, and then nipping back to the **Design** tab to see these changes rendered on the canvas. Using this approach, you'll have a working prototype in no time, which you can then test on your own Android device or in an emulator.

Creating prototypes in Android Studio is also a great way of testing how your design looks and functions across a range of different screen configurations. If you simply happen to have multiple Android devices lying around, you can install your prototype on all of them, but most Android developers test their project across difference devices by creating multiple **Android Virtual Devices** (**AVDs**).

Note

We'll explore deploying and testing your digital prototypes in more detail in the next chapter.

You can also get a quick preview of your prototype across different devices using Android Studio's built-in **Preview** pane. To access this pane, make sure the **Text** tab is selected, and then click on the **Preview** tab along the left-hand side of the screen (where the cursor is positioned in the following screenshot):



With this tab open, you'll see a preview of your app as it appears on a single Android device. To check how your layout will appear on a different Android device, click on the name of the currently selected device (where the cursor is positioned in the following screenshot), and then choose a new device from the drop-down menu. You can even preview your UI across multiple devices at once by selecting **Preview all screen sizes** from the drop-down menu:

Creating your first prototype

You can start by prototyping any screen in your app, but I find that it makes sense to start with the first screen users see when they

launch your app. In our recipe app example, this is the home screen, so this is the first screen I'm going to prototype.

Referring back to my wireframe, I can see that the home screen should include a welcome message and photos from some of the recipes featured in the app. I also want to include a menu, so users can easily navigate to the app's most important screens. Specifically, I want this menu to include a link to the search screen, the user's recipe scrapbook, and all the different recipe categories.

Since this menu needs to include quite a few options, I'm going to use a navigation drawer. A navigation drawer is an effective solution here, as it's scrollable and can be neatly tucked out of the way when it's no longer needed.

Note

If you're not familiar with the concept of navigation drawers, then you may want to look at Android's Material Design guidelines, which has a whole section dedicated to navigation drawers https://www.google.com/design/spec/patterns/navigation-drawer.html .

Boot up Android Studio and create a new project. Choose whatever settings make the most sense for your particular project, but if you're following along with this example, then select the **Navigation Drawer** template. As you might already have guessed from the name, this template has a built-in navigation drawer, which will make creating your home screen digital prototype much easier.

Click on **Finish** to go ahead and create your project. By default, this project already includes quite a bit of code and resources. My mission is to modify all this automatically generated code to match my home screen wireframe. Let's start with the most straightforward task: creating the screen's welcome message and adding our images.

Open the project's `content_main.xml` file and make sure it's using `RelativeLayout` as the parent container. Then, select the **Design** tab and drag one `TextView` and two `ImageView` widgets from the **Palette**, and drop them onto your canvas.

Next, populate these widgets with text and images. Create a string resource that contains the welcome text of your choice. Find two images you want to use and add them to your project's `drawable` folder (in my project, I'm using `mushroomtoast.jpeg` and `sundayroast.jpeg`). Update `TextView` to display the string resource, and update `ImageViews` to display your chosen images.

Tip

Digital prototyping: Don't waste your time!

A this point, you may be tempted to invest lots of time positioning your `ImageView` and `TextView` objects so that they look *exactly* as they should in your finished app. However, be wary of spending too much time perfecting your prototypes.

The key thing to remember is that the purpose of digital prototyping isn't to create an early version of your project but to test the *theory* of your project's design. If you spend too much time getting your digital prototypes perfect, then you might as well have started creating the first version of your app!

Your digital prototype may end up highlighting some underlying problems with your design, which means you'll need to go back and make some changes to your wireframes or even redo them completely. If this happens, then at least you can console yourself with the knowledge that it's better to uncover problems with your design at this early stage, rather than *after* you've actually started building your app. Digital prototyping is a powerful and time-saving tool in your early detection arsenal—but *only* if you don't invest lots of time in perfecting your prototypes.

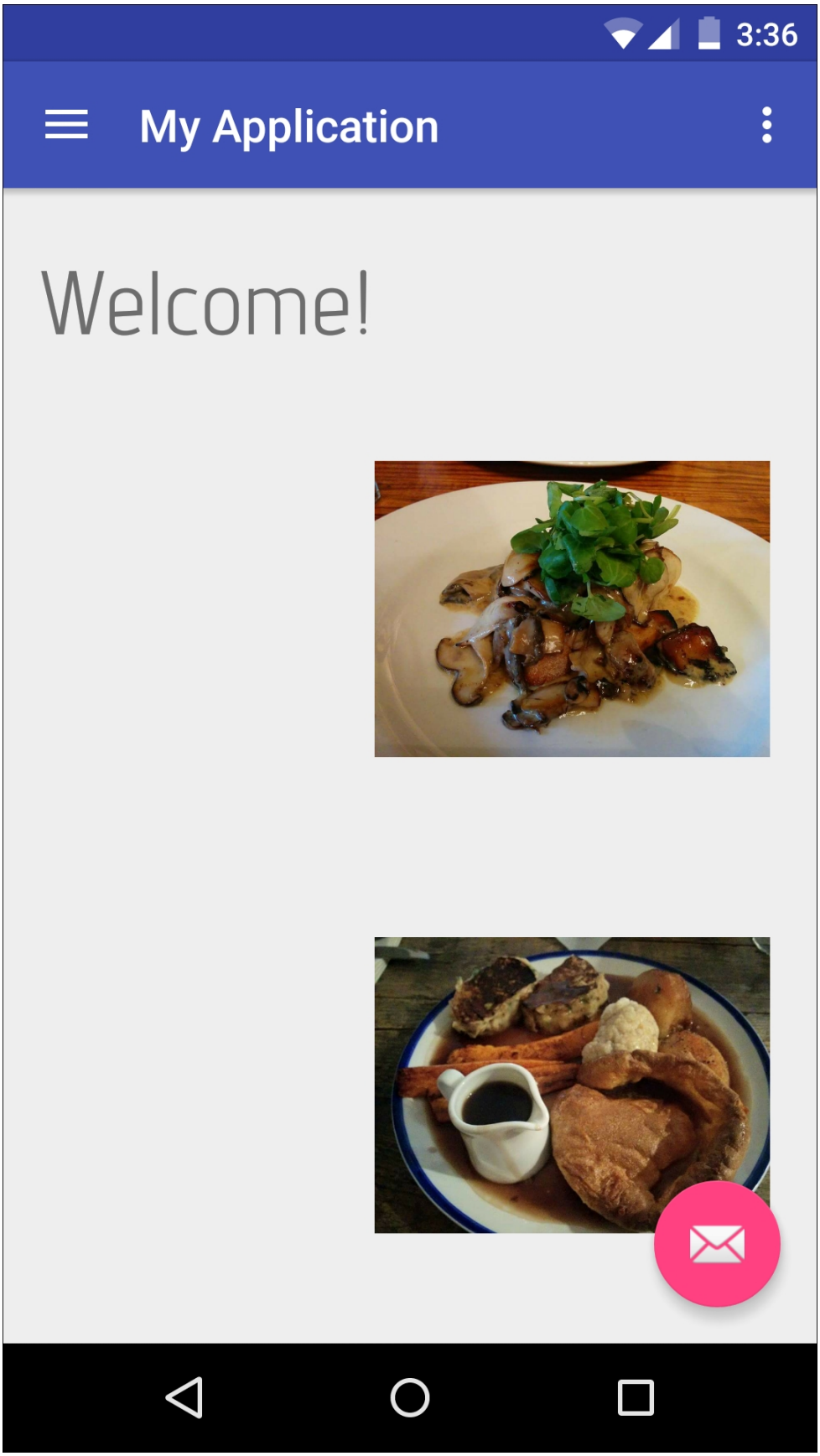
Don't fall into the trap of worrying about the finer details; instead, aim to create your prototypes as quickly as possible. This usually means creating screens that are only a rough representation of how the finished screen will look, and that typically has little or no functionality.

Here is the XML for my home screen prototype. Since this is only a prototype, I haven't invested too much time in aligning each UI element:

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout
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:paddingBottom="@dimen/activity_vertical_ma
rgin"

android:paddingLeft="@dimen/activity_horizontal_ma
rgin"

android:paddingRight="@dimen/activity_horizontal_m
argin"

android:paddingTop="@dimen/activity_vertical_margi
n"

app:layout_behavior="@string/appbar_scrolling_view
_behavior"

tools:context="com.example.jessica.myapplication.M
ainActivity"
  tools:showIn="@layout/app_bar_main">

  <ImageView
      android:layout_width="175dp"
      android:layout_height="225dp"
      android:id="@+id/imageView2"
      android:src="@drawable/mushroomtoast"
```

```
        android:layout_below="@+id/textView2"
        android:layout_toEndOf="@+id/textView2" />

    <ImageView
        android:layout_width="175dp"
        android:layout_height="225dp"
        android:id="@+id/imageView3"
        android:src="@drawable/sundayroast"
        android:layout_below="@+id/imageView2"
        android:layout_alignStart="@+id/imageView2"
/>

    <TextView
        android:paddingTop="10dp"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:textSize="40sp"
        android:text="@string/welcome!"
        android:id="@+id/textView2"
        android:layout_alignParentTop="true"
        android:layout_alignParentStart="true" />

</RelativeLayout>
```

Check how this looks on your emulator or Android device:


3:36
My Application
Welcome!

The next task is more complex: adding our own menu items to the navigation drawer.

You'll find all the navigation drawer code in the project's `res/menu/activity_main_drawer.xml` file. There's already quite a bit of code in `activity_main_drawer.xml`, but this code nicely illustrates how to add items to the navigation drawer, so rather than simply deleting it, let's use it as our template.

Currently, the `activity_main_drawer.xml` file looks like this:

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

  <group

//This is an optional, invisible container that
groups related <items> together in the navigation
drawer//

android:checkableBehavior="single">

//This is another optional element that defines
the checkable behavior for either individual menu
items (via the android:checkable attribute in an
<item> element) or for an entire group, via
android:checkableBehavior in a <group> element. In
this example, we're marking all <items> in this
<group> as checkable. The "single" value means
that the user can only select one item at a time,
within this group. The other potential values are
"none" (the user can't check any items in the
group) or "all" (the user can select multiple
items in this group at any one time)//

    <item

//Creates a new menu item//

          android:id="@+id/nav_camera"

//Assigns the menu item a unique resource ID so
the app recognizes when the user selects this item
```

```
in the menu//

          android:icon="@drawable/ic_menu_camera"

//Defines what drawable to use as this item's
icon//

          android: />

//The text you want to display in the navigation
drawer//

      <item
     ............
     ............
     ............

  </group>

//This is the end of this particular group//

  <item android:>
      <menu>

//This is the start of a new group//

     ............
     ............
     ............

      </menu>
  </item>

</menu>
```

As already mentioned, the menu items I want to add are the scrapbook, the search screen, and the different recipe categories. The first task is deciding what icons I want to use for each of these menu items.

Wherever possible, you should use the system icons as most users will already be familiar with them, and will therefore know what they mean. You'll find the entire list of Material Design system icons at https://design.google.com/icons/ .

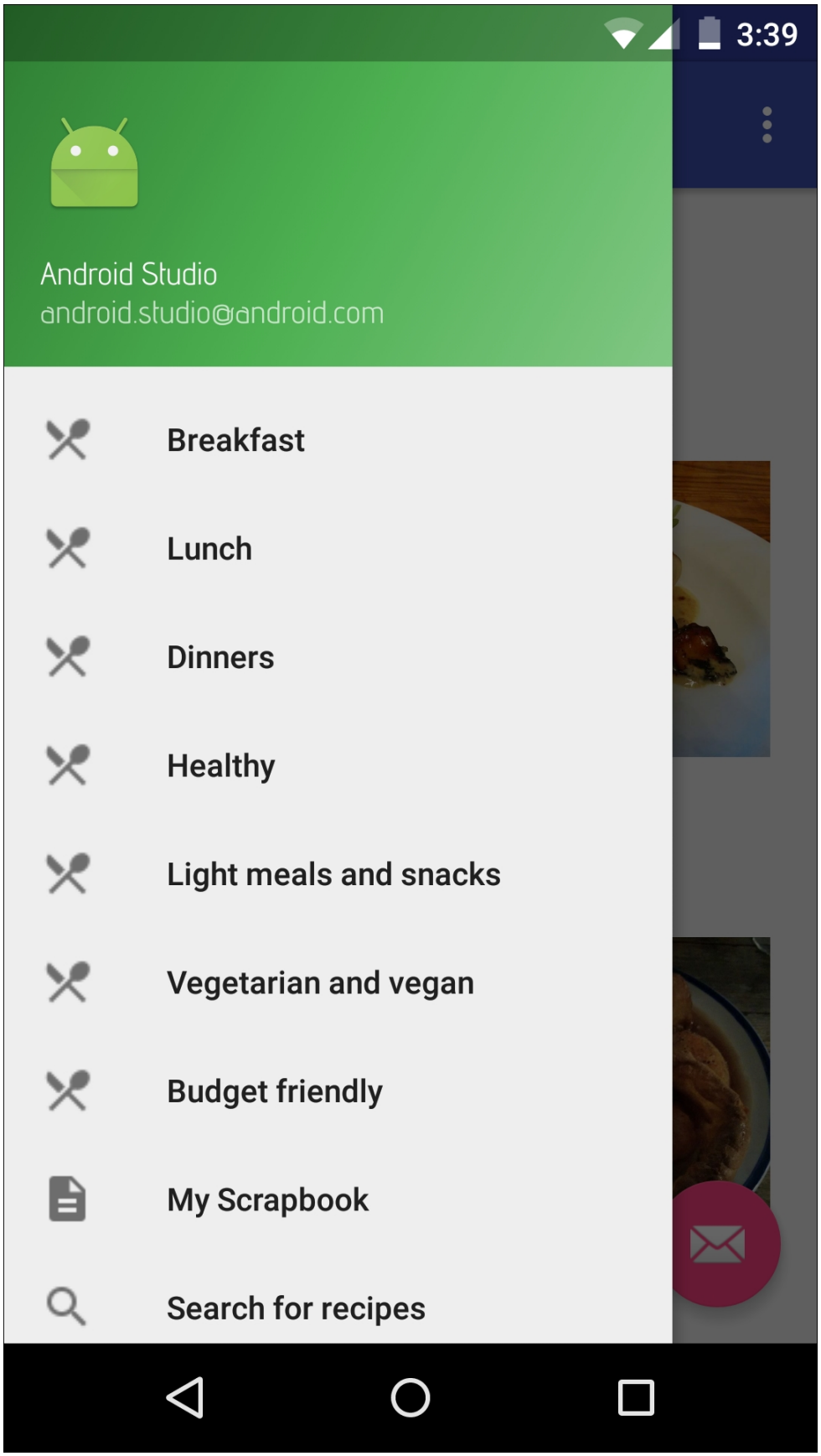
From this long list of icons, I'm going to use the standard Search icon, so download this image and drop it into your project's `drawable` folder. Finding icons for all the other items isn't quite as straightforward, but since this is just a prototype, I'm going to use some placeholder icons for now. I have decided to use the Description icon as it kind of looks like a notebook (or a scrapbook), and I'm also going to use the knife-and-fork local dining icon for each of the recipe categories.

Once all these images are safely tucked away in the `drawable` folder, it's time to add a few new entries to the navigation drawer code:

```
<?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/breakfast"
          android:icon="@drawable/ic_local_dining"
          android: />
      <item
          android:id="@+id/lunch"
          android:icon="@drawable/ic_local_dining"
          android: />
      <item
          android:id="@+id/dinners"
          android:icon="@drawable/ic_local_dining"
          android: />
      <item
          android:id="@+id/healthy"
          android:icon="@drawable/ic_local_dining"
          android: />
      <item
          android:id="@+id/lightmeals"
          android:icon="@drawable/ic_local_dining"
          android: />
      <item
          android:id="@+id/vegan"
          android:icon="@drawable/ic_local_dining"
          android: />
      <item
          android:id="@+id/budget"
```

```
            android:icon="@drawable/ic_local_dining"
            android: />

    </group>

    <item
        android:id="@+id/scrapbook"
        android:icon="@drawable/ic_scrapbook"
        android: />
    <item
        android:id="@+id/search"
        android:icon="@drawable/ic_search"
        android: />
       ............
       ............
       ............

</menu>
```

And that's it! Run the finished prototype on your Android device or emulator, but this time, be sure to open the navigation drawer by dragging it on the screen. You will see some new additions to this drawer:


3:39
Android Studio
android.studio@android.com
Breakfast
Lunch
Dinners
Healthy
Light meals and snacks
Vegetarian and vegan
Budget friendly
My Scrapbook
Search for recipes

Creating your second prototype

Wireframing gives you a clearer picture of your UI, but sometimes even wireframes can't effectively illustrate how *every* version of a certain screen will look. This is the case with our app's search results screen, as this screen will vary depending on what the user has searched for. So, how do we create a single prototype that effectively represents our search results screen?

The answer is to create a flexible prototype layout, and feed it some fake data. This fake data should accurately represent the complete range of real-life content this screen will ultimately have to display. You should go out of your way to find the most awkward examples of your app's content, so you can really put your design to the test. In our recipe app example, this means using recipes that have unusually long or short titles.

To create a prototype of the tricky search results screen, I'm going to create `ListView`, and then I will create the arrays of sample recipe titles and images, which I'll feed to `ListView`. The `ListView` will then display these arrays in my layout as though they're real search results.

Once you've created your first prototype, you can either add more screens to this project or create each subsequent prototype as a separate project. Personally, I feel like the latter is the most straightforward option, so I'm going to create my search results prototype as a new project—this time using a blank template.

If you also decide to prototype each screen as a separate project, then make sure you give each project a different package name, so you can install and test them all on the same Android device.

Referring back to my wireframes, I can see that the search results screen needs to include a list of recipe titles and accompanying images along with a header. Let's start with the most straightforward

task: creating the header. Open your project's `strings.xml` file and create a string that'll provide the search screen's header text:

```
<string name="sResults">Search Results...
</string>)
```

Open the `activity_main.xml` file and make sure its parent container is a vertical `LinearLayout`. Create a `TextView` and set it to display the `sResults` string.

Next it's time to tackle the part of the wireframe that makes this screen more difficult to prototype: the search results. As already mentioned, I'm going to create a `ListView` and feed it some fake data, so the first step is adding the `ListView` to our layout resource file:

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

  <TextView
      android:id="@+id/textview"
      android:text="@string/sResults"
      android:textSize="30dp"
      android:layout_width="match_parent"
      android:layout_height="wrap_content"
      android:paddingTop="10dp"
      android:paddingLeft="10dp"/>

  <ListView
      android:id="@+id/listview"
      android:layout_width="match_parent"
      android:layout_height="match_parent"
      />

</LinearLayout>
```

Tip

ListViews, adapters, and list items

A `ListView` is a view group that displays items as a vertical, scrollable list. Users can select any item in the list by giving it a tap, which makes `ListViews` perfect for displaying our search results, as eventually we want the user to be able to open a recipe by tapping its title.

To see an example of a `ListView` in action, grab your Android device and take a look at the stock **Contacts** app. It contains a scrollable list of all your contacts; you can tap any item in the list to see more information about that particular contact. Sound familiar?

Once you've identified one `ListView`, you'll start noticing them *everywhere*. Flick through a few of the apps installed on your device, and chances are you'll be surprised by just how often you come across `ListViews`.

`ListViews` are made up of *list items*. Each list item is displayed as a row in the `ListView`, and these items have their own layout that you define in a separate XML file. You can create a simple layout for your list items, or you can also create more complex list items that feature multiple pieces of text and several images arranged in a `RelativeLayout`.

The final piece of the puzzle is an adapter that acts as a bridge between the `ListView` and its underlying data. The adapter pulls content from a specified source, such as an array, and adapts this content into views. The adapter then takes each view and places it inside the `ListView` as an individual row.

In this chapter, I'm going to be using `SimpleAdapter`, which is a type of adapter that maps static data to views defined in an XML file. This is a pretty straightforward adapter, but it does have one

catch: `SimpleAdapter` require an `ArrayList` of `Maps` in order to define each row in the `ListView`.

If you want to learn more about `ListViews` and the different kinds of adapters, the best place to go is the official Android docs: http://developer.android.com/guide/topics/ui/layout/listview.html.

Let's define the layout of the individual items/rows that'll appear in our `ListView`. Create a new layout resource file called `simple_list_layout.xml`, and then create `TextView` and `ImageView` that will eventually hold the recipe titles and accompanying images:

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

  <ImageView
      android:id="@+id/images"
      android:layout_width="250dp"
      android:layout_height="150dp"
      android:paddingTop="10dp"
      android:paddingRight="5dp"
      android:paddingBottom="10dp"  />

      <TextView
          android:id="@+id/recipe"
          android:layout_width="wrap_content"
          android:layout_height="wrap_content"
          android:textSize="20dp"
          android:paddingRight="10dp"/>

  </LinearLayout>
```

Next, add all the images for your fake search results. I'm adding the following to my project's `drawable` folder:

- blueberrypancake.jpg
- brownies.jpg
- calamari.jpg
- chilli.jpg
- fryup.jpg
- kashmiricurry.jpeg
- pie.jpeg
- redberrypancake.jpeg
- scallops.jpeg
- surfandturf.jpg
- sweetandsourprawns.jpeg
- tuna.jpeg

I'm going to add all these images plus the corresponding recipe titles to separate arrays. Then, I'm going to feed these arrays to the `ListView` via an adapter.

> ## Note
>
> This is much more complex than the first prototype, so to speed things up I'm writing this code as quickly as possible, forsaking a bit of efficiency and optimization along the way.

Open your project's `MainActivity.java` and add the following:

```
package com.example.jessica.myapplication;

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

import android.app.Activity;
import android.widget.ListView;
import android.os.Bundle;
import android.widget.SimpleAdapter;

public class MainActivity extends Activity {
```

```
    String[] recipes = new String[] {

//Create a recipes array//

          "Easy veggie chilli",
          "Deep fried calamari with garlic and
lemon mayo",
          "Best-ever chocolate brownies",
          "Everyday fish pie with cheesy sauce",
          "Seared tuna with stir-fried veggies",
          "American style blueberry pancakes with
strawberry and banana",
          "Full English fry up",
          "Kashmiri curry",
          "Red berry pancakes with cream",
          "Sticky sweet and sour prawns",
          "Surf and turf for two"
  };

//Add all the preceding titles to the recipes
array//

  int[] images = new int[]{

//Create an images array//

          R.drawable.chilli,
          R.drawable.calamari,
          R.drawable.brownies,
          R.drawable.pie,
          R.drawable.tuna,
          R.drawable.blueberrypancake,
          R.drawable.fryup,
          R.drawable.kashmiricurry,
          R.drawable.redberrypancake,
          R.drawable.sweetandsourprawns,
          R.drawable.surfandturf
  };

//Add all the preceding image files to the images
array//

  @Override
  public void onCreate(Bundle savedInstanceState)
{
      super.onCreate(savedInstanceState);
      setContentView(R.layout.activity_main);
```

```
        List<HashMap<String,String>> aList = new
        ArrayList<HashMap<String,String>>();

//Create an array list, called aList//

        for(int i=0;i<10;i++){
            HashMap<String, String> myMap = new
            HashMap<String,String>();
            myMap.put("recipe", recipes[i]);
            myMap.put("images",
Integer.toString(images[i]) );
            aList.add(myMap);
        }

//Add recipes and images to aList//

        String[] from = { "images","recipe", };

        int[] to = { R.id.images,R.id.recipe};

//Use the ImageView and TextView from
simple_list_layout.xm//

        SimpleAdapter adapter = new
        SimpleAdapter(getBaseContext(), aList,
        R.layout.simple_list_layout, from, to);

//Create a new SimpleAdapter and feed it three
parameters. The first parameter is the context
reference (getBaseContext); the second is the
collection of data we want to display (aList), and
the third is the layout we want to use for each
row (simple_layout_list.xml)//

        ListView listView = ( ListView )
        findViewById(R.id.listview);

//Get the ListView object//

         listView.setAdapter(adapter);

//Assign the adapter to the ListView object//

    }
}
```

Attach your Android device to your computer or boot up the emulator and take a look at the results:

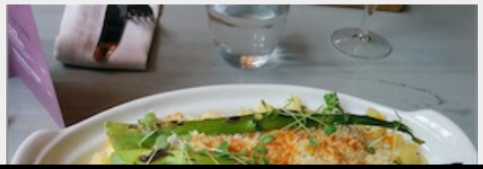
2:06

Search Results...

Easy veggie chilli

Deep fried calamari with garlic and lemon mayo

Best-ever chocolate brownies



Everyday fish pie with cheesy

Spend some time interacting with your prototype and testing it across different screen configurations, whether that's via Android Studio's preview function or by creating multiple AVDs.

Now you have two working prototypes; ask yourself whether there's any way you can improve these prototypes. In particular, be on the lookout for any opportunities to split your UI into fragments that you can then combine into multipane layouts that adapt to different screen configurations.

If your digital prototypes spark any new ideas, then grab a piece of paper and try wireframing these ideas. If these new paper-and-pencil wireframes have potential, then ideally you should put them through the same vigorous digital wireframing, paper prototyping, and usability testing stages that we covered in the previous chapter. At the very least, you should create a digital prototype of your new designs, and spend some time testing these prototypes—again—across as many different screen configurations as possible.

This may feel like taking a step backward, but this is all invaluable for ensuring that your design really is the best it possibly can be; and if it improves your app, then it's time well spent. If you release the first version of your app when you suspect you could have improved the underlying design, you're not going to have a good time—and neither are your users.

Also, that old cliché is right—you don't get a second chance to make a first impression. Your average user is going to be far more excited by an amazing app that seems to come out of nowhere, rather than version 2.0 of an app they tried ages ago but weren't too keen on.

Even if you explore your ideas and then discard them in favor of your original design, at least you'll be confident that you really are releasing the best possible version of your app. Basically, if your digital prototypes have stirred up any new ideas, then now's the time to explore them.

> **Note**
>
> If you do create a new series of wireframes and prototypes, it's always good to get a second opinion, so you may want to consider performing another round of user testing—especially if you find yourself torn between two different design directions for your app.

Once you have a set of digital prototypes that you're 100% happy with, it's time to finalize these designs.

Finalizing your design

At this point, you should have a firm grasp of the fundamentals of your app's design, including what UI elements should appear on each screen, where they'll appear on the screen, and the navigation patterns that you want to use. However, we still haven't put much thought into the finer details of our app's look and feel. What kind of font are you going to use for the headings? What color should the buttons be? Will there be any background music or sound effects? And how is your app going to use Material Design properties such as elevation and shadow?

Here are some of the things you need to consider when finalizing your design.

Visuals

This is the finer point of how your UI elements will look. Although you should try and use Android's standard widgets wherever possible, there's still lots of scope for putting your own twist on standard elements such as buttons and menus, particularly in terms of the color palette you use, plus Material Design elements such as elevation and shadow.

The ratio of your visual content is also important; should your screen feature lots of images and animation, or should it be more text heavy? As always, make sure you're designing something that'll appeal to your specific target audience. An app that features lots of images and a bright and bold palette might be appropriate for a fun app that's targeting a younger audience, but it's probably not so appropriate for an app that helps busy professionals balance their finances.

Background music and sound effects

Sound is a powerful way of setting the mood, evoking an emotional response, and giving your users an immediate feeling of what they can expect from your app.

However, don't feel like you have to include sound in your app; there's nothing wrong with silence, and in some situations, a completely silent app is preferable, especially if people are likely to use your app in situations where noise could be annoying or inappropriate.

If you do include some form of sound, always consider how integral sound should be to the user being able to successfully interact with your app. As a general rule, your app should be usable regardless of whether users have their volume turned way up, way down, or if they're in a location where they can't hear your app clearly, such as in a busy bar or restaurant.

The only exceptions are apps that fundamentally rely on sound, such as music streaming apps, an app that gives the user driving directions, or some gaming apps. In these instances, you can be confident that no one will expect to be able to use your app without turning up the volume on their device.

Your text

Text is your most direct method of communicating with your users. When crafting your text, you need to consider its content along with its visual appearance.

Your text's appearance has a massive impact on how easily your users can read and understand what that text is saying, but appearance can also send your users subtle messages about the text. For example, you can use `textSize` to communicate how important each piece of text is and what category it falls into, such as whether it's a heading, a caption, or a block of code.

To create text styles that really speaks to your users, you can use the following attributes:

- **Text size**: You should decide the size of all your app's major *types* of text early in the design process and *not* on a screen-by-screen basis. By limiting yourself to a selection of predefined sizes and using them consistently, your users will quickly learn what each `textSize` means. They can then use this information to decipher each new screen they encounter in your app, since they'll instantly know which text is the subheading, the body text, and so on, based on its size. You specify the font size with the `android:textSize` attribute, for example, `android:textSize="10sp."`. The most common method of setting `textSize` in advance is to use themes and styles, which we'll discuss later in this chapter.
- **Text style**: Android supports bold, italic, and underlined effects, but make sure you use them sparingly. If you use these effects too often, you're adding lots of visual clutter that can make your text difficult to read, plus you'll end up diminishing the effect's impact. You apply bold, italic, underlined, and `bold|italic` effects using the `android:textStyle` attribute, for example, `android:textStyle="bold."`. The underlining text is a bit trickier, as you'll need to create a string resource and apply underlining to the text inside that resource, for example, `<string name="main_title"><u>This text is underlined</u>This text isn't</string>`. Then, reference this resource from your

project's layout resource file (`android:text="@string/main_title"`), and the text will appear underlined in your UI.
- **Capitalization**: Like bold, italic, and underlined effects, you should use capitalization sparingly so you don't reduce its impact. You should also avoid using all capitals, as the all-caps text is commonly interpreted as shouting, plus it's much harder for your users to read.

The next thing you need to worry about is what your perfectly styled text is actually saying.

Think of text as your app's voice. This voice should align with your app's overall design. If your design is colorful and upbeat with cheerful music, then your text should be equally friendly. If your design is a bit more edgy, then you may want to give your text more attitude.

Just don't get carried away! Most of the time, it won't make sense to go to extremes with your text, and you should opt for a plain and neutral tone instead. Remember that the tone and content of your text should reflect the rest of your app's design. So, if your UI is streamlined with a focus on getting the job done, then chances are you'll want your text to be equally to the point.

If in doubt, play it safe with a neutral tone.

Your app's personality

I've left this one until last as it's a bit of a vague concept that incorporates a little bit of everything we've discussed so far.

Your app's *personality* is a combination of lots of different design elements, from visuals, through to background music and the tone of your text.

If your app is going to make a clear and distinct impression on your users, then all these different aspects need to align. If these design

elements aren't consistent, then chances are that you're going to leave your users with the uneasy feeling that something is a little *off* about your app.

Imagine you created an app that uses a neutral, pretty pastel palette; is packed full of cutesy graphics and ends every line of cheerful, upbeat text with at least three exclamation points—but the background music is trash metal!

Take a critical look at all the design decisions you've just made so far. Do they all come together to form a cohesive whole? Or does one (or more) elements stick out like the proverbial sore thumb?

Tip

Rules are made to be broken!

Although most of the time you *will* want all your design aspects to align, occasionally you may decide to throw the user a curveball in the form of an element that intentionally clashes with the rest of your app. This technique can create some very powerful effects; for example, you might combine cheerful colors and enthusiastic text with unsettling music to create a creepy and unnerving experience, or you may want to combine cutesy graphics with dry and sarcastic text to get your users laughing.

If you do decide to break the rules, just make sure you have a clear idea of what you're trying to achieve and how you're trying to achieve it, as it's easy to get this technique wrong!

If you're struggling to decide what your app's overall personality should be, then think about what'll appeal the most to your target audience.

As always, if you're struggling for inspiration, head over to the Google Play store and hunt down a few apps that are similar to your

project or that target the same audience.

Since we want our recipe app to appeal to students, I'm going to visit the Play store and search for apps that include the word `student`. Instantly, I see a whole bunch of timetable and agenda apps, forums that specifically target university students, and homework planners. In this scenario, I'd download a few of the most highly-rated apps, and then spend some time flicking through them to get a feel for their personality. I'd also probably download a few 5 star-rated recipe apps and give them the same treatment before combining all my findings and using them to influence the look and feel of my own recipe app.

Creating themes and styles

Now that you've made some decisions regarding the finer details of your UI, it's time to think about how you might implement them. This is where themes and styles come in useful, as they're one of the quickest and easiest ways of implementing design decisions consistently across your views, activities, and even your entire application.

Once you've made these design decisions, it's a good idea to create styles and perhaps even a theme to help you implement your designs easily and consistently.

Styles and themes are essentially the same thing: a collection of properties. These properties can be anything from the color of your text, to the size of an `ImageView`, or even the `"wrap_content"` attribute. The difference isn't how you create this collection of properties, but how you *apply* them to your project:

- Styles are the groups of properties that control what a view looks like; for example, you might apply a style to a `TextView` to specify the size and color of all the text within it.
- Themes are groups of properties that you apply to an activity or even an entire application. When you apply a theme to an activity, every view inside the activity will use any property that applies to it. If you apply a theme to an app, *every* view throughout that app will use all the applicable properties from this theme. You'll typically extend another theme rather than create one from scratch.

Styles and themes do take a bit of prep work, so why should you go to the effort of creating themes and styles, when you could just apply properties to views directly? There are a few reasons:

- **It's efficient**: If you're going to use the same collection of attributes multiple times throughout an app, then defining these

properties in advance makes applying them *much* easier—as easy as typing `style="@style/captionFont"` in fact.
- **It's consistent**: Defining a collection of attributes as a style or theme helps create a consistent look and feel across your app. And as we've already discussed, users *love* consistency.
- **It's flexible**: Design changes are pretty much a fact of life, so you should expect to be tweaking your app's visuals throughout the entire development process. Themes and styles provide a central place where you can make changes once and have them appear instantly across your app. Decided that your headings should be 10dp larger? Easy, just increase the `android:textSize` attribute in your `style="@style/heading"`.

Note

Remember when we were talking about the `textSize` attribute and how using only a small selection of text sizes can help users decode your UI? This rule also applies to styles. If you use a limited number of styles, your users will quickly pick up what these styles mean, and then use this information to help them decipher new screens. To get the most out of styles (and, to a lesser extent, themes) pay attention to that old cliché: less is more.

Defining styles

To create a style (or series of styles), you need to create a `styles.xml` file in your project's `res/values` folder, if it doesn't contain this file already.

