Jason Morris

Hands-On Android UI Development

Design and develop attractive user interfaces for Android applications

Packt

Hands-On Android UI Development

Design and develop attractive user interfaces for Android applications

Jason Morris

BIRMINGHAM - MUMBAI

Hands-On Android UI Development

First published: November 2017

Production reference: 1161117

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham
B3 2PB, UK.
ISBN 978-1-78847-505-1

www.packtpub.com

Credits

Author
Jason Morris

Reviewers
Jessica Thornsby
Michael Duivestein

Commissioning Editor
Kunal Chaudhari

Acquisition Editor
Noyonika Das

Content Development Editor
Onkar Wani

Technical Editor
Sachin Sunilkumar

Copy Editor
Shaila Kusanale

Project Coordinator
Devanshi Doshi

Proofreader
Safis Editing

Indexer
Francy Puthiry

Graphics
Jason Monteiro

Production Coordinator
Arvindkumar Gupta

About the Author

Jason Morris is a multi-discipline software developer. He has been developing software for as long as he can remember. He's written software for the desktop, server, feature phones and smartphones, and the web and microcontrollers. Jason programs in a wide range of programming languages, and loves a new programming challenge. When he's not writing code or spending time with his family, taking photos or camping, he's probably thinking about programming. In 2010/2011, he wrote *Android User Interface Development: A Beginner's Guide*, which helped many beginner Android developers take their first steps into the realm of User Interface design and development for mobile devices.

I thank my wife and daughter, who have been so patient while I was writing this book.

About the Reviewers

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 freelance technical writer and full-time Android enthusiast, the author of Android UI Design, and the coauthor of *iWork: The Missing Manual*.

Michael Duivestein is a software engineer living in Cape Town, South Africa.
His curiosity about all things code has meant he has enjoyed a career as a software engineer spanning many industries, including Point of Sale systems, investment management systems, and mobile payment systems.
Currently employed as a backend/API engineer for `Travelstart`, Africa's leading online travel agency, Michael spends his whole day, practically every day, working with Java and Android. In his spare time, Michael is a Kendo practitioner and enjoys playing Ingress to get out and about.

www.PacktPub.com

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

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 `service@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://www.packtpub.com/mapt`

Get the most in-demand software skills with Mapt. Mapt gives you full access to all Packt books and video courses, as well as industry-leading tools to help you plan your personal development and advance your career.

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

Customer Feedback

Thanks for purchasing this Packt book. At Packt, quality is at the heart of our editorial process. To help us improve, please leave us an honest review on this book's Amazon page at `https://www.amazon.com/dp/1788475054`.

If you'd like to join our team of regular reviewers, you can email us at `customerreviews@packtpub.com`. We award our regular reviewers with free eBooks and videos in exchange for their valuable feedback. Help us be relentless in improving our products!

Table of Contents

Preface

User interfaces represent the single most important aspect of any modern mobile application. They are the primary point of contact with your user, and a good user interface can be the difference between the success and failure of an application. This book will guide you down the path of user interface excellence, teaching you the techniques of great user interface design for Android, and showing you how to implement them.

Building user interfaces is all about building for the user and making their lives easier. It's important that the user interface helps the user achieve their goals within the application without distracting them. This means that each screen must serve a purpose, and every widget must work for its place on the screen. Widgets that are seldom used are distractions and should be moved, or removed, from the application entirely.

User interfaces are not just about pretty colors, fonts, and graphics; they are at the heart of the user experience within an application. How you create your user interfaces is built on the underlying structures you build in your applications code base. If the foundations of the application are not solid, the user experience will suffer. Every action taken by the user should have a quick and positive result, reinforcing their confidence that they can reach their goals within the application.

In this book, we'll be exploring not just how to write user interface code, but how to go about designing great user interfaces as well. We'll look at how lower layers like data storage can impact the user experience, and how to leverage the Android platform and its support libraries to build better applications, applications that look great, run fast, and help ensure that the user can understand the application quickly and is guided through with minimal distractions.

What this book covers

`Chapter 1`, *Creating Android Layouts*, will introduce or reintroduce Android Studio and help create a new Android application project from a template. We'll look at how an Android project is structured and how a user interface is wired together.

`Chapter 2`, *Designing Form Screens*, will show you how to design a form screen from scratch. You will learn techniques to decide what to put on the screen and how to layout a form screen to maximize its effectiveness while not disrupting your user's flow of thought.

`Chapter 3`, *Taking Actions,* will show you how to handle events in Android. It'll walk you through various types of events and provide you patterns and techniques to keep your application as responsive as possible. It'll also show you techniques to avoid complexity overload in your code base and how to keep your application's internal structure as clean as possible.

`Chapter 4`, *Composing User Interfaces,* will give you tools to build modular user interface components. It'll show you how to wrap related logic and user interface structures so that they can be reused, to reduce your code complexity while also making the user experience more consistent.

`Chapter 5`, *Binding Data to Widgets,* will introduce the data binding framework in Android. You'll find out why data binding exists, how it works, and how to use it effectively. You'll learn how to create user interface structures that automatically update when the data model changes.

`Chapter 6`, *Storing and Retrieving Data,* will cover how the storage and retrieval of data in a mobile app has a direct effect on the user experience. You will learn how best to build offline-first, reactive applications using the Room data access layer and data binding.

`Chapter 7`, *Creating Overview Screens,* will explore the `RecyclerView` and how it is often used in overview screens and dashboards to provide information. You'll learn how to create the data structures that support a `RecyclerView` and how to leverage data binding to massively reduce the boilerplate traditionally associated with these structures.

`Chapter 8`, *Designing Material Layouts,* will introduce several Material Design-specific patterns and widgets for Android apps. You'll learn how to leverage collapsing title bars and customize them to show your user additional information. You'll also learn how to add swipe-to-dismiss behavior, undo snackbars, and elevation to your user interfaces.

`Chapter 9`, *Navigating Effectively*, will help you learn to make a design-effective application navigation to guide your user to their intended goals intuitively. This chapter introduces several navigation-specific widgets and layout techniques, and shows where and how they can be most effectively applied.

`Chapter 10`, *Making Overviews Even Better*, will revisit the overview screen built in `Chapter 7`, *Creating Overview Screens,* and show how to leverage the Android platform API to produce polished overview screens. You'll learn how to use `DiffUtil` to automatically produce animations in `RecyclerView` and patterns that allow you to produce more readable overview lists.

`Chapter 11`, *Polishing Your Design*, will help you with the polishing of a good design. It'll introduce tools and techniques to help you choose a color scheme for your application. You'll also learn how to generate color schemes on the fly and how to create and use animations to guide your users.

`Chapter 12`, *Customizing Widgets and Layouts*, will introduce building your own custom widgets for Android. You'll learn how to render 2D graphics directly from Java code and how to create your own custom layouts. This chapter also shows how to build animation widgets that self-animate while they are visible.

`Appendix A`, *Activity Lifecycle,* a diagram and short description covering the lifecycle events delivered to an Android `Activity`, and when they can be delivered.

`Appendix B`, *Test Your Knowledge Answers,* the answers to the Test Your Knowledge section of each chapter.

What you need for this book

You'll need to download and install at least Android Studio 3.0. Using Android Studio, you will need to download a recent Android SDK so that you can compile and run your code.

Who this book is for

This book is for novice Android and Java developers who have basic knowledge of Android development and want to start developing stunning user interfaces.

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: "The next lines of code read the link and assign it to the `constantSize` attribute." A block of code is set as follows:

```
<selector
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:constantSize="true"
android:exitFadeDuration="@android:integer/config_shortAnimTime"
android:enterFadeDuration="@android:integer/config_shortAnimTime">
```

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

```
<selector
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:constantSize="true"
android:exitFadeDuration="@android:integer/config_shortAnimTime"
android:enterFadeDuration="@android:integer/config_shortAnimTime">
```

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: "In order to download new modules, we will go to **Files** | **Settings** | **Project Name** | **Project Interpreter**."

Warnings or important notes appear like this.

Tips and tricks appear like this.

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 email `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 emailed directly to you. You can download the code files by following these steps:

1. Log in or register to our website using your email 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/Hands-On-Android-UI-Development`. We also have other code bundles from our rich catalog of books and videos available at `https://github.com/PacktPublishing/`. Check them out!

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.

1
Creating Android Layouts

Mobile App user interfaces have come a long way from the early days, and while users have more devices to choose from than ever, they all expect consistently high-quality experiences from their apps. Apps are expected to run fast and the user interfaces are expected to be smooth; all this while running on a massive array of devices of varied capabilities. Your app needs to run on devices with screens as big as televisions on the one end of the scale, and smartwatches with screens as small as 2.5 cm or even smaller on the other end of the scale. At first glance, this may seem like a nightmare, but there are simple tricks to make building responsive Android apps easy.

In this book, you'll learn a diverse set of skills as well as some theoretical knowledge that you can apply to build fast, responsive, and great-looking Android applications. You'll learn how to go about designing the screens that your application will actually need, and then how to build them for maximum flexibility and performance while keeping your code easy to read and avoiding bugs.

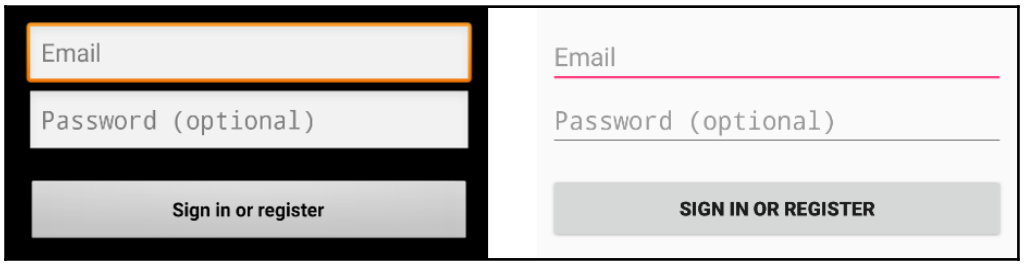
In this chapter, we will look at the basic principles used to build user interfaces for Android applications. You'll need to be familiar with concepts to build even the simplest Android application, so in this chapter, we'll cover the following topics:

- The basic structure of an Android application
- Creating a simple Activity and layout files using Android Studio
- Where to find the most useful parts of the Android Studio layout editor
- How a well-organized project is structured

Material Design

Since Android was first introduced in 2008, the user interface design has changed radically, from the first early versions of the grey, black, and orange theme, to the **Holo theme**, which was more of a stylistic change than a fundamental shift in the design language, and eventually culminating in material design. **Material Design** is more than just a style; it's a design language complete with concepts for navigation, and overall application flow. Central to this idea is the notion of paper and card, the idea that the items on the screen are not simply next to each other but may also be above and below in the third dimension (although this is virtual). This is achieved using the elevation property that is common to all widgets in Android. Along with this basic principle, material design offers some common patterns to help the user identify which components are likely to take which actions, even the first time they are used in the application.