You can then create styles using the following format:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>

//<Resources> must always be the root node of your
styles.xml file //
```

```
   <style name="FooterFont"
parent="@android:style/FooterText">

//Create a new style and assign it a name. In this
example, I'm using FooterFont//

       <item
name="android:layout_width">match_parent</item>

//Add each property to your style using the <item>
element//

       <item
name="android:layout_height">wrap_content</item>

//The value of each <item> can be a string, a
color, a reference to another resource, or another
valid value. You'll find a few examples of
possible <item> elements below//

      <item
name="android:layout_width">wrap_content</item>
       <item
name="android:textColor">#ffff0000</item>
      <item name="android:textSize">12sp</item>
       <item
name="android:typeface">sans_serif</item>

   </style>

//End of the FooterFont style//

<style name="CodeFont"
parent="@android:style/TextAppearance.Medium">

//Start of a new CodeFont style//

     ............
     ............
     ............

//This is where you'd define the CodeFont
attributes//

   </style>

//End of the CodeFont style//
```

```
</resources>
```

Okay, so this may look like a lot of work, but bear in mind that you wouldn't create a style to use just once. Typically, you'll define a style in advance, and then use it multiple times throughout your project. Plus once you've defined a style, you can use inheritance to create variations of this style (which we'll explore in more detail later in this chapter).

Tip

What's in a name?

Put some thought into what you call each style, as a style's name can communicate valuable information about its purpose and relationship with other styles.

For the best results, always name your styles based on their purpose rather than their appearance, as this might change as you refine your UI.

As an example, `CaptionStyle` is a good name but `ItalicsLightCaption` isn't, as your UI might evolve to a point where light, italicized text no longer fits with the rest of your UI. At this point, you have two options. You can continue using this inconsistent style name, which is potentially confusing, especially if you're collaborating on this project with others.

Alternatively, you'll need to open `styles.xml`, and give your style a new name, but you'd then also need to manually change every reference to this style throughout your project. As you can see, neither of these are ideal solutions!

Once you've created your style, applying it to a view is straightforward; just open an XML layout resource file and use the

`@style` attribute, followed by the name of the style you want to apply:

```
<TextView
   style="@style/CodeFont"
   android:text="@string/helloworld" />
```

Not all views accept the same style attributes; for example, an `ImageView` won't accept an `android:textAlignment`. The official Android docs is the best place to check what properties a particular view supports, specifically the view's corresponding class reference where you'll find a table of supported XML attributes. For example, if you were creating a text style, then you might want to take a look at the `TextView` class reference (http://developer.android.com/reference/android/widget/TextView.html). Also, bear in mind that some views extend other views, so if you were creating a style to apply to `EditText`, it'd be worth taking a look at the `TextView` class reference too, as `EditText` extends the `TextView` class.

Sometimes, you'll inadvertently apply incompatible properties to a view, particularly when you're applying a style to an entire activity, or when you're applying a theme. So, what happens then? The answer is *not much*. A view will only accept properties that it supports, and it'll ignore the rest.

Note

When you apply a style to a view, this style is applied to that view only. If you apply a style to `ViewGroup`, the child view elements won't inherit the style's properties. If you want to apply a style to multiple views at once, then you should apply the style as a theme instead.

Some style properties are not supported by any view, and you can only apply them as a theme to an activity or across an entire app. For example, properties that hide the application title won't have any

impact when applied to a single view. These attributes are easy to spot as they all begin with `window`, such as `windowNotTitle`, `windowTitleSize`, and `windowNoDisplay`. You'll find the complete list at the `R.attr` reference that's available as part of the official Android docs at http://developer.android.com/reference/android/R.attr.html.

Inheritance

You can quickly and easily create variations on existing styles by using them as **parent styles** and inheriting their attributes.

Use the `<parent>` attribute to specify the style you want to inherit from. You can then add properties and change the existing properties by overriding them with new values. You can inherit from the styles that you've created yourself or from the styles that are built into the Android platform, such as the following:

```
<style name="MyAppTheme"
parent="Theme.AppCompat.Light.DarkActionBar">

//Creates a new style/theme called MyAppTheme that
inherits from the
Theme.AppCompat.Light.DarkActionBar platform
theme. MyAppTheme will inherit all the
characteristics of its parent//

  <item
name="colorPrimary">@color/colorPrimary</item>

  <item
name="colorPrimaryDark">@color/colorPrimaryDark</i
tem>
  <item
name="colorAccent">@color/colorAccent</item>

//Define whatever properties you want to add or
change. Here, I'm overriding the parent theme's
colorPrimary, colorPrimaryDark and colorAccent
values with values I've defined in my project's
colors.xml file//
```

```
</style>
```

If you want to inherit from the styles that you've defined yourself, you don't have to use the `<parent>` attribute. However, it's generally considered best practice to prefix the name of the parent style to the name of your new style, and separate the two with a full stop.

Imagine I wanted to create a new style that inherits from a custom `FooterText` style which I created earlier, but that increases the `textsize` from 12sp to 20sp. I'd do this with the following code:

```
   <style name="FooterText.Large">

//Notice there's no parent attribute in the
<style> tag. Instead, I'm using the name of the
original style (FooterText), and affixing the name
of the new style (Large)//

            <item
name="android:textSize">20sp</item>
   </style>
```

You reference this inherited style in exactly the same way as any other style; in this instance that's `style="@style/FooterText.Large"`.

You can continue inheriting like this as many times as you like; just keep chaining these style names together with full stops. There's nothing stopping you from creating `FooterText.Large.Blue`, `FooterText.Large.Italic`, `FooterText.Large.Blue.Bold`, and so on.

However, to make it easier to remember all your style names correctly and help distinguish between styles, it's a good idea not to get carried away! After all, `FooterText.Large.Blue.Bold.Underlined.Monospace` isn't the easiest thing to remember.

> **Note**
>
> Inheriting properties by chaining style names together only works for styles you create yourself. If you want to inherit from Android's built-in styles, you need to use the parent attribute.

Working with themes

At this point you may be wondering why we've spent so much time focusing on styles without really discussing themes. This is because you define a theme in exactly the same way you define a style. You could take any of the styles we've defined above, and apply them as a theme.

To apply a theme, open your project's `Manifest.xml` and add the `android:theme` attribute to either of the following:

- **The <application> tag**: If you want to apply a theme across your entire app, find the `<application>` tag and add the `android:theme` attribute, plus your style name, for example, `<application android:theme="@style/FooterFont">`
- **The <activity> tag**: To apply your theme to a specific activity, add `android:theme` to the relevant `<activity>` tag

When you apply a theme, every view within the activity or across the app will use all the theme properties that it can support. If you apply `FooterFont` to an activity, all `TextViews` (and views that extend `TextView`) within that activity will use this collection of properties. If you apply `FooterFont` across your app, all `TextViews` and views that extend `TextView` will assume these properties, but any view that *doesn't* support the `FooterFont` properties will simply ignore them. And if a view supports some of the `FooterFont` properties and not others, it'll apply the supported properties and ignore the rest.

To save you time and effort, the Android platform includes many predefined themes that you can use, including a selection of Material

themes. You'll find a list of all the available styles in the **R**.style section of the official Android docs (http://developer.android.com/reference/android/R.style.html). To use any of the platform styles and themes, replace all the underscores in the style name with a period, so `Theme_Dialog` becomes `"@android:style/Theme.Dialog"`.

If you want to customize a built-in theme, you should use the theme as the `<parent>` tag of your own custom theme, and then add your own properties or override the existing properties, as follows:

```
<style name="MyTheme"
parent="android:Theme.Light">

//Create a new style called MyTheme, that inherits
from the android:Theme.Light platform theme//

   <item

name="android:windowBackground">@color/custom_them
e_color</item>

//This is where you add your own attributes, or
override existing attributes//

   <item
name="android:colorBackground">@color/custom_theme
_color</item>

</style>
```

The final step is updating your project's `Manifest` to use your new theme. So, if you wanted to use `MyTheme` in a specific activity, you'd write the following:

```
<activity android:theme="@style/CustomTheme">
```

To use `MyTheme` across your entire application, you'd write the following:

```
<application android:theme="@style/CustomTheme">
```

Get ready for errors!

In a perfect world, users will only ever interact with your app under ideal conditions. If your app pulls content from the Internet, people will only ever open your app when they have a lightning-fast connection to the World Wide Web. Your app will never have to deal with interruptions such as pesky incoming calls or SMS messages, and your users will certainly *never* enter their username or password incorrectly.

Sadly, this isn't the way the real world works. Your users will make mistakes, run out of storage space, lose their Internet connection, and get phone calls when they're midway through the final boss battle in your RPG app. Your app must be able to handle all these less-than-ideal conditions and errors.

It may seem strange to plan for errors, but unfortunately errors happen to the best of us. Not only should your app be able to handle errors when they do occur, but it should handle them *gracefully*.

In the final section of this chapter, we'll look at how to design your app so that it can handle whatever the real world throws at it.

User input errors

User input errors are where users enter incorrect information or leave a field blank. The most common places where user errors occur are forms such as login screens or payment screens.

Assuming that you clearly communicated what information the user needed to enter, technically these errors aren't your fault, but you should still design your app to be able to cope with these kinds of error.

When user errors occur, your app should clearly communicate what's happened and give the user enough information to be able to

fix the problem. Simply displaying a popup that says *Error* isn't going to cut it.

Ideally, your error messages will include an action that helps the user fix the error. In our popup example, once the user has dismissed the popup, our app could automatically select the `EditText` where the user had entered the incorrect information, ready for them to start typing.

You should only offer actions that you can actually support, so if the error is caused by the user running out of storage space, then offering them a **Try Again** button isn't going to solve the problem. If you can't help the user resolve an error, it's far better to explain to them what the error is, what's caused it, and how they can fix it, rather than offering an empty gesture that's probably going to leave them feeling even more frustrated.

When your app does encounter user error, it should try to preserve as much user-entered information as possible. So, if a user fills in a form but then enter their password incorrectly, your app should preserve their username, address, and any other fields they've filled in correctly, while highlighting the fact that they've entered the wrong password.

However, while you handle user input errors, don't overlook the importance of good design. If your app is going to fail, then it might as well fail in style! A well-designed error message should feel like a natural extension of your app. In our popup example, we could edit the popup so its colors better complement our app's overall color scheme, or we could change the font or include a custom sound effect that reflects our app's personality.

Even though I've used a popup message as an example, you should always look for ways to make your error messages as unobtrusive as possible, and popups aren't particularly subtle. More unobtrusive methods include underlining, highlighting, automatically selecting a field where the user has entered something incorrectly, reloading the screen with the addition of some text explaining the error that's

occurred, or taking the user back to the screen where they have entered the wrong info.

Although user input is one of the most common causes of errors, there are other errors and less-than-ideal conditions that your app needs to be able to handle. These include the following:

- **Lack of connectivity**: If the user is offline and tries to access a feature that requires the Internet or network access, you should display an unobtrusive message explaining that they need an Internet connection in order to complete this action.
- **Incompatible state**: These errors occur when the user attempts to run conflicting operations, such as trying to send a message via an online messaging app when they're offline. When these errors occur, your app should display a message explaining the error and ideally provide the user with an easy way of changing their current state.
- **Missing permissions**: If the user selects an action that requires a certain permission, your app should notify the user and give them the option of granting that permission. If the user denies this permission request, you should disable all related features, so the user doesn't keep trying to access features that your app can't currently support. You should also give the user an easy way of granting previously denied permissions, just in case they change their mind. If certain permissions are essential for your app, you should request them in advance. We'll look at permissions in much more detail in Chapter 10, *Best Practices and Securing Your Application*.
- **Empty states:** These occur when your app can't display regular content; for example, a `ListView` that contains no search results. You should design your app to be able to deal with missing content, although the best way to handle the situation will vary depending on the context and the type of content that's missing. In our search results example, you could create a **No results found** screen, or the app could automatically widen the search query and display content that's the best possible match. For example, if the user searches for `burito`, your app could display

all recipes that include the word `burrito` instead of displaying nothing.

    - If you do decide to take a best-possible-match approach, then you should signal that this content isn't an exact match. In our search results example, we might display a message along the lines of **"We couldn't find any results for burito, did you mean burrito?"**
- **Content unavailable:** This is subtly different to empty states as it involves content that should appear in your app but for some reason isn't available. For example, perhaps your UI includes images or videos that need to be downloaded from the Internet, but the user is currently offline. If your app includes this kind of content, you should create placeholders that'll appear in your layout when the content is unavailable. If you don't, then there's the risk that your layout will suddenly jump about as the missing content becomes available and is inserted into the layout, something you may have encountered before when browsing slow-to-load webpages on your mobile device. Creating placeholders also ensures that your layout retains its intended structure even when some of its content is missing.

Summary

In this chapter, you learned how to quickly create digital prototypes based on your wireframes. Although you created paper prototypes previously, digital prototyping is particularly useful for testing how your screen designs will look and function on a real Android device and across different screen configurations.

The most important thing to take away from this chapter is that wireframes are a way of planning how your screens will look and function in *theory*, but digital prototypes are how you put that theory to the test. Sometimes, you may find that the theory doesn't work in practice and that you need to make some changes to your wireframes or even redo them completely, but that's okay—discovering problems with your design is all part of creating the best possible version of your app.

Now that you've finalized your design, in the next chapter we'll take a closer look at how you can ensure this newly finalized design translates correctly across as many different Android devices as possible.

Chapter 8. Reaching a Wider Audience – Supporting Multiple Devices

Android is a mobile operating system that places very few restrictions on hardware. Manufacturers are free to create Android devices that are packed with high-end hardware such as DSLR-worthy cameras, huge internal memories, and lightning-fast CPUs; or they can create more budget-friendly devices that take a no-frills approach to hardware. Android screens can also come in all sorts of sizes, shapes, and screen densities.

The only thing these devices have in common is that they all run the Android operating system—and even in this, Android smartphones and tablets are inconsistent. The current Android smartphone and tablet market is made up of lots of different versions of the Android operating system—from legacy versions, right through to the very latest release of Android. And even if two devices are running exactly the same version of Android, there's no guarantee that they'll be exactly the same, as manufacturers have a nasty habit of tweaking the Android operating system to come up with their own **original equipment manufacturer** (**OEM**) versions. A Samsung smartphone running Android 7.0 may not necessarily be the same as a Sony smartphone running Android 7.0.

This flexibility is great for manufacturers who can make their devices stand out from the competition by coming up with new and innovative hardware, software, and screen configurations. It's also good news for consumers who can shop around and find exactly the Android device that's right for them.

But is this good for developers? *Sort of*.

All these variations on hardware, software, and screen configuration mean lots of opportunities to innovate and come up with a truly original app. However, it also poses a massive challenge, as you'll need to create an app that provides a consistent experience across all the different hardware, software, and screen configurations that your app might encounter.

Unfortunately, there's no quick fix. Essentially, the key to creating a flexible UI is to provide a wide range of alternate resources that are optimized for all the different hardware, software, screen configurations, languages, and region settings that your app may encounter, which is arguably the most time-consuming part of Android app development.

In this chapter, I'm going to cover all the major aspects you need to bear in mind, if you're going to create a flexible app that's compatible with as many different Android devices as possible.

Supporting different versions of Android

While most new versions of the Android platform introduce exciting new features that you'll be desperate to use in your app, if you're going to reach the widest possible audience then you'll also need to support as many older versions of the Android platform as possible.

This is a delicate balancing act. Supporting older versions of the Android platform takes time and effort, and the further you go, the harder you'll have to work to get your app playing nicely with older versions of Android.

If you continue striving to support older and older versions of Android, at some point you'll inevitably find yourself compromising your app's UI, functionality, and general user experience as you encounter versions of Android that simply aren't powerful enough to

support the kind of app you originally wanted to create. In our recipe app example, we're relying on the device being powerful enough to load multiple high-res images whenever the user performs a search. If the user is running an older version of the Android operating system, processing these images may cause the search results to load more slowly. Although the problem lies with the user's device, your typical Android user is far more likely to blame your app than their outdated smartphone or tablet.

We could shrink the images or even remove them completely, but would you want to try a recipe when you hadn't even seen a photo of the finished product? At this point, you should take a step back and ask yourself whether all this time, effort, and compromise is really *worth it*.

To identify the point at which supporting older versions of Android becomes more trouble than it's worth, you'll need to look at the current Android market—especially how many devices are running each version of the Android operating system. Once you have this information in front of you, you can make an informed decision about the point at which it no longer makes sense to keep supporting the older versions of Android.

One source of this information is Google's Dashboard (http://developer.android.com/about/dashboards/index.html), which gives you a percentage of the relative number of devices running each version of Android. Just be aware that this information is gathered in a very specific way. It's essentially a snapshot of all the devices that visited the Google Play store in the previous 7 days. This data is collected from the Google Play app, and is not necessarily representative of the current state of the entire Android market. It's also worth noting that the Google Play app is only compatible with Android 2.2 and higher, so any device running versions of Android lower than 2.2 aren't included in this data; although according to Google way back in August 2013, devices running versions lower than Android 2.2 only accounted for about 1% of devices, so we're talking about a very small percentage here.

Spend some time exploring the Dashboard data, and come to a decision about which versions of Android you're going to support and which versions you aren't.

Specifying minimum and target API levels

Once you've decided what versions of Android you're going to support, you need to include this information in your project. How you add this information will vary depending on the IDE you're using, so you'll need to open one of the following files:

- Manifest file (Eclipse)
- Module-level `build.gradle` file (Android Studio)

We will discuss the components of this file in the next section.

minSdkVersion

This attribute identifies the lowest API level that your app is compatible with, for example, `minSdkVersion 16`. Google Play will use your app's `minSdkVersion` attribute to determine whether a user can install it on device.

When debating your app's `minSdkVersion` value, make sure you consult the Dashboard stats as this provides a snapshot of your potential audience. Ultimately, you'll need to decide whether supporting each additional slice of this audience is worth additional time and effort.

targetSdkVersion

This attribute identifies the highest API level that you've tested your app against.

The `targetSdkVersion` value is particularly important for forward compatibility, as the system won't apply any behavior changes introduced in new Android releases until you update your app's `targetSdkVersion` value. To ensure your app benefits from the latest Android features, you should aim to set your app's `targetSdkVersion` value to the very latest version of Android. Updating your app to target a new SDK should always be a high priority whenever Google release a new version of Android, but you should only do this after thoroughly testing your app against the latest SDK release. *Never* blindly update your `targetSdkVersion` value without testing it first.

Ideally your `targetSdkVersion` and `compileSdkVersion` value should always correspond to the very latest version of the Android SDK.

compileSdkVersion

This attribute tells Gradle what version of Android SDK it should compile your app with.

Your `compileSdkVersion` value is not included in your published APK; it's purely used at compile time. Changing your `compileSdkVersion` value does *not* change the runtime behavior, so it's recommended you always compile with the latest SDK.

Check version at runtime

Sometimes, you'll have a clear cut-off point where it makes sense for your app to stop supporting earlier versions of Android, but this line may not always be so clear cut.

Imagine your app includes a non-essential feature that isn't supported on Android 5.0 and earlier versions, but apart from this feature, your app is compatible with earlier versions of Android. Since this Marshmallow-and-higher feature isn't essential, it doesn't make sense to prevent everyone running Android 5.0 or earlier versions from installing your app. In this scenario, you can disable this feature by ensuring any code that depends on higher API levels

is only executed when these APIs are available. Basically, this feature won't be available to users who are running Android 5.0 or a lower version, but these users will still be able to install and use your app.

You achieve this using the `Build` constants class to specify when the related code should run; for example, the following code verifies whether your app is running on Lollipop or a higher version:

```
if (Build.VERSION.SDK_INT >=
Build.VERSION_CODES.LOLLIPOP)
```

Android provides a unique code for each platform version, which you should use alongside the `Build` constants class (in this example, the code is `LOLLIPOP`). You'll find a complete list of these codes at the official Android docs (http://developer.android.com/reference/android/os/Build.VERSION_CODES.html).

Supporting different screens

Android devices come in many different shapes and sizes. Your task, as a developer, is to create a UI that looks just as good on the small space available to a budget-friendly smartphone, as it does on the large space available to a top-of-the-line Android tablet and everything in between.

So let's break it down. Android categorizes screens in two ways:

- **Screen sizes**: Traditionally, Android supported four generalized screen sizes: `small`, `normal`, `large`, and `xlarge`. However, Android 3.2 (API level 13) introduced some new configuration qualifiers that allow you to be more specific about screen sizes.
- **Screen densities**: A device's screen density is a combination of its resolution and display size, and is measured in **dots per inch** (**dpi**). The higher a device's dpi, the smaller each individual pixel, which means greater clarity and more detail per inch. Android supports six generalized densities: low (`ldpi`), medium (`mdpi`), high (`hdpi`), extra high (`xhdpi`), extra-extra-high (`xxhdpi`), and extra-extra-extra-high (`xxxhdpi`).

You can anticipate that your app will be installed on devices spanning a wide range of screen sizes and densities. And you should aim high; it's not good enough for your app to be compatible with these different screen configurations, you should give users the impression that your app was designed specifically for their particular screen, whatever size and density that screen happens to be.

In this section, I'll show you how to create an app that can handle a wide range of different sizes and densities. The underlying theme you'll encounter time and time again is that your app's layout and drawables must render at a size that's appropriate for the current screen. The Android system is clever enough to handle most of this rendering work automatically, and it will scale your layouts and

resources to fit the current size and density, but you shouldn't rely on the Android system to do all the hard work for you.

Android's automatic rendering alone won't provide the best possible user experience. You'll need to give it a hand by providing multiple versions of your app's resources that are optimized for different screen sizes and densities. These resources could be strings, layouts, graphics, or any other static resources that your app requires.

To add these resources to your project, you'll need to create alternate versions of your project's directories, and then tag them with the correct configuration qualifiers; for example, if you have a layout that's optimized for landscape orientation, you'll need to create a `res/layout-land` directory, and then place your landscape layout file inside this directory. Then, when a user runs your app, the Android system will automatically load the resource that best matches the current screen configuration, whether that's the default layout or your landscape-optimized `res/layout-land` layout.

Configuration qualifiers

Android supports a wide range of configuration qualifiers that you can append to your project's resource directories. These configuration qualifiers are the key to controlling which version of a resource the system displays.

Configuration qualifiers specify the characteristics that a resource was designed for, such as an image that was designed for a particular screen size or screen density. You'll find a complete list of valid configuration qualifiers in the official Android docs, specifically **Table 2** of the **Providing Resources** page (http://developer.android.com/guide/topics/resources/providing-resources.html#AlternativeResources).

The configuration qualifiers you'll need to use will vary depending on your app and the kind of devices you're targeting, but as a minimum,

you'll typically use size, density, and screen orientation qualifiers to provide drawable resources and layouts that are optimized for different screen configurations.

To use a configuration qualifier, create a new directory in your project's `res/` directory and name it using the following format:

```
<resources_name>-<config_qualifier>
```

So, if you were creating a directory to hold layouts that were optimized for devices held in landscape mode, you'd use the `land` qualifier and create a `res/layout-land` directory, then place your layout-optimized layouts inside this directory.

Note

Never place any resources directly inside your project's `res/` directory as this will cause a compiler error. You also cannot nest alternative resources, so you cannot create a `res/drawable/drawable-xxhdpi/` directory.

You can use more than one qualifier at a time by separating each qualifier with a dash; for example, a `res/drawable-en-hdpi` directory would contain drawable resources that are designed for devices set to the English language (`en`) with a screen density that falls into the high-density bucket. If you do use multiple qualifiers, the order they appear in your directory name is crucial. They must appear in the same order as in the **Providing Resources** page (http://developer.android.com/guide/topics/resources/providing-resources.html#AlternativeResources). If you use qualifiers in the wrong order, the Android system won't recognize the directory, and will subsequently ignore all the resources that it contains.

Don't worry about the case you use for your directory names, as the resource compiler converts directory names into lowercase before

processing anyway. If your project includes directories with long-winded names consisting of multiple configuration qualifiers, then you may want to take advantage of this automatic conversion, and use capitalization to make your directory names easier to read.

Naming is also important for the resources that you place inside your directories. When you create multiple versions of the same resource, they must all have exactly the same name as the default resource. Any variation and the Android system won't recognize them as alternate versions of the same resource.

> **Note**
>
> If your project contains resources that the system should *never* scale to suit the current screen configuration, place these resources in a directory with the `nodpi` configuration qualifier, for example, `res/drawable-nodpi`.

How Android selects the perfect resource

When your app features multiple versions of the same resource, the Android system follows a strict set of rules when deciding which version it should display at runtime.

When the Android system is looking for a resource based on the screen size or density, it starts by searching for an exact match that it can display without scaling. If it can't find a suitable size or the density-specific version, then Android switches to plan *B* and searches for a version that's designed for screens smaller than the current screen.

If the only available resources are larger than the current screen, the system will use the default version of the resource instead. The Android system assumes that your project's default resources are designed for the baseline screen size and density, which is a `normal`

size and medium-density. Therefore, the system scales default resources up for high-density or larger screens, and down for low-density screens.

If the system can't find a suitable density-specific resource, or even a default version of the required resource, then your app will crash—which is why it's essential that you provide a default version of *every* resource.

Even if you're confident that you've provided all the alternative resources your project could ever need, it's possible that your app may wind up on a device that has hardware, software, screen size, screen density, or language settings that you hadn't anticipated and therefore didn't provide specific resources for. In this scenario, the system may fall back on your project's default resources, and if your app doesn't include these default resources, then your app will crash.

> **Note**
>
> Default resources are all the resources that are stored in a directory without a configuration qualifier, such as `res/drawable`.

Creating alias resources

Sometimes, you'll have a resource that's suitable for more than one configuration qualifier; for example, you might have a drawable that you want to add to both your project's `res/drawable-hdpi` and `res/drawable-xhdpi` directories.

> **Note**

You can't create a directory that uses multiple configuration qualifiers of the same type, so it's not possible to create a `res/drawable-hdpi-xhdpi` directory.

You could copy/paste the resource so that it appears in both directories; but this isn't very efficient, plus duplicating resources increases the size of your project, which is bad news for your end users. The best solution is to use an **alias**.

Imagine you have a `scene.png` drawable that you want to use for both `hdpi` and `xhdpi` screens; this is the perfect opportunity to use an alias. In this scenario, you need to place the default version inside your project's `res/drawable` folder as normal. Then, save the version of the image you want to use for `hdpi` and `xhdpi` screens inside the `res/drawable` folder, but give it a different name to the default resource, for example, `scenery_alias.png`.

At this point, you have two drawables:

- `res/drawable/scenery.png`
- `res/drawable/scenery_alias.png`

Next, create an XML file inside both of the density-specific directories. Inside these XML files, add some code that points toward your project's `res/drawable/scenery_alias.png` resource:

```
<?xml version="1.0" encoding="utf-8"?>
<bitmap
xmlns:android="http://schemas.android.com/apk/res/
android"   android:src="@drawable/scenery_alias"
/>
```

When the Android system tries to load the scenery resource from `res/drawable-hdpi` or `res/drawable-xhdpi`, it'll recognize the alias and display `res/drawable/scenery_alias.png` instead. In this way, you can replace memory-hogging and inefficient duplicate resources with small XML files.

You can also use the alias function to reuse the same layout resource files in multiple directories, as follows:

1. Create a default layout (`main.xml`), and place it in your project's `res/layout` directory.
2. Create the layout you want to use across multiple directories. Give this layout a different name to the default layout (I'm going to use `main_alias.xml`), and place it inside your project's `res/layout` directory.
3. Create XML files in all the directories where you want to use the `layout_alias.xml` file.
4. Add some XML code that references `layout_alias.xml`:

```
<?xml version="1.0" encoding="utf-8"?>
<merge>
   <include
layout="@layout/main_alias"/>
</merge>
```

Although less commonly used, you can also create aliases for strings and other simple values, as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
   <color name="yellow"> #ffff00
</color>
   <color
name="highlightColor">@color/yellow</color>
</resources>
```

In this example, `highlightColor` is now an alias for `yellow`. You can also create aliases for strings:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
   <string name="title">Student Recipe</string>
   <string name="appTitle">@string/title</string>
</resources>
```

In this example, `appTitle` is now an alias for `title`.

Screen density

When designing your UI, you should aim to achieve **density independence**. This is where the Android system preserves the physical size of your UI elements across screens with different densities.

Density independence is *crucial* to providing a good user experience. A high-density screen has more pixels per inch, which means the same amount of pixels fits into a smaller area. Low-density screens have less pixels, so the same amount of pixels fits into a much larger area. If you use absolute units of measure, such as specifying UI elements in pixels, then your UI elements are going to appear larger on low-density screens and smaller on high-density screens.

Note

It's a common misconception that devices with the same screen resolution automatically have the same screen density. Even if two devices have the same screen resolution, these screens may be different sizes. This means the screens are displaying their content in a different amount of space, which equates to a different number of dpi.

If your UI elements change sizes on different devices, then this is going to lead to problems in your layout and usability issues, and this will make for a generally poor user experience.

In most cases, you can ensure density independence by specifying your layout dimensions in density-independent pixels, and by replacing static, hardcoded sizes with more flexible elements such as `"wrap_content"` and `"match_parent"`.

When it comes to drawables, Android automatically scales each drawable based on the current screen's density, so your drawables

render at the appropriate physical size for the current device. However, this automatic scaling can result in blurry or pixelated images. To make sure your drawables look their very best, you need to create alternate versions of each drawable, which are optimized for different screen densities.

The problem is that the Android marketplace includes more screen densities than you could *ever* hope to support; and even if you could, providing so many alternate drawables would cause the size of your project to balloon out of control, to the point where it's unlikely it'd even fit onto your average Android smartphone or tablet.

This is why Android groups screen densities into the following generalized **density buckets**:

- `ldpi` (low-density): 120dpi
- `mdpi` (medium-density): 160dpi
- `hdpi` (high-density): 240dpi
- `xhdpi` (extra-high-density): 320dpi
- `xxhdpi` (extra-extra-high-density): 480dpi
- `xxxhdpi` (extra-extra-extra-high-density): 640dpi

The first step to a flexible layout is to create a directory for each of these density buckets, for example, `res/drawable-ldpi`, `res/drawable-mdpi`, and so on. Then, just create a version of each resource that targets these different density buckets, and the Android system will handle the rest.

To create optimized drawables for each density bucket, you need to apply the 3:4:6:8:12:16 scaling ratio across the six generalized densities. For the best results, start by creating a version of the image at the largest supported density, and then scale this graphic down proportionally for each subsequent density bucket.

Note

Most of the time, creating a graphic at the largest supported density means creating an image for the `xxhdpi` density bucket and *not* the `xxxhdpi` bucket. This is because the `xxxhdpi` directory is reserved for launcher icons.

Converting dpi into pixels and vice versa

Since different screens have different pixel densities, the same number of pixels will translate into different physical sizes on different devices. Pixels aren't a density-independent unit of measurement, so 40 pixels isn't the same size on every device.

For this reason, you should never use absolute pixels to define distances or sizes. Occasionally, you may need to convert dpi values into pixel values and vice versa. You do this using the following formula: *dpi = (width in pixels * 160) / screen density*.

Note

Android uses `mdpi` (160dpi) as its baseline density where one pixel neatly equates to one density-independent pixel. This is where the 160 value in the formula comes from.

Let's try this with some numbers:

```
(180 px x 160) / 120 = 240 dpi
```

You may also want to use an online converter (http://jennift.com/dpical.html) to play around with dpi values.

If you want to convert a dpi value into pixels, then use the following formula:

```
dp * (dpi / 160) = px
```

For example, take a look at the following:

```
120 x (240 / 160) = 180
```

Providing different layouts for different screen sizes

Android supports a wide range of screen sizes, and it automatically resizes your UI to fit the current screen. As I've already mentioned, you shouldn't rely on the Android system to do all the hard work for you, as this autoscaling often doesn't make the best use of the space available on larger screens, particularly tablet-sized devices.

If you thoroughly test your app across a range of different screen sizes using the emulator and multiple **Android Virtual Devices** (**AVDs**), you may encounter your app struggling to display or function correctly on certain screens. For example, you may discover that Android's automatic scaling makes your UI look cramped once the device's screen dips below a certain dpi value; or at the other end of the scale, you may find that your UI has large areas of empty space on larger, tablet-sized screens.

If your UI is having problems with certain screen sizes, you should create layouts that are optimized for these screens.

The process for providing alternate layouts is the same as providing any alternate resource: create directories that have the appropriate configuration qualifiers. Then, create layout resource files that are optimized for specific screen sizes, and make sure these layouts have the same name as the corresponding default layout. The Android system will then select the appropriate layout based on the current screen configuration.

However, the screen size configuration qualifiers aren't as straightforward as the density configuration qualifiers, as Android 3.2 introduced some new configuration qualifiers that allow you to define the specific width and/or height required by each of your layouts, in

dpi units. These new configuration qualifiers give you greater control over your resources, but they are a bit more difficult to get your head around.

> **Note**
>
> Prior to version 3.2, Android supported screen-size groups: `small`, `normal`, `large`, and `xlarge`. In order to accommodate a greater variety of screen sizes, the Android team replaced these groups with new configuration qualifiers. This chapter focuses on the new configuration qualifiers, but if you want to find out more about the now-deprecated screen size groups, you can find more information at the official Android docs at (http://developer.android.com/guide/practices/screens_support.html).

These powerful new qualifiers are discussed in the next sections.

smallestWidth – sw<N>dp

As the name suggests, you use the `smallestWidth` qualifier to define the minimum width in dpi that must be available before the Android system can use a particular layout. For example, if your layout requires a minimum of 700dpi, the `smallestWidth` configuration qualifier would be `sw700dp`. In this scenario, you'd create a `res/layout-sw700dp` directory and place your layout inside. The system will only use this layout when the current device has at least 700dpi of available width.

This qualifier is particularly useful as width is often a major factor when designing your layouts. Many apps scroll vertically, but it's pretty rare to encounter a UI that scrolls horizontally. Most apps have strict constraints about the minimum space they need horizontally, and this configuration qualifier gives you a way of specifying that minimum as a dpi value.

A device's width is a fixed characteristic that doesn't change when the user switches between portrait and landscape mode. The user's perception of their screen's width and height may change, but the system's perception of a device's `smallestWidth` never does, even when users switch their device from portrait to landscape mode and vice versa.

Available screen width – w<N>dp

Sometimes, your app may need to react to how much width or height is currently available, which means taking the device's current orientation into consideration. For example, imagine your UI has the option to display two fragments side by side in a multi-pane layout. If the user is viewing your app in landscape mode, it makes sense to display the multi-pane layout, but as soon as the user switches the device into portrait mode, there may no longer be enough width available to display these fragments side by side.

This is where the `sw<N>dp` configuration qualifier comes in. You can use this qualifier to set the minimum width a resource requires, for example, `res/layout-w700dp`. However, unlike the `smallestWidth` qualifier, `w<N>dp` represents the width that's currently available to your UI, taking into account the current screen orientation. This allows your app to react to the width currently available to it and not just the device's fixed `smallestWidth` value.

Available screen height – h<number>dp

As already mentioned, your typical Android app is pretty flexible about the minimum height it needs. However, if you do need to specify the minimum screen height a layout or resource needs, you can use the `h<number>dp` qualifier, for example, `res/layout-h720dp`.

The value the system assigns to the screen height changes when the screen's orientation changes, so similar to `w<N>dp`, you can use this qualifier to detect whether your app is currently being viewed in portrait or landscape mode.

While using these new configuration qualifiers may appear more complicated than using the traditional screen size groups, they do give you more control over how your UI translates across different screens, and they allow you to specify the exact point at which your app switches from a layout that's optimized for Android smartphones, to a layout that's optimized for tablets.

> **Note**
>
> Not all versions of Android support all qualifiers, for example `sw<N>dp` was introduced in API level 13. However, when you use a qualifier in your project, the system automatically and implicitly adds the platform version qualifier, so older versions of Android can at least recognize unsupported qualifiers and subsequently ignore them.

Designing for different screen orientations

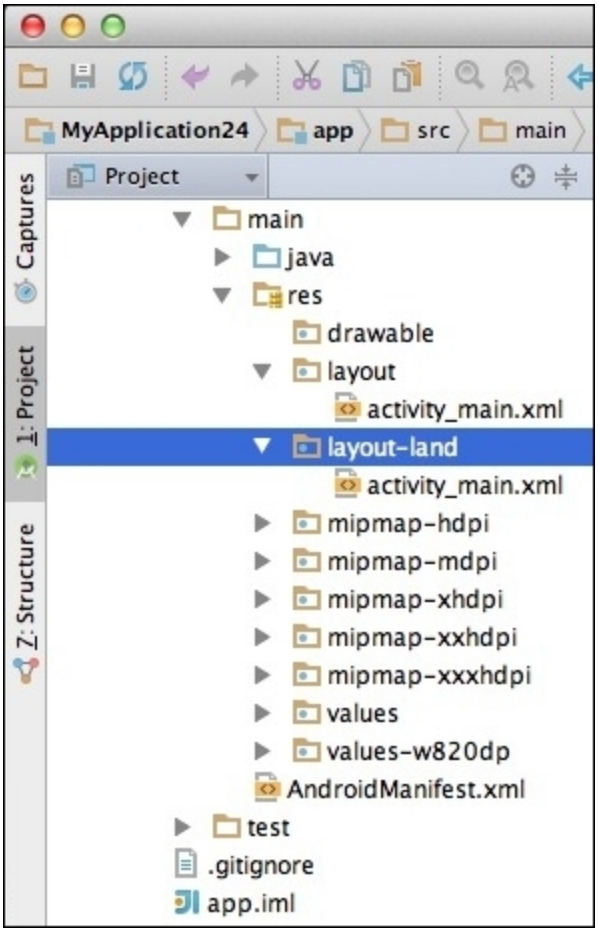
You may also want to create versions of your layouts that are optimized for landscape and portrait orientations. You can provide layouts that are optimized for alternate orientations in the same way you provide layouts for different screen sizes and densities: you create an extra directory with the correct orientation qualifier and place your orientation-specific layouts inside that directory.

Even though Android supports two orientations—portrait and landscape—you only ever need to create one additional directory, since you'll use your project's default `res/layout` directory for one of these orientations.

Choose which orientation you want to use as your app's default, and then create a directory for the other orientation, so your project may consist of a `res/layout` directory that contains your default/portrait

layouts and a `res/layout-land` directory which contains your project's landscape layouts.

Alternatively, if you want to use landscape orientation as your project's default, create a `res/layout-port` directory and use `res/layout` for your project's landscape layouts:



Note

When it's time to test your app, make sure you test it in both landscape and portrait mode across a range of screen configurations.

Reacting to orientation changes

Sometimes, your app will need to register configuration changes and then modify its behavior accordingly.

One of the most common scenarios is reacting to whether the screen is in landscape or portrait mode. For example, imagine your app consists of two fragments: one fragment displays a list and the other fragment displays information about the currently-selected item. The app displays each fragment separately on smaller screens in a single-pane layout and side by side on larger screens in a multi-pane layout. When the user selects an item in the multi-pane layout, the information is displayed in the same activity. However, when the user selects an item in the single-pane layout, your app will need to display this information in a new activity. In this scenario, your app needs to know what layout the user is currently viewing (single or multi-pane), so it can react accordingly.

One method is to identify a view that's only visible in the multi-pane layout, and then query whether this view is currently visible:

```
public class MyActivity extends FragmentActivity {
   boolean mIsDualPane;

   @Override
   public void onCreate(Bundle savedInstanceState)
{
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main_layout);

//Check whether detailsView is currently visible//

        View detailsView =
findViewById(R.id.article);
        mIsDualPane = detailsView != null &&
                        detailsView.getVisibility()
==
                        View.VISIBLE;

//If this view is visible, and we're in multi-pane
mode....//

if (mDualPane) {
```

```
...
...

//This is where you'd define your app's multi-pane
behavior//

} else {

//If this view isn't visible, then we're in
single-pane mode//

...
...

//This is where you'd define your app's single-
pane behavior//
```

You can retrieve the device's current configuration using `getConfiguration()`:

```
Configuration config =
getResources().getConfiguration();
```

To retrieve the device's current screen orientation and act on the results, run the following code:

```
if (getResources().getConfiguration().orientation

//If the screen is currently in landscape
mode...//

                ==
Configuration.ORIENTATION_LANDSCAPE) {

...
...

//This is where you'd define your app's landscape
behavior//

} else

//If not, then the device is in portrait
orientation//
```

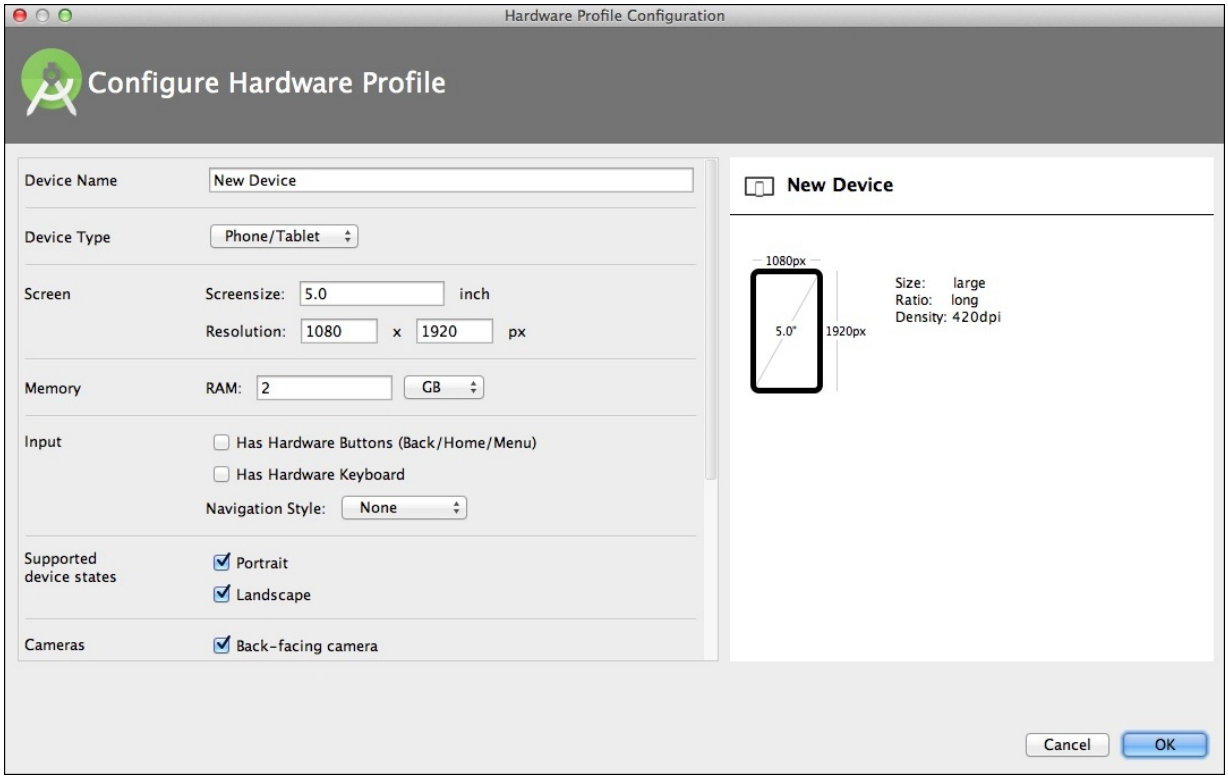
```
...
...

//This is where you'd define your app's portrait
behavior//
```

Testing across multiple screens

Before publishing your app, you should thoroughly test it across all supported screen sizes and screen densities, in both landscape and portrait mode. Unless you happen to have access to a bunch of different Android smartphones and tablets, the most practical way is to use the emulator that comes as part of the Android SDK.

When you launch the **AVD Manager** and select **Create Virtual Device...**, you can choose from a wide range of ready-made **Android Virtual Devices** (**AVDs**) that are based on real-life devices, or you can create your own by selecting **New Hardware Profile**. When creating a new AVD, you can then choose whether your AVD supports landscape and/or portrait mode, but you should typically test your app in both landscape and portrait orientation on *every* AVD you create.

You can also enter the screen size and resolution of your choice. As you enter these settings, you'll see the **Density** of this particular device displayed in the window's right-hand pane:



To test your app's screen support, you'll need to create multiple AVDs that represent all the most common Android screen configurations. There's no shortcut here, and generally speaking, the more time you spend testing your app, the better the user experience will be, regardless of the device it winds up on.

Note

I'll be using the emulator that comes as a part of the Android SDK throughout this book, but if you're not a fan of this default emulator, there are alternatives available. One popular alternative is Genymotion (https://www.genymotion.com).

Showing off your screen support

When it's finally time to prepare your app's Google Play listing, you should use screenshots that show your app in the best possible light.

Ideally, you should take all your screenshots on a large, high-density screen that's running the very latest version of Android, so potential users can see the most impressive version of your UI.

If you need to create promotional images for your website, blog, social media accounts, or anywhere else, then you should place your screenshots in context by wrapping them in device artwork. The easiest way to do this is using Android's drag-and-drop **Device Art Generator** (http://developer.android.com/distribute/tools/promote/device-art.html).

Attracting an international audience

Your Android app has the potential to reach a worldwide audience, but *only* if you invest time and effort into localizing your app.

Your biggest task when localizing a project is translating its text into your target language, but you'll also need to translate any drawables that contain text, any video that features text or dialogue, and audio that contains dialogue. You'll also need to ensure that any numbers, currency, times, or dates are formatted correctly for your target audience, as formatting can vary between languages and countries.

You provide these alternate resources in the same way you provide any other resources: create new directories and use an appropriate configuration qualifier.

When it comes to localization, you'll need to use *locale* configuration qualifiers, which consist of the following:

- **A language code**: These are two-letter lowercase ISO codes, as defined by ISO 639-1 (https://en.wikipedia.org/wiki/ISO_639-1).
- **Country or regional code (optional)**: Two-letter uppercase ISO codes as defined by ISO 3166-1 (https://en.wikipedia.org/wiki/ISO_3166-1_alpha-3) and proceeded by a lowercase `r`. You can use country/region codes in combination with the language code to provide resources that target devices set to a particular language and located in a specific country or region. For example, you could provide resources for French speakers located in Canada by combining the language code (`fr`) with the regional code (`CA`) plus a lowercase `r`, so your directory would be `res/values/fr-rCA`. You can't use a country or regional code on its own; it must always be preceded by a language code.

Note

Although it's tempting to think of a locale as being synonymous with a country, this isn't always the case. While you may create a directory that uses the French language and country code, you can also create a directory that combines the French language code with the Canadian country code.

Identifying the target languages and regions

The first step to localizing your app is to identify the languages, regions, and countries you want to support.

Put your business head on and look for locales where there's a potential market for your app. In particular, you should look for languages, regions, or countries where:

- There's a large or growing number of Android users
- International languages, such as English, aren't widely used
- There's a gap in the market for an app of your genre or subject

All the preceding factors mean that localizing your app for this locale could be particularly lucrative.

Decide what countries or regions you want to target first, and then determine what language(s) your app needs to support in order to appeal to people in this part of the world.

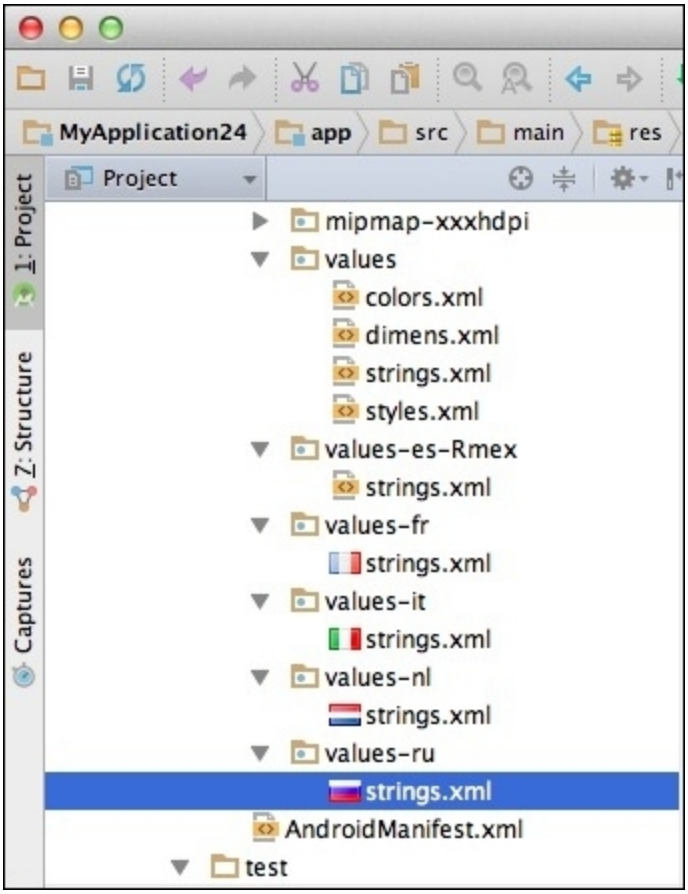
Once you have a list of all the locales you're going to support, grab their language codes from ISO 639-1 (https://en.wikipedia.org/wiki/ISO_639-1), plus any necessary region or country codes (https://en.wikipedia.org/wiki/ISO_3166-1_alpha-3).

Providing alternate text

Since translating your app's text is usually the biggest localization task, we're going to tackle this first.

Open your project's `res` folder and create all the alternate `values` directories your app needs to support your target locales. So, if you wanted your app to support the Spanish language, you'd create a directory called `res/values-es`. If you wanted to provide Spanish text that targets Spanish-speakers in Mexico, you'd create a `res/values-es-Rmex` directory.

Create the `strings.xml` files inside each of these directories:



Place all your translated string resources inside the corresponding `strings.xml` file (we'll look at some options for getting your text translated in a moment).

Continuing with our Spanish theme, our `res/values-es/strings.xml` file may look something like this:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
     <string name="hello_world">Hola
Mundo</string>
</resources>
```

At runtime, the Android system logs the device's locale settings, checks your project for any suitable locale-specific resources, and then loads either the resources from the corresponding locale-specific directory or falls back on your project's default resources.

Tip

Keep your text separate from your application code

It's good practice to keep the localized aspects of your app separate from your application code and to *never* hardcode any text into your app, as this will make your text far more difficult to localize. Instead, declare all your text as string resources, as it's easy to extract strings from your project, send them to a translator, and then integrate them back into your app without making any changes to your compiled code.

What other resources might you need localizing?

A large part of the localization process is translating your project's text, specifically its sting resources, but you may also want to provide locale-specific versions of other resources. For example, you may want to provide translated versions of the following:

- Drawables that feature text
- Video that contains text or dialogue

- Audio that features dialogue

You may also want to provide alternate versions of resources that aren't appropriate for the locale you're targeting. For example, you may want to provide alternatives to scenes that feature a particular city or scenery if this image isn't representative of the locale you're targeting. Sun-drenched beaches and crystal-clear seas aren't your typical scenery in every country (unfortunately).

You should also be aware that some images maybe considered inappropriate or offensive in certain cultures.

You provide locale-specific versions of any resource in exactly the same way you provide alternate string resources; create a new directory and use the same combination of the language and country/region code. For example, if you wanted to provide multiple versions of a `typical_highshcool.jpg` drawable, you'd create multiple drawable folders, such as `res/drawable-ja-rJPN`, `res/drawable-en-rUSA`, and `res/drawable-sv`.

Why default resources are important

The key to ensuring that your app runs correctly on devices with any language, country, or region settings is to provide a complete set of default resources.

One of the most important default resources you need to provide is `res/values/strings.xml`. This is where you'll define every string resource used throughout your project in your app's *default language*.

> ## Note
>
> The default language is the language that's spoken by the majority of your target audience. This may not necessarily be your audience's first language.

If you don't provide a default version of every string resource, then your app will crash whenever it winds up on a device with locale settings that you haven't provided a specific string resource for. Even if the user doesn't understand your app's default language, this is still preferable to your app crashing or refusing to open.

You'll also need to provide a default version of every other locale-specific resources you use in your app, because if your project is missing one default resource, then it may not run on devices with unsupported locale settings.

Default string resources are also crucial when there's some text you want to keep consistent across your app, regardless of the device's locale settings. One common example is your app's title, which you may want to keep consistent across all versions of your app.

To use text consistently across locales, you just need to define the string once in your project's default `strings.xml` file, and then omit it from all your locale-specific `strings.xml` files. Then, when the system tries to load your app's title from a locale-specific strings file, it'll realize that this string is missing and fall back on the default version. The end result is that the same string resource will be used in every version of your app.

Tip

Should I translate my app's title?

There's no straightforward answer to this tricky question. Having multiple titles makes an app more difficult to maintain, and it can make usually straightforward decisions more complex; if your app goes by multiple names, then what should your app's Twitter handle be? Do you need to create multiple versions of your app's logo and app icon? Do you need to create multiple support e-mail addresses?

You'll also have to work much harder to promote a product that goes by several different names, particularly when it comes to things such as **Search Engine Optimisation** (**SEO**).

Despite all these drawbacks, there are a few good reasons why you might want to publish your app under multiple names. The most common reason is that your app's title may not make much sense to people who aren't fluent in your app's default language. If we decided to call our recipe app Best Student Recipes, English-speaking users would instantly know what to expect, but for users who speak different languages, *best student recipes* could be completely meaningless. A nonsensical title isn't particularly appealing to potential users who stumble across your app in the Google Play store, so you may want to consider releasing your app under multiple names.

There's no real right or wrong answer here. All that matters is that you make the best decision based on the app you're developing and your target locales.

Whatever decision you do make, make sure you stick to it! Changing your app's name after you've launched it is only going to confuse your users.

Which configuration qualifiers are the most important?

At this point, we've covered quite a few different configuration qualifiers: locale-specific qualifiers, screen orientation, screen density, and screen size qualifiers.

If you use a mixture of these in your projects, there's a strong possibility that more than one version of a resource is going to be suitable for the current device. Imagine your app winds up on a device with a high-density screen and Spanish language settings, and that's currently being held in landscape mode. The system

needs to display a drawable, but multiple versions of this drawable are suitable for the current device:

- `res/drawable`
- `res/drawable-es`
- `res/drawable-land`
- `res/drawable-hdpi`

Which drawable will the system use? When Android is spoilt for choice, it follows a strict set of rules. Of all the qualifiers you can use, locale almost always gets preferential treatment. In this example, the device is configured to use Spanish, so Android would load the graphic from the `res/drawable-es` directory *even though* the device in question also has a high-density screen and is currently in landscape mode.

> ## Note
>
> The only qualifiers that take precedence over locale qualifiers are **mobile country codes** (**MCC**) and **mobile network codes** (**MNC**). If you use any of these codes in your project's directories, they'll always take precedence over your locale qualifiers.

Getting your app translated

Translating your app's resources is a crucial part of the localization process, and unless you're fluent in the language that you're targeting, at some point you'll need to enlist the help of a translator, whether that's a paid professional or a kind volunteer.

Although you may be tempted to go down the quick and easy route of a machine translator, services such as Google Translate can never compare to a human translator, and relying on machine translators will almost always result in a bad experience for your international users.

If you do decide to hire a professional translator, you should start contacting translators as early as possible, so you can get an idea of turnaround time and incorporate this information into your development plan. It also gives you time to shop around and find the best possible translator for your budget.

You can find translators online, or you may want to explore the Google Play app Translation service, which is offered through Developer Console.

To access these services, perform the following steps:

1. Log in to the Developer Console (https://play.google.com/apps/publish).
2. If you haven't already done so, upload your APK by selecting **Add new application**.
3. Once you've uploaded your app, it'll appear in your **All Applications** dashboard. Select the app you want to purchase translation services for.
4. Make sure **APK** is selected in the left-hand menu.
5. Under **APK Translation Service**, select **Start**.
6. Select the source language used in your original text.
7. Under **Translate API**, add the XML source file that contains the strings that you want to translate.
8. Select your target languages.
9. Choose a translation vendor.

Just be aware that even though you're making a purchase through the Developer Console, this is a direct business agreement between you and your chosen vendor, so you'll need to work with the translator directly. After purchasing a translation through the Developer Console, you will receive an e-mail from your vendor, and it's your responsibility to manage the project from there.

Getting the most out of your translations

The quality of your translations depends on the skill of the translator, but it also depends on the quality of your input. If you provide your chosen translator with a higher quality input, then it stands to reason that you're going to get a higher quality output.

In this section, I'll show you how to increase your chances of receiving a high quality, accurate translation.

Putting your strings in context

When you're declaring your string resources, you should provide as much information about each string as possible, as this will help your translator to better understand your string resources.

At the very least, you should give each string a descriptive name, but ideally you should also supply a comment that explains what this string is for, along with the information about when and where the string will appear in your app, plus any restrictions such as the maximum amount of characters this part of your UI can display. For example, take a look at the following:

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

//Text that appears on a button that can fit 9
characters. Must be one word. The user taps this
button to submit a search query//
```

If your app features any specialist language or technical terms, then you should thoroughly explain what each term means. If this turns out to be too much text to fit into a comment, or you find yourself explaining multiple strings, then you may want to create a separate glossary document that explains the meaning and use of all the specialist terms used throughout your `strings.xml` file. You can then send this additional document to the translator along with your strings.

Using terms consistently

You can greatly increase your chances of a successful and accurate translation by using terms consistently throughout your app. For example, if the main method of navigating your app is via a series of buttons that allow the user to progress to the next screen, you should label these buttons consistently, instead of using a mix of terms that all mean the same thing, such as **Forward**, **Next**, **OK**, and **Submit**. This will make the translator's job much easier.

> ## Note
>
> Using terms consistently will improve your app's general user experience too.

Getting your strings in order

To help the translator make sense of your project's string resources, you should take a few moments to perform some admin on your `strings.xml` file. Remove any unused strings that you might have added early on and then never actually used in your project, and remove the strings that have disappeared from your project over time. You should also remove any duplicate strings, and be on the lookout for any spelling mistakes or typos. Finally, make sure your strings are formatted correctly and consistently. This housekeeping may seem simple, but it can make your translator's job much easier, particularly if they're not an Android developer themselves.

Not creating unnecessary strings

More strings mean more work! Even if you're supporting several languages, you probably won't need to create a locale-specific version of every string in every language your app supports.

There may be pieces of text that you want to use consistently across all locales, such as your app's title, which means there's no need to add this string to every locale-specific `strings.xml` file. Remember, if you don't include a particular string in a locale-specific directory, your app will use the default string instead.

Your app may also support languages that are variations of the same parent language. For example, imagine your app's default language is British English, but it also supports American English. Chances are that most of the resources you define in your project's `res/values/strings.xml` file will be suitable for your American English audience. You should still create a `res/values-en-rUSA/strings.xml` file, but this file should only contain the strings where American English spelling differs from British English spelling.

Always marking non-translatable text

Your project may include some string resources that you don't want translating, such as your app's title, URLs, promotional codes, social media links, and e-mail addresses. You should clearly mark all the text you don't want translating before handing your project over to the translator.

Wrap the `<xliff:g>` tags around the text that you don't want translating, and use an `id` attribute to explain why this text shouldn't be translated:

```
<string name="name"> This text will be
translated<xliff:g id="This is my
explanation">this text won't be
translated</xliff:g> this text will be translated.
</string>
```

Imagine you want to display a welcome message that includes your app's title. You want the actual welcome text to be translated, but you want the app's title to remain consistent across all locales. In this scenario, you'd use the following XML:

```
<string name="welcomeMessage"> Welcome to
the<xliff:g id="appTitle">Student
Recipes</xliff:g>app.</string>
```

Finally, to prevent your project from throwing an error, you need to update the resources tag in your `strings.xml` file:

```
<resources
xmlns:xliff="urn:oasis:names:tc:xliff:document:1.2
">
```

Tip

Marking dynamic values as non-translatable

Sometimes, you'll want to combine some predetermined and dynamic text in a single string. For example, your app could ask the student for their name, take their input, combine it with some of the predetermined text, and then display the **Welcome to the Student Recipe app, Nicole!** message.

So, what does this have to do with translation? When you create strings that feature dynamic text, you represent the dynamic text with a placeholder, such as follows: `<string name="name">Welcome to the app %1$s</string>`

You should make it clear to your translator that this placeholder isn't an error or typo, and therefore, they shouldn't try to translate it. This becomes particularly important when you're working with a translator who isn't familiar with Android development.

To make it clear that a placeholder shouldn't be translated, you can use the regular `<xliff:g>` tags and the `id` attribute. However, for the sake of clarity, you should also provide an example of the kind of content this placeholder text might eventually display. In our welcome message example, this looks something like the following: `<string name="welcomeMessage">Welcome to the Recipe App<xliff:g id="userName" example="Nicole">%1$s</xliff:g>! </string>`

Other things to consider

Although language may be the most obvious factor, there are several less obvious things you need to take into consideration if your app is really going to make a splash on the international market.

Right to left support

Don't automatically assume that your audience will read left to right (LTR). In some countries, reading right to left (RTL) is the norm. Depending on the locales you're targeting, you may want to consider implementing support for RTL and LTR.

The good news is that in Android 4.2 and higher, the system automatically mirrors your app's UI when the user switches the system language to a right-to-left script.

To take advantage of this automatic mirroring, open your project's `Manifest` file and add the following line:

```
android:supportsRtl="true"
```

Then, change all your app's `left/right` layout properties to the `start/end` equivalents:

- If you're targeting Android 4.2 and higher, use `start` and `end` instead of `left` and `right`, for example, `android:paddingRight` becomes `android:paddingEnd`
- If you're targeting versions of Android lower than 4.2, you should use `start` and `end` in addition to `left` and `right`; for example, you should use both `android:paddingRight` and `android:paddingEnd`

Formatting values

Keep in mind that not every locale formats values, such as dates and numbers, in the same way. You should *never* hardcode formats based on assumptions about the user's locale as this can cause problems when the user switches to another locale.

Instead, always use the system-provided formats and utilities such as follows:

- DateUtils (http://developer.android.com/reference/android/text/format/DateUtils.html) and DateFormat (http://developer.android.com/reference/java/text/DateFormat.html) for dates
- String.format (http://developer.android.com/reference/java/lang/String.html#format(java.lang.String,%20java.lang.Object) or DecimalFormat (http://developer.android.com/reference/java/text/DecimalFormat.html) for numbers and currency
- PhoneNumberUtils (http://developer.android.com/reference/android/telephony/PhoneNumberUtils.html) for phone numbers

Localizing your app – best practices

Localizing your app is a powerful way of attracting potentially *millions* more users. Why limit yourself to just one market? Implementing the following best practices will increase your app's chances of connecting with a worldwide audience.

Design a single set of flexible layouts

Depending on the languages that you choose to support, some of your alternative string resources may expand or shrink dramatically during the translation process to the point where they no longer fit your layout.

To minimize the chances of this happening in your app, you should create flexible layouts that can accommodate alternate string resources of all sizes. One useful trick is to give your app's default text more space than it requires, so your UI already has some wiggle room when it comes to accommodating slight variations in size.

You should design UI elements that contain text to be able to expand horizontally and vertically in order to accommodate minor changes in text height and width, for example, buttons that can grow and shrink depending on the size of their button labels. However, you should also consider how these expanding and shrinking UI elements might impact the rest of your UI. Could an expanding button make your UI look cramped or lopsided? Or worse, could it push the neighboring buttons off the screen, rendering them unreachable?

Pay particular attention to UI elements that the user can interact with, such as buttons; if these elements change size, it can quickly impact your user experience. Two touchable objects that inch closer to one another as they expand can result in frustrated users who keep catching the wrong button by accident.

If you design your layouts with flexibility in mind, then typically you'll be able to use the same layout across all locales. However, if you're struggling to design a layout that's flexible enough to accommodate all your alternate strings, you may want to make some changes to your text.

Simple, straight-to-the-point text typically means less variation during the translation process, so if you're struggling with text that shrinks or expands dramatically, start at the root of the problem—your app's default text. Look for any opportunities to simplify this text, any unnecessary text you can remove from your UI, plus any words that you can replace with a universally understood picture or symbol, such as replacing the **OK** text with a checkmark or **Cancel** with a cross**.**

Note

Simplifying your app's text and removing any words that aren't strictly necessary will provide a better user experience all round, regardless of whether the user is viewing a localized version of your app or accessing your app in its default language.

Depending on the amount of text you need to display and the locales your app supports, it may be impossible to create a single layout that can accommodate text across *all* your target languages. Layouts can only flex so far after all!

As a final resort, you can create alternate layouts specifically for the language (or languages) that are causing your app so many problems, which leads onto the next point.

Create alternate languages only when needed

You can create a layout that targets a specific language by creating a new layout directory with the relevant configuration qualifier. For example, if you wanted to create a version of your project's `main.xml` layout that's optimized to display your tricky German translations, you'd create a `res/layout-de/main.xml` directory, and then place your German layout resource file inside this directory.

However, alternate layouts do make your app harder to maintain, so only use them as a last resort.

Testing your app across different locales

Once you've created all your locale-specific layouts and resources, you need to make sure that your app really is ready for an international audience. This means testing the localized versions of your app across a wide range of AVDs with localization settings that correspond to your different target locales, plus at least one AVD that's set to a language and locale that your app *doesn't* explicitly support.

When testing your app across different locales, be sure to check for the following:

- Your project is displaying the correct string resources, layouts, and any other locale-specific resources for the device it's currently installed on. Placing a `strings.xml` file in the wrong directory and missing a configuration qualifier or even a simple typo is all it takes for your app to display Korean text (`res/values-ko/strings.xml`) when it should really be displaying Kurdish text (`res/values-ku/strings.xml`).
- Your locale-specific resources are displaying correctly in the current layout and aren't causing any usability problems.

You should also be aware that some screen configurations, hardware, and software may be more common in certain locales. Make sure you research the most popular devices in all your target locales to see whether you need to incorporate support for specific hardware, software, or screen configurations into the localized versions of your app.

Testing for different locales

You can test your app's language settings on a physical device by opening that device's **Settings** app, selecting **Language**, and then choosing a new language from the menu. This is useful for getting an idea of how the localized version of your app functions across devices with different language settings. However, you should also test how your app functions with different country and region settings, something you can't easily change on a physical Android device, which is where AVDs come in.

Although you can't create an AVD with specific language, locale, or region settings, you can change these settings once you have an AVD up and running in the emulator.

Start by creating an AVD with the screen configuration, hardware, and software settings you want to test your app against, and launch your project in this AVD.

Open your Mac terminal or Windows command prompt and **change directory** (`cd`) so it's pointing at the directory where the**Android Debug Bridge** (`adb`) is located. For example, the command I'm using is:

```
cd /Users/jessica/Downloads/adt-bundle-
mac/sdk/platform-tools
```

Once your AVD is up and running, you can change its locale by issuing adb commands. To do this, you'll need the ISO code of the language you're targeting (`ISO 639-1`), and any country or region code (`ISO 3166-1`) you're using.

In your `adb` window, enter the following command:

```
adb shell
```

After a few moments, you will see a `#` prompt. At this point, you can enter the following command:

```
setprop persist.sys.locale [ISO language code,
with optional country/region code] ;stop;sleep
5;start
```

For example, if you wanted to test your app on a device that's set to Spanish (`es`), you'd run the following:

```
setprop persist.sys.locale es;stop;sleep 5;start
```

If you wanted to check your app on a device that's set to use the Spanish language in the Mexican locale, you'd run the following:

```
setprop persist.sys.locale es-Rmex;stop;sleep
5;start
```

At this point, the emulator will restart with the new locale settings. Relaunch your app and that's it—you're ready to test your app against your new locale settings.

Look for common localization issues

So, once you've set up your test environment, what *specific* issues should you be looking for?

Firstly, you should check every screen in your app to make sure there's no instances of clipped text, poor line wrapping, strange-looking word or line breaks, or incorrect alphabetical sorting. You should also be aware of any locales where the text should run RTL and check that this is actually what's happening in your layouts.

You should also be on the lookout for any unintentionally untranslated text; unintentional because there may be instances where you want to use the same text consistently across your app, such as your app's title and your contact details. If you spot any *unintentionally* untranslated text, then you should dig into your app's code to see exactly what's going wrong.

Also, be on the lookout for any wrongly translated text. This kind of error can be difficult to spot, as chances are that you're not fluent in all the languages your app supports. This is one of the few occasions where it's acceptable to use an online translator, such as Google Translate, as it's often the quickest and easiest way of double-checking that your app's Spanish text really is Spanish.

You should also look for any instances where text or other locale-specific resources don't fit with your current layout. This could be the following:

- Resources that are smaller than the default version, such as the text that shrinks dramatically during the translation process. Shrinking resources can leave gaping holes in your layout, or they can throw other UI elements off balance and create a layout that, while not completely broken, does look pretty odd.

- Resources that are present in some versions of your UI and missing in others. Again, this can cause awkward gaps and aesthetic problems in your layout. If you're intentionally omitting certain resources, you might want to look for something to fill the gap this resource leaves behind, or as a last resort, supply an alternative layout that's designed specifically for this locale.
- Resources that are much larger than the default version, which can result in a cluttered, cramped-looking UI or usability problems, such as touchable areas that are pushed too close to one another.
- Resources that aren't appropriate for the current screen configuration. As already mentioned, different locales may also mean different hardware, software, and screen configurations. If you're targeting multiple countries and regions, make sure you research the most popular Android devices within these specific demographics, and then test your app across AVDs that reflect the hardware, software, and screen configurations that are most commonly used within these different areas.

Testing for default resources

After you've tested your app across all the languages, countries, and regions that you want to support, you should install your app on an AVD with locale settings that it *doesn't* explicitly support. This may sound strange, but it's the most effective way of checking that you've included a default version of every resource your app requires.

As I've mentioned several times throughout this chapter, including default resources means that if your app gets into difficulties, it has something to gracefully fall back on rather than crashing. Providing default versions of all your project's strings, drawables, and other resources guarantees that your app will run even if it winds up on a device with locale settings that it doesn't explicitly support.

To test your app, open your Terminal or Command Prompt window and tell your emulator to switch to a locale that your app doesn't explicitly support. If you've provided the necessary default resources,

your app should load these resources, including the contents of your project's `res/values/strings.xml` file. If your app crashes, then this is an indication that your project is missing at least one default resource.

Plan a beta release in key countries

Once you've finished testing your app across different locales, and you're confident that it'll functions correctly regardless of the user's language, country, or region settings, then you may want to consider opening your app up to feedback from your potential target audience, in particular the native speakers of all the languages your app supports. This usually takes the form of beta testing.

> **Note**
>
> Users who install beta versions of your app cannot leave reviews on your Google Play page, so don't worry about these early versions of your app negatively (and unfairly!) impacting your Google Play rating.

Getting real-world feedback is always valuable, but this feedback becomes even more valuable when you're planning an international launch. Native speakers will be able to easily pinpoint any grammar or spelling mistakes in your text, or even something that just sounds a bit *off*. They can also quickly flag up anything about your app's content that's inappropriate, confusing, or difficult for them to relate to.

This is all invaluable feedback that can help you perfect the localized versions of your app and create something that feels like it was created specifically for this section of your target audience, rather than a translation that was knocked together in a last-minute bid to rack up some extra downloads.

If you're looking to launch a beta testing program, then Google Play can help. After you sign into Developer Console and upload your APK, you can set up groups of users for beta testing.

You may want to start small with a closed beta test, where you specify a group of testers by entering their Gmail addresses. If you happen to have a list of testers to hand, you can launch a closed beta program by doing the following:

1. Log in to the Developer Console and selecting **APK** from the left-hand side menu.
2. Select the **Upload your first APK to beta** button and following the onscreen instructions to upload your APK.
3. Once your APK is uploaded, select **Beta testing**.
4. If you see a **Choose a testing method** option, open the accompanying drop-down menu and select **Set Up Closed Beta Listing**, followed by **Create list**.
5. Enter the e-mail addresses of all the people who you want to participate in the beta test. Click on **Save**.
6. Copy **Beta opt-in URL**. You can then share this link with your testers. When a potential tester clicks this link, they'll see an explanation about what being a beta tester entails. They can then decide whether to participate in the test or not.
7. After copying **Beta opt-in URL**, you'll need to enter an e-mail address or URL that you'll use to collect feedback from your testers. Enter this information in the **Feedback channel** field, then click on **Save**.

Tip

Gathering tester feedback

Since testers can't leave reviews on your app's Google Play page, you'll need to provide them with an alternative way of sending you their feedback. Whenever you launch a testing program through Developer Console, it's crucial you complete

the **Feedback channel** field as this is your testers' primary way of communicating with you.

This feedback channel could take the form of an address or URL, but you may also want to consider setting up a Google+ page or Google Group for your testers. Whatever your preferred method of gathering feedback, it's important to recognize that anyone who tests your app is doing you a favor, so you should make it as easy as possible for them to send you their feedback.

If providing feedback feels like hard work, then your testers simply won't bother—they'll keep their thoughts to themselves, and your project will suffer as a result.

After a round of closed beta testing, you may want to move on to open beta testing, where you specify a maximum number of testers but don't specify their e-mail addresses.

To launch an open beta testing program, make sure you're logged into Developer Console, and then select **APK** from the left-hand side menu. Select **Beta testing**, open the drop-down menu, select **Set up Open Beta Testing**, and then specify the maximum number of users who'll be able to beta test your app.

Copy **Beta opt-in URL** and share it with your testers. Again, make sure you enter an e-mail address or URL in **Feedback channel**, and then click on **Save**.

Get ready for lift off!

If you want your app to make a splash in the international market, supporting multiple languages and locales is just the beginning. You'll also need to devise some form of international marketing campaign.

This may sound like the kind of thing that requires a massive budget and a dedicated marketing team, but you can tailor the scope of your

campaign to suit your budget and how much free time you have to promote your app. Even if your app is a one-person project, there's no end to budget-friendly (and in some cases, free) ways of promoting your app to an international audience, especially when you're promoting your app online.

A marketing campaign could be as simple as writing a press release announcing the release of your app, and then arranging for this text to be translated into the various languages your app supports. You could then spread your press release across the World Wide Web via social networks, free news and press release websites, your own blog, or any other places where you can post content for free.

Note

Don't get carried away and post your marketing materials absolutely everywhere just for the sake of it. Spamming is only going to hurt your app's reputation in the long run.

You should also translate any other promotional materials you create, such as banners, virtual badges, adverts, or memes.

Even if your promotional graphics don't feature any text, you should still look for opportunities to tweak these graphics, so they're more likely to resonate with each of your target locales. For example, could you include any geographical or cultural references that would make your graphics more appealing to users in specific regions? In our recipe app example, this might mean identifying recipes that are more likely to appeal to sections of our target audience and then featuring these recipes more prominently in the app's promotional banners and adverts.

If you create any promotional videos, you should also create localized versions of these videos for every language your app supports. This may mean recording entirely new audio for every video or providing subtitles.

Remember, the key to attracting an international audience is to create the illusion that your app was designed with each user's specific locale in mind.

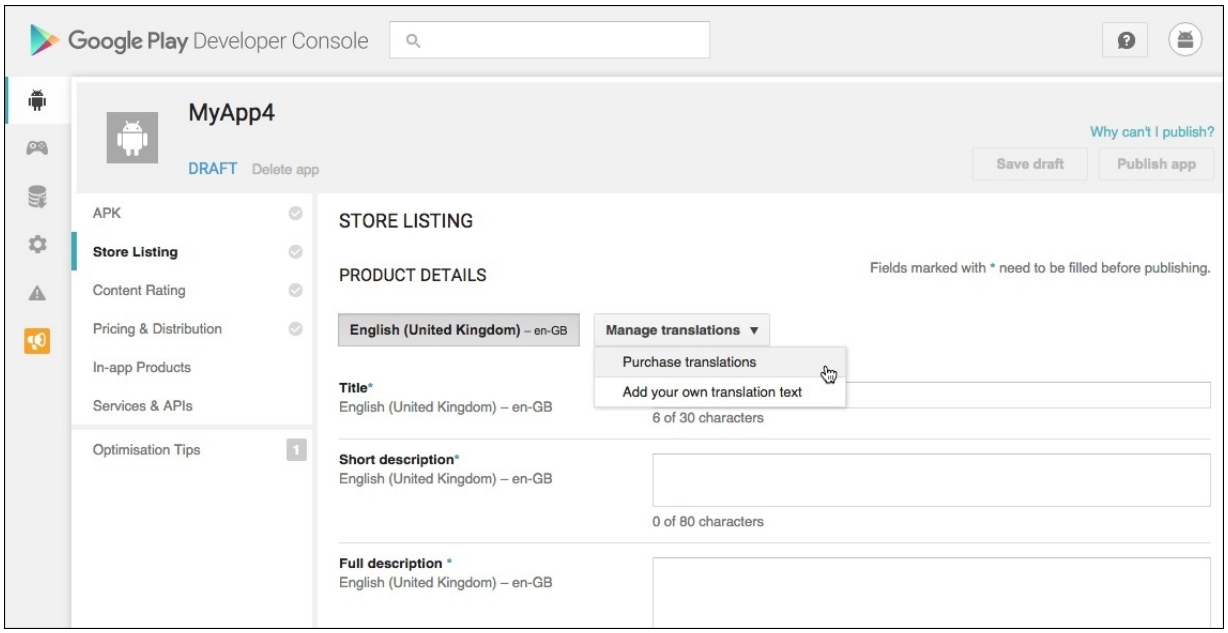
Localize your Google Play store listing

Your app's Google Play listing is the first impression that international users will have of your app. To make sure this first impression is a positive one, you'll need to localize your store listing, which means translating its text for every locale your app supports.

You can create multiple localized versions of your app's Google Play listing via Developer Console. Then, when a user opens your app's listing, Google Play will automatically determine that user's location, and it'll display the version of your listing that's the most appropriate for their language, country, or region settings. If Google Play can't find a suitable locale-specific version of your app's listing, it'll display your app's default Google Play page instead.

To create localized Google Play listings, log into your Developer Console account, and select **App applications** from the left-hand side menu. Select the app you want to work with, followed by **Store listing**.

To create a new, localized version, click on the **Manage translations** button. At this point, you can purchase a translation by clicking on **Purchase translations** and following the onscreen instructions:



If you already have some translated text you want to add to your localized store listing, select **Add your own translation**. Select all the languages you want to create a localized listing for (don't worry, you can always add more later on), and click on **Add**.

At this point, Developer Console will take you back to the store listing form, but now, if you click on the **Languages** button, you'll see a drop-down list of all the languages you've just added. Clicking on any of these languages will take you to a separate form where you can craft an entirely new store listing, which will be displayed whenever someone views your app in Google Play with these locale settings.

If you need to delete one of your localized pages at any point, simply click on the **Manage Translations** button, followed by **Remove translations**, and then choose the language you want to remove. If you want to change what version of your Google Play page is considered the default, click on the **Manage Translations** button again, and select **Change default language**.

Creating a localized store listing isn't just about translating your app's description; you should also supply alternate graphics. Even when a user is viewing a localized version of your Google Play page, they

may still be unsure whether your app actually supports their language. Don't leave any room for doubt; take screenshots of your UI in all the languages it supports, and then upload these screenshots to the corresponding version of your app's Google Play page.

> **Note**
>
> As you're building your app's Google Play page, Developer Console will offer hints about how your app can reach the widest possible audience. These tips contain some useful information, so make sure you check them out by selecting **Optimization Tips** from the left-hand side menu.

After launching your project

The hard work doesn't end when you successfully launch your app. Once you hit the **Publish** button and send your app out into the big wild world, there's a whole new bunch of things you can do to help that app reach the widest possible audience.

Following a successful launch – supporting international users

Once you've attracted an international audience, you need to hang on to that audience—which means offering support in a variety of languages.

Exactly how much support you can offer will vary depending on whether you have created your app as part of a team, or whether you're a solo developer creating Android apps in your spare time. But offering *some* level of support is crucial if you're going to hang on to the users that you worked so hard to gain in the first place.

At the very least, you should monitor your app's Google Play reviews. The Google Play store helpfully translates all reviews for you, and you should try and respond to any question or suggestions that come in via Google Play, regardless of the language they're written in.

Wherever possible, you should respond to Google Play reviews in the poster's language, even if it means resorting to an online translation tool. Just be aware that translation tools tend to generate responses that are a little *off,* so it's a good idea to let users know that you're using a translation tool, just so they know to overlook any clunky phrasing or grammatical errors.

Tip

Create a Google Play badge

Once you app is live, your goal is to drive as many people to its Google Play page as possible. One way of doing this is to create a Google Play badge. Whenever someone clicks on this badge, it'll take them direct to your app's Google Play listing.

You can post this badge anywhere you want—on your website, blog, social media accounts, or you can incorporate it into your e-mail signature. Just don't get carried away; no-one likes a spammer, and spamming your app's Play badge isn't going to earn your app a loyal following in the long run.

You can create a badge using the Google Play Badges generator (https://play.google.com/intl/en_us/badges).

Monitor your app's international performance

When you support multiple locales, your app is inevitability going to be more popular in one part of the world than it is in others. There's a whole host of reasons why this might be the case, ranging from

things you can control (such as the quality of your translations), to thing you *can't* control (such as a rival app that's already dominating the market in one of your target locales).

Having a clear picture of your app's popularity across all supported locales is essential for identifying areas where you need to focus more time and effort, and areas where it might make more sense to scale back your efforts or even stop supporting them completely.

If your app is proving massively popular in one locale, you may decide to put increased effort into meeting the needs of this part of your demographic, such as performing more market research in this area, creating localized content specifically for these users, or even creating language-specific user groups, such as a Google+ community, LinkedIn group, or Facebook page.

So, how do you identify locales where users *love* your app? Developer Console provides stats about the number of downloads, installs, and uninstalls that your app is racking up across all the different locales it supports. To access this information, log in to your Developer Console account, select the app you want to investigate, and then select **Statistics** from the left-hand side menu.

But, let's be realistic, these stats aren't always going to be good news, and sometimes they may reveal that your app is struggling to attract users in certain locales. If this is the case, then you should try and figure out what might be hindering your app's success.

It might be something that you can fix, such as a less-than-perfect translation, or the fact that your app has hardware, software, or screen configuration requirements that aren't representative of the Android devices commonly used in this locale. However, the problem may be something that isn't quite so easy to fix, such as a lack of interest in your app's genre or subject matter, or a rival app that's already dominating the marketplace.

Depending on the reasons behind your app's poor performance, you may decide to try and address the issue, or you may decide that it

makes more sense to drop support for this particular locale and redirect the time and effort into areas where your app has already found an enthusiastic audience.

Just remember that launching your app internationally is only phase one. If your app is going to find global success, then you'll have to make an ongoing effort to support users from all over the world, as well as continuously monitor how your app is faring in all the different locales.

Summary

In this chapter, we looked at how to reach the widest possible audience, from supporting different versions of the Android operating system, to optimizing your app for different hardware, software, screen sizes, screen densities, and orientations. We even covered translating our app into different languages. We also touched on how to promote your app via advertising and marketing campaigns.

Now that you know how to attract an audience, in the next chapter, we'll look at how to wow this audience by optimizing our app's UI.

Chapter 9. Optimizing Your UI

Your app's user interface is your most direct connection to the user, so you'll want to make sure it's *perfect*.

Up until now we've concentrated on creating a beautifully designed user interface that's packed with useful features—but this is only half the battle. If your app is going to rack up those 5-star Google Play reviews, you'll need to create a UI that's quick to render, responsive, and generally delivers an all-around great user experience.

Performance is crucial to creating a successful UI, and a successful app in general. If your app is laggy, prone to crashing, gobbles up data and memory, or drains the user's battery, then no one is going to want to use it, no matter how good your UI looks!

In this chapter, I'm going to show you how to hunt out all the most common performance problems that may be impacting your app. Since you'll want to fix any problems you do encounter, along the way I'll also be touching on some of the reasons why these problems occur in the first place, and the steps you can take to address them.

By the time you've completed this chapter, you'll know how to create a smooth and responsive UI that people will *love* using.

Timing your code

Don't wait for your app to start throwing errors before you go looking for performance-related problems. Your app could be slowly leaking memory, allocating too many objects, or struggling under the weight of a complex view hierarchy. None of these will necessarily throw an error, but they'll definitely have a negative impact on your app's performance.

If you're going to create a high-performance app, you need to go *looking* for problems.

Timing your code is a powerful way of seeing *exactly* what's going on in your app, including any sections of code that are running slower or longer than others. The first Android SDK tool we're going to look at lets you do just that.

TraceView is a graphical viewer that can profile any Android app running on your device.

TraceView, like most of the tools we're going to cover in this chapter, can only measure a running application. So the first step is to install the app you want to test on your Android device, and then attach that device to your computer. Alternatively, you could use an emulator and suitable AVD. Make sure your app is running and its UI is visible.

Tip

Before and after—measuring your app's performance

When you discover a problem with your app, obviously you're going to want to fix that problem. But after you've made some changes, how do you know that the problem is truly fixed?

If you use TraceView to measure your app's performance before and after you make your changes, you'll have all the data you need to see whether your optimizations have had any significant impact on your app's performance. So even if you don't identify any code that needs optimizing, TraceView's output is still valuable data that you should keep to hand and refer back to as you work on optimizing your app.

To launch TraceView, select **Tools** from the Android Studio toolbar, followed by **Android** and **Android Device Monitor**. The **Android Device Monitor** will open in a new window.

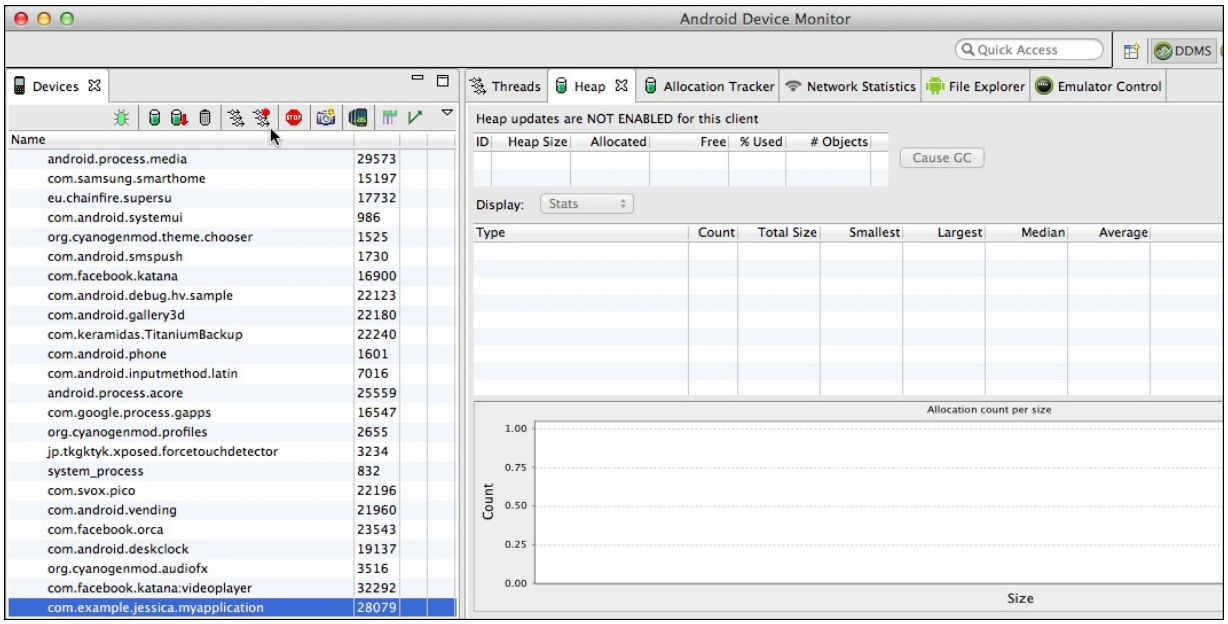
Note

The **Android Device Monitor** is a standalone tool that's included as part of the Android SDK. Throughout this chapter, I'll be using several tools that are included in the **Android Device Monitor**. Although I'll be accessing the **Android Device Monitor** via the Android Studio user interface throughout this chapter, you can also launch the **Android Device Monitor** separately. If you want to bypass Android Studio, or if you're using Eclipse, you can launch the **Android Device Monitor** by finding the `monitor` file in your Android SDK download and double-clicking it. The **Android Device Monitor** will then open in a new window.

In the **Android Device Monitor** window, select the **DDMS** tab. Along the left-hand side of the screen you'll find a **Devices** tab that lists all the currently detected devices and emulators. Select the device or AVD that contains your app, and you'll see a list of all the processes that are running on this device. Select the process you want to profile.

If you don't see your application in the list, check it's running and that its UI is visible.