If you look at the original Android theme alongside the Holo Light theme, you can see that while the style changed dramatically, many elements stayed similar or the same. The grey tones flattened, but are very similar, and many of the borders were removed, but the spacing remains very close to the original theme. The material design language is often very similar in its basic styling and design to Holo:



The design language is an essential part of modern user interface design and development. It not only defines the look and feel of your widget toolkit, but also defines how the application should behave on different devices and in different circumstances. For example, on Android, it is common to have a navigation drawer since the slides are from the left, while this would not feel natural to the user on other platforms. Material design defines much more than the look and feel of navigation, it also includes guidelines for motion and animation, how to display various types of errors, and how to guide your users through an application for the first time. As a developer or designer, you may feel like this limits your creative freedom, and to some degree it actually does, but it also creates a clear message for your users on how your app expects to be used. This means that your users can use your app more easily, and it requires less cognitive load.

Another aspect of application development, which is of vital importance in any modern mobile application, is its performance. Users have come to expect that applications will always run smoothly, no matter the actual load on the system. The benchmark for all modern applications is 60 frames per second, that is, a full render event delivered to the user every 16.6 milliseconds.

Users don't just expect an application to perform well, they expect it to react instantly to external changes. When data is changed on the server side, users expect to see it on their device instantly. This makes the challenges of developing a mobile application, especially one with good performance, become even more difficult. Fortunately, Android comes with a fantastic toolset and an enormous ecosystem for dealing with these problems.

Android attempts to enforce good threading and performance behavior by timing each event that happens on the main thread and ensures that none of them take too long (and producing an **Application Not Responding** (**ANR**) error if they do). It further requires that no form of networking is conducted on the main thread, since these will invariably affect the application's performance. However, where this approach gets hard to work with is--any code related to the user interface must happen on the main thread, where all the input events are processed, and where all the graphics rendering code is run. This helps the user interface framework by avoiding any need for thread locks in what is very performance-centric code.

The Android Platform is a complete alternative to the Java Platform. While at a high level, the Android platform APIs are a form of Java framework; there are noticeable differences. The most obvious is that Android does not run Java bytecode, and does not include most of the Java standard APIs. Instead, most of the classes and structures you'll use are specific to Android. From this perspective, the Android platform is a bit like a large opinionated Java Framework. It attempts to reduce the amount of boilerplate code you write by providing you with skeleton structures to develop your applications.

The most common way to build user interfaces for Android is to do it declaratively in the layout XML files. You can also write user interfaces using pure Java code, but while potentially faster, it's not commonly used and carries some critical pitfalls. Most notably, the Java code is much more complex when handling multiple screen sizes. Instead of simply being able to reference a different layout file and have the resource system link in the best fit for the device, you have to handle these differences in your code. While parsing XML may seem a crazy idea on a mobile device, it's not nearly that bad; the XML is parsed and validated at compile time, and turned into a binary format which is what is actually **read at runtime** by your application.

Another reason it's really nice to write Android layouts in XML is the **Android Studio layout editor**. This gives you a real-time preview of what your layout will look like on a real device, and the blueprint view helps enormously when debugging issues like spacing and styling. There is also an excellent linting support in Android Studio that helps you avoid common problems before you're even finished writing your layout files.

Android Studio

Android Studio is a fully-featured IDE built on top of the **IntelliJ** platform, designed specifically for developing Android applications. It has a huge suite of built-in tools that will make your life better, and help you write better applications more rapidly.

You can download Android Studio for your favorite **Operating System** (**OS**) from `https://developer.android.com/studio/`. The setup instructions for each operating system vary slightly, and are available on the website. This book has been written assuming an Android Studio version of at least 3.0.

Once installed, Android Studio will also need to download and install an Android SDK for you to develop your applications on. There are platform options for virtually every version of Android ever released, including emulated hardware, which allows you to test how your application will run on different hardware and Android releases. It's best to download the latest Android SDK, and one of the older versions as well to check backward compatibility (4.1 or 4.4 are good options).

Android application structure

An Android application is very different in its internal structure when compared to applications on other platforms, in even the simplest details. Most platforms see their applications as monolithic systems with a fixed entry point. When the entry point returns or exits, the platform assumes that the application has finished running. On Android, an application may have several different entry points for the user, and others for the system. Each entry point has a different type, and different ways for the system to reach it (called **intent filters**). The most important part of an application from the user perspective is its activities. These (as the name suggests) are supposed to represent an action that the user will take with the application, such as the following:

- List my emails
- Compose an email
- Edit a contact

Each `Activity` is a non-abstract class extending the `Activity` class (or any descendant of `Activity`), and registers itself and its intent filters in the application manifest file. Here's an example of how the manifest entry for an `Activity` that can view and edit contacts might look:

```
<activity android:name=".ContactActivity">
 <intent-filter>
   <!-- Appear in the launcher screen as the main entry point of the
application -->
   <action android:name="android.intent.action.MAIN" />
   <category android:name="android.intent.category.LAUNCHER" />
 </intent-filter>
 <intent-filter>
   <!-- Handle requests to VIEW Uris with a mime-type of 'data/contact' -->
   <action android:name="android.intent.action.VIEW" />
   <data android:mimeType="data/contact"/>
 </intent-filter>
 <intent-filter>
   <!-- Handle requests to EDIT Uris with a mime-type of 'data/contact' -->
   <action android:name="android.intent.action.EDIT" />
   <data android:mimeType="data/contact"/>
 </intent-filter>
</activity>
```

The application manifest file (always named as `AndroidManifest.xml`) is how the Android system knows what components your application has, and how to reach each of them. It also contains information such as the permissions your application will need from the user, and which versions of the Android system the application will run on.

Each `Activity` is typically intended to do a single thing from the user's perspective, but this is not always the case. In the preceding case, there are three possible intent filters, each of which telling the system something different about the `ContactActivity` class:

- The first one tells the system that `ContactActivity` should have its icon displayed on the launcher screens, effectively making it the main entry point of the application
- The second tells the system that `ContactActivity` can be used to `VIEW` content with a mime-type of `"data/contact"`
- The third tells the system that `ContactActivity` can also be used to `EDIT` content with the `"data/contact"` mime-type

The system resolves `Activity` classes through Intents. Each Intent specifies how and what the application would like to do on behalf of the user, and the system uses the information to find a matching intent-filter somewhere in the system. However, you won't typically add intent-filters to all of your `Activity` entries; you'll launch most by specifying the class directly within your application. Intent-filters are normally used to implement abstract inter-application interactions for example, when an application needs to "open a web page for browsing," the system can automatically launch the user's preferred web browser.

An `Activity` will generally have a primary layout file defined as an XML resource. These layout resource files are generally not standalone, but will make use of other resources and even other layout files.

Keep your activities simple! Avoid loading too much behavior into a single `Activity` class, and try and keep it bound to a single layout (and its variants, such as "landscape"). At worst, allow multiple behaviors with common layout widgets (for example, a single `Activity` to view, or edit a single contact). We'll go through some techniques for this in `Chapter 4`, *Composing User Interfaces*.

The resource system in Android needs some special attention, as it allows for multiple files to collaborate together to create complex behavior out of simple components. At its heart, the resource system selects the most appropriate of each resource when it is requested (including from inside other resources). This not only allows you to create screen layouts for portrait and landscape mode, but allows you to do the same for dimensions, text, colors, or any other resource. Consider the following example:

```
<!-- res/values/dimens.xml -->
<dimen name="grid_spacer1">8dp</dimen>
```

The above dimension resource can now be used in layout resource files by name:

```
<!-- res/layouts/my_layout.xml -->
<LinearLayout
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:layout_margin="@dimen/grid_spacer1">
```

Using this sort of technique, you can adjust layouts for different screen sizes by simply changing the distance measurements used to space and size the widgets, rather than having to define completely new screen layouts.

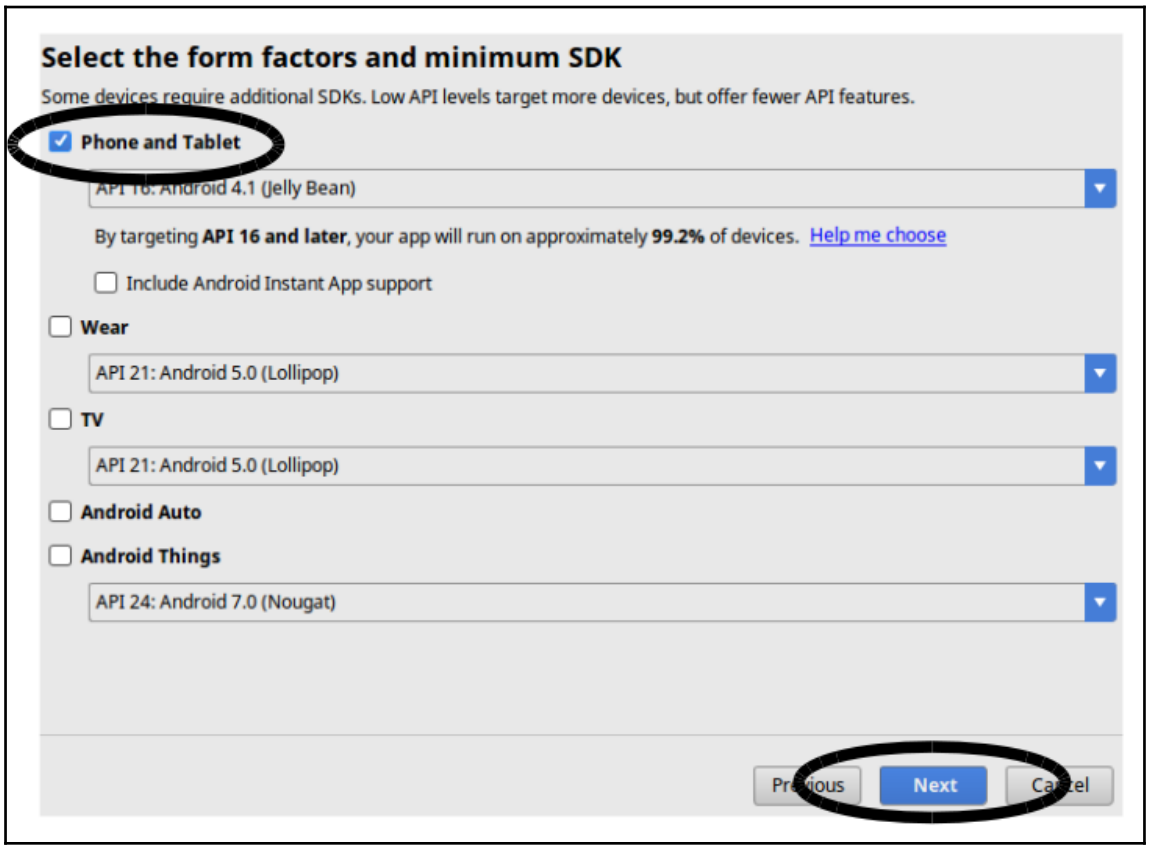
It's a good idea to try and be generic with resources such as dimensions and colors. This will help keep your user interface consistent for your users.
Consistent user interfaces are often more important than trying to innovate. The less mental effort your user needs to understand your application, the more they can engage with it.

Creating SimpleLayout

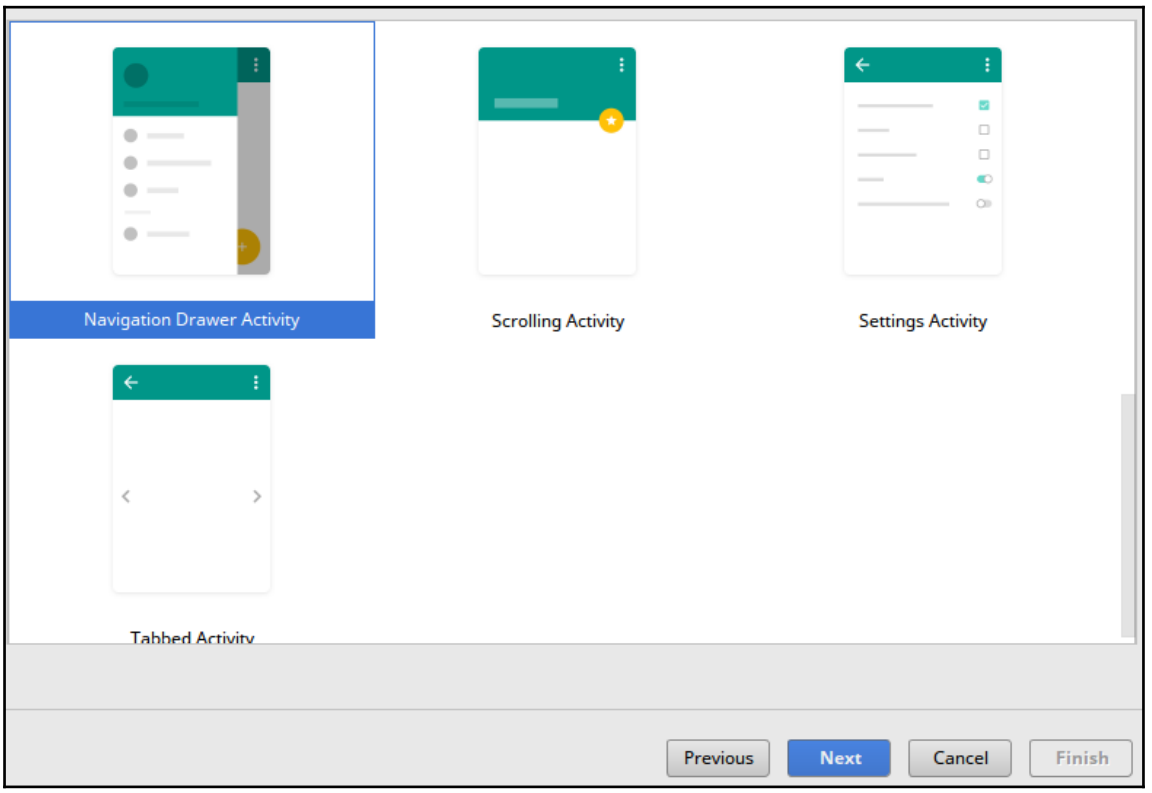
Now that we've gone over the basics of an Android application structure, let's create a simple screen and see how things all fit together. We'll use Android Studio and one of its wonderful template activities. Just follow these easy steps:

1. Start by opening Android Studio on your computer.
2. Start a new project using the **File** menu, or the quickstart dialog (depending on which one shows up for you).
3. Name the project as `SimpleLayout`, and leave any additional support (C++, Kotlin) off:

4. On the next screen of the New Project wizard, ensure that you support Android 4.1 or higher, but leave only **Phone and Tablet** checked for this task:

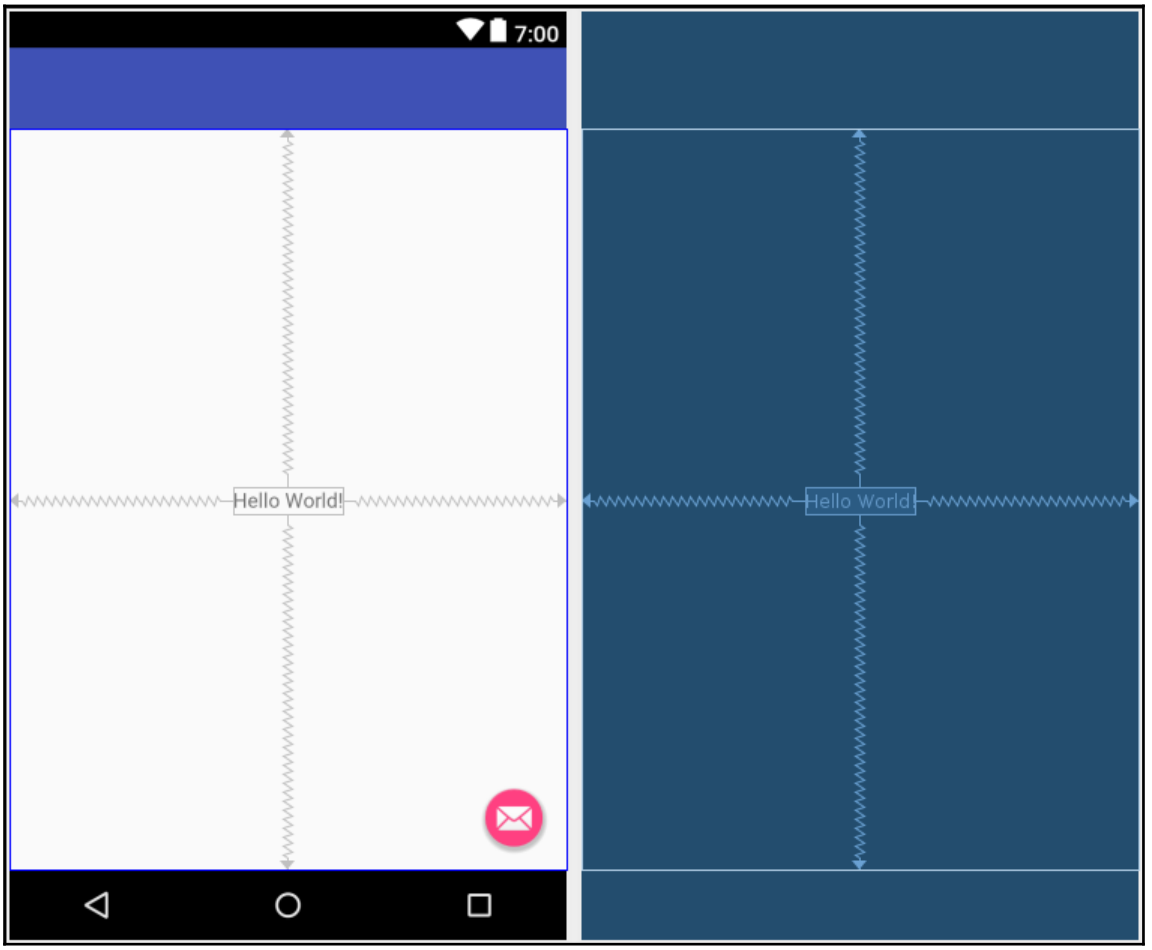


5. Android Studio comes with a fantastic selection of **Activity templates** available on the next screen. This will be the first `Activity` generated to get you started with your project. For this example, you'll want to scroll down the list and find **Navigation Drawer Activity**. Select it and click on **Next**:



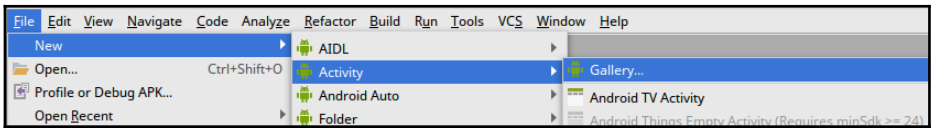
Leave the `Activity` details as their defaults (**MainActivity**, and so on) and click on **Finish** to complete the New Project wizard. Android Studio now creates your project and runs a first build-sync over to get everything working.

6. Once your project has finished being generated, you'll be presented with the Android Studio layout editor, looking something like this:



Congratulations, this template provides an excellent starting point to explore how Android applications and their user interfaces are built and fit together.

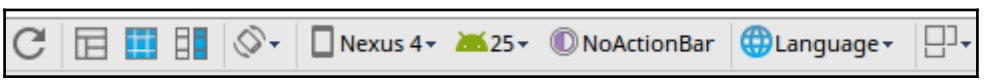
If you ever want to get back to the Activity templates screen, you can use the **Gallery...** option in the Android Studio **File** | **New** | **Activity** menu:



Discovering the layout editor

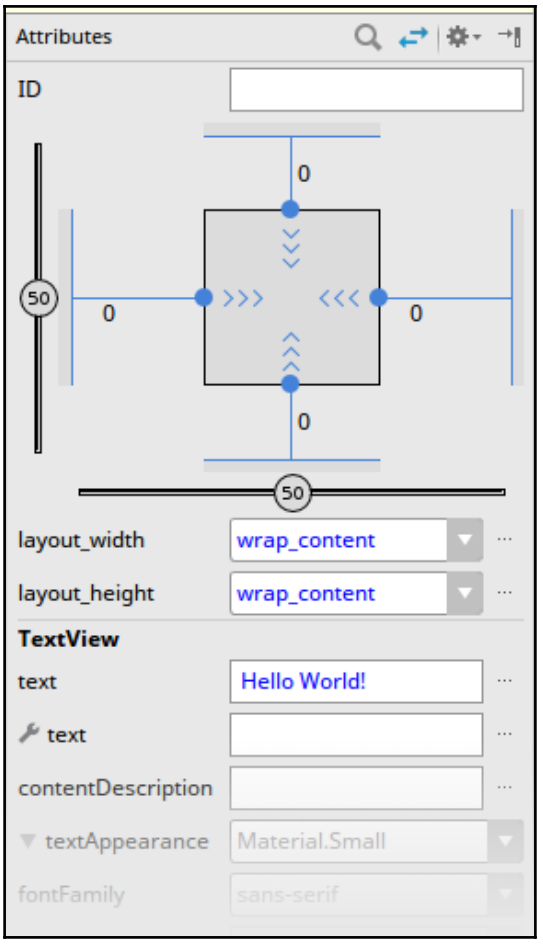
At first glance, the layout editor in Android Studio is a standard WYSIWYG editor; however, it has several important features that you need to be aware of. Most importantly, it actually runs the code for the widgets in order to render them within the editor. This means that if you write a custom layout or widget, it will look and behave as it will on the emulator or on a device. This is fantastically useful for rapidly prototyping screens, and can drastically cut down on development time when used properly.