Start method profiling by clicking the Start Method Profiling icon (where the cursor is positioned in the following screenshot):



At this point, you'll be presented with two profiling options:

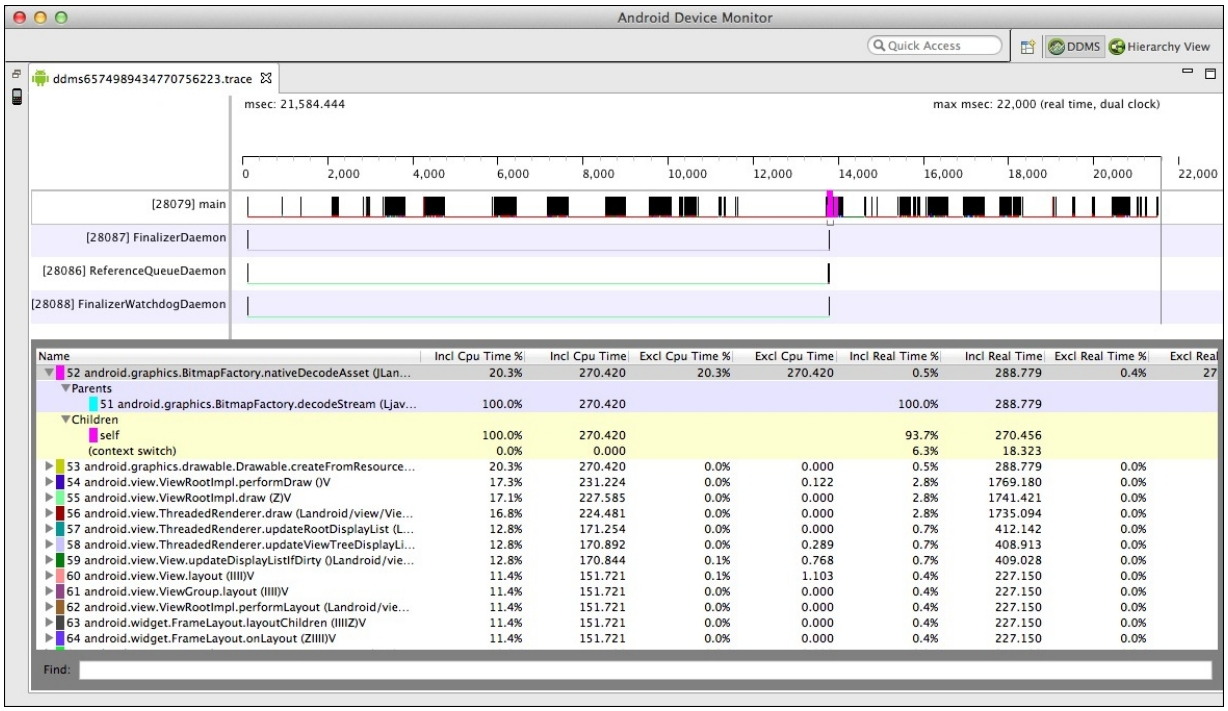
- **Trace-based profiling**: Traces the entry and exit of *every* method, no matter how small. This kind of profiling has a massive overhead, so you should only trace everything when you have absolutely no idea what to profile. You can then use this data to narrow your search for a follow-up round of sample-based profiling.
- **Sample-based profiling**: Collects the call stacks at a frequency specified by you. With this kind of profiling, the overhead is proportional to the sampling frequency, so the overhead is generally much more manageable.

Make your selection and TraceView will start recording. Spend some time interacting with your app, making sure to interact with all the sections and methods you want to profile. Then, click the Stop Method Profiling icon and DDMS will pull the trace file from your device and load it in the viewer.

Note

Depending on how much recording you've done, this data may take a few moments to launch, so you may need to be patient.

You'll end up with a trace file that looks something like this:



A trace file consists of:

- Timeline panel:

  This panel displays where each thread and method started and stopped, so you can track how your code is executing over time.

  The timeline panel displays each thread's execution on its own row, in a different color. You'll see a spike where each method starts (the left bar) and stops (the right bar). The line between these points is the amount of time it took the method to execute.

  If you spot any long lines of the same color, this is an indication that this method is eating up processing time and you should gather more information about this method. Click on any method

you want to learn more about, and its stats will appear in the Profile panel.

- Profile Panel:

  This panel provides lots of information about what happened inside the currently selected method, including how many times that method was called, how many times it was called recursively, and the percentage of total CPU time used by this method, inclusive and exclusive of the execution time of all its child methods. This data helps you identify any methods that need optimizing.

Identifying overdraw

When Android draws the screen, it starts with the highest-level container and then draws all the children and grandchildren on top of this parent view. This means that an individual pixel may be drawn more than once in a single cycle, in a process known as **overdraw**.

Coloring pixels that are only going to end up getting covered by subsequent views is a waste of processing power, and the more times you paint the screen, the more overdraw you're adding.

Overdraw is a particular problem for mobile devices such as Android, which have limited memory bandwidth to begin with and may struggle with GPU-intensive drawing tasks. By identifying and rectifying unnecessary or excessive overdraw, you'll increase your app's rendering speed.

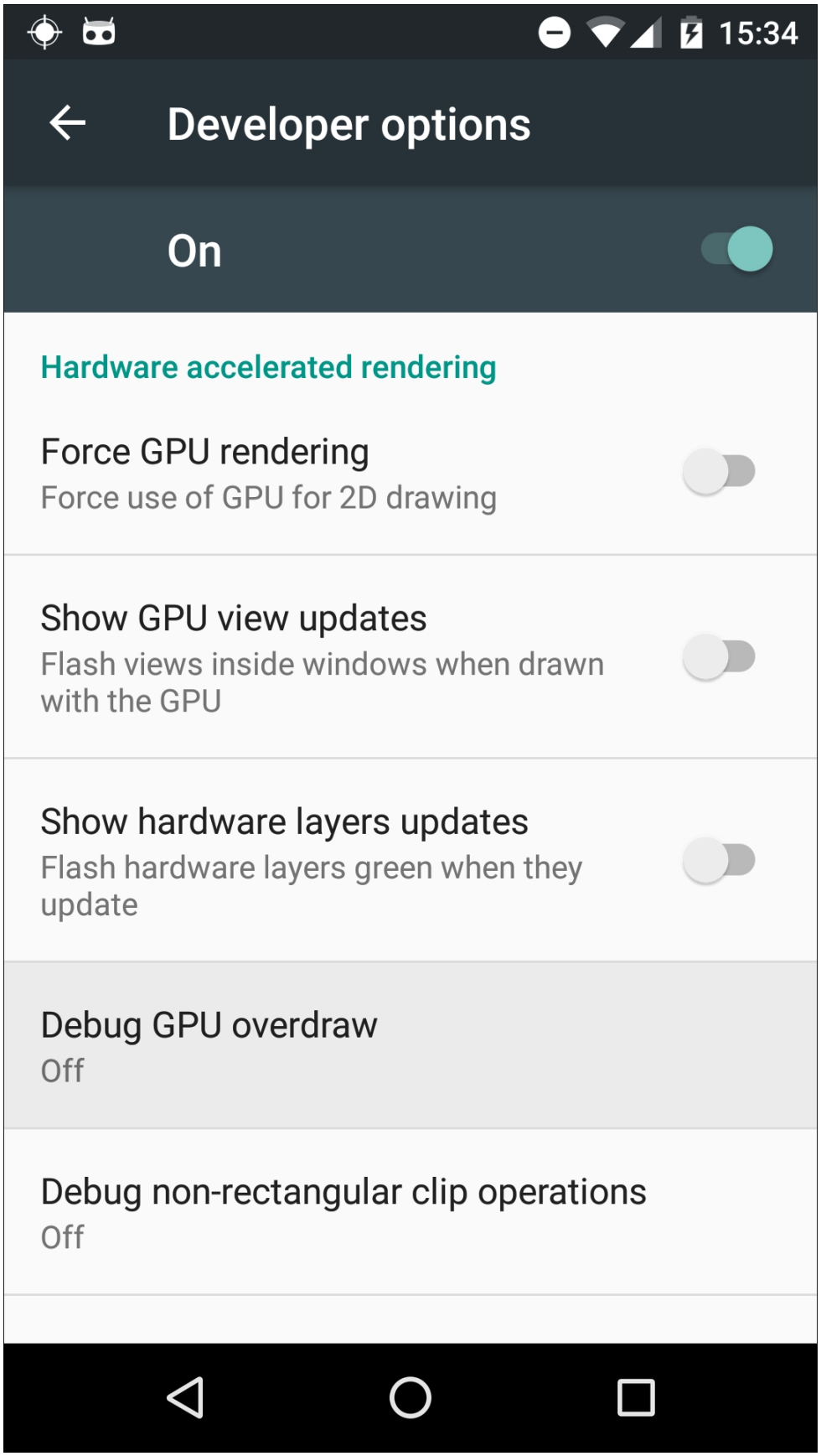
> ## Note
>
> Large amounts of overdraw can also indicate more serious problems with your user interface, so checking the amount of overdraw in your app can also point you in the direction of areas you should investigate further.

It's unrealistic to try and eliminate *every* instance of overdraw, as some areas of overdraw are normal and unavoidable. If your app has a blue background, then every element you place on top of that background is going to cause some overdraw, but this is unavoidable—can you imagine releasing an app that's just a single blue screen, and nothing else?

You only need to worry about excessive amounts of overdraw, such as multiple full-screen layers, or overdraw that doesn't contribute anything to the final image the user sees on screen, such as content that's completely hidden behind other views.

Android devices running 4.2 and higher have a built-in feature that lets you see the amount of overdraw present in any app installed on your Android device (and even the amount of overdraw present throughout the Android system, if you're curious about that sort of thing).

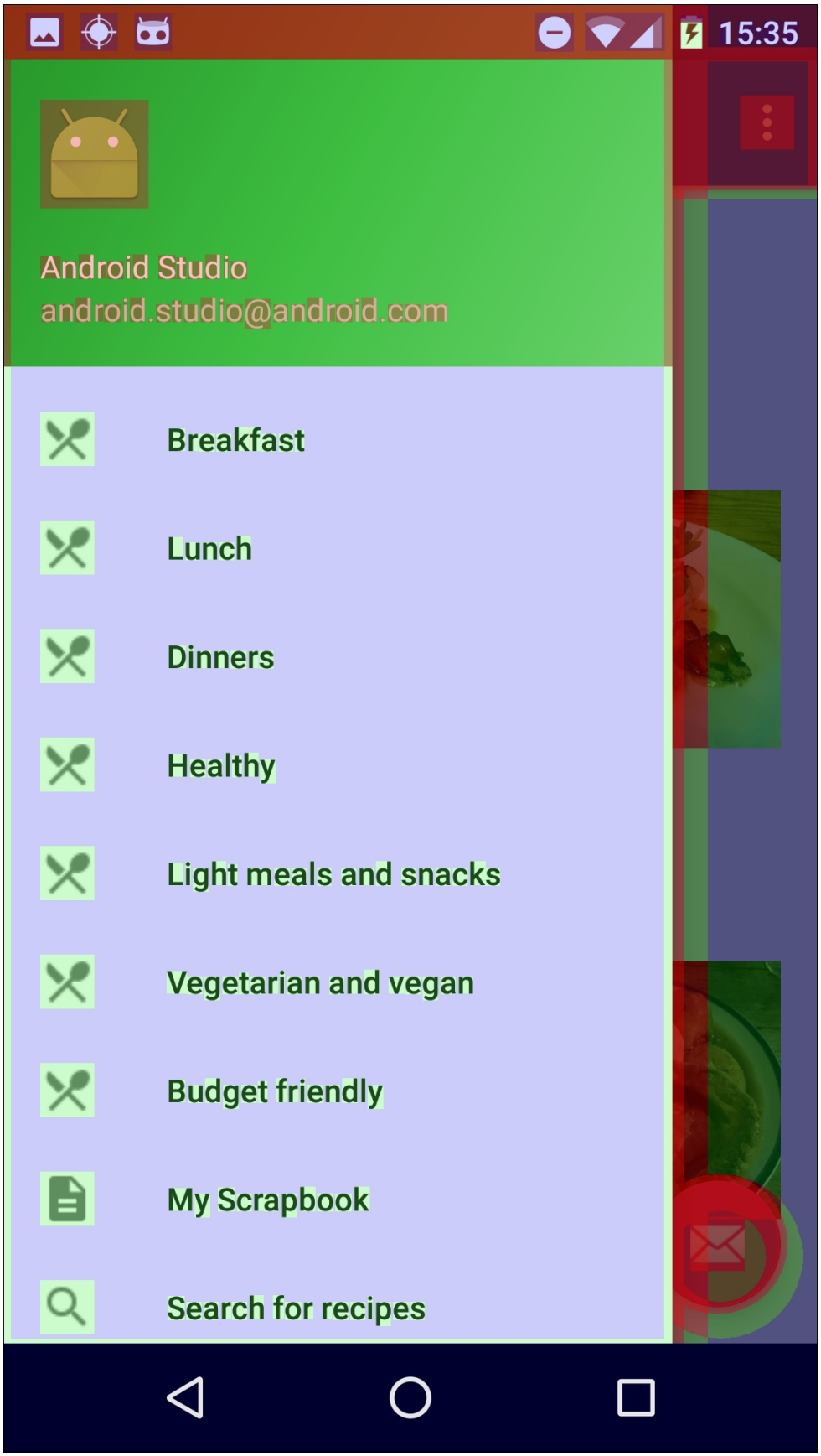
To see the amount of overdraw that's occurring in your app, install the app you want to test on a physical Android device running 4.2 or later. Then open your device's **Settings**, followed by **Developer options**. Give **Debug GPU Overdraw** a tap, and select **Show overdraw areas** from the pop-up that appears:


15:34
Developer options
On
Hardware accelerated rendering
Force GPU rendering
Force use of GPU for 2D drawing
Show GPU view updates
Flash views inside windows when drawn with the GPU
Show hardware layers updates
Flash hardware layers green when they update
Debug GPU overdraw
Off
Debug non-rectangular clip operations
Off

Note

The **Debug GPU overdraw** pop-up contains a **Show areas for Deuteranomaly** option, which serves the same purpose as the standard **Debug GPU Overdraw**, but with color correction to compensate for people with deuteranopia color blindness (reduced sensitivity to green light).

With this option selected, the Android system colors each area of the screen differently, depending on how many times each pixel has been drawn and redrawn.


15:35
Android Studio
android.studio@android.com
Breakfast
Lunch
Dinners
Healthy
Light meals and snacks
Vegetarian and vegan
Budget friendly
My Scrapbook
Search for recipes

These colors are your guide to areas where overdraw is a particular problem:

- **No Color = no overdraw**: These pixels were painted once.
- **Blue**: An overdraw of 1x, so these pixels were painted twice. In other words, the screen was drawn once, and then drawn again on top. Although the amount of overdraw you can afford varies from device to device, the majority of devices should be able to handle a single level of overdraw. Large areas of blue are acceptable, but if the entire window is blue then you may want to investigate further to see whether you can strip away some of this overdraw.
- **Green**: An overdraw of 2x. These pixels were painted three times. Medium-sized areas of green are okay, but if more than half of your screen is green you should look into whether you can optimize some of this green away.
- **Light red**: An overdraw of 3x. Some small areas of light red may be unavoidable, but any medium or large areas of red are a cause for concern.
- **Dark red**: An overdraw of 4x, which means this pixel was painted at least five times—possibly even more! You should *always* investigate any dark red areas.

When you're investigating an area of excessive overdraw, your first stop should be checking out the layout's corresponding XML file to see whether there's any obvious areas of overlap. In particular, be on the lookout for:

- Any drawables that aren't visible to the user.
- Any backgrounds that are being drawn on top of one another, particularly full-screen backgrounds. If your UI has several layers of background before your app even starts drawing its content, then you're inevitability going to end up with some serious overdraw.
- Any areas that draw a white background on top of another white background.

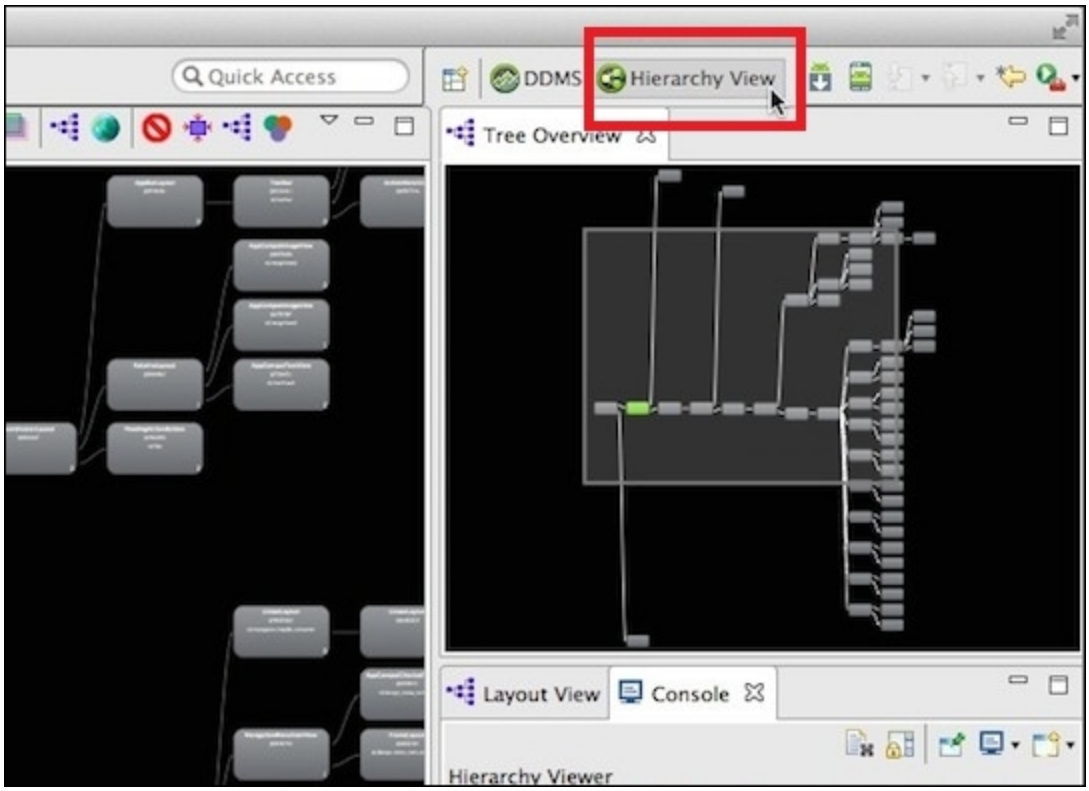
- `ViewGroups` nested inside `ViewGroups`. Look into whether you can replace these multiple `ViewGroups` with a `RelativeLayout`.

Another tool that's useful for identifying unnecessary views and nested layouts is the **Hierarchy View** tool that comes with the Android SDK and is accessed through the **Android Device Monitor**.

Like many of the diagnostic tools I'll be using throughout this chapter, **Hierarchy View** can only communicate with an app that's running in an AVD or a physical Android device. However, unlike the other diagnostic tools, **Hierarchy View** can only connect to a device that's running a developer version of the Android operating system. If you *don't* have a developer device, you can get around this restriction by adding the `ViewServer` class (https://github.com/romainguy/ViewServer) to your project.

If you're using a physical Android device, you also need to make sure debugging is enabled. Open your device's **Settings**, followed by **Developer options**, and then drag the **Android debugging** slider to the **On** position.

Open the Android Device Monitor (by selecting **Tools** | **Android** | **Android Device Monitor**) and click the **Hierarchy View** button (where the cursor is positioned in the following screenshot):



Select your device from the **Windows** tab and you'll see a list of all the Activity objects running on the selected device, listed by component name.

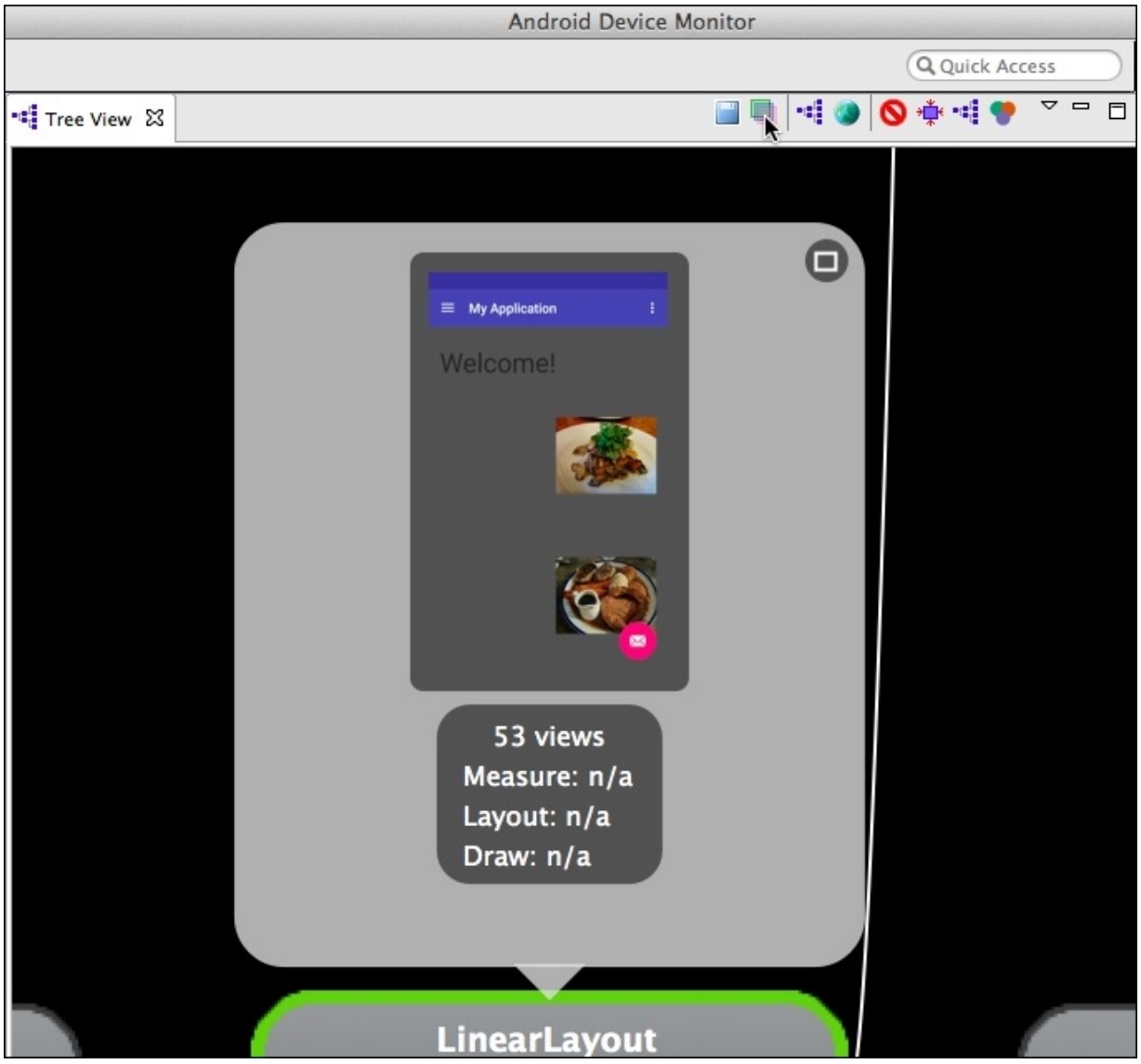
To populate the various **Hierarchy View** panes, click the blue **load the view hierarchy into the tree view** icon. Depending on how complex your app's view hierarchy is, the tree may take some time to load.

Once your project's **Hierarchy View** has loaded, you may want to spend some time exploring that hierarchy in the various windows (I'll be discussing these in more detail in the next section), but one of the quickest and easiest ways of identifying large areas of overdraw is to export the Activity's hierarchy as a Photoshop document.

When you create a PSD document from the **Hierarchy View** output, each `View` is displayed as a separate layer. This means you can use the PSD document to peel back each layer of your app, and see exactly what each layer is contributing to your UI. Inspecting each

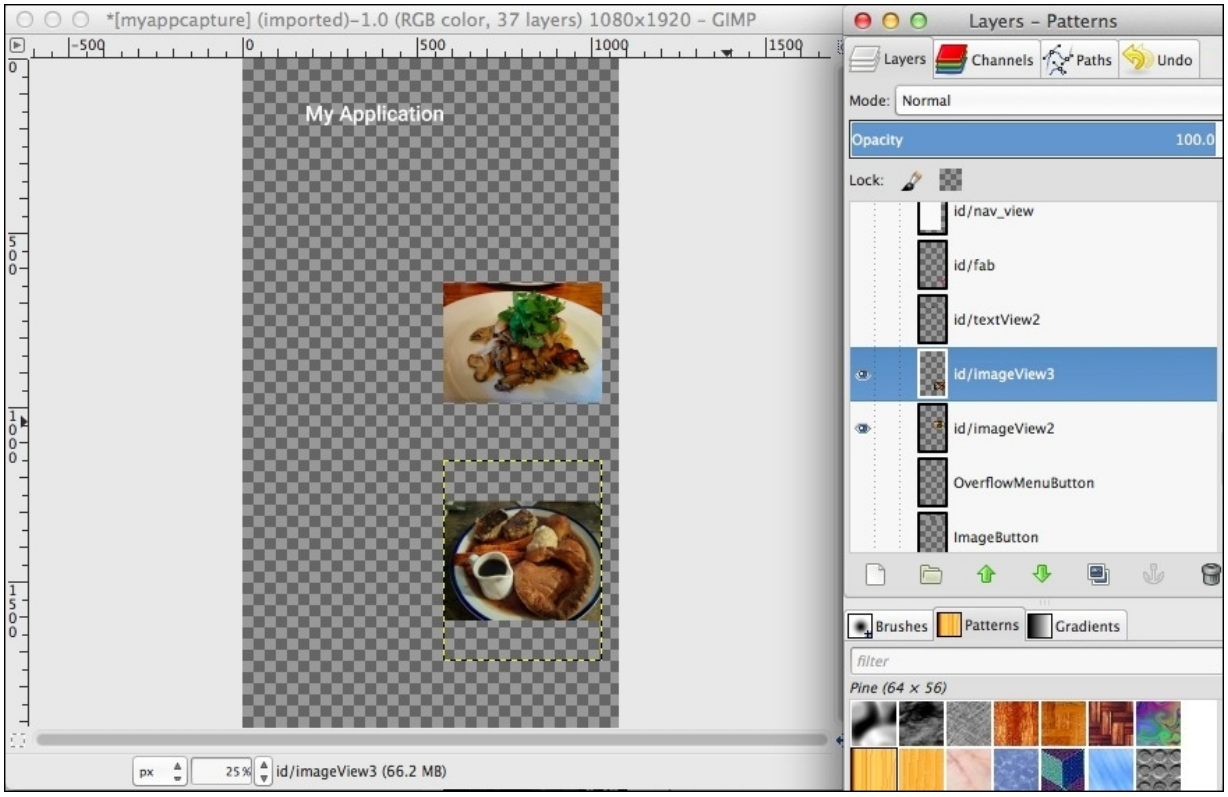
layer can help you identify sources of overdraw, or if you already suspect that certain areas of your app are suffering from overdraw, then you can put your theory to the test by hiding the different layers and seeing how this impacts the final rendered image the user sees on screen.

To export your hierarchy as a Photoshop document, click the **Capture the window layers as a Photoshop document** icon (where the cursor is positioned in the following screenshot):



This button generates a PSD file that you can inspect in Adobe Photoshop or in any image-editing tool that supports PSD files, such

as the free and open source Gimp program (https://www.gimp.org).



Examining the layers of our application in GIMP

Spend some time exploring the different layers that make up your UI.

This PSD document is particularly useful for identifying one of the major causes of overdraw: multiple white backgrounds. Multiple white backgrounds can be difficult to spot, so one trick is to replace the white backgrounds in your PSD file with different images. You can then check what portions of these images are visible as you move through the different layers of your UI.

Simplifying your Hierarchy View

Another common cause of performance problems is your app's **Hierarchy View**.

When the Android system renders each view, it goes through three stages: measure, layout, and draw. The time it takes the system to complete each stage is affected by the number of views in your hierarchy, and how these views are arranged.

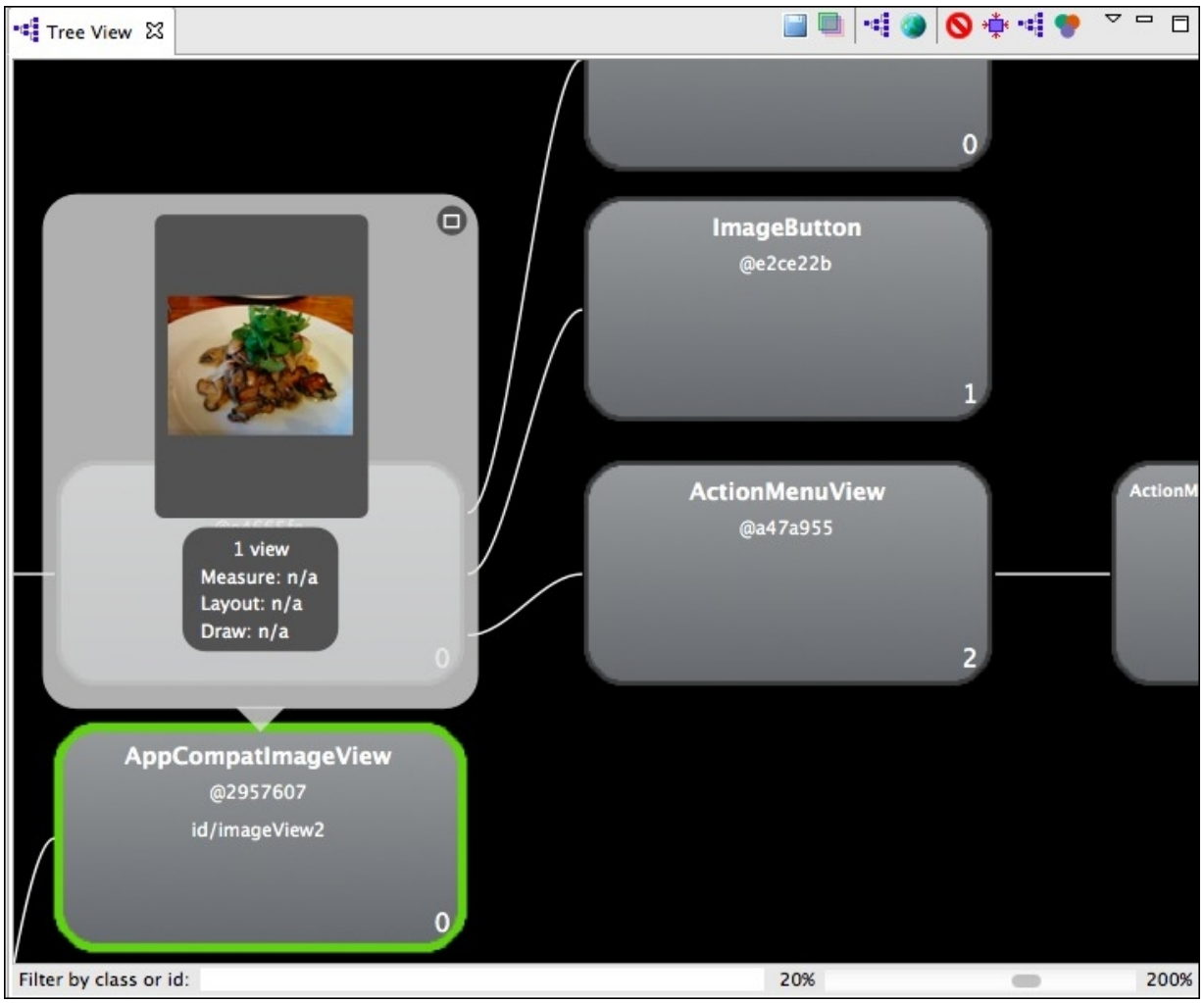
Arranging your views in deeper, more complex, hierarchies will have a noticeable impact on your app's rendering speed. You should be on the lookout for any opportunities to flatten your view hierarchy and remove nested layouts.

As well as highlighting areas of overdraw, **Hierarchy View** helps you visualize your app's view hierarchy and provides some really useful performance information about how long each view takes to render.

The **Hierarchy View** tool consists of three different windows.

Tree View

This window provides a bird's eye view of the currently selected Activity's view hierarchy:



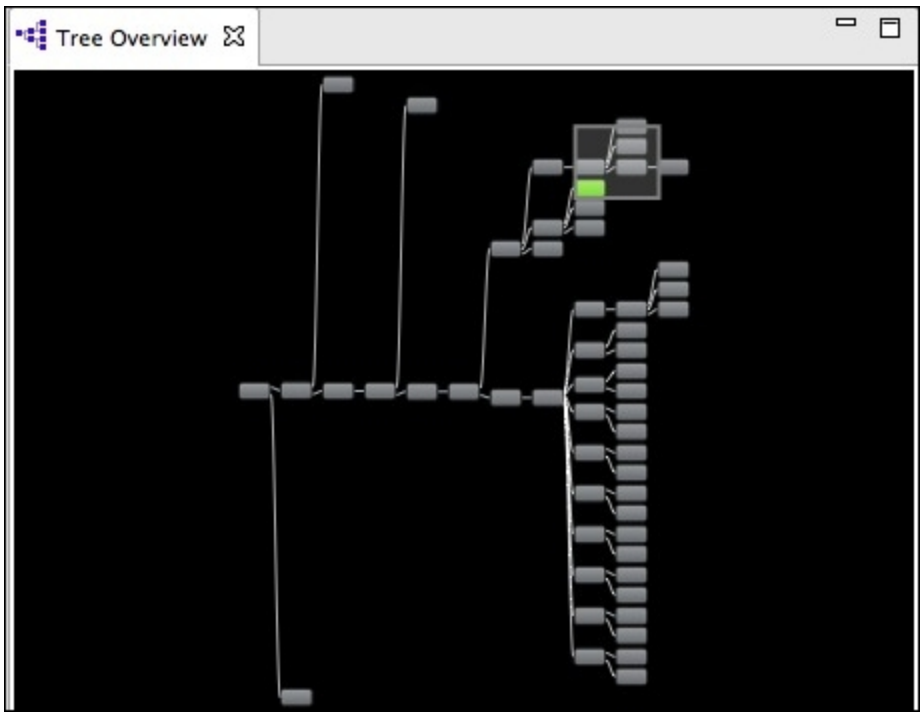
Each node in Tree View represents a single `View`. When you select a node, additional information about that `View` appears in a small window above the node:

- **View class**: The object's class
- **View object address**: A pointer to the `View` object
- **View object ID**: The value of the object's `android:id` attribute

You'll also see a preview of how this `View` will appear on an Android device. By seeing exactly what each `View` is contributing to the final UI, you can decide whether this `View` is adding anything of value. If it isn't, then you should remove it from your app.

Tree overview

This window contains a map representation of the Activity's entire view hierarchy:

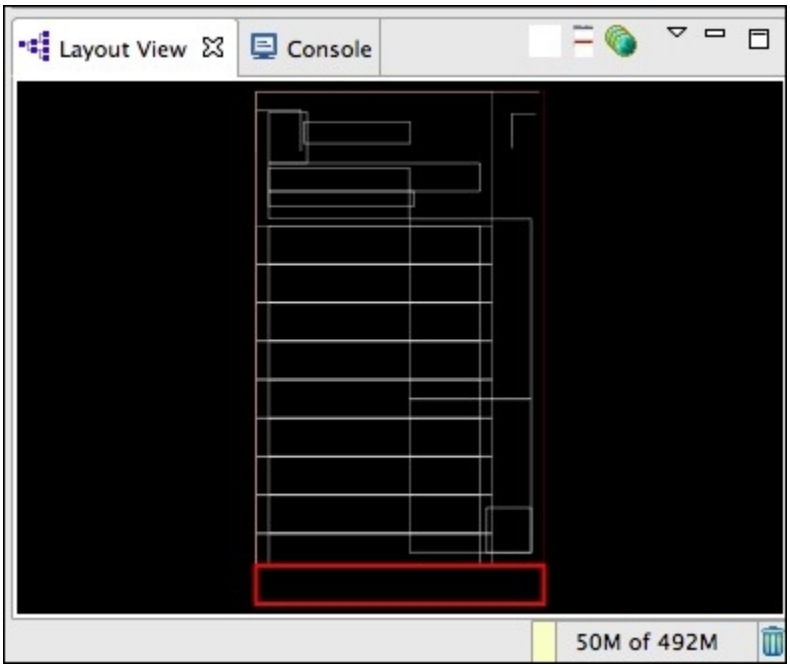


This high-level view of your hierarchy structure is particularly useful for seeing just how complex your view hierarchy really is, as well as helping you identify nested layouts and other opportunities to flatten your layout.

Layout View

This window displays a skeleton of your Activity's UI.

When you select a node in either the **Tree View** or **Tree Overview** window, the **Layout View** highlights the areas that this `View` paints. Again, this helps you weed out redundant `Views`:



While **Hierarchy View** is great for spotting nested layouts and redundant views, there are some view hierarchy issues that you can't spot just by looking at these three windows.

To help you identify any problems that may be lurking beneath the surface, you can use the **Hierarchy View** tool to measure how long it takes each `View` to move through each phase of the rendering process (measure, layout, and draw). Armed with this information, you'll know exactly what `Views` you need to optimize.

Hierarchy View's **Tree Overview** doesn't display render times by default. To add this information to the **Tree Overview**, you need to select the root node of the part of the tree you want to profile. Then click the green, red, and purple Venn diagram icon (when you hover over this icon, you'll see an **Obtain layout times for tree rooted at selected node** tooltip).

After a few moments, three colored dots will appear on each node within this section of the view hierarchy. These dots indicate the rendering speed of this `View` relative to the other profiled `Views`.

From left to right, these dots indicate the time it takes to:

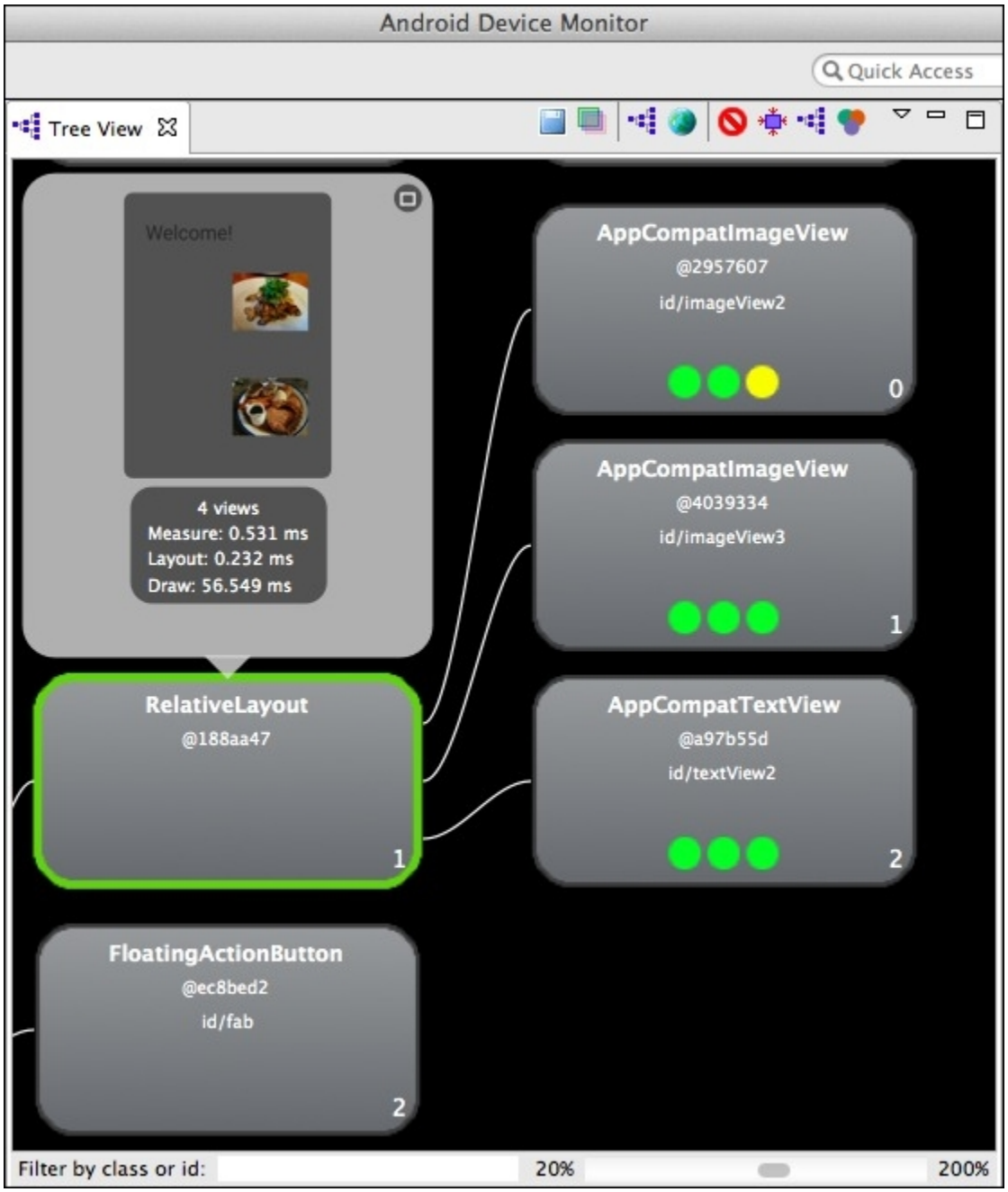
- Measure the view
- Layout the view
- Draw the view

The color of each dot indicates the time it takes the `View` to move through each phase of the rendering process:

- **Green**: This view is faster than at least half of the other profiled nodes. A green dot in the measure position means that this view has a faster layout time than at least 50% of the other profiled nodes.
- **Yellow**: This view is in the slowest 50% of all the profiled nodes.
- **Red**: This view is the slowest out of all the profiled nodes.

You can use this information to identify which `Views` are the slowest to measure, layout, and draw, so you not only know which `Views` you need to optimize but also the part of the rendering process you should be focusing on.

When you click a profiled node you'll also see the measure, layout, and draw times for that `View`, displayed in milliseconds.



Just remember that these performance indicators are relative to one another, so your view hierarchy is always going to include some red and yellow nodes.

Before you start looking for ways to optimize `Views` with yellow and red dots, ask yourself whether these `Views` have a good reason for rendering more slowly, for example views that have more children are always going to lag behind nodes that have less children.

Spotting memory leaks

Android may be a memory-managed environment, but you still need to scrutinize how your app is handling memory.

Garbage collection (**GC**) can *only* remove objects that it recognizes as unreachable. If your app allocates objects that the Android system doesn't identify as unreachable, then those objects are never going to get garbage collected. They're going to hang around, polluting your heap, and taking up valuable space.

As your app continues to leak objects that the system can't garbage collect, the amount of usable space will get smaller and smaller. The Android system will try to compensate for this shrinking memory by running longer and more frequent GC events.

While your typical GC event won't have a noticeable impact on your app's performance, as more and longer GC events start occurring in a small space of time, your users may notice a drop in performance, and may even encounter an `OutOfMemoryError`.

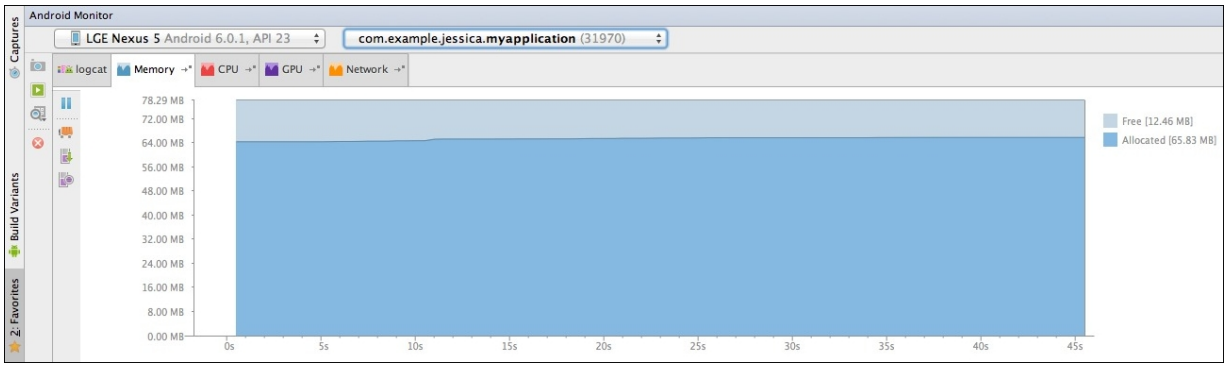
Memory leaks can be difficult to detect, but the Android SDK comes with several tools that you can use to scour your app for those sometimes subtle signs of memory-management problems.

Memory monitor

Memory monitor tracks your app's memory usage over time. This is another tool that can only communicate with a running app, so make sure to install your app on a physical device or an emulator before you proceed.

You can access memory monitor from the main Android Studio screen by selecting the **Android Monitor** tab towards the bottom of the screen, and then selecting the **Memory** tab.

Make sure the app you want to test is visible onscreen. As soon as memory monitor detects your running app, it'll start recording memory usage, displaying the memory your app is using in dark blue and the unallocated memory in light blue.



Tip

Troubleshooting

If Memory Monitor displays a **No debuggable applications** message, open Android Studio's **Tools** menu, select **Android,** and make sure **Enable adb integration** is selected. This feature can be temperamental, so if it doesn't work at first then try the *very* high-tech solution of toggling **Enable adb integration** on and off a few times. If you're using a physical Android device, it may also help to disconnect your device and reconnect it to a different USB port.

Spend some time interacting with your app while keeping an eye on memory monitor. Eventually your app's memory usage will swell until there's no unallocated memory left. At this point, the system will free up some memory by triggering a GC event, causing the allocated memory to drop.

Most GC events are perfectly normal, but if you see GC events becoming longer and more frequent then this is an indication that a memory leak may be occurring in your app.

If you track a suspected memory leak over a period of time, you may eventually see the Android system try to accommodate your app's insatiable thirst for memory by granting it a higher memory ceiling. If you see this happening in memory monitor, then this is a sign that a serious memory leak is occurring in your app, and you should investigate it in more detail.

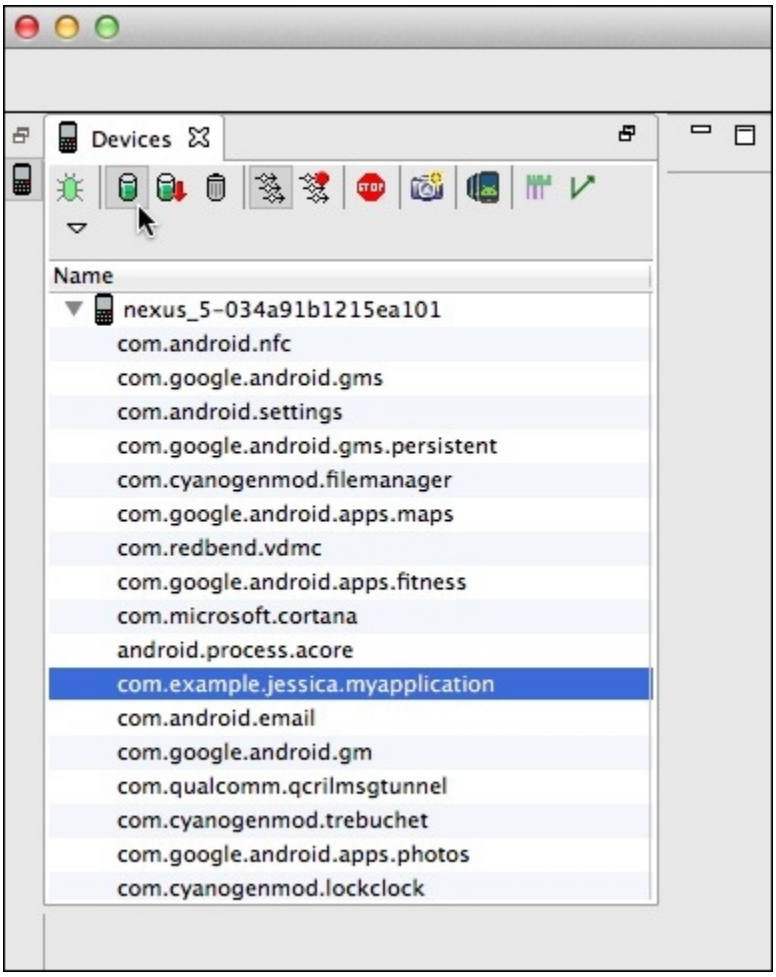
Heap tab

If you spot strange memory usage in the memory monitor, you can use the Android Device Monitor's Heap tab to gather more information about how your app is using memory.

As the name suggests, this tab provides data about your app's heap usage, including the kind of objects your app is allocating, the number of allocated objects, and how much space these objects are taking up.

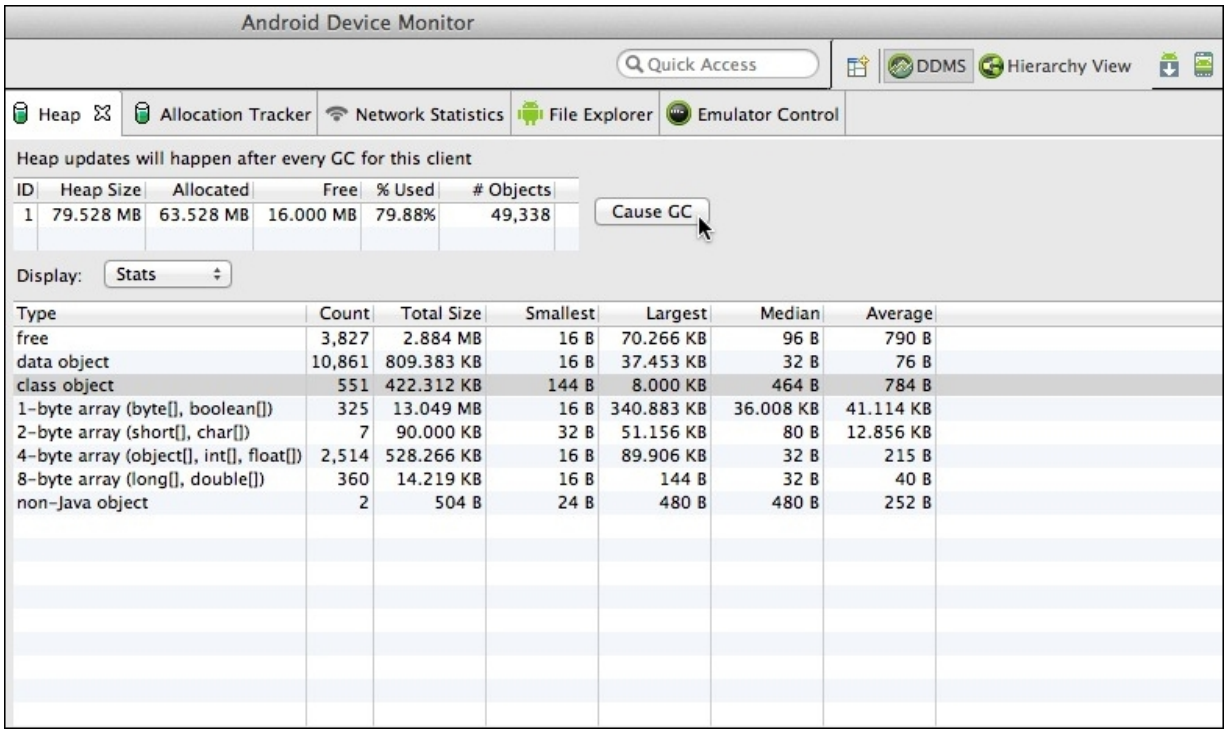
Android smartphones and tablets have a finite amount of heap space that can only accommodate a certain number of objects. As the heap grows, the Android system will try to free up memory by triggering a GC event—which we already know is bad news for performance.

To access the heap tab, launch the **Android Device Monitor**, select the **DDMS** tab, and then select your device or AVD from the **Devices** panel, followed by the process you want to examine. Click the **Update heap** button (where the cursor is positioned in the following screenshot):



Select the **Heap** tab and spend some time interacting with your app.

The heap output is only displayed after a GC event has occurred, so you'll either have to be patient and wait for an organic GC event, or you can force a GC event by clicking the **Cause GC** button. Once a GC has occurred, the Heap tab will display information about your app's heap usage.



Spend some time interacting with your app, triggering GC events before *and* after you perform different actions so you can compare how these actions impact the heap. In this way, you can isolate the actions that are causing the memory leak, as well as any other memory-related problems your app is experiencing.

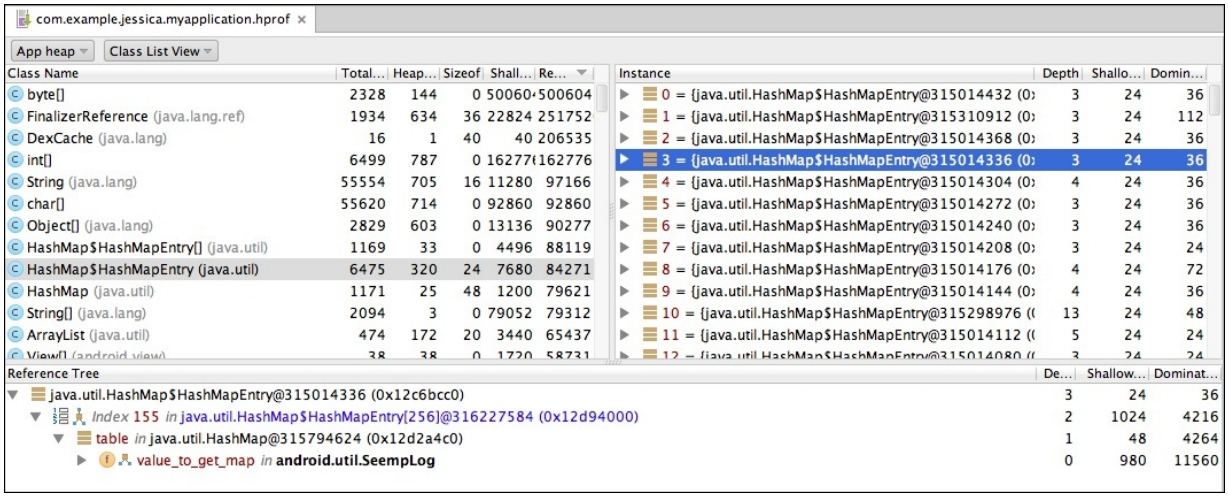
When you're tracking down problems in your app's heap, it may to help to generate a HPROF file, which is a snapshot of all the objects in your app's heap plus detailed information about the related classes and instances.

Once you've generated a HPROF file, you can view it in Android Studio or in a separate profiling tool, such as the Eclipse Memory Analyzer (http://www.eclipse.org/mat).

To retrieve a HPROF file, click the **Dump HPROF** icon (next to the **Update Heap** icon). Give your file a name and save it.

To analyze your heap dump in Android Studio, open your HPROF file as a new Android project. Android Studio will automatically open the

file in its Android Memory HPROF Viewer, ready for you to analyze in more detail.



Object allocation – understanding memory churn

Another common memory problem that you should check for is memory churn.

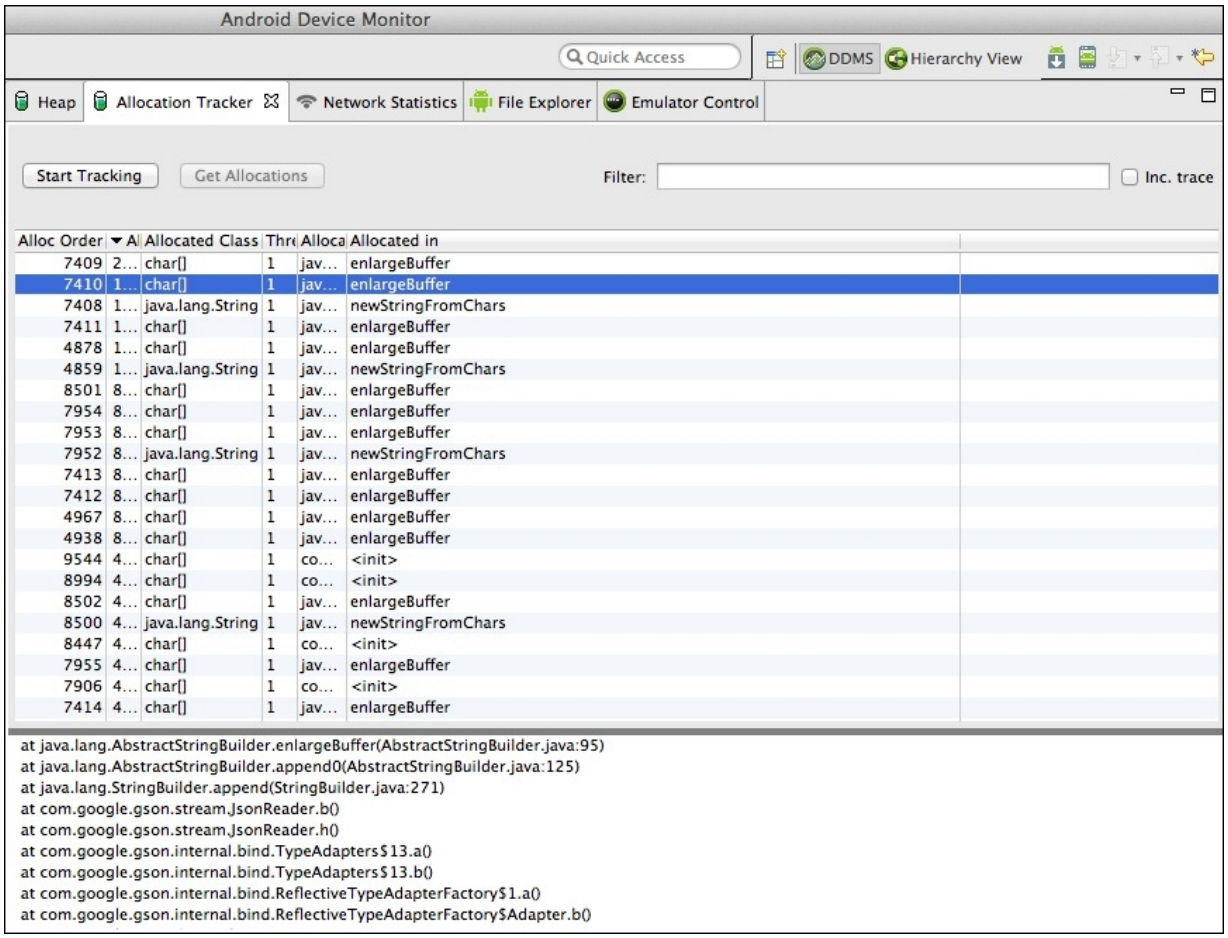
Memory churn occurs when your app allocates lots of temporary objects in a short period of time, which can quickly gobble up a device's available memory, triggering those performance-sapping GC events.

You can check for memory churn using the Android SDK's **Allocation Tracker**, which lists all the objects your app is allocating to memory when you perform different actions. If you spot any suspicious-looking allocations, you can then use **Allocation Tracker** to inspect the classes and threads that are responsible for allocating these objects.

Inside the Android Device Monitor, select the **DDMS** tab and then open the **Allocation Tracker** tab. Select your device or AVD from the **Devices** tab, followed by the process you want to examine.

In the **Allocation Tracker** tab, click the **Start Tracking** button and spend some time interacting with your app. To see a list of all the objects that have been allocated since you started tracking, click the **Get Allocation**s button. Android Studio will then open a tab that displays all the allocations that occurred during this sampling period.

Click on any allocated object to see more information about that object:



When you're ready to stop collecting data, click the **Stop Tracking** button.

Each row in the **Allocation Tracking** tab represents a specific allocation, and provides the following information for that allocation:

- Allocation order

- Allocation size
- Allocated class
- Thread ID. This is the thread that made the allocation
- Allocated In. This is the function in your code that's responsible for this allocation

Debugging your project

It's crucial that you thoroughly test your app for bugs before releasing it. Android Studio provides a range of tools that you can use to debug an app running on an emulator or a physical Android device, although these tools can only test a debuggable version of your app, which means you'll need to run your app in debug mode.

To run your app in debug mode, open your project's module-level `build.gradle` file and add the following:

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

.........

.........
  }
   buildTypes {
           debug {
                   debuggable true
             }
    }
}
```

After editing your Gradle build file, make sure you sync your project. Next, click the **Debug** icon or select **Run** from the Android Studio toolbar, followed by **Debug**.

Select the AVD or physical Android device where you want to install and test the debuggable version of your app. Once your app has loaded, Android Studio's **Debug** perspective should open automatically.

Note

If the **Debug** perspective doesn't open automatically, select **View** from the Android Studio toolbar, followed by **Tool window**

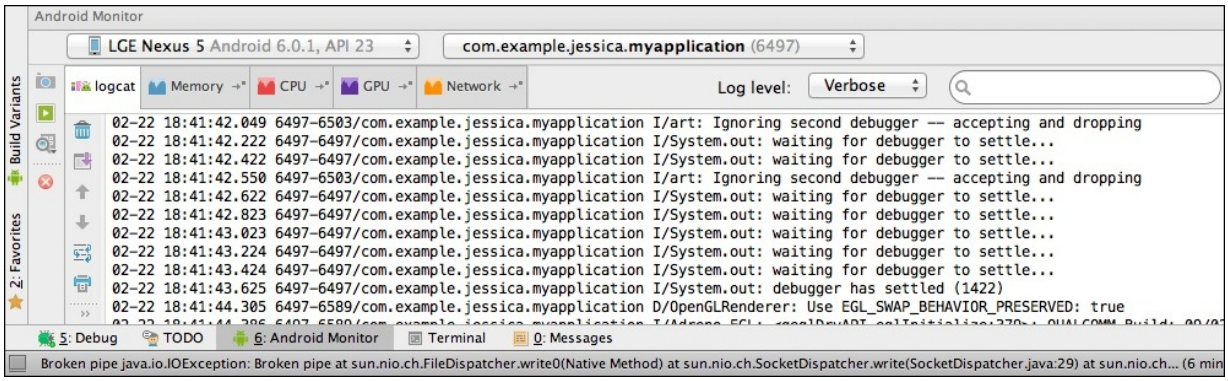
and **Debug**.

The Debug perspective consists of:

- **Debugger**: Displays threads and variables
- **Console**: Displays the device status

To debug an app that's already running, click **Attach debugger to Android process**. In the **Choose Process** window, select the device and the app you want to attach the debugger to and then click **OK**.

Once you have a debuggable version of your app up and running, you can view the log messages related to your application. To view these messages, select the **Android Monitor** tab that appears along the bottom of the Android Studio screen and then select the l**ogcat** tab:



Logcat can sometimes be a case of information overload, so Android Studio provides several ways of filtering the logcat output. One method is to use the **Log Level** drop-down menu. By default, this menu is set to **Verbose**, which displays all log messages, but there are several more specific options you can choose from:

- **Debug**: Displays log messages that are useful during development, plus message levels lower in this list
- **Info**: Displays expected log messages for regular usage, plus message levels lower in this list

- **Warn**: Displays possible issues that are not yet errors, plus message levels lower in this list
- **Error**: Displays issues that have caused errors, plus message levels lower in this list
- **Assert**: Displays issues that should *never* happen

If none of these filters meet your debugging needs, you can create custom filters. Open the **Show only selected application** dropdown (towards the right side of the **logcat** panel) and then select **Edit Filter Configuration**.

This opens a **Create New LogCat Filter** window, where you can craft a new filter by providing the following information:

- **Filter name**: If you're creating a new filter, you should give this filter a unique name. If you're modifying an existing filter, select it from the left-hand pane and its name will appear in this field.
- **Log Tag**: Every log message has a tag associated with it, which indicates the system component the message originated from. If you want to see messages that originate from a certain system component only, you can enter that component's tag here.
- **Log Message**: If you only want to see messages that contain certain elements or character strings, specify them in the **Log Message** field.
- **Package Name**: If you want your filter to display messages that relate to a certain package only, enter this package name here.
- **PID**: If you only want to see messages that refer to a specific process, enter that process ID here.
- **Log Level**: To filter based on log level, open this dropdown and select anything other than the default **Verbose** option.

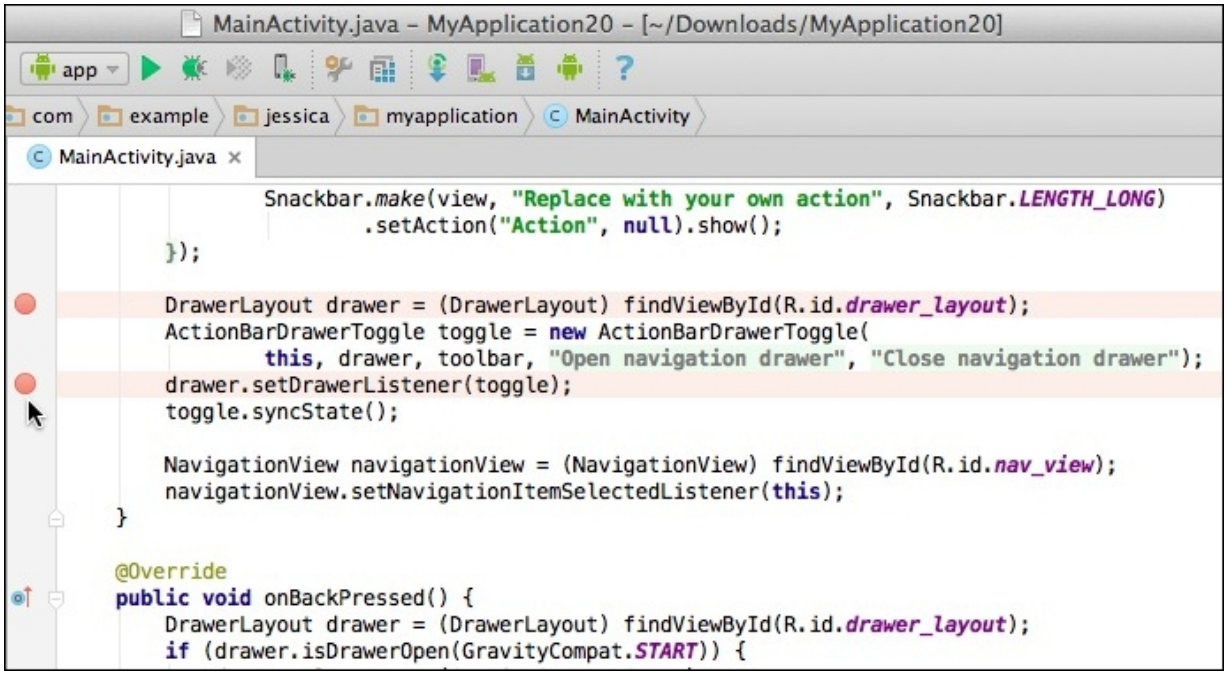
Working with breakpoints

Having trouble working out where an error is originating from? You can use breakpoints to pause the execution of your app at a particular line of code. By creating multiple breakpoints and

scrutinizing your app every time it comes to a stop, you can gradually isolate the portion of code that's responsible for the error.

To set a breakpoint:

1. Open the file where you want to create your breakpoint.
2. Locate the line where you want to set your breakpoint and click on this line.
3. Click the yellow portion that appears in the left-hand sidebar. A red dot will appear, indicating that you've successfully created a breakpoint.



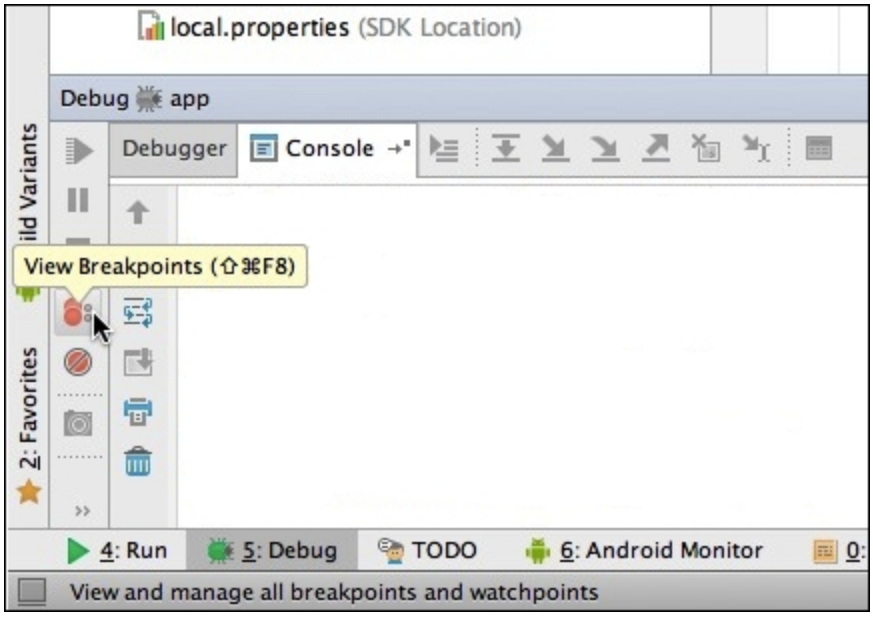
After creating your breakpoints, click the Rerun app icon (the green **Play** icon along the left-hand toolbar).

Every time Android Studio reaches a breakpoint, it'll pause the execution of your app and highlight the triggered breakpoint in your code. You can then open the **Debug** window (by selecting the **Debug** tab in the bottom toolbar) and use the debugger and console to gather more information about what's going on in your app at this exact point in the execution of your code. You may also

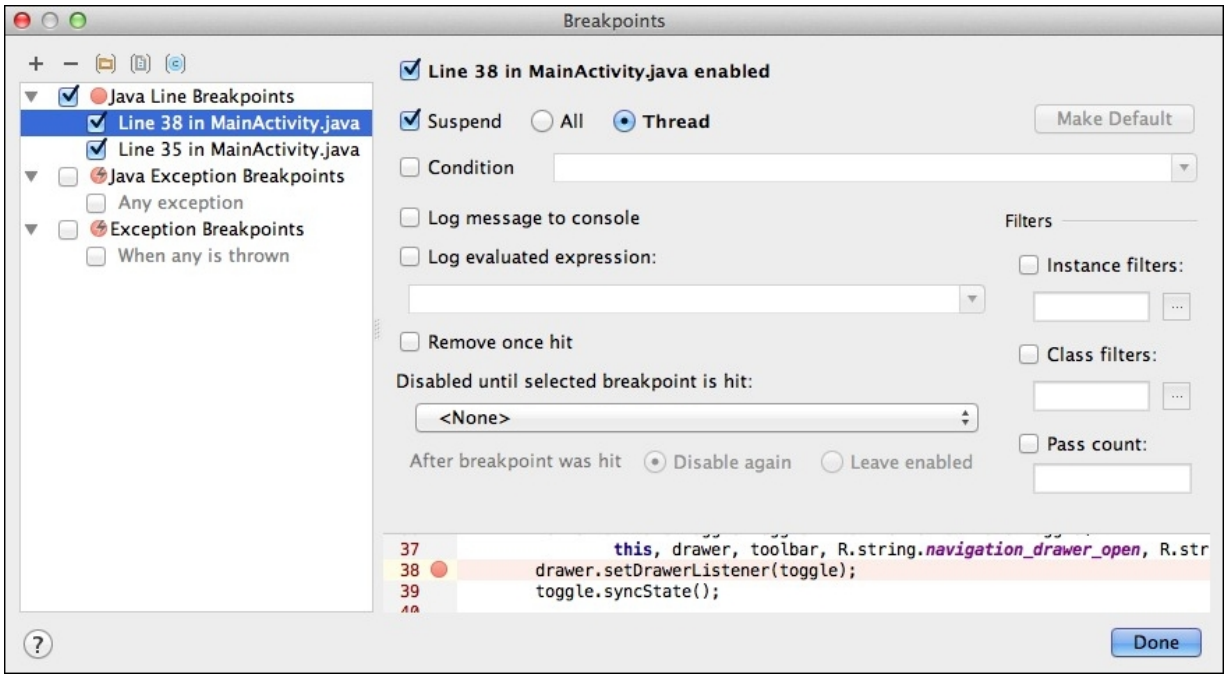
want to take a look at the **logcat** output. Repeat this process until you've isolated the code that's causing the problem.

Configuring your breakpoints

If you have a very specific kind of breakpoint in mind, you can make some changes to your breakpoint settings. Start by clicking the **View Breakpoints** icon (where the cursor is positioned in the following screenshot):



The Breakpoint window appears and lists all the breakpoints you've created in the current project. To see what changes you can make to a breakpoint, select that breakpoint from the left-hand list.



This window gives you lots of different options for configuring the selected breakpoint:

- **Suspend**: Select this checkbox to enable a suspend policy for this breakpoint, then choose from **All** (when a breakpoint is hit, all threads are suspended) or **Thread** (when a breakpoint is hit, only the thread where the breakpoint is hit is suspended).
- **Condition**: Select this checkbox, then in the accompanying textbox specify a condition for hitting this breakpoint. The condition must be a Java Boolean expression with a `true/false` value. This condition is evaluated each time the breakpoint is reached, and if the result is true the specified action is performed.
- **Log message to console**: Select this checkbox to display a log message in the console when this breakpoint is hit.
- **Log evaluated expression**: Select this checkbox to evaluate an expression when this breakpoint is hit and display the results in Android Studio's console.
- **Remove once hit**: When enabled, this breakpoint will be triggered once and then removed.
- **Disabled until the selected breakpoint is hit**: This breakpoint is dependent on another breakpoint, and will only be enabled

once the specified breakpoint has been triggered.

- **Instance filters**: To limit breakpoint hits to instances of a particular object, select this checkbox and then provide the ID value of the instance you want to use.
- **Class filters**: Select this checkbox to have this breakpoint behave differently in reaction to different classes. Then, specify the classes that'll trigger the breakpoint in the accompanying textbox. To define classes where the breakpoint *shouldn't* be triggered, add these classes to the textbox but preface them with a minus symbol.
- **Pass count**: Define the number of times a breakpoint should be reached, but ignored, by selecting this checkbox and then specifying the number of times the breakpoint should be skipped. After the specified number of passes, the breakpoint will be triggered as normal.

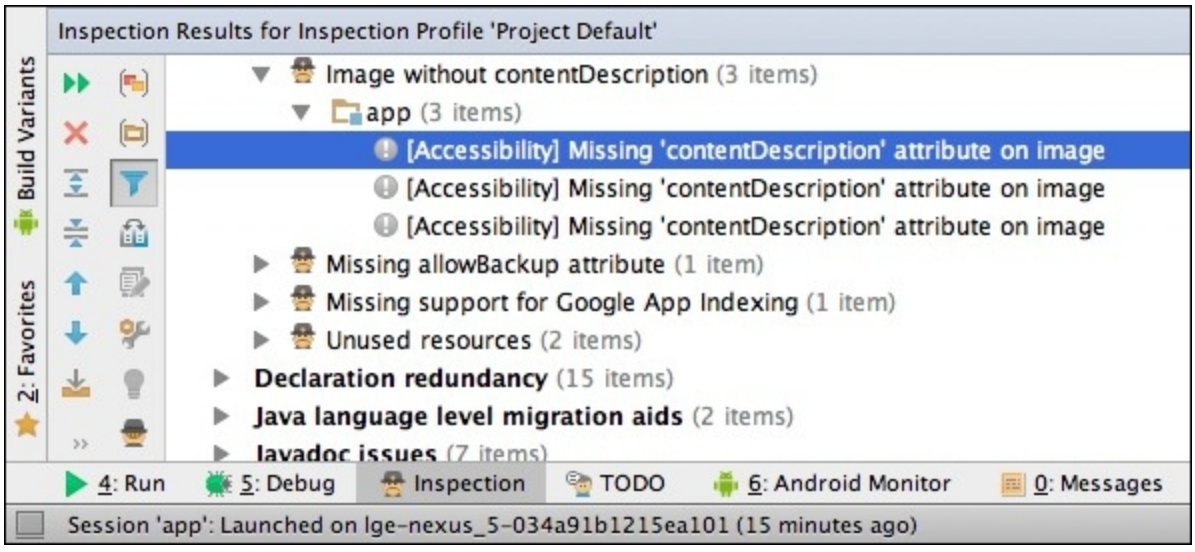
Examining your code with Lint

It's important to check that there are no problems with the structural quality of your code, as these can cause errors and have a negative impact on your app's performance. Conveniently, the Android SDK comes with Lint, a static code-scanning tool that's *perfect* for identifying and correcting structural problems with your code.

In Android Studio, the configured Lint expressions run every time you build your app and print the output to Android Studio's **Events log**, which you can access by selecting the **Events log** tab.

However, you can also run Lint on a specific module at any time, by right-clicking on that module's file or folder and then selecting **Analyze | Inspect code,** followed by the area you want to inspect (**Whole project, Module**, or **Custom Scope**). Make your selection, and Android Studio automatically opens a new **Inspection** tab where you can view the Lint output.

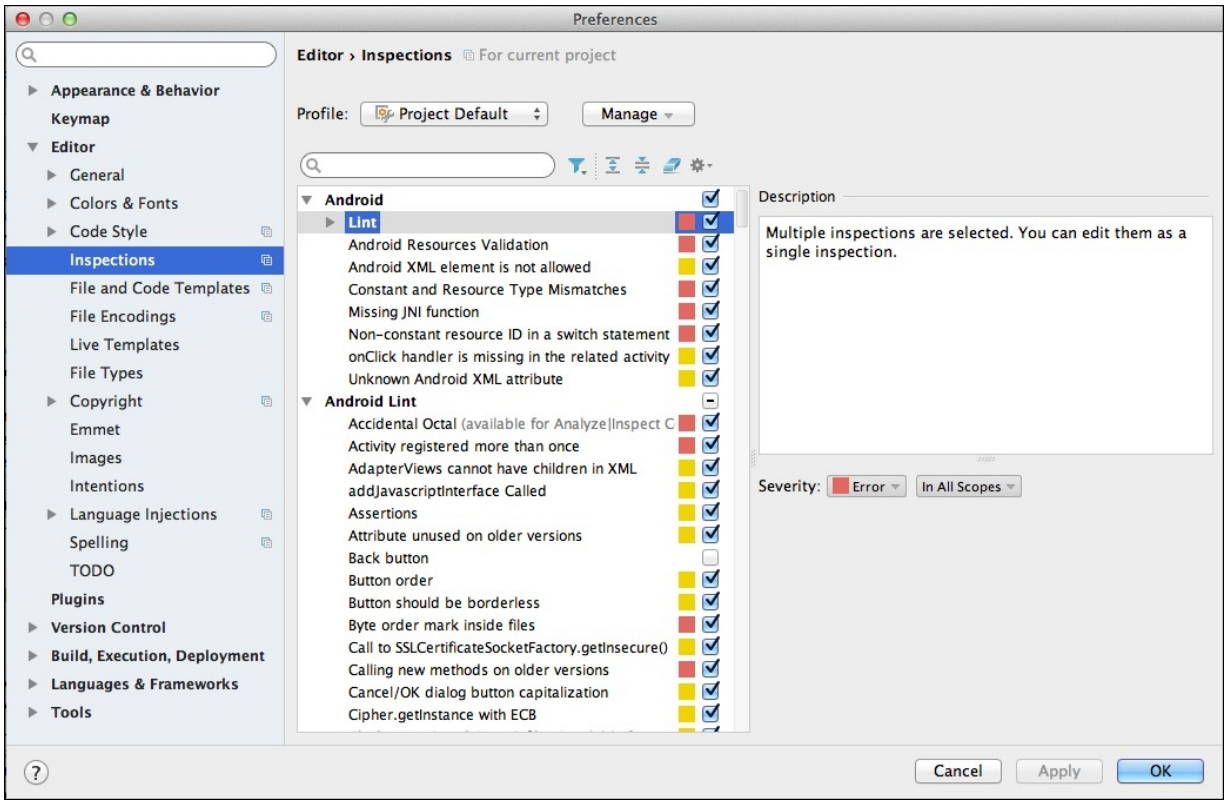
To help you zero in on critical problems, Lint provides a description and a severity level for every issue it reports.



Make sure you correct *all* errors that Lint detects before you release your app.

If you need to make some changes to the default Lint settings, select **Android Studio** from the toolbar, followed by **Preferences**. In the window that appears, double-click **Editor** and select **Inspections**.

This opens the **Inspection Configuration** page, which lists all the supported Lint profiles and inspections.



Here, you can explore the different inspections and make edits such as changing their severity level and scope.