To ensure that your layout is being rendered correctly, you'll sometimes need to ensure that the layout editor is configured correctly. From the toolbar at the top of the layout editor, you can select the virtual device configuration you would like it to emulate. This includes whether the layout is being viewed in portrait or landscape mode, and even what language settings to use for the layout rendering and resource selection:



It's important to keep in mind that the list of available Android platform versions that the layout editor can emulate is limited, and it is not connected to the list that you have installed as virtual devices (so you cannot add new versions to the layout editor by installing additional platform versions). If you want to see how your user interfaces of versions that Android Studio doesn't directly support look, the only way to do it is to run the application.

The next really important thing to note is the attributes panel, which is docked to the right of the layout editor by default. When you select a component in the design area, the attributes panel allows tweaking of all the attributes that can be changed in XML, and of course, you get to see the results of any changes live in the layout editor:

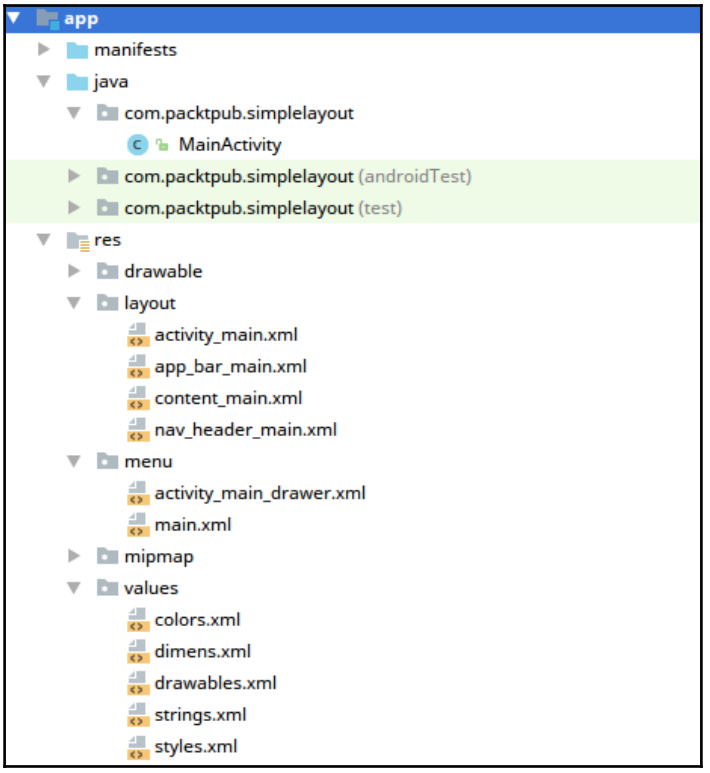


The number of attributes is generally kept well under control by Android Studio. The default panel only shows the most commonly used attributes for the selected widget. To toggle between this shortlist and the list of all the available attributes (something you'll do more often than you think), you'll want to use the toggle button ( ) at the top of the attributes panel.

However, when you look at the **All Attributes** view, you'll note that their sheer number makes the view rather difficult to use. The easiest way to solve this is to use the search button ( ) to find the attribute you're looking for. This will allow you to search for attributes by name, and is the quickest way to filter the list and get to the attribute, or group of attributes, that you're looking for (that is, `scroll` will give you all the attributes containing the word `scroll`, including `scrollIndicators`, `scrollbarSize`, `scrollbarStyle`, and so on).

Organizing project files

Android Studio gives you a fairly standard Java project structure, that is, you have your main source sets, tests, a resources directory, and so on, but that doesn't really cover all of your organizational needs. If you check the project structure we created, you might note some patterns:



1. You'll first note that only a single `Activity` was created--`MainActivity`, but this `Activity` template has generated four layout files.

2. Only `activity_main.xml` is actually referenced by `MainActivity`; all the other files are included via the resource system.

3. The next thing to note is that the layout file referenced by `MainActivity` is named as `actvitity_main.xml`; this is a standard naming pattern that Android Studio will actually suggest when creating new `Activity` classes. It's a good idea, because it helps separate layouts used for `Activity` classes from those used elsewhere.

4. Next, take a look at the names of the other layout files. Each of them is also prefixed with `nav`, `app_bar`, and `content`. These prefixes help group the layout files logically in a file manager and in the IDE.

5. Finally, you'll note that the `values` directory has several XML files in it. The entire `values` directory is actually treated as one big XML file by the resource compiler, but it helps keep it organized by the type of resources being declared.

Use filename prefixes in the resources directories (especially layouts) to keep things organized. You cannot break things down into subdirectories, so a prefix is the only way to group files together logically. Common prefixes are "activity", "fragment", "content", and "item", which are commonly used to prefix layouts that are used to render list items and so on.

6. If you open the `MainActivity` class now, you'll see how the layout is loaded and bound. The first thing `MainActivity` does when it's created is to call `onCreate` to its parent class (which is a mandatory step, and failure to do so will result in an exception). Then, it loads its layout file using the `setContentView` method. This method call does two things at once: it loads the layout XML file, and adds its root widget as the root of the `Activity` (replacing any widgets that were already there). The `R` class is defined by the resource compiler, and kept in sync for you by Android Studio. Every file and value resource will have its own unique identifier, which allows you to keep things tightly bound together. Rename a resource file, and its corresponding field will change:

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

7. You'll then note that `MainActivity` retrieves various widgets that were included in the layout files by their own IDs (also defined in the `R` class). The `findViewById` method searches through the `Activity` layout for a widget with the corresponding `id`, and then returns it:

```
// MainActivity.java
Toolbar toolbar = (Toolbar) findViewById(R.id.toolbar);
setSupportActionBar(toolbar);
```

The `findviewById` method works by traversing all of the widgets in an `Activity` in a series of loops. There is no lookup table or optimize this process. As such, you should call the `findViewById` method in `onCreate` and keep a class-field reference to each of the `View` objects you'll need.

8. The preceding code snippet will return the `Toolbar` object declared in the `app_bar_main.xml` layout resource file:

```
<!-- app_bar_main.xml -->
<android.support.v7.widget.Toolbar
    android:id="@+id/toolbar"
    android:layout_width="match_parent"
    android:layout_height="?attr/actionBarSize"
    android:background="?attr/colorPrimary"
    app:popupTheme="@style/AppTheme.PopupOverlay" />
```

`findViewById` can also be found on the `View` class, but it's a relatively expensive operation, so when you have widgets that will be used again in an `Activity`, they should be assigned to fields in the class.

Summary

As you saw, an Android application comprises of more modular components, which assemble in layers, and are often directly accessible from the platform. The resource management system is your greatest ally and should be leveraged to provide your users with a consistent experience, and keep your user interface consistent. When it comes to arranging your application, Android Studio has a variety of tools and features that it will use to help you keep things organized and within commonly understood patterns. However, it's also important to stick to your own patterns and keep things organized. The Android toolkits have their own requirements, and you'll need to obey their rules if you want to benefit from them.

Android Studio also has an excellent collection of template projects and Activities, and they should be used to get your projects kick-started. They can also often serve with explanations for how common user interface design patterns are implemented in Android.

In the next chapter, we'll take a look at starting a layout from scratch and how to approach designing a form screen.

2
Designing Form Screens

Form screens are an essential part of user interface design in many ways because their history is a lesson in how not to do things. Most applications need to capture input from their users at some point, and you need input widgets for that, but you should always consider the minimum amount of information you need to ask the user for, rather than try to get all the information you may need in the future. This approach will keep the user focused on the task they are trying to carry out. Presenting them with a wall of input fields is overwhelming to most users and breaks their focus, which in turn can lead to them abandoning what they were trying to do with your application.

This chapter is focused on form screens, and will walk you through a bit of their history before diving into a method for actually designing form screens. This approach can and should be reused whenever you need to design a screen for an app. It's always important to take a step back from your code work and consider how things will look and fit together for the user; it's often the difference between a successful app and failure.

In this chapter, we'll develop a practical form screen using Android Studio and the layout editor. Starting from an empty template in a new project, you'll learn the following:

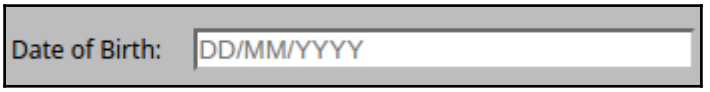
- How to break up and then arrange a form layout to be most effective for your users
- How to use resources to keep your user interface consistent
- How to style widgets to help the user understand what the widget should be used for
- How to build drawable resources that respond to state changes

Exploring form screens

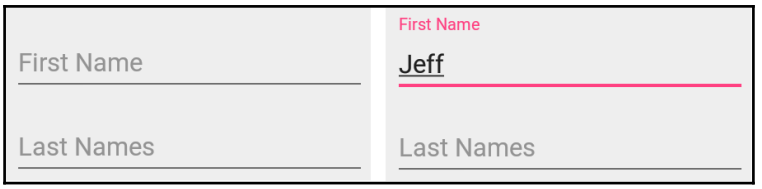
While not the most glamorous component of an application's user experience, form screens are a long-time staple of software. A form screen can be defined as any screen where the user is expected to explicitly enter or change data, as opposed to viewing or navigating it. Good examples of form screens are login screens, edit profile screens, or the add contact screen from a phonebook app. Over the years, the idea of what constitutes a good form screen has changed, with some people going as far as to shun them completely. However, you can't capture the user's data out of thin air.

The Android standard toolkit provides an excellent and diverse collection of widgets and layout structures to facilitate building excellent forms, and in Material Design applications, form screens can often double as a *view* screen (what will usually be a read-only version of the form screen) thanks to the placement of labels. A good way to understand this principle is to consider the evolution of textboxes. As soon as you have a blank space to be filled by your user, you need to tell the user what to put there, and when we started labeling textboxes, we simply copied how we did this on paper forms--by putting a label to the one side of the textbox:

The issue with this is that the label always takes up quite a bit of space, and takes up even more if you need to include some validation rules for the user (such as date inputs--**DD/MM/YYYY**). This is where we started adding hints to the input boxes. The label will explain what to add in the **Date of Birth** textbox, and a hint within the textbox will tell the user how to input valid data:



From this pattern, many mobile applications began to drop the label completely and instead used the hint/placeholder to contain the data on the theory that from the context of the form, the user will be able to infer what data was in each of the textboxes. However, this means that the user has to do a bit of extra thinking in order to make sense of the screen when they see it for the first time. This extra delay can quickly turn to frustration, and reduces how usable your application is. For this reason, Material Design text inputs turn their hints into small labels that move above the textbox when the user focuses on the textbox, making it easier for them to keep track of what information they are entering:



This also reduces the amount of work that needs to be done on form screens as a developer, because you typically won't need to separate the *view* and *edit* screens of your application, since the form will always have all of its labeling available. However, it's important to avoid overcrowding your screens with input widgets. Nobody likes to have to fill in lots of data, even if most of it is optional. Instead, always consider the minimum amount of data you need from your user at each point in your application. It's also important to consider how you will ask the user for their data.

We'll start our first form screen as an information capturing screen. We'll be building an imaginary app to track someone's travel expenses, allowing them to capture, tag, and store each of their expenses to be filtered and reviewed later. The first thing we need is a screen where the user can capture an expense and any additional information that goes with it.

As best as possible, you should make input fields optional, but you can always encourage people to give more data by telling them how complete something is. This is a common technique when dealing with a user profile--"Your profile is 50% complete", helps encourage the user to provide more data to raise that number. This is a simple form of gamification, but it's also very effective.

Designing a layout

Good user interface design is rooted in some simple rules, and there are processes you can follow to design a great user interface. For example, imagine that you're building an app to capture travel expenses so that they can be claimed easily at a later time. The foremost thing we'll build over here is the screen that captures the details of a single claim. This is a perfect example of a modern form screen design.

When designing a layout, it's a good idea to use a mockup tool such as Balsamiq (`https://balsamiq.com/`), or even paper and pencil to think about the layout of the screen. Physical index cards make excellent thinking spaces as they have similar proportions to a phone or tablet. Using paper, especially, helps you think about the arrangement of the screen instead of being distracted by the exact colors, fonts, and spacing that should be dealt with in a common set of theming rules.

To start designing the screen, we need to consider what data we'll need from the user and how we might be able to fill some of it out for them. We also need to try and stick to the platform design language so that the application doesn't feel out of place to the user. It's also important when designing form screens to ensure that the entire input form will fit onto the device's display. Scrolling an input screen requires your user to remember what is not on the screen, and causes frustration and anxiety. Whenever you design a form screen, ensure that all the inputs will fit onto one display. If they don't all immediately fit onto the display together, first consider whether you can remove some of them. After removing any that aren't absolutely required, consider grouping some on a single line, ensuring that you put no more than two inputs per line. More than two input fields on a single line implies that you can probably turn them into a single input.

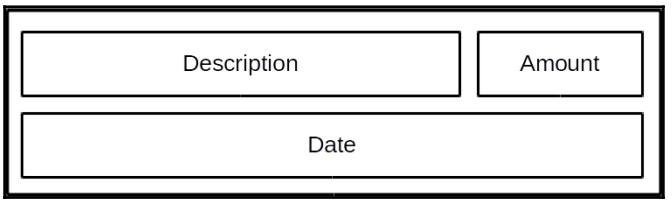
So, to get started, consider what information a user will want to capture for a travel expense:

- How much the expense was
- Some photos of the invoice, or maybe of the item purchased
- The date that they captured the expense on
- What sort of expense they're capturing such as food, transport, accommodation, and more
- A short description to help them remember what the expense was

Great, that seems like a good starting point, but they're not in a great order and they're not grouped at all. We need to consider what is most important, and what groups logically fit together well on-screen. For starters, let's focus on developing a portrait layout for a phone, since that will be our most common use case. So, the next thing to do is group the input components in a way that will feel logical and familiar to a user. When looking at an overview of claims, the things we'll want to list are as follows:

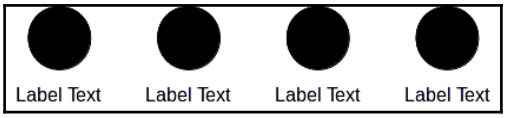
- The date of the expense:
    - The date that they captured the expense on
- The amount the claim is for:
    - How much the expense was
    - Some photos of the invoice, or maybe of the item purchased
- The description of the claim:
    - What sort of expense they're capturing such as food, transport, accommodation, and more
    - A short description to help them remember what the expense was

So, we'll group these three fields together, and put them at the top of the screen. This particular grouping will feel common to anyone who has used any budget or expense tracking software:

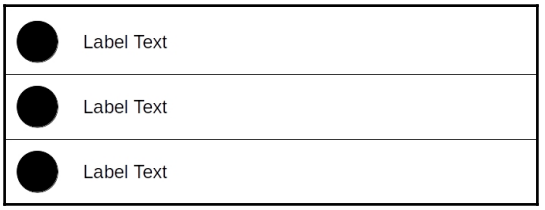


The **Date** is a special field, because we can easily populate it with the current date. It's most likely when the user enters this screen that they are capturing an expense for the same day. We still need to capture a category and attachments for the expense. Attachments will need a large amount of space so that the user can preview them without having to open each one to know what it is, so we'll put them at the bottom of the screen and have them take any space left over. That just leaves the category. The expense categories are best represented using icons, but we need some space for text so that the user knows what each icon means. We can do this in one of several ways:

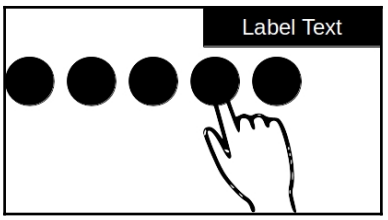
1. Place a tiny label above or below each icon:
    - Pros: All the labels are always on the screen
    - Cons: The labels can be hard to read on smaller screens, and the icons take up more screen space:



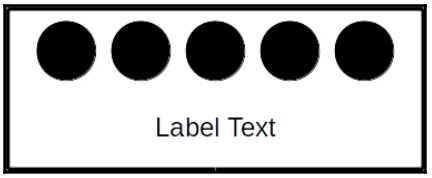
2. Create a vertical list of icons and put a nice large label to the right of each:
    - Pros: The labels are easy to read, and always associated with their icons
    - Cons: This will take up lots of the vertical space that is best used to display attachment previews:



3. Show only the icons, and display the label when the user holds their finger over the icon (long presses):
    - Pros: The text takes up no screen space
    - Cons: This sort of behavior is not intuitive to users, and requires that the user selects the category to know what its label is:

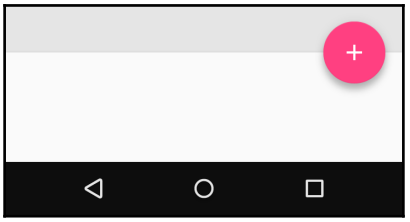


4. Show the list of icons and the text of the selected category below them:
    - Pros: The text label can be large and easy to read, and takes up less screen space because only one label is shown at a time
    - Cons: The user must select the category to know what its label is:



In order to keep the label in a nice, easily readable size, while also drawing extra attention to which category is currently selected, this example will show you how to create option four, where the currently selected category name is shown below a horizontal list of the category icons. We'll also highlight the selected icon to help keep a connection between the two user interface elements.

The one remaining thing that the user needs to be able to do is attach files to the expense claim before saving it. There should be a nice, large area at the bottom of this layout that will make a perfect area to preview a single attachment, and if the user has more than one attachment, they can swipe left and right to switch between their previews. However, how can they attach them in the first place? This is where a floating action button is an ideal solution. You'll see floating action buttons everywhere in Android applications. They are usually near the bottom-right of the screen, where a right-handed person will have their thumb if they hold the phone one-handed, and out of the way of most Western content, which will be to the left of the screen (normally):

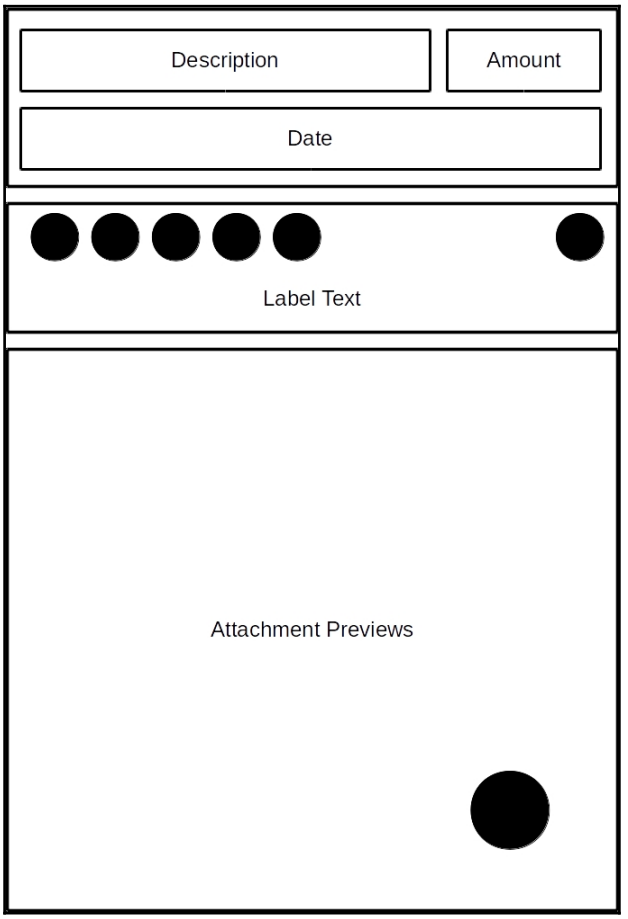


Floating action buttons are normally the most common *creative* (as opposed to navigation or destructive) actions on the screen; for example, creating a new email in the Gmail or Inbox app, attaching a file, and so on.

So now, we have the screen broken down into three logical areas, outside of the normal decorations:

- Claim details
- Categorization
- Attachment

Putting them together into a single screen layout concept gives you a wireframe, looking something like this:



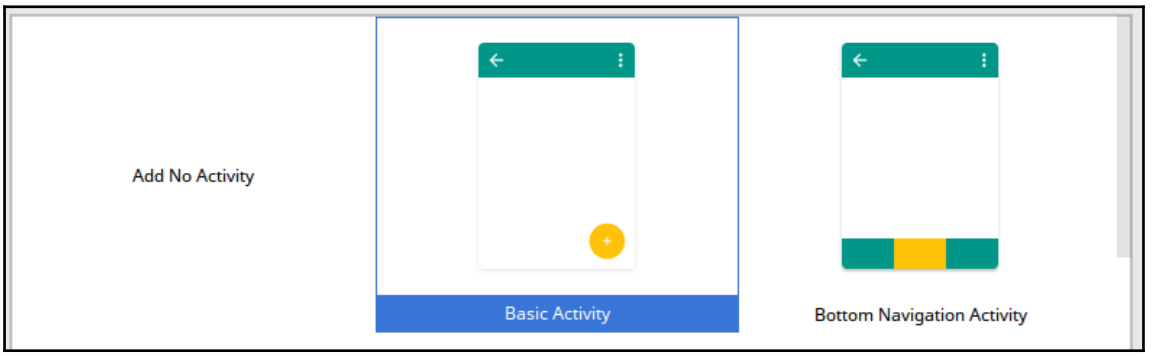
This exercise of wireframing your screens before you start developing them is an extremely valuable phase, because it gives you time and space to think about each of the choices you could be making, rather than just grabbing the first available widgets in the toolbox and putting them onto the screen. Now that you have a wireframe, you're ready to get started with building the user interface for the application.

Creating a form layout

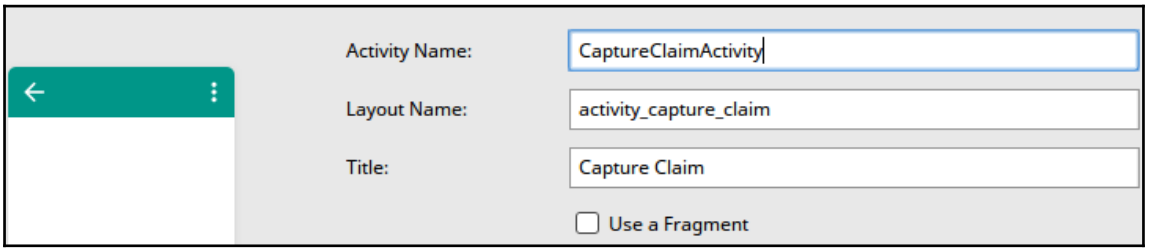
Once you have a good wireframe to work from, you'll want to start developing the user interface screen. For this, we'll use Android Studio and its wonderful layout editor.

Since this is a brand new project, you'll need to open Android Studio and use **File | New | New Project** to get it started. Then, follow these steps:

1. Name the project `Claim`, and leave any non-Java support turned off.
2. Target Android 4.1 on **Phone & Tablet** only.
3. In the Activity Gallery, choose the **Basic Activity**:



4. Name the new Activity `CaptureClaimActivity`, and then change the title to `Capture Claim`. Leave the other parameters at their default values:



5. Finish the New Project wizard, and wait for the project to be generated.
6. When the project has been generated and synchronized, Android Studio will open the `content_capture_claim.xml` file in its layout editor.

7. By default, Android Studio assumes that you will be using a `ConstraintLayout` as the root of your layout. This is an incredibly powerful and flexible tool, but also not well suited as the root element of this user interface. You'll need to switch over to the **Text** view at the bottom of the screen in order to change to something more suitable:

8. The file will currently have something like the following XML in place:

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.constraint.ConstraintLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    app:layout_behavior="@string/appbar_scrolling_view_behavior"
    tools:context="com.packtpub.claim.CaptureClaimActivity"
    tools:showIn="@layout/activity_capture_claim">

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hello World!"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintRight_toRightOf="parent"
        app:layout_constraintTop_toTopOf="parent" />

</android.support.constraint.ConstraintLayout>
```

9. Change the `ConstraintLayout` to a simple `LinearLayout`. `LinearLayout` is one of the simplest layouts available on Android. It renders each of its children in a straight line, either horizontal or vertical, depending on its orientation attribute. Replace the whole of the `content_capture_claim.xml` file with the following content:

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

```
    android:orientation="vertical"
    app:layout_behavior="@string/appbar_scrolling_view_behavior"
    tools:context="com.packtpub.claim.CaptureClaimActivity"
    tools:showIn="@layout/activity_capture_claim">

</LinearLayout>
```

Choosing the right layout to use is about more than just keeping your code simple; less flexible layouts are much faster at runtime and lead to a much smoother user experience. Try to stick to simpler layouts where possible, but also avoid nesting layouts too deep (one inside the other), as this also leads to performance problems.

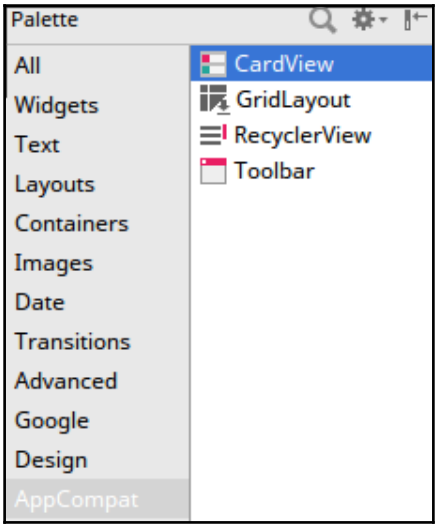
10. Change back to the **Design** view in the layout editor, and you'll notice that the **Component Tree** to the left of the design view now has a **LinearLayout (vertical)** as its only component.

Creating the description box

Now that the base layout is set up, it's time to start adding the widgets to the user interface and make it useful. In this next stage, you'll be using several Material Design widgets that help produce great user interfaces, such as **CardView** and the **TextInputLayout** widget. Before Material Design, text input boxes were just plain `EditText` widgets, which while still available, are now generally discouraged in favor of a `TextInputLayout`. The `TextInputLayout` is a specialized layout that contains a single `EditText` widget for the user to enter text data. The `TextInputLayout` then also provides the floating hint/label effect and animations, transitioning the `EditText` widgets hint to a label space just above the input area. This means that even when the user has filled in the text, the hint for the `EditText` is still visible above their input:

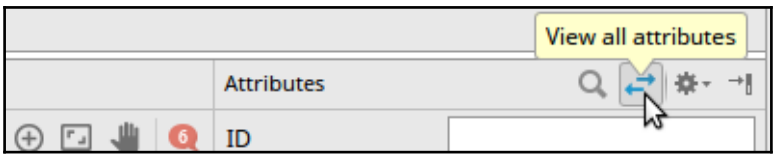
You'll be wrapping this first group of widgets in a `CardView`, which will serve as a visual grouping for the user. Follow these steps to add the description input box:

1. Open the **AppCompat** section of the **Widget Palette**. This contains widgets that come from special APIs that are parts of the extended Android platform. They're not included on the platform by default, and instead, are included in each application they are used in, a bit like static linking a library.
2. Drag and drop a `CardView` into your user interface design; you can drop it anywhere on the design canvas. This will serve as the grouping for the description, amount, and date input boxes. Ensure that in the **Component Tree**, the `CardView` appears as a child of the **LinearLayout (vertical)**:

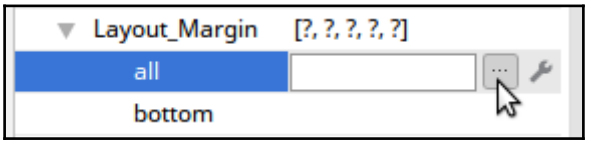


3. A `CardView` stacks its widgets on top of each other in layers (back to front). This is not what's needed in this case, so you'll need to open the **Layouts** section of the `Palette` and drag a `ConstraintLayout` into the `CardView` on your design. Ensure that in the **Component Tree**, the `ConstraintLayout` appears as a child of the `CardView`.
4. Select the new `ConstraintLayout` in the **Component Tree**.

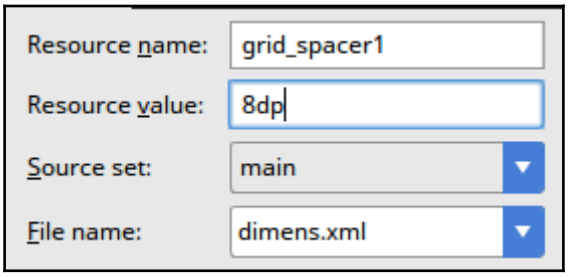
5. In the **Attributes** panel, select the **View all attributes** button:



6. Open the section titled **Layout_Margin**.
7. Click on the resource editor button for the **all** line, as shown in the screenshot:



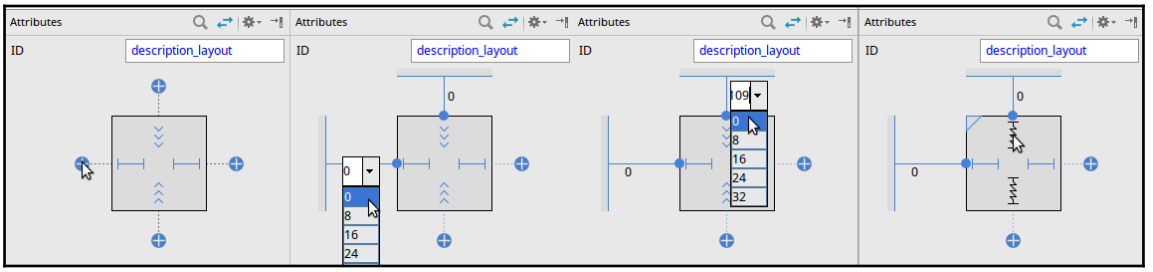
8. In the resource editor, select the **Add new resource** button in the top-left, and choose **New dimen value** (dimen is short for dimension. A dimension resource can be used to specify sizes in non-pixel units, which are then converted according to the actual display system on the user's device).
9. Name the resource `grid_spacer1`, and give it a value of `8dp`:



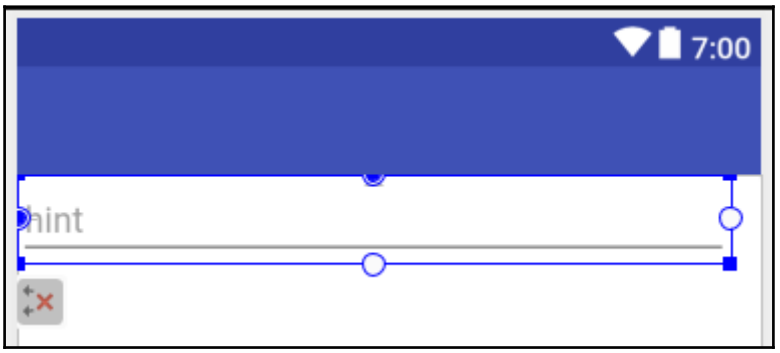
Material Design interfaces on Android use a spacing grid in an increment of **8dp**, which is *8 density-independent pixels*. This is a special unit of measurement that varies the actual number of pixels used based on the density of the screen. These are also the most common unit of on-screen measurement in Android. A **1dp** measurement will be 1 physical pixel on a 160dpi screen, and scaled to 2 pixels on a 320dpi screen. This means that by measuring your layout in terms of density-independent pixels rather than physical pixels, your user interface will translate better over the range of screen densities it might encounter on various devices.

10. Click **OK** to create the dimension resource and go back to the layout editor.
11. Now, you'll need to start building up the input boxes for the user to fill in. The first of these will be the description box. Open the **Design** section of the **Palette**, and drag a `TextInputLayout` into the **Component Tree** as a child of the **ConstraintLayout:**

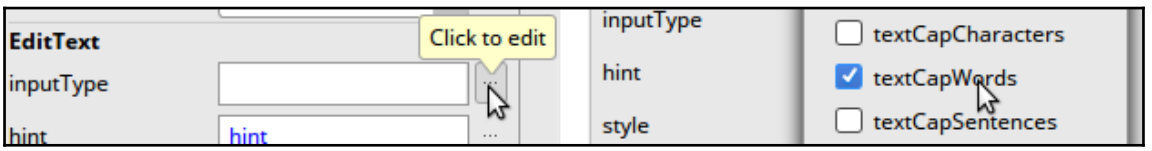
12. In the **Attributes** panel, click on the **View fewer attributes** button (it's the same one as **View all attributes**).
13. At the top of the **Attributes** panel, set the **ID** of the `TextInputLayout` to `description_layout`.
14. Use the Constraint editor (just below the **ID** attribute) to create connections to the left, and above the `TextInputLayout` by clicking on the blue circles with the **+** signs in them. Then, change the constrained margins to zero on both the new constraints, as shown:



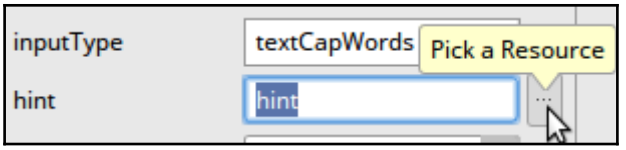
15. Your `TextInputLayout`, now named `description_layout`, should have snapped to the top-left corner of the layout editor:



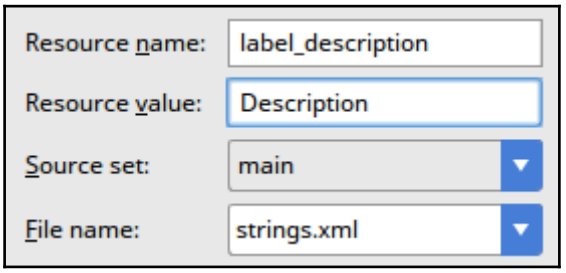
16. Change the `layout_width` attribute to `match_constraint`, and the `layout_height` parameter to `wrap_content`. The `TextInputLayout` will shrink to the minimum space it can occupy in the top-left corner.
17. Now, using the **Component Tree**, select the `TextInputEditText` inside the `description_layout TextInputLayout`.
18. In the **Attributes** panel, change the **ID** to `description`, since this is the field you actually want to capture the contents of.
19. Change the **inputType** to **textCapWords**; this will instruct software keyboards to place a capital letter at the beginning of each word:



20. The hint/label for the description box is currently **hint**, and it's hardcoded into the layout. We want to change it to `Description`, and make it localizable (so that it's easy to translate the app into new languages). Use the edit button to open the string resource editor, and choose **Add new resource** | **New string value**:



21. Fill in the **Resource name** as `label_description`. You'll notice that this follows another prefix rule, which helps when dealing with large numbers of string resources in your source code.
22. Fill in the **Resource value** as `Description`, and leave the remaining fields unchanged:



23. Click on **OK** to create the new string resource and go back to the layout editor.

In this section, you created a grouping component (the `CardView`) that will serve to visually group the description amount and date fields for the user, and you have populated it with its first component--the description box. You have also created a dimension resource that can be reused throughout your application to represent a single grid spacing unit, allowing you to adjust the size of the grid for the entire application. A consistent grid spacing in the application helps define a consistent look and feel for the application, and keeping this value as a resource provides you with a single place where you can change it if required.

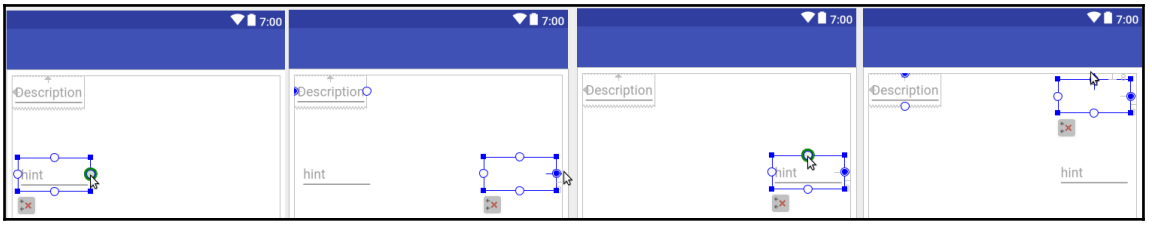
Adding the amount and date inputs

In this next section, we'll finish building the description box by adding the `amount` and `date` fields. This will involve using some more complex constraints on the widgets you will be adding, as they will need to be positioned relative to each other. Follow these steps to finish the description box:

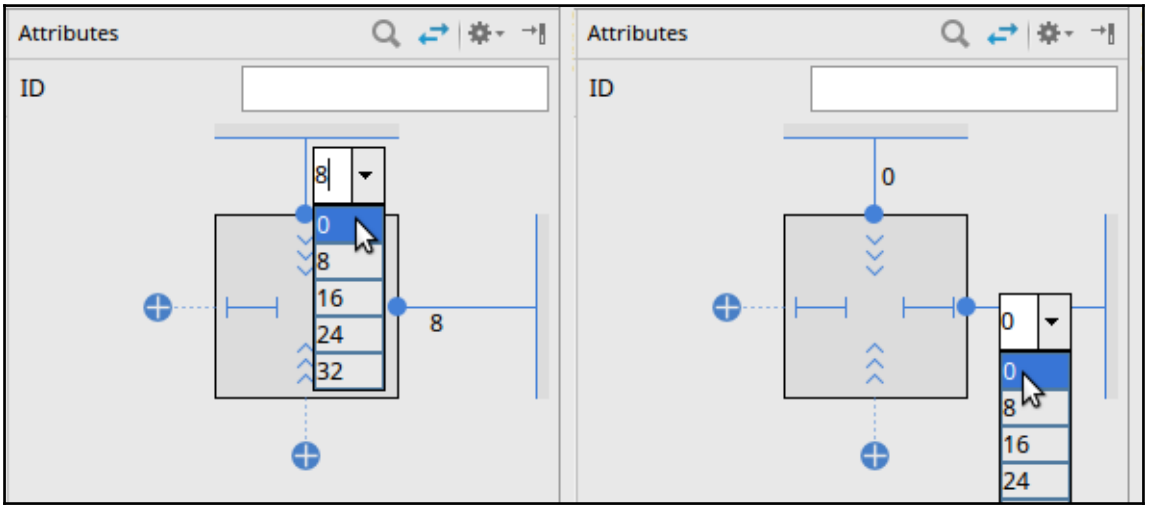
1. Drag another `TextInputLayout` into your design and place it somewhere below the **Description** field. This new box has no constraints as of yet.
2. In the **Attributes** panel, change the **ID** to `amount_layout`.
3. In the **Attributes** panel, open the resource editor for `layout_width` as you did to create the `grid_spacer1` resource earlier.
4. Create a new resource named `input_size_amount`, and set its value as `100sp`.

Much like dp, **sp** (**scale-independent pixels**) is a relative pixel size, but unlike density-independent pixels, scale-independent pixels are scaled according to the user's font preferences. Normally, these are used to specify font sizes, but they can also be useful when specifying fixed sizes for text input widgets.

5. Now, drag the right constraint handle to the right of the layout, and then drag the top constraint handle to the top of the layout, as shown:



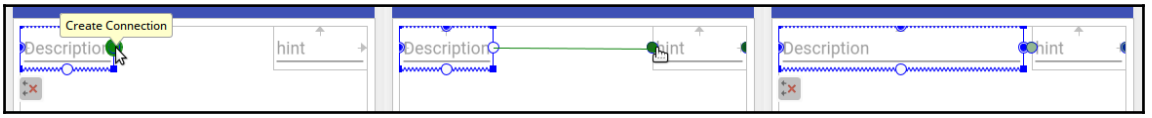
6. Now, zero the margins using the constraint editor in the **Attributes** panel:



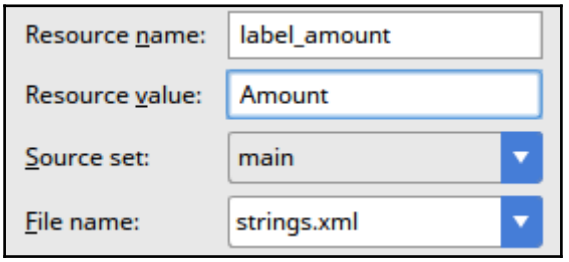
7. Now, use the **Component Tree** to select the `description_layout TextInputLayout` widget.

When selecting widgets directly in the design view, the editor will pick the deepest child of the **Component Tree** that you clicked on. This means that if you click on the **Description** field directly, you will select the `TextInputEditText` box, instead of the `TextInputLayout`. So, when dealing with the `ConstraintLayout` in particular, it's often better to select widgets in the Component Tree to ensure that you pick the right one.

8. In the layout view, drag the right constraint handle of the description `TextInputLayout` to meet up with the left constraint handle of the new `amount_layout` and `TextInputLayout`:

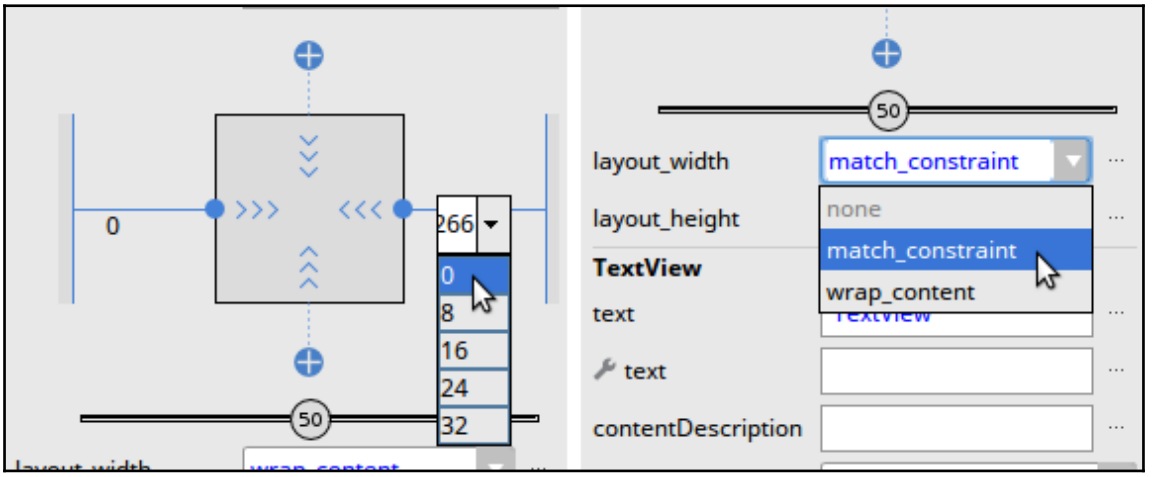


9. Click on the new `TextInputEditText` widget in the **Component Tree** panel.
10. In the **Attributes** panel, change the **ID** to `amount`.
11. Use the attribute editor to change the **inputType** to **number**.
12. For the **hint** attribute, open the resource editor to create a new string resource.
13. Name the resource as `label_amount`, and give it a value `Amount`:



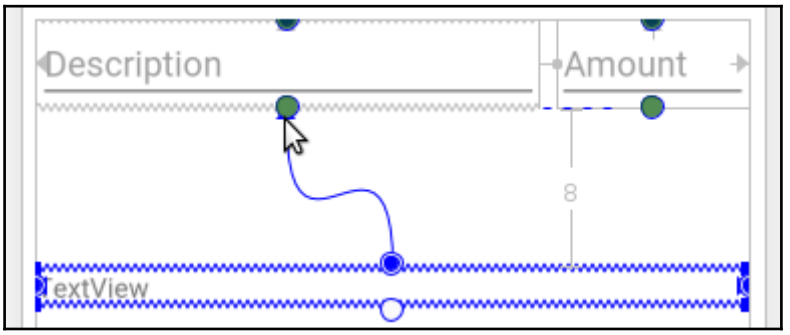
14. Now, we'll add a label for the `Date` input field; in the **Palette** panel, open the **Text** section and drag a new `TextView` into the layout editor.
15. Using the constraint editor in the **Attributes** panel, add a constraint to the left and right, and then zero their margins.

16. Change the `layout_width` to `match_constraint` so that the label takes up all the available width:



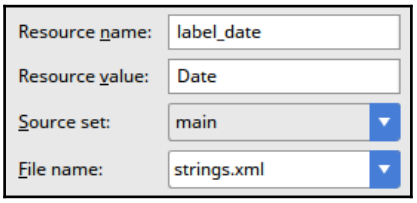
The `match_contstraint` value available to the children of a `ConstraintLayout` is a special marker attribute that will cause the widget to fill the space made available by its constraints. This is similar to how the `match_parent` value will cause a widget to take up all the space made available by its parent.

17. Now, drag a new constraint from the top of the new `TextView` to the bottom of the Description `TextInputLayout`:

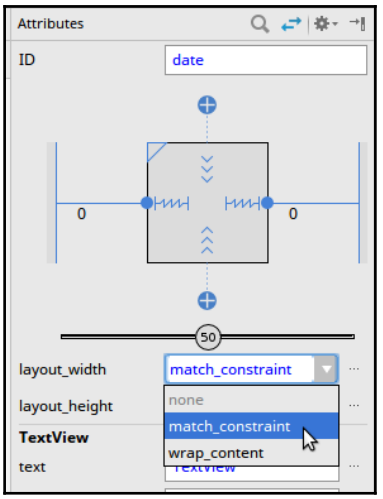


18. Use the resource editor to create a new string resource for the **text** attribute.

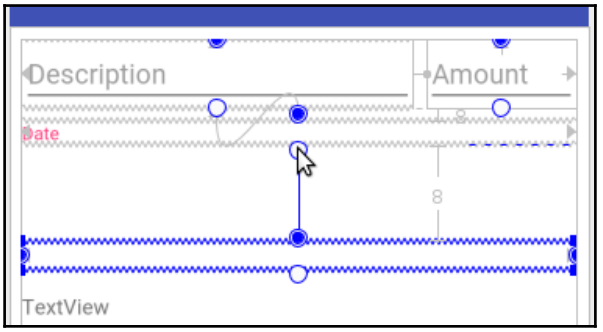
19. Name the new resource `label_date`, and make its value `Date`:



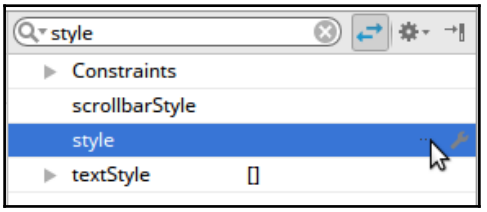
20. Still in the **Attributes** panel, change the **textAppearance** attribute to `AppCompat.Caption`. This is the same **textAppearance** style used by `TextInputLayout` for the hovering label when the cursor is focused on its `EditText`.
21. Now, use the resource selector on the **textColor** attribute to choose the **colorAccent** color resource. This is the highlight color that is generated by Android Studio for you, and is also used by `TextInputLayout`. Your `TextView` should now look like the focused label for a `TextInputLayout`, which is exactly what you want, because the next widget should look like an `EditText`, but isn't.
22. From the **Palette** panel, drag another `TextView` into the design layout.
23. Use the **Attributes** panel to change its **ID** to `date`.
24. Create left and right constraints, and set them to zero.
25. Change the `layout_width` to `match_constraint` so that the `date TextView` takes up all the horizontal space:



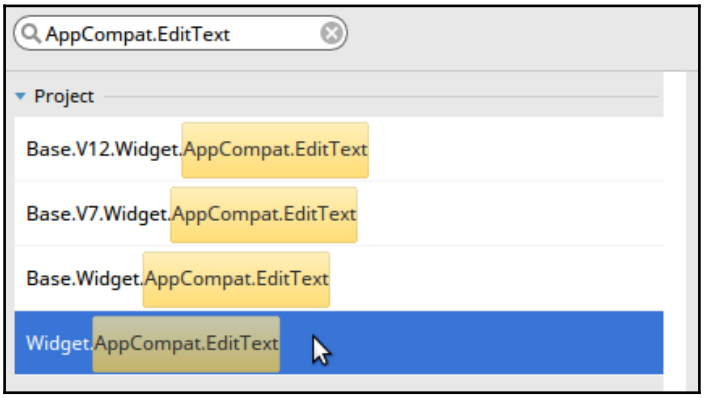
26. Drag the constraint handle from the top of the `date TextView` to the bottom of its `TextView` label:



27. At the top of the **Attributes** panel, use the **View all attributes** toggle button to view all the available attributes.
28. Using the **Attributes** search box, find the **style** attribute:



29. Open the resource selector for the **style** attribute.
30. Using the search box, find the `AppCompat.EditText` style:



31. Clear the search box, and toggle back to the **View fewer attributes** panel.
32. Clear the **text** attribute by deleting its content (this `TextView` should be empty in the layout file).
33. In the **Component Tree**, select the `CardView`.
34. In the **Attributes** panel, change its `layout_height` to `wrap_content`. The `CardView` will roll upward, taking up just enough space to contain the widgets that now make up the description, amount, and date inputs.

Unlike the description and amount input boxes, the date is actually made up of two labels that are styled so that together, they look like a focused `TextInputLayout` widget. This is important because the user will populate the date using a calendar dialog, rather than typing the date using a keyboard. A calendar dialog is more user-friendly, and less error-prone than manual date entry. Also, like this, the component looks familiar to the user, giving them a suggestion of how it should be used. This sort of styling capability is very important and useful in Android, and it's worth learning how standard components are composed together and styled so that you can build these sort of emulations.

Your completed **Description**, **Amount**, and **Date**, so the capture box should look like this in the Android Studio layout editor:

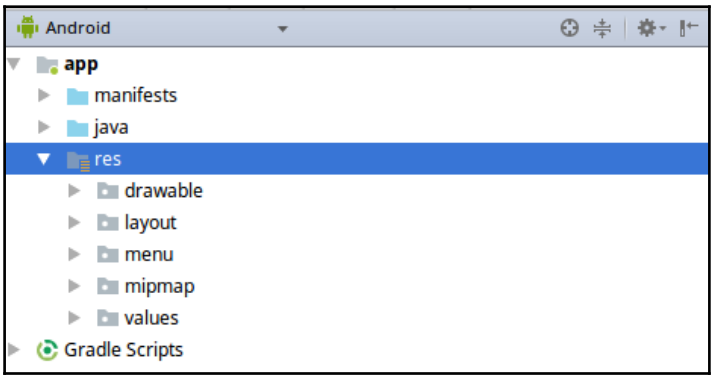
Creating the category chooser

The category chooser is where the user will select how to file their expense claims. There'll be a fairly small number of these, and they will be represented by icons in the user interface. Fortunately for Android developers, Material specifies a huge range of standard icons, and Android Studio has features to import them as bitmap or vector graphics files. When deciding whether to use bitmap images or SVGs, it's important to consider the trade-off between these two formats, specifically in relation to Android. Especially so since in Android, multiple copies of a bitmap are often provided for different screen sizes and densities, leading to much higher-quality scaling (as most will only ever be scaled down slightly). Here's a quick table to compare them:

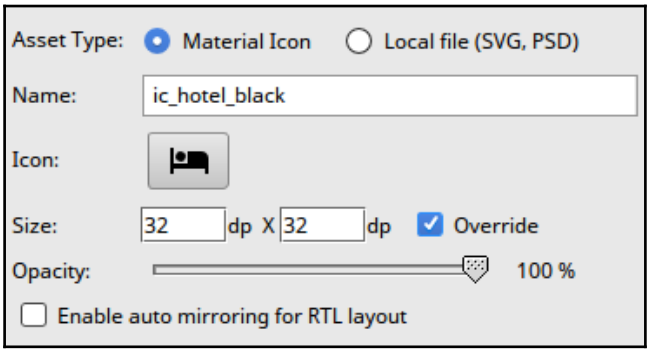
Bitmaps	**Vector Graphics**
Are supported on all platforms natively	May require a support library to work
Can be handled by the GPU and rendered with full acceleration	Must be rendered into a bitmap before they can be rendered onto the screen, which takes time
Take up more space in your app's APK, especially as you may need to provide different copies for different screen sizes and densities	Are stored as binary XML files, and take up very little space in the APK
Suffer massive loss of quality when scaled up, and loss of detail when scaled down	Can be rendered at virtually any size with no perceptible loss of quality or detail

For the category chooser widget, you'll be importing vector graphics icons and using them as radio buttons. Let's get things started:

1. In the files view to the extreme left of Android Studio, right-click on the **res** directory and select **New**, **Vector Asset** to open the vector import tool:



2. Where it says **Icon**, click on the button with the Android robot.
3. Use the search box at the top-left of the dialog to find the "hotel" icon, and select it.
4. Click on **OK** to return to the import tool.
5. The import tool will have changed the proposed name to `ic_hotel_black_24dp`; change this to `ic_accommodation_black`:



6. In the **Size** boxes, select the **Override