You can also run Lint inspections for a specific build variant or for all build variants, by adding the `lintOptions` property to your project's module-level `build.gradle` file. For example, if you wanted to set your Lint's `abortOnError` option to `false`, you'd need to add the following to your `build.gradle` file:

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

android {
```

```
...........
............
............

  lintOptions {

//Lint shouldn't exit the process when it
discovers errors//

          abortOnError false
    }
```

You can find a complete list of all your Lint configuration options at Google's GitHub (http://google.github.io/android-gradle-dsl/current/com.android.build.gradle.internal.dsl.LintOptions.html#com.android.build.gradle.internal.dsl.LintOptions).

Optimizing your code with ProGuard

The ProGuard tool shrinks and optimizes your code by removing unused code and renaming classes, fields, and methods with semantically obscure names. The end result is a smaller `APK` that's more difficult to reverse engineer, something that's particularly important if your app has access to sensitive information.

ProGuard runs automatically when you build your app in release mode. To build a release version of your app, you need to enable the `minifyEnabled` property in your project's module-level `build.gradle` file and make sure `buildTypes` is set to `release`. For example:

```
android {   ...  buildTypes {

 //The buildTypes element controls whether your
app is built in debug or release mode//
        release {

//In this example, we're building a release
version of our app so we can run ProGuard. If your
build.gradle file contains the debug attribute,
then make sure you remove it so it doesn't prevent
ProGuard from running//

           minifyEnabled true

//To enable ProGuard, add the minifyEnabled
property and set it to true//

            proguardFiles
getDefaultProguardFile('proguard-android.txt'),
           'proguard-rules.pro'
        }
    }
  }
```

The `getDefaultProguardFile` attribute obtains the default ProGuard settings specified in the `Android/sdk/tools/proguard/proguard-android.txt` file that you downloaded as part of the Android SDK. Alternatively, you can use the `proguard-android-optimize.txt` file, which contains the same rules but has optimization enabled.

If you want to add some project-specific options to the default ProGuard settings, open your project's `Gradle Scripts/proguard-rules.pro` file and add your new rules.

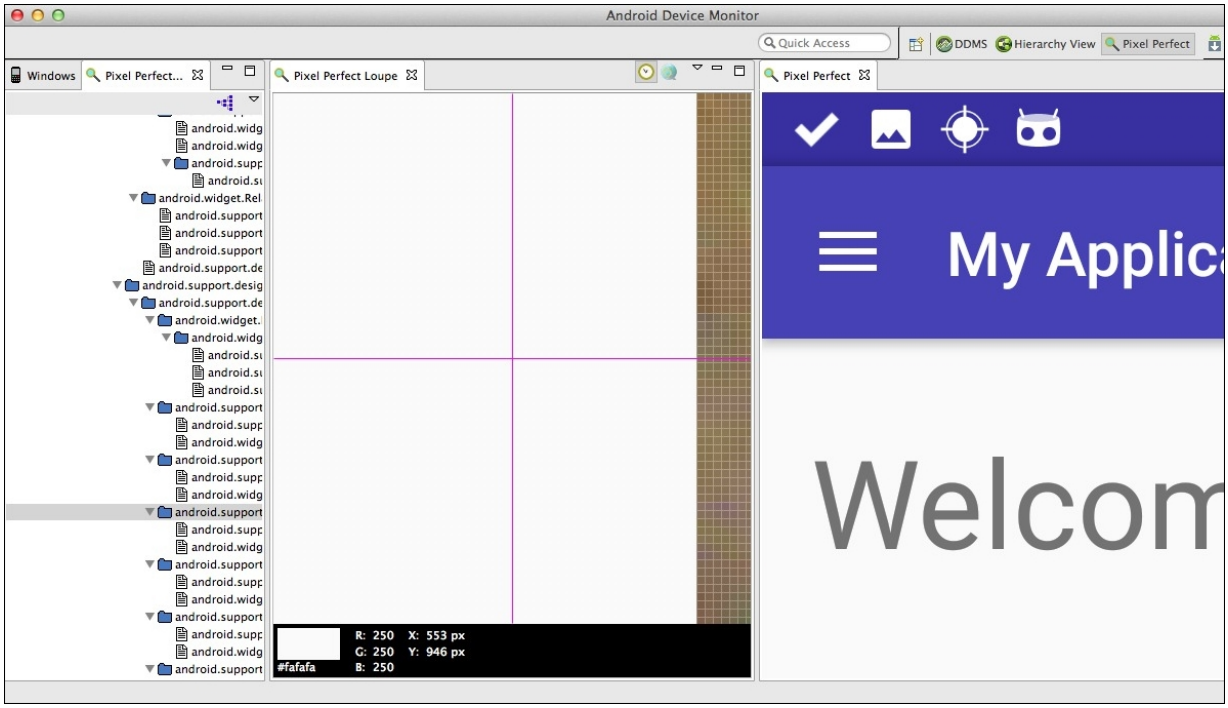
For more information about the different ProGuard settings that you can add to this file, check out the ProGuard manual (https://stuff.mit.edu/afs/sipb/project/android/sdk/android-sdk-linux/tools/proguard/docs/index.html#manual/retrace/examples.html).

Scrutinize each pixel

Another tool you may want to explore is **Pixel Perfect**.

The **Pixel Perfect** window displays a magnified version of the screen that's currently visible on the attached Android device or emulator, and lets you scrutinize the individual pixels that make up your UI. You can also use it to overlay an image over your UI, which is handy for checking how your UI compares to your digital wireframes.

Pixel Perfect is integrated into the **Android Device Monitor**. To launch **Pixel Perfect**, select **Window** from the **Android Device Monitor** toolbar, followed by **Open Perspective**, **Pixel Perfect,** and **OK**.



The **Pixel Perfect** window contains the following areas.

Pixel Perfect pane

This window displays a magnified version of the UI that's currently visible on the connected Android device or AVD.

By default, Pixel Perfect doesn't update automatically to reflect changes that occur on screen, so you'll need to keep clicking this pane's **Refresh Screenshot** icon. Alternatively, if you want Pixel Perfect to update automatically, then select the **Automatically Refresh the screenshot** icon.

One of Pixel Perfect's most useful features is the ability to load `jpg`, `jpeg`, `png`, `gif`, or `bmp` images as an overlay. This is particularly useful if you want to take stock of how the current screen compares to your original design, as you can load a digital wireframe (or any other digital design docs you have to hand) as an overlay.

To load an image as an overlay, navigate to the screen you want to work with and make sure it's being displayed in the Pixel Perfect window. Select the **Load an image to overlay the screenshot** icon, and then select the image you want to use as your overlay.

Pixel Perfect tree

This is a hierarchical list of all the `View` objects that are currently visible. Just be aware that system objects also appear in this list.

Pixel Perfect Loupe pane

This pane contains the magnified screen image, overlaid by a grid where one square represents one pixel. To see more information about a particular pixel, select it and the Loupe pane will display the following:

- **Pixel swatch**: A rectangle filled with the same color as the selected pixel
- **HTML color code**: The pixel's corresponding hexadecimal RGB code
- **RGB color values**: The pixel's R, G, and B color values

- **X and Y coordinates**: The pixel's coordinates, as a px value

Processes and threads

How your app handles threads and processes has a significant impact on your app's performance. By default, when the user launches an app, the Android system creates a single thread of execution for that application. All components run in this single thread, which is known as the *main* thread or the *UI* thread.

Unless you specify otherwise, most of the operations you perform in your app run in the foreground on this main thread. Most of the time this single-thread model works fine, but if your app needs to perform particularly intensive work or long-running operations, then the main thread can become blocked. This can cause your app to freeze, display system errors, and potentially even crash.

If you're going to deliver a good user experience, then it's vital you don't block the UI thread with intensive or long-running operations. If you do need to run demanding processes, then you should create additional threads.

This involves specifying which process a certain component belongs to in your project's `Manifest` file. The `Manifest` entry for each type of component (activity, service, receiver, or provider) includes an `android:process` attribute that specifies the process where this component should run. If you want the system to create a new process for this component, preface the `android:process` value with a colon, for example `android:process=":myprocess"`.

If you need to handle more complex interactions, you may want to consider using a `Handler` to process messages delivered from the main thread, or you could use the `AsyncTask` class to simplify the execution of worker thread tasks that need to interact with your UI.

Tip

Using AsyncTask

`AsyncTask` provides you with an easy way of performing synchronous work on your user interface without blocking the main thread. Using `AsyncTask`, you can separate tasks into the work that should be performed on the main thread and operations that should be performed on a separate worker thread. In this way, `AsyncTask` is a way of executing some work in a background thread and then publishing the results back to the main thread.

To use `AsyncTask`, you need to subclass `AsyncTask` and then implement the `doInBackground()` callback method, which executes automatically on a worker thread and performs background operations. The value returned by `doInBackground()` is then sent to `onPostExecute()`, and from here you can run the task by calling `execute()` on the main thread.

Ideally, you should only use `AsyncTasks` for short operations. If you need to keep threads running for more than a few seconds, it's recommended that you use APIs such as `Executor`, `ThreadPoolExecutor`, and `FutureTask`, which are provided as part of the `java.util.concurrent` package.

Terminating processes

When deciding how your application should handle processes, it's worth bearing in mind that even though the Android system will try and maintain all processes for as long as possible, it will kill off processes if memory starts getting low.

When deciding what processes to keep alive and which processes to terminate, the system decides how important each process is to the user by placing these processes in an **importance hierarchy**. Processes with the lowest importance are the first to go, while processes at the top of the importance hierarchy are rarely killed off.

> **Note**
>
> The Android system will increase the ranking of a process if another, more important, process depends on it. A process that's supporting another process can *never* be ranked lower than the most important process it's supporting.

Ranked from most important to least important, the different levels of Android's importance hierarchy are detailed in the following sections.

Foreground processes

This is a process that's essential for the actions the user is currently performing. The Android system assigns a process this ranking if it hosts either:

- An `Activity` that the user is interacting with
- A `Service` that's running in the foreground
- A `Service` that's bound to the activity the user is currently interacting with
- A `Service` that's executing one of its lifecycle callbacks (`onCreate`, `onStart`, or `onDestroy`)
- A `BroadcastReceiver` that's executing its `onReceive()` method

If any of these conditions are true, that process is considered a foreground process. The Android system only kills foreground processes as a very last resort.

Visible processes

This is a process that doesn't have any foreground components but may still affect what the user is seeing on screen. The Android system assigns this ranking if the process hosts a `Service` that's bound to a visible Activity, or an Activity that's not in the foreground

but is still visible to the user, for example an Activity where the `onPause()` method has been called.

The Android system will only kill a visible process if there's not enough memory to support all foreground processes.

Service processes

This is a process that's running a `Service`, but doesn't fall into either of the two higher categories.

Although service processes are not directly related to anything the user sees, they usually perform actions that the user cares about. The system will avoid killing service processes unless doing so is the only way to keep all foreground and visible processes running.

Background processes

Background processes hold an Activity that's not currently visible to the user. Since background processes don't have a direct impact on the user experience, the system may kill background processes off at any time if it needs to reclaim memory for foreground, visible, or service processes.

Empty processes

This is a process that doesn't hold any active components. The system may keep empty processes alive for caching purposes, but make no mistake, they're the first to go if the system needs to free up some memory.

Re-using layouts with > and <merge/>

The Android platform provides a range of simple, reusable UI components known as widgets, but sometimes there may be larger or more complex UI components that you need to reuse across multiple screens, such as a panel containing a progress bar and a **Cancel** button, or a user profile consisting of a username and an avatar.

If your project features elements that you want to use multiple times, you can save yourself time and effort by implementing these elements as a reusable layout. You can then import these reusable elements into as many layout files as you want, using Android's `>` and `<merge/>` tags.

To extract common elements into a reusable layout, create a new XML layout resource file and then define the UI elements you want to reuse. Pay special attention to the file's root view as this will also be included every time you embed this component in other layouts.

To import your reusable component into a layout resource file, just add the `include` tag and reference the layout file you want to import. For example, if you'd created a reusable layout called `contactslist` and wanted to import this component into another layout, you'd use the following:

```
<include layout="@layout/contactslist"/>
```

However, be aware that using the `include` tag can introduce redundant `ViewGroups` into your view hierarchy.

Imagine you have a layout called `main_layout.xml` that uses a vertical `LinearLayout` as its root view. You have a re-usable `contactslist_layout.xml` component that you want to include

in this `main_layout` file, but the `contactslist_layout` also uses a vertical `LinearLayout` as its root view.

If you include `contactslist_layout.xml` in your main layout, you're going to end up with a vertical `LinearLayout` inside a vertical `LinearLayout`. This duplicate vertical layout isn't contributing anything to the UI, but it *is* making your view hierarchy more complex and potentially slowing down your app. So how do we get rid of this duplicate `LinearLayout`? The answer is by using the `merge` tag.

The `merge` element helps to eliminate the redundant `ViewGroups` that can work their way into your view hierarchy when you include reusable layouts.

When you use the `merge` element as the root view of a reusable layout, the `LayoutInflator` skips the `merge` tag and inserts the reusable views into the `main_layout` as though they've always been part of that layout. As a result, your view hierarchy is simpler—which is nothing but good news for your app's performance!

Loading views only when needed

Depending on your app, you may find yourself with a user interface that contains a large number of complex views that you rarely use, such as pop-ups and progress indicators.

One possible solution to this problem is to add some of these complex `Views` via a `ViewStub`, which is a variation of the `include` tag. A `ViewStub` is a lightweight view that isn't included in your layout directly, so it's very cheap to keep in your view hierarchy.

When you add a `View` via a `ViewStub`, the `ViewStub` only loads the `View` as and when it's needed. This allows you to create complex layouts consisting of lots of small views, and your UI will still render quickly and smoothly as you're not immediately populating your user interface with lots of complex `Views`.

To use a `ViewStub`, you need to specify the layout you want to inflate using the `android:id` attribute, for example:

```
<ViewStub
   android:id="@+id/popup"
   android:inflatedId="@+id/popup_import"
   android:layout="@layout/basic_popup"
   android:layout_height="wrap_content"
   android:layout_width="match_parent" />
```

When it's time to load your `ViewStub's` layout, you just need to set the `ViewStub` to visible. To do this, either change the visibility of the stub by calling `setVisibility(View.VISIBLE)`:

```
((ViewStub)
findViewById(R.id.popup)).setVisibility(View.VISIB
LE);
```

Or invoke the `inflate()` method:

```
View importPopup = ((ViewStub)
findViewById(R.id.popup)).inflate();
```

The inflated layout replaces the `ViewStub`, and at this point the `ViewStub` element is no longer part of your view hierarchy.

Note

`ViewStub` currently doesn't support the `merge` tag in the layouts to be inflated.

Summary

In this chapter, I covered some of the most common performance problems you need to be aware of when developing Android apps, including overdraw, memory leaks, and complex view hierarchies.

We took an in-depth look at numerous tools you can use to check whether some of the most common performance problems are affecting your Android projects. We also looked at how to gather more information about any problems you *do* diagnose, so you're in a better position to fix them.

There's just one chapter left to go! In the final chapter, I'm going to cover all the best practices and guidelines that didn't fit neatly into any of the previous chapters. And since security is such a big concern for mobile users and developers at the moment, I'll also show you how to lock down your UI (and your app in general) so you can be confident that your app isn't leaving users open to security vulnerabilities.

Chapter 10. Best Practices and Securing Your Application

The Android operating system was designed in anticipation of hackers attempting to perform common attacks, such as social engineering attacks that try to trick the user into handing over their personal information or installing malware.

Android comes with built-in security features that significantly reduce the chances of security exploits succeeding, and that limit the impact of any attacks that *do* succeed.

These built-in security controls provide you, your application, and your users with a certain level of protection by default. Nevertheless, following security best practices is essential for further reducing the chances of your app leaving users vulnerable to exploits, data leaks, and other security-related issues.

In this chapter, we're going to look at how you can make the most of Android's built-in security features. Towards the end of this chapter, I'll also look at some best practices we haven't covered in detail in any of the previous chapters, including how to design more effective notifications and ensuring that your app is accessible to all users.

Keeping user data secure

The most common security concern for your typical Android user is whether the applications they've chosen to install can access the sensitive data stored on their device.

If your app has access to the user's data, then you have a responsibility to make sure that data remains secure. One of the quickest and easiest ways of protecting user data is to take a long hard look at whether your app really needs access to this data at all.

If you minimize the data your app has access to in the first place, then you minimize the risk of your app inadvertently exposing personal information. You also reduce the chances of attackers being tempted to try and exploit your application in order to gain access to the sensitive data it's privy to. You should always be on the lookout for ways to create the same effect *without* your app requiring direct access to sensitive data.

Wherever possible, don't store usernames or passwords on the user's device. Instead, your app should perform initial authentication using the username and password supplied by the user, and then switch to a short-lived, service-specific authorization token.

To help protect your users from phishing attacks, you should also minimize the number of times your app asks for user credentials. This way, a phishing attack is more likely to strike the user as suspicious, as it'll be out of character for your app to ask for sensitive information.

> **Note**
>
> If your app does need access to passwords and usernames, keep in mind that you may be legally required to provide a privacy policy explaining how your app uses and stores this data.

Connecting to a network

Network transactions pose an inherent security risk, particularly since mobile users are more likely to connect to unsecured wireless networks such as public Wi-Fi hotspots.

Whenever your app connects to a network, it's crucial that you implement the following best practices, in order to help keep your users safe:

- Use `HTTPS` over `HTTP` wherever possible. Also, never automatically trust *any* data downloaded from insecure protocols such as `HTTP`.
- Implement authenticated, encrypted, socket-level communication using the `SSLSocket` class.
- When you're handling sensitive IPC, use an Android IPC mechanism that can authenticate the identity of the application connecting to your IPC, such as a `Binder`, `Intent`, `BroadcastReceiver`, or `Messenger with a Service`. This is more secure than using localhost network ports.
- Don't use unauthenticated SMS data to perform sensitive commands. SMS are neither encrypted nor strongly authenticated by default, and are therefore susceptible to interception on the network.
- When sending data messages from a web server to your app, you should use the **Google Cloud Messaging APIs** and IP networking wherever possible.

Android N also introduces the concept of a **network security configuration** file, which you can use to create custom network security settings for your app without having to modify your actual application code. You can use this new network security configuration file to:

- Specify which certificate authorities are trusted for your app's secure connections
- Restrict an app's secure connections to specific, named certificates
- Debug secure connections in your app without added risk to the installed base
- Protect apps from accidentally using clear-text traffic, which is a particular security risk as it transmits potentially sensitive data in a human-readable format

To create a network security configuration file, create a new XML values resource file with the following path:

```
res/xml/network_security_config.xml:
```

Then reference this file in your project's Manifest:

```
<?xml version="1.0" encoding="utf-8"?>

...

<app ...>

   <meta-data
android:name="android.security.net.config"

android:resource="@xml/network_security_config" />

...

</app>
```

The structure of a simple network configuration file is as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<network-security-config>
   <base-config>

//This is the default configuration used by all
connections that are not covered by a specific
domain-config//

       <trust-anchors>

<certificates src="@raw/trusted_cas"/>

        //This is a reference to an XML file where
you'd name any trusted Certificate Authorities//

       </trust-anchors>

   </base-config>

   <domain-config>

       <domain>mydomain.com</domain>
```

```
...
...

 //This is where you'd create a configuration to
use for connections to a specific destination, in
this example that's www.mydomain.com//

        </domain-config>
</network-security-config>
```

Requesting permissions

Android devices do a lot—from snapping photos and recording videos, to providing directions, posting to social media, and sending SMS messages. This means your typical Android device has access to *huge* amounts of sensitive information.

The good news is that Android does a lot to help keep this information secure. The platform is based on a privilege-separated system where apps run separately from one another and from the system, in a limited-access sandbox. This restricts the data and features each app has access to. By default, no Android app has permission to perform any operation that could adversely affect the operating system, the user, or other applications, which helps prevent malicious apps from corrupting data or accessing sensitive information. These permissions also restrict each app's access to device features not provided by the basic sandbox, protecting the user from apps making unauthorized use of hardware (such as accessing the device's camera) or external communication channels (such as connecting to the Internet).

However, there are legitimate reasons why your app might need to access the user's information or device capabilities, such as an SMS app that requires access to your contacts list or a video recording app that needs to access the device's camera. If your app does require access to protected information or features, then you'll need to prompt the user for access. The user then has the choice to accept or deny this permission request.

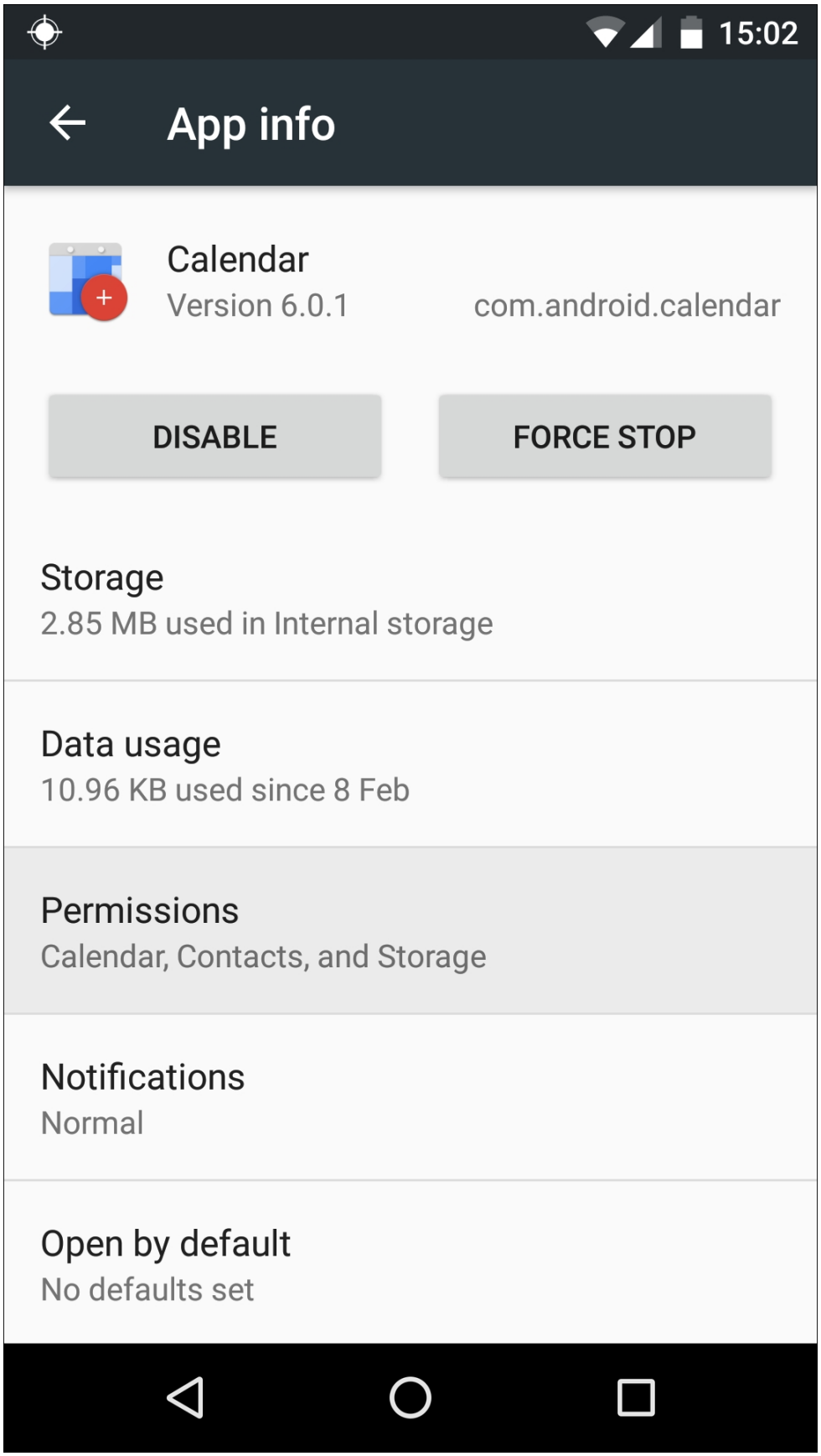
Originally, Android apps requested all the permissions they might possibly need up front before the user could even download the app from Google Play. If the user didn't want to grant the app one or more of the requested permissions, their only option was to abandon the installation altogether, and go looking for an alternative app.

Android 6.0 completely overhauled this permissions model, replacing it with new **runtime permissions**.

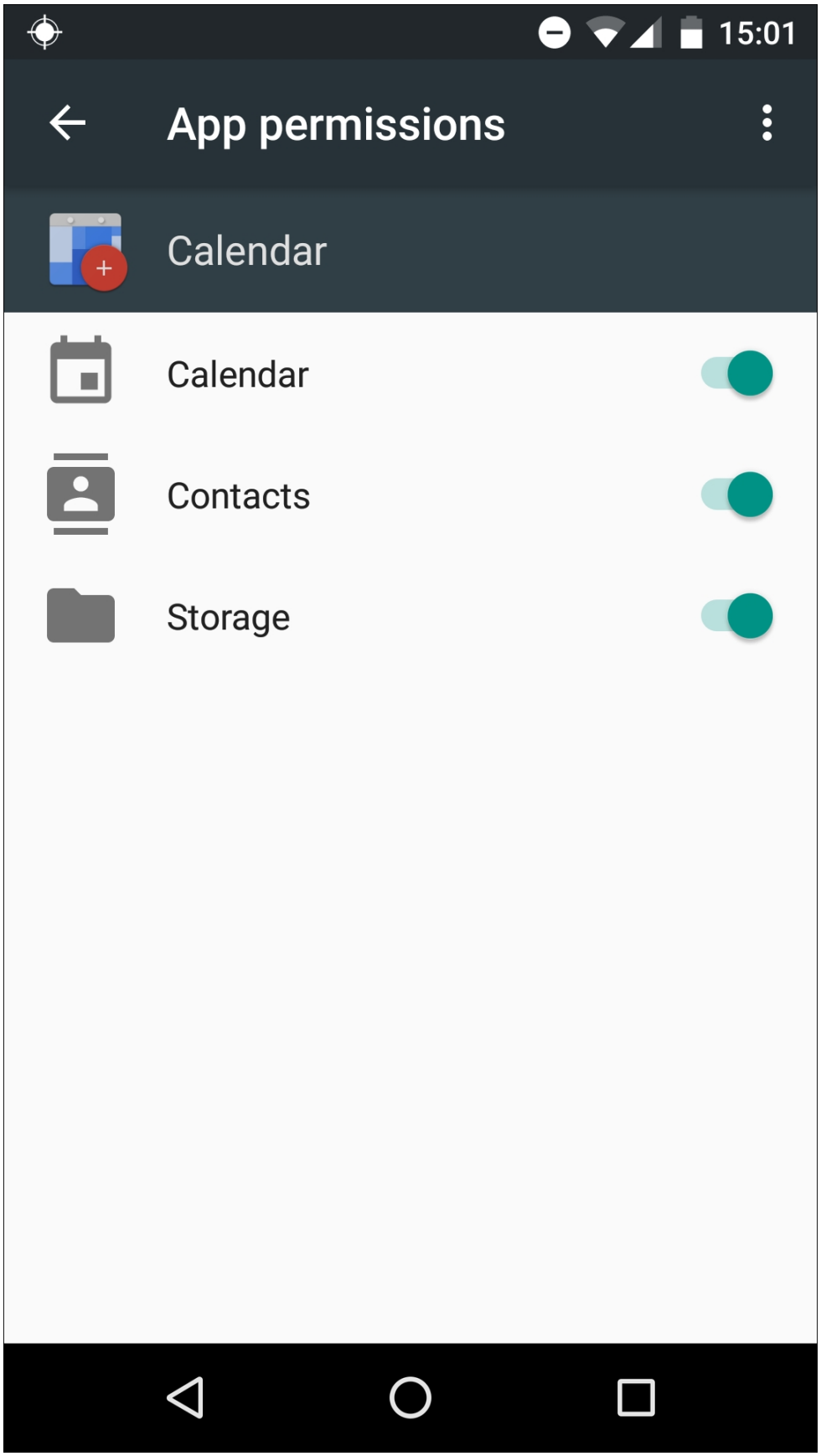
Beginning with Android 6.0, apps request permissions one by one while the app is running, as and when that app requires access to protected services, data, or device features.

For example, imagine you'd developed a note-taking app that supports traditional text notes, but also gives users the option to record voice memos. Before Android 6.0, the user would have to grant this app permission to access the microphone at install time. However, under the new permissions model, the user can launch your note-taking app and write as many text memos as they want, potentially without the app ever requesting access to the microphone. Your app will only request microphone access when the user tries to record a voice memo for the first time, as this permission is required to complete the current task.

In Android 6.0 and higher, users can change an app's permissions manually at any time, by opening their device's **Settings** and selecting **Apps**. At this point, they'll see a list of all the apps installed on their device. They can then select any app from the list, and tap **Permissions** to see all the permission categories this particular app has access to.


15:02
App info
Calendar
Version 6.0.1
com.android.calendar
DISABLE
FORCE STOP
Storage
2.85 MB used in Internal storage
Data usage
10.96 KB used since 8 Feb
Permissions
Calendar, Contacts, and Storage
Notifications
Normal
Open by default
No defaults set

The user can revoke any permission, by dragging that permission's slider to the **Off** position.


15:01
App permissions
Calendar
Calendar
Contacts
Storage

This new permission model has a number of benefits for both users and developers:

- The user no longer has to read through a list of permissions before they can install an app, which makes for a more streamlined installation process.
- If you release an update that requires new permissions, previously the user had to accept these permissions before they could install the update. In Android 6.0 and higher, apps update automatically and will then make any new permission requests associated with the update, as and when (and if) those permissions are required.
- Users have more control over the information and features each app has access to. The new permission model gives users the option to deny individual permission requests, something that wasn't possible in previous versions of Android. For example, a user who has downloaded an image-editing app may be happy to give that app access to their gallery, but more hesitate to grant that same app access to their device's camera. In previous versions of Android, the user *had* to grant that app all the permissions it requested up front, or not install the app at all. However in Android 6.0 and higher, the user is free to deny the app access to their camera, if they're not comfortable with that particular permission request.

- Users are more likely to understand why your app is requesting each permission, as requests now happen in context when the user is trying to access a related feature for the first time. And when users understand why your app is requesting access to sensitive information or device features, they're more likely to grant those requests.

The new permissions model – backwards compatibility

Under the new permissions model, users can revoke any permission they've previously granted to any app, at any time, even if that app targets API level 22 or lower. This means you need to test that your app continues to function normally if the user denies one or more of its permission requests—*regardless* of your app's `targetSdkVersion` value.

Initially, if your app winds up on a device running Android 5.1 (API level 22) or lower, *or* your app's `targetSdkVersion` is 22 or lower, the system defaults to the old permissions model and makes all permission requests up front, at install time. If the user denies these permissions requests, then the system won't install the app. Similarly, if you add a new permission to an updated version of your app, the system will ask the user to grant these new permissions before they can install the update.

However, users can *still* revoke previously granted permissions manually by selecting their device's **Settings**, followed by **Apps**, and then selecting your app from the list.

If your app targets an earlier version of the Android platform and you *haven't* accounted for the new permissions model, your app may no longer function normally (or at all) if the user chooses to revoke one or more of your app's permissions manually. The system will warn the user that revoking permissions for apps targeting earlier versions of Android can cause the app in question to stop working, but the user still has the option to go ahead and revoke these permissions, if they really want to.

Even if your app targets pre-6.0 versions of the Android platform only, you shouldn't just assume that you can ignore the new permissions model. You should still aim to create an app that provides a good user experience, even if the user chooses to revoke some (or all) of its previously granted permissions.

Permission groups

Android's permissions are divided into two categories:

- Normal permissions:

  These permissions give your app access to data or resources outside of its sandbox, but pose very little direct risk to the user's privacy, the operation of the device, or to other apps.

  If you declare that your app requires a normal permission in its `Manifest`, the system will automatically grant your app that permission.

- Dangerous permissions:

  These permissions give your app access to data or resources that could pose a risk to the user's privacy, or that may affect the user's stored data, other apps, or the device's normal operation. For example, being able to read the user's contacts list is considered a dangerous permission.

  If the user's device is running Android 6.0 (API Level 23) or higher *and* your app's `targetSdkVersion` is 23 or higher, then dangerous permissions trigger a permission request only when the user tries to perform an operation that requires this permission. For example, if your app requires the `READ_CONTACTS` permission, the system may ask for that permission the first time the user tries to create an SMS message.

  If your app winds up on a device running Android 5.1 or lower, or its `targetSdkVersion` is 22 or lower, the Android system will ask the user to grant all dangerous permissions at install time.

In addition to the *normal* and *dangerous* ratings, Android groups related permissions into permission groups. When your app requests access to a permission, the user will be presented with a dialogue requesting access to that entire permissions group.

If your app requests a dangerous permission and the user has already granted your app access to another dangerous permission

from that same group, the system will grant that permission automatically without requiring any additional input from the user.

This approach helps mobile users make more informed decisions about what parts of their device and what information each app should have access to *without* overwhelming them with technical information or too many permission requests.

Permissions are divided into nine groups:

- Calendar
- Camera
- Contacts
- Location
- Microphone
- Phone
- Body sensors
- SMS
- Storage

Declaring permissions

As you develop your app, make a note every time your app requires access to resources or information that it doesn't create itself, or whenever it attempts to perform actions that could impact the user's privacy, the behavior of other apps, or the device in general. Most of the time these actions will require your app to make a permissions request.

To declare a permission, open your project's `Manifest` and add a `<uses-permission>` element as a child of the top-level `<manifest>` element:

```
<manifest

xmlns:android="http://schemas.android.com/apk/res/
android"
 package="com.me.app.myapp" >
```

```
<uses-permission
android:name="android.permission.ACCESS_FINE_LOCAT
ION"/>

android:label="@string/permlab_fineLocation"

//Provide a label for each permission your app
requests, using android:label. This label is
displayed to the user when they're viewing a list
of permissions. Try to keep this label short - a
few words will usually be sufficient//


android:description="@string/permdesc_fineLocation
"

//Describe what granting this permission request
will allow your app to do, using the
android:description attribute. Ideally, your
descriptions will be two sentences long - the
first should describe the permission that's being
requested, and the second should warn the user
about the dangers of granting this particular
permission//

       android:protectionLevel="dangerous" />

//Indicates the potential risk associated with
this permission. The possible values are normal,
dangerous, signature, and SignatureOrSystem//
   ...
</manifest>
```

Here's how the string resources for this permission's label and description might look:

```
 <string name="permlab_fineLocation">Precise
location (GPS and network-based)</string>

   <string name="permdesc_fineLocation">
This app will be able to retrieve your precise
location using the GPS or network location
sources. Apps may consume additional battery power
while using location services</string>
```

Verifying permissions

Your app won't always need to ask for permission to access protected information or features—if your app currently has access to the microphone, then it won't need to request this permission, for example. However, since users running Android 6.0 and higher can revoke previously granted permissions at any time, your app will need to check whether it currently has access to protected information or features, every time it needs to act on the related permission. Even if the user granted this permission previously, in Android 6.0 and higher there's no guarantee the user hasn't manually revoked that permission at some point.

To determine whether your app currently has permission to access information or features, you need to call the `ContextCompat.checkSelfPermission()` method. For example, this code snippet shows you how to check whether your app has permission to access the Internet:

```
int permissionCheck =
ContextCompat.checkSelfPermission(this,
Manifest.permission.INTERNET);
```

Note

`ContextCompat.checkSelfPermission()` is available as part of revision 23 of the `support-v4 library` for backwards compatibility (http://developer.android.com/tools/support-library/features.html?utm_campaign=runtime-permissions-827&utm_source=dac&utm_medium=blog#v4).

If your app currently has access to the requested permission, `ContextCompat.checkSelfPermission` will return `PackageManager.PERMISSION_GRANTED` and your app can proceed with the operation. If the user hasn't granted your app this permission, or they granted the permission but then revoked it at

some point, this method returns `PackageManager.PERMISSION_DENIED` and you'll need to request that permission by calling one of the `requestPermission` methods:

```
if (ContextCompat.checkSelfPermission(myActivity,
                Manifest.permission.INTERNET)

//Does this app have permission to access the
Internet?//

        != PackageManager.PERMISSION_GRANTED) {
      if
(ActivityCompat.shouldShowRequestPermissionRationa
le(myActivity,
           Manifest.permission.INTERNET)) {

//If shouldShowRequestPermissionRationale returns
true, the app doesn't currently have this
permission, and you'll need to request access//
           ..........
      ..........

   } else {


ActivityCompat.requestPermissions(myActivity,

//Request the permission. Note, when your app
calls requestPermissions(),the system displays a
standard dialogue that you cannot customize. If
you want to provide some additional information,
such as an explanation about why your app needs
this permission, you must do so before calling
requestPermissions()//

                new String[]
{Manifest.permission.INTERNET},
                MY_PERMISSIONS_REQUEST_INTERNET);

   }
}
```

Handling the permissions request response

When your app makes a permission request, the system presents a dialogue to the user. When the user responds, the system invokes your app's `onRequestPermissionsResult()` method and passes it the user's response. To find out whether the user has granted or denied the permission request, your app needs to override that method:

```
@Override
public void onRequestPermissionsResult(int
requestCode,
       String permissions[], int[] grantResults) {
//The Activity's onRequestPermissionsResult method
is called and passes the user's response//
   switch (requestCode) {
       case MY_PERMISSIONS_REQUEST_INTERNET: {
                    if (grantResults[0] ==
PackageManager.PERMISSION_GRANTED) {
........
.........
// If the user has granted the permission, this is
where you'd perform your app's Internet-related
tasks//

           } else {
.......
.......
// The user has denied the permission request.
This is where you'd disable the functionality that
depends on Internet access. Since denying a
permission may prevent parts of your app from
working correctly, you may also want to provide
some feedback explaining what impact denying this
permission will have on your app's functionality//

           }
           return;
       }


   }
}
```

Permissions and <uses-feature>

It's possible that your app may rely on certain hardware or software features being present on the user's device, for example your typical camera app will almost certainly require a device with camera hardware.

To prevent users from installing your app on devices that don't have the hardware or software necessary to support it, you need to add the `<uses-feature>` declaration to your project's `Manifest`:

```
<uses-feature
android:name="android.hardware.camera" />
```

You then specify whether your app requires this feature to function at all (`true`) or whether it prefers to have this feature but can function without it if necessary (`false`):

```
<uses-feature
android:name="android.hardware.camera"
              android:required="true"/>
```

Although the Android system won't check your app's `Manifest` for `<uses-feature>` elements, Google Play does use your app's `<uses-feature>` elements to decide whether your app is compatible with the user's device. Google Play won't allow any user to install an app that it deems incompatible with their device's hardware or software.

Note

If you add `<uses-feature>` elements to your `Manifest` but don't include the `android:required` attribute, Google Play assumes that your app requires this feature (`android:required="true"`).

Ideally, you should declare all your project's hardware and software requirements, but there are a few safeguards in place just in case you forget to mention one or more `<uses-feature>` elements.

Google Play checks your app's *regular* permissions for any implicit hardware-related features. If it finds any, it adds the corresponding hardware or software feature to your app's metadata, and then takes these into account when deciding whether each potential user can or cannot install your app.

This does help to prevent users from downloading incompatible apps, but it can also be a bit of a nuisance. Imagine your app requests `android.permission.CAMERA` but you didn't include `<uses-feature android:name="android.hardware.camera">` in your `Manifest`. In this scenario, Google Play will assume that your app requires camera hardware to function, and consequently won't allow anyone to install your app on a device that doesn't have a camera. If your app doesn't actually require a camera, this misunderstanding could prevent people with perfectly compatible devices from installing your app.

To prevent this misunderstanding, you need to open your project's `Manifest` and specify that a camera is preferred, but not essential:

```
<uses-feature
android:name="android.hardware.camera"
android:required="false" />
```

For a full list of all the permissions that imply feature requirements, check out the official Android docs (http://developer.android.com/guide/topics/manifest/uses-feature-element.html#permissions).

Note

Declaring a `<uses-feature>` doesn't automatically grant your app permission to access the related features or information. You'll *still* need to request every permission your app requires, as normal.

Best practices for app permissions

Despite everything this new runtime permissions model has to offer, it does present developers with a few new challenges. Firstly, since your app can request permissions at any time, issuing your permission requests at exactly the right time is now an important part of the user experience.

Secondly, since the user can now deny individual permissions, you'll need to ensure that your app continues to provide a good experience regardless of whether the user chooses to deny one or more of your application's permission requests.

> **Note**
>
> This doesn't mean that your app has to function even if the user consistently denies its requests. It's possible that denying a certain permission may render your app completely unable to perform its core functionality. However, even if this is the case, you can still provide a good user experience by letting the user know *why* your app has suddenly stopped working. For example, whenever the user tries to launch your app, you might display a dialogue box listing all the missing permissions that your app requires in order to function, and then giving the user an easy way of granting these permissions.

In this section, we're going to look at all the best practices surrounding permissions, so you can make more informed decisions about which permissions your app requests, in addition to *when* and *where* it issues those requests.

Making as few permission requests as possible

This is the golden rule of permission requests.

Permissions are designed to protect the user and the device, so your app should *only* request permissions that it requires to function. Research (http://repository.cmu.edu/hcii/268) suggests that the amount and type of permissions an app requests directly affects user behavior. When confronted with two apps that have similar functionality, users are more likely to opt for the one that requires fewer permissions.

If you're going to reach the widest possible audience, you need to limit the number of permissions your app requires. This is particularly important under the new permissions model, as requests are made as the user is interacting with your app. Imagine you've got lost on your way to an important meeting—you open up the map application you downloaded the night before, only to be confronted with permission requests every time you try to interact with the app. Not what you want when you're in a rush! It's incredibly easy to overwhelm a busy, multi-tasking mobile user with too much information.

If in doubt, remember that it's easier to add permissions to your app later via updates, than it is to remove them—so it's better to play it safe with a less-is-more approach.

Tip

Automatic permission adjustments

New releases of the Android platform may bring new restrictions, which can result in your app needing to request permissions that it didn't previously require. To make sure your app continues to work for users who are running the latest and greatest version of Android, the system may step in and automatically add these new permission requests to your project's `Manifest`.

When deciding whether to automatically add permissions to your app, Android takes a look at your app's `targetSdkVersion`. If the value is lower than the version where the new permissions were added, the system may add these permissions automatically. This can result in your app requesting permissions that it doesn't necessarily need—definitely not a good thing!

To avoid this, make it a priority to test your project against new releases of the Android platform, and then update your `targetSdkVersion` as soon as you're confident that your app is compatible with this latest release.

Requesting critical permissions up front

If your app's core functionality hinges on having access to certain permissions, you should request these critical permissions up front. This is because users are more likely to take the time to read and therefore *grant* permission dialogues the first time they launch your app, rather than grant permission requests that interrupt them when they're midway through performing an important task in your app.

For non-critical permissions, you should wait until the user tries to access a related feature and then request that permission in context.

Providing extra information where necessary

Ideally, your permission requests should be self-explanatory, *especially* on devices running Android 6.0 or higher, where apps request permissions in context. For example, if a user taps your app's **Take a photo for your profile picture** button, chances are they aren't going to be confused when your app requests access to their camera.

However, if you suspect that it may not be immediately obvious why your app is requesting a certain permission, then you'll want to provide the user with some more information. Your users are always going to feel more comfortable when they understand why an app is

requesting access to information or features. If you *don't* provide this information, then your users may suspect that you're leaving them in the dark on purpose.

Just be wary of explaining *everything*. You shouldn't need to provide an explanation for every permission your app requests. If you overwhelm the user with too much information, they're going to get frustrated by the constant interruptions, and may end up either skim-reading your explanations (bad) or skipping them completely (worse).

If you find yourself providing an explanation *every time* your app makes a permission request, this may be an indication of a deeper problem with how your app is handling sensitive data or using device features. Maybe your app is requesting too many permissions? Or maybe it's requesting miscellaneous permissions that are unrelated to its core functionality?

Remember that permission request dialogues aren't the only way of communicating with your users. You may want to consider other ways of giving the user the information they need to better understand the context of your app's permission requests. For example, you could:

- Add this information to your app's Google Play description.
- Create a privacy policy outlining the information your app needs access to and how it uses this information.
- Create support documentation that your users can turn to if they need more information, such as an online user manual or a **Help** section inside your app.
- Update your UI so permission requests are more self-explanatory. Your users may be confused if they tap a stock avatar and your app suddenly requests access to their camera, however if the stock photo has a **Tap here to take a profile picture** label, then your users are *far* less likely to be confused.
- Create dedicated support channels for your users such as forums, social media pages, or an e-mail address where they can reach out to you with any questions.

Paying attention to permissions required by libraries

When you include a library in your project, your project inherits any permissions that library requests. Before you use *any* library, always check what permissions that library requires, and what it uses those permissions for.

Adding multiple libraries to your project without checking their permissions first can quickly land you in hot water with your users, as from the user's perspective it's your app that's making these requests, and not an external library. These libraries may even request permissions that seem completely unrelated to your app, potentially making your users even more suspicious.

If you need to include a library in your project but are concerned that it requires too many permissions, then you can always look for an alternative library that provides similar functionality but doesn't make quite so many permission requests.

Being transparent about accessing the camera and microphone

Sometimes your app may need to access particularly sensitive features, such as the device's camera or microphone. Most of the time your app won't require *constant* access to these features, but once the user has granted your app access, how do they know that it isn't constantly pulling information from their microphone and camera?

No one likes to feel they're being watched, so you should be up front about when your app is accessing the device's camera or microphone. For example, you could display a pop-up with a timer indicating that your app is going to turn the device's camera on in 3, 2, 1, or you could display a flashing microphone icon in the corner of the screen whenever your app is *listening* to the user.

Considering alternatives

Even if your app does require access to protected information or features, you may be able to engineer your app so it doesn't have to make these requests itself.

An app only needs to request permissions for actions that it performs directly - *not* when it's asking other apps to perform a task on its behalf, or supply it with information. For example, instead of requesting access to the device's camera, you could use `MediaStore.ACTION_IMAGE_CAPTURE` to launch a camera app that the user has already installed on their device. You can also use system intents to request information from other apps, such as requesting contact information from the user's Contacts app, rather than issuing the `READ_CONTACTS` permission.

The major downside to this approach is that it presents the user with a dialogue every time your app needs to access the protected information or feature, so consider whether the frequency or timing of these requests is likely to irritate your users.

Notifications

Notifications underwent a massive overhaul in Android Lollipop, getting both a Material Design makeover *and* new methods that help Android sort notifications more intelligently.

In Android 5.0 and higher, you can annotate your notifications with the `setPriority()` method. Low-priority notifications may be hidden from the user, while higher-priority notifications are more likely to interrupt whatever the user is currently doing.

The different priority levels are:

- `Notification.PRIORITY_MAX`: Used for critical and urgent notifications that alert the user to a condition that's time-sensitive, or that they may need to resolve before they can continue with the current task.
- `Notification.PRIORITY_HIGH`: Typically used for important communications such as chat messages. High-priority notifications appear in a floating *heads-up* window that contains action buttons. Action buttons allow the user to act on, or dismiss, the heads-up notification without having to navigate away from the current screen.
- `Notification.PRIORITY_LOW`: Used for less urgent notifications.
- `Notification.PRIORITY_MIN`: Used for contextual or background information. Minimum-priority notifications aren't usually shown to the user except under special circumstances, such as detailed notification logs.
- `Notification.PRIORITY_DEFAULT`: You can use this priority flag for all notifications that don't fall into any of the more specific priority categories.

Many Android devices feature a notification LED that you can use to inform users about events that are happening inside your app, even when the screen is off. Assigning your app's notifications a priority level of `MAX`, `HIGH`, or `DEFAULT` should cause the LED to glow.

In Android 5.0 and higher, notifications also appear on the device's lockscreen, which means you need to consider whether your notifications contain information that users might not want to appear on their lockscreen for the whole world to see.

You can use `setVisibility()` to tell Android how much information it should display on the lockscreen, and assign it one of the following values:

- `VISIBILITY_SECRET`: No part of the notification appears on the lockscreen. This is recommended if your app's notifications contain personal or potentially embarrassing information.
- `VISIBILITY_PRIVATE`: The notification displays some basic information on the lockscreen, but most of the information is hidden.
- `VISIBILITY_PUBLIC`: The notification appears on the lockscreen in its entirety.

Notification best practices

Design your notifications well, and they'll provide real value to your users—plus, they'll serve as a handy way of enticing users back to your app, by presenting them with interesting and timely updates about events that are happening inside your application.

To make the most out of Android's notification system, you should keep the following best practices in mind.

Providing the right content

At the very least, your notifications should contain:

- A title (`setContentTitle`) and secondary text (`setContentText`)
- A timestamp; note this indicates when the event occurred and *not* when the notification was posted
- The notification type (optional)

To make sure the user can recognize your app's pending notifications in the system bar, you should include a distinct app icon (`setSmallIcon`).

Your notification icon should be simple and avoid any excessive detail, but at the same time it must be eye-catching and distinct from the other notification icons users may encounter. You should use the Material Light action bar icon style and avoid opaque backgrounds—basically, your notification icon should be a white-on-transparent background image.

If you're struggling for inspiration, try booting up your Android device and taking a look at the kind of notification icons other applications are using.

Using notifications sparingly

Every time you issue a notification, you're interrupting whatever the user is doing at that moment, so it's important you use notifications sparingly.

You should *never* use notifications to alert the user about background operations that don't require their input or that aren't time-sensitive. You also shouldn't use notifications to alert the user about events that are already happening on screen. Instead, inform the user about these events in context via your app's UI. For example, if the user has unlocked a new level in your gaming app, and that app is currently on screen, you should alert them via an on screen message rather than firing off a notification.

Even though notifications can be useful for reminding the user about your application, you should *never* use unnecessary notifications to tempt the user into launching your app. In the long run, unimportant or unwanted notifications are only going to make users view your app as an attention-seeker, which may even lead to them uninstalling your app and leaving you a negative review on Google Play.

Giving users a choice

Ideally, you should give your users the option to change your app's notifications settings, such as switching between sound alerts and vibration alerts, or even allowing users to disable notifications entirely.

Categorising notifications

When ranking and filtering notifications, the Android system may take an app's category into consideration, so you should assign a suitable category to each of your app's notifications. To do this, use the `setCategory()` option and choose from the following supported categories:

- `CATEGORY_ALARM`: Alarm or timer
- `CATEGORY_CALL`: An incoming voice or video call
- `CATEGORY_EMAIL`: Asynchronous bulk message
- `CATEGORY_ERROR`: An error in a background operation
- `CATEGORY_EVENT`: Calendar event
- `CATEGORY_MESSAGE`: Incoming direct message, such as an SMS
- `CATEGORY_PROGRESS`: The progress of an operation running in the background
- `CATEGORY_PROMO`: A promotion or advert
- `CATEGORY_RECOMMENDATION`: A specific, timely recommendation
- `CATEGORY_SERVICE`: An indication of a service that's running in the background
- `CATEGORY_SOCIAL`: A social network or sharing update
- `CATEGORY_STATUS`: Ongoing information about device or contextual status
- `CATEGORY_SYSTEM`: This category is reserved for system or device status updates
- `CATEGORY_TRANSPORT`: Media transport control for playback

Making use of actions

You can also add action buttons to your notifications. Action buttons allow users to perform common tasks from the notification UI, without them having to open the originating app.

You add a button to your notification using the following syntax:

```
NotificationCompat.Builder nBuilder = new
NotificationCompat.Builder(context)
       .setContentTitle("This is a notification
title")

//Sets the notification's title//

      .setContentText("This is the notification's
body text.")

//Sets the second line of the notification's
text//

            .addAction(R.drawable.accept,
"Download", pIntent)

//Adds a 'Download' action to this notification,
complete with icon//

      .addAction(R.drawable.cancel, "Cancel",
pIntent);

//Adds a 'Cancel' button, complete with icon -
this time the R.drawable.cancel icon//
```

Tip

Each action should have its own icon and its own name.

Although optional, it's generally a good idea to add at least one action to each of your app's notifications. Just don't get carried away—limit yourself to a maximum of three actions per notification.

Using expanded layouts

For users running Android 4.1 and higher, you can supply two different visual styles for each of your app's notifications:

- **Normal view**: This is the default, compact layout.
- **Big view style**: A separate layout that appears when the user expands your notification by pinching or dragging it open.

You have a choice of three different big view styles.

Big text style

This layout provides additional text that's displayed in the detail area of the expanded notification, in place of the notification's regular `setContentText`:

```
.setStyle(new NotificationCompat.BigTextStyle()
                .bigText("This text replaces the
context text in the big view"))
```

Big picture style

This layout includes a large image attachment:

```
.setStyle(new Notification.BigPictureStyle()
        .bigPicture(aBigImage))
```

Inbox style

This layout includes a list of up to five items:

```
.setStyle(new Notification.InboxStyle()
        .addLine(string1)
        .addLine(string2)
        .addLine(string3)
        .setSummaryText("+2 more"))
```

Direct reply notifications

Google are adding a few new features to notifications in the upcoming release of Android *N*, including an **inline reply** action

button that allows users to reply directly from the notification UI.

Direct reply notifications are particularly big news for messaging apps, as they give users the ability to reply without even having to launch the messaging app. You may have already encountered direct reply notifications in Google Hangouts.

To create a notification action that supports direct reply, you need to create an instance of `RemoteInput.Builder` and then add it to your notification action.

The following code adds a `RemoteInput` to a `Notification.Action`, and creates a **Quick Reply** key. When the user triggers the action, the notification prompts the user to input their response:

```
private static final String KEY_QUICK_REPLY =
"key_quick_reply";
String replyLabel =
getResources().getString(R.string.reply_label);
RemoteInput remoteInput = new
RemoteInput.Builder(KEY_QUICK_REPLY)
       .setLabel(replyLabel)
       .build();
```

To retrieve the user's input from the notification interface, you need to call `getResultsFromIntent(Intent)` and pass the notification action's intent as the input parameter:

```
   Bundle remoteInput =
 RemoteInput.getResultsFromIntent(intent);

//This method returns a Bundle that contains the
text response//

   if (remoteInput != null) {
          return
remoteInput.getCharSequence(KEY_QUICK_REPLY);

//Query the bundle using the result key, which is
provided to the RemoteInput.Builder constructor//
```

Bundled notifications

Don't you just hate it when you connect to the Internet first thing in the morning and your Gmail app instantly bombards you with a **4 new messages** notification, but doesn't give you any more information about those individual e-mails? Not particularly helpful!

When you receive a notification that consists of multiple items, the only thing you can really do is open the app and take a look at the individual events that make up this grouped notification.

However, the upcoming release of Android N promises to fix this problem by introducing **bundled notifications**. This new notification style allows you to group multiple notifications from the same app into a single, bundled notification. A bundled notification consists of a parent notification that displays summary information for that group, plus individual notification items.

If the user wants to see more information about these individual items, they can *unfurl* the bundled notification card into separate, smaller notifications by swiping down with two fingers. When the user expands a bundled notification, the system reveals more information about each child notification. The user can then act on each of these mini-notifications individually, for example they might choose to dismiss the first three notifications about spam e-mails, but open the fourth e-mail. This is similar to the Notification Stacks feature you may have encountered in Android Wear.

Note

Bundled notifications are particularly useful when your app has the potential to generate multiple notifications where each child notification is actionable.

To group notifications, call `setGroup()` for each notification you want to add to the same notification stack, and then assign these notifications the same key:

```
final static String GROUP_KEY_MESSAGES =
"group_key_messages";
Notification notif = new
NotificationCompat.Builder(mContext)
        .setContentTitle("New SMS from " +
sender1)
        .setContentText(subject1)
        .setSmallIcon(R.drawable.new_message)
        .setGroup(GROUP_KEY_MESSAGES)
        .build();
```

When you create another notification that belongs to this stack, you just need to assign it the same group key:

```
Notification notif2 = new
NotificationCompat.Builder(mContext)
   .setContentTitle("New SMS from " + sender1)
    .setContentText(subject2)
     .setGroup(GROUP_KEY_MESSAGES)
  .build();
```

Application widgets

Widgets provide users with a small, convenient sample of your application's most important data, typically from the comfort of their home screen.

To create a basic app widget, you need to complete the process described in the following sections.

Declaring an AppWidgetProvider class in your project's Manifest

The `AppWidgetProvider` class defines the methods that allow you to programmatically interact with your app widget based on broadcast events.

Here's an example of a basic `AppWidgetProvider` implementation:

```
<receiver android:name="MyAppWidgetProvider" >

//Specifies the AppWidgetProvider that's used by
this widget//

   <intent-filter>

       <action
android:name="android.appwidget.action.APPWIDGET_U
PDATE" />

//The <action> element specifies that this
AppWidgetProvider accepts the
ACTION_APPWIDGET_UPDATE. This is the only
broadcast event you need to declare explicitly, as
the AppWidgetManager class forwards all other app
widget broadcasts to the AppWidgetProvider
automatically//

   </intent-filter>
```

```
   <meta-data
android:name="android.appwidget.provider"

//Specifies the AppWidgetProviderInfo resource,
which in this example is
android.appwidget.provider//


android:resource="@xml/my_appwidget_info" />

//The <meta-data> element also requires an
android:resource attribute, which specifies the
location of the AppWidgetProviderInfo resource//

</receiver>
```

Creating an AppWidgetProviderInfo file

The next step is defining the basic qualities of your widget, such as its minimum width and height, and how often it's updated. Create a `res/xml` directory, if your project doesn't already contain one, and then create a new XML layout file. In this example, we're using `my_appwidget_info.xml`:

```
<appwidget-provider
xmlns:android="http://schemas.android.com/apk/res/
android"

   android:minWidth="50dp"

   android:minHeight="50dp"

//Specifies the minimum height and width the
widget consumes by default//

   android:updatePeriodMillis="80000000"

//This is how often the app widget framework
requests an update from the AppWidgetProvider. By
default, if the device is asleep when the widget
requests an update, the device will wake up in
order to perform this operation. For the sake of
```

```
the user's battery, don't set this value too low//

   android:initialLayout="@layout/my_appwidget"

//The layout resource where you'll define the
widget's layout//

   android:resizeMode="horizontal"

//Specifies how the widget can be resized. The
possible values are horizontal, vertical, and
none. You can also combine these values, for
example android:resizeMode="horizontal|vertical"//

   android:widgetCategory="home_screen">

//Declares whether your widget can be displayed on
the home screen (home_screen), the lockscreen
(keyguard) or both. Only devices running lower
than Android 5.0 support lockscreen widgets//

</appwidget-provider>
```

Creating the widget's layout

You define your widget's layout in XML in the same way you create a regular layout resource file. The only major difference is that app widget layouts are based on `RemoteViews`, which don't support every layout and view.

As a result, app widgets support the following layout classes only:

- `FrameLayout`
- `GridLayout`
- `LinearLayout`
- `RelativeLayout`

They also only support the following widget classes:

- `AdapterViewFlipper`
- `AnalogClock`
- `Button`

- `Chronometer`
- `GridView`
- `ImageButton`
- `ImageView`
- `ListView`
- `ProgressBar`
- `StackView`
- `TextView`
- `ViewFlipper`

> **Note**
>
> `RemoteViews` also support `ViewStubs`.

App widget best practices

Well-designed app widgets that display useful information serve as a constant reminder about how great your app is, tempting the user to launch your application—or at the very least, making them less likely to uninstall your app.

To make sure your widget presents your application in the best possible light, make sure you adhere to the following best practices.

Including margins for earlier versions of Android

Your widgets shouldn't extend to the edges of the screen or bump up against other widgets, as this can make the user's home screen look messy and cluttered. The solution is to add a margin around each edge of your widgets, something that Android 4.0 and higher does automatically whenever the user places a widget on their home screen. To take advantage of this behavior, you just need to set your app's `targetSdkVersion` to `14` or greater.

Since Android 4.0 and greater applies margins to your app automatically, you shouldn't add any extra margins to your widget

when it's installed on devices running Android 4.0 and over. However you *will* still need to add margins when your app is installed on devices running earlier versions of Android.

The solution is to create two `dimens.xml` files that specify different margins for different versions of Android:

- `res/values-v14/dimens.xml`: This file defines the 0dp margins for devices running Android 4.0 and greater:

```
<dimen name="widget_margins">0dp</dimen>
```

- `res/values/dimens.xml`: This file defines the margins for devices running versions of Android lower than 4.0:

```
<dimen name="widget_margins">10dp</dimen>
```

You can then reference these `dimens.xml` values in your widget's layout resource file, and the Android system will select the appropriate `widget_margins` value at runtime:

```
android:padding="@dimen/widget_margins"
```

Providing flexible graphics and layouts

The Android home screen is divided into a grid. The user can place app widgets inside free cells, and then stretch them horizontally and/or vertically to occupy a different number of cells.

To ensure your widget's layout is flexible enough to be able to adapt to different grid sizes, you should define your widget's background using stretchable nine-patch images, and then use flexible layouts such as `LinearLayout`, `RelativeLayout` or `FrameLayout`.

Not updating too often

If the device is asleep when the app widget framework requests an update from the `AppWidgetProvider`, the device will wake up in order

to perform this operation. Requesting too many updates is a surefire way to drain the user's battery and may result in them uninstalling your app, or at the very least banishing your app widget from their home screen.

It's important to consider how often your widget really needs to receive new information, for example you'll typically need to update a weather or news widget less often than a widget that alerts users about incoming e-mails.

If you're building a widget that *does* require frequent updates, then it's generally a good idea to perform these updates based on an alarm that won't wake the device. If this alarm goes off while the device is asleep, this update won't be performed until the next time the device wakes up, so it has less impact on the device's battery.

To create this kind of alarm, use the `AlarmManager` to set an alarm with an `Intent` that your `AppWidgetProvider` receives, and then set the alarm type to either `ELAPSED_REALTIME` or `RTC`, for example:

```
alarmManager.setRepeating(AlarmManager.RTC...
```

Finally, set the widget's `updatePeriodMillis` to zero so it won't override your alarm and wake the device.

Accessibility best practices

Your app is only truly accessible when everyone can navigate it, understand it, and use it successfully, including people who may have visual, physical, or age-related limitations.

Android has several in built accessibility features that help you optimize your app for users with visual or physical disabilities, so the good news is that, in most cases, creating an accessible app won't require any extensive changes to your code.

In this section, I'll show you how to use these built-in platform features to make sure *everyone* can enjoy using your app.

> **Note**
>
> If your app is part of a team effort, then it's important that everyone on your team keeps accessibility in mind. If you're working with a designer or a team of testers, then it's a good idea to make sure they're also familiar with the following accessibility guidelines.

Adding descriptive text to your UI controls

If you've designed your UI well, you shouldn't have to add an explicit label to every on screen element, for example a button with a telephone icon inside a **Contacts** app has a pretty obvious purpose. However, users with vision impairments may not be able to pick up on these visual clues, so you'll need to provide them with some additional information.

You should provide content descriptions for every UI component that doesn't feature visible text. Also consider whether these descriptions alone provide sufficient context for the user to fully understand the related visual elements—without any visual context, a **Delete** or **Call the selected contact** content description may not be particularly helpful.

The text in the description attribute *doesn't* appear on screen, but if the user enables speech-based accessibility services such as TalkBack, this description is read aloud when the user navigates to that item.

You can add a description using the `android:contentDescription` XML layout attribute:

```
<ImageButton
   android:id="@+id/newSMS"
   android:src="@drawable/newSMS"
   android:contentDescription="@string/newSMS"/>
```

Imagine that the value of `@string/newSMS` is `Create a new SMS message`. When the user hovers over this icon with an accessibility service enabled, this description will be read aloud and the user will then understand what this UI element does.

Note

For `EditTexts,` your top priority should be helping the user to understand what content they're expected to enter into this empty field, so you should provide an `android:hint` attribute *instead* of a content description. Once the user has entered some text into the `EditText`, the accessibility service will read this text aloud, instead of the `android:hint` value.

There'll be situations where you want to base an item's content description on dynamic elements, such as the state of a slider or the

currently selected text in a list. If this is the case, then you'll need to edit the content description at runtime using the `setContentDescription()` method.

Providing descriptions is particularly important for `ImageButton`, `ImageView`, and `Checkbox` components, but you should add content descriptions wherever you suspect that users with different abilities might benefit from some additional information. Just don't get carried away and add unnecessary descriptions, as this just increases the noise the user encounters as they're trying to decipher your UI, making it more difficult for them to pull useful information from their accessibility service.

Wherever possible, use Android's standard controls as they have `ContentDescriptions` built in by default, and therefore work automatically with accessibility services such as TalkBack.

Designing for focus navigation

Focus navigation is where your users employ directional controls to navigate the individual elements that make up your app's UI, similar to the four-way remote control navigation on a television. Users with limited vision or limited manual dexterity often use this mode of navigation instead of touchscreen navigation.

Directional controllers can be software-based or hardware-based, such as a trackball, D-pad, or external keyboard. Users may also choose to enable the gestures navigation mode available in devices running Android 4.1 and higher.

To make sure your users can successfully navigate your app using a directional controller, you need to verify that all your UI input controls can be reached and activated without using the touchscreen. You should also verify that clicking a directional controller's center, or **OK** button, has the same effect as touching a control that already has focus.

In order to support focus navigation, you should ensure that all your app's navigational elements are focusable. You can achieve this by adding the `android:focusable="true"` attribute to your UI elements, or perform this modification at runtime using the `View.setFocusable()` method on each UI control.

The UI controls provided by the Android framework are focusable by default, and the system visually indicates focus by changing the control's appearance.

When a user navigates in any direction using directional controls, focus is passed from one UI element to another, as determined by the focus order. The system determines the focus order automatically based on an algorithm that finds the nearest neighbor in a given direction.

However, sometimes the results may not be quite what you had in mind, or they may not provide the best experience for the user. If this is the case, Android provides four optional XML attributes that you can use to override this automatic focus order and dictate *exactly* which view will receive focus when the user navigates in that direction:

- `android:nextFocusUp`: Defines the next view to receive focus when the user navigates up
- `android:nextFocusDown`: Defines the next view to receive focus when the user navigates down
- `android:nextFocusLeft`: Defines the next view to receive focus when the user navigates left
- `android:nextFocusRight`: Defines the next view to receive focus when the user navigates right

The following example XML shows two focusable UI elements where the `android:nextFocusDown` and `android:nextFocusUp` attributes have been set explicitly. The `Button` is located to the right of the `TextView`, however thanks to the magic of `nextFocus` properties, the user can reach the `Button` element by pressing the down arrow when the focus is on `TextView`:

```
<TextView android:id="@+id/text"
   android:text="Hello, world"
     android:focusable="true"
    android:nextFocusDown="@+id/Button"
    ... />
<Button android:id="@+id/button"
    android:focusable="true"

    android:nextFocusUp="@id/text"
    ... />
```

The easiest way to test your navigation is to run your app in the emulator and navigate around your UI using the emulator's arrow keys and **OK** button only. Check that the navigation works as expected in all directions, including when you're navigating in reverse.

> ## Note
>
> You can also modify the focus order of UI components at runtime, using methods such as `setNextFocusDownId()` and `setNextFocusRightId()`.

Custom view controls

If you build custom interface controls, make sure you implement accessibility interfaces for these custom views and provide content descriptions.

> ## Note
>
> If you want your custom controls to be compatible with all versions of Android back to 1.6, you'll need to use the Support Library to implement the latest accessibility features.

When creating custom views, you need to make sure these views are successfully creating `AccessibilityEvents` whenever the user selects an item or changes focus, as accessibility events are an important part of providing accessibility features such as text-to-speech.

To generate `AccessibilityEvents`, call `sendAccessibilityEvent(int)` with a parameter representing the type of event that's occurred. You'll find a complete list of the event types Android currently supports in the `AccessibilityEvent` reference documentation (http://developer.android.com/reference/android/view/accessibility/AccessibilityEvent.html).

Providing alternatives to audio prompts

To assist users who are hearing impaired, you should avoid including any audio-only feedback in your app. You should always accompany your app's audio feedback with a secondary feedback mechanism, such as closed captions, a transcript, on screen notifications, or another visual alternative.

Testing various font sizes

Android users can change the size of the font that appears across their device manually, from their device's **Settings | Accessibility** screen. To make sure these size changes also affect the text within your app, define your app's text and associated containers in scaled pixels (`sp`).

Also, keep in mind that when users have large fonts enabled, your app's text may become larger than the space you originally allocated for it, so you'll need to check that your text and UI looks good and functions normally even when the user has large text enabled. In particular, make sure that your UI elements don't overlap and that all touchable elements remain reachable across various text sizes.

Using recommended touch target sizes

Make sure all your app's touch targets are at least 48 x 48dp, and ensure that the space between on screen elements is always at least 8dp. This helps to ensure your UI is easier to navigate for people with manual dexterity challenges, but also for children with developing motor skills.

Providing alternatives to affordances that time out

Some apps feature icons or controls that disappear after a certain amount of time, for example it's common for video playback controls to fade out once the user is a few seconds into the video.

This poses a problem for people who are using accessibility features such as TalkBack, as TalkBack waits until the user has focused on a control before it reads that control's description. If your UI has controls that fade out quickly, these controls may actually vanish before the user has a chance to focus on them—which means they never get read out, and therefore the user won't be aware of their existence.

For this reason, you shouldn't rely on timed-out controls for high-priority tasks or important functions. If your UI does feature timed-out controls, you may want to disable this functionality when accessibility services are enabled, so these timed-out controls no longer disappear.

To ensure users with visual impairments can read your text more easily, it's recommended that you use a contrast ratio of 4.5:1 between your app's background and text. As a general rule, the smaller your text, the more contrast you'll need to use.

Also bear in mind that some of your users may be colorblind, so you shouldn't use color signals as the only means of conveying important information. In addition to color, you can use elements such as patterns, shapes, size, textures, or text.

Testing your application's accessibility features

Testing is a crucial part of creating an accessible app, as it can uncover problems with user interaction that you might not otherwise have spotted.

Testing your app's accessibility features typically involves:

- Using your app with audible feedback enabled:

  Audible accessibility services provide audio prompts that dictate the on screen content to your users as they move around your UI. The most effective way of testing the experience you're providing to users with visual impairments is to enable an audible accessibility service on your Android device and then interact with your app using sound alone.

  For Android users, feedback is typically provided via the TalkBack accessibility service. TalkBack comes preinstalled on many Android devices (open your device's **Settings** and select **Accessibility**, followed by **TalkBack**) but you can also download TalkBack for free from Google Play.

  Once you've enabled an audible accessibility service, spend some time navigating around your app using the spoken feedback only. Look for any opportunities to improve the experience for users who might be interacting with your app without sighted assistance.

  You should also check that your app provides enough information for the user to understand and act on each on screen element using a service such as TalkBack, *without* overloading them with too much information. This can be a tricky balancing act—too much or too little information will make it equally difficult for the user to decipher your UI.

- Navigating your app using directional controls only, instead of the touchscreen

As part of accessibility testing, you should verify that your app is easy to navigate using directional controls only, meaning without using the touchscreen, and ensure that users can move focus between your app's UI elements in a way that makes sense.

If available, you can use a physical device with a D-pad or trackball, but if your device doesn't have these hardware features you can use a software-based directional controller instead, or use the Android emulator and its simulated keyboard controls. You may even want to use TalkBack gestures (https://support.google.com/accessibility/android/answer/6151827), which allow users to navigate apps (and their device in general) using very specific gestures.

Summary

In this final chapter, we covered some best practices that I hadn't explored in detail in previous chapters, including app security and accessibility best practices. You also learned how to create more useful notifications, and began to explore the new notification options coming up in Android N.

If you want to learn more about developing effective Android apps, you'll find lots of additional information in the Android docs (http://developer.android.com/training/index.html), at the Android blog (http://android-developers.blogspot.co.uk/), or by checking out some of Google's code samples (http://developer.android.com/samples/index.html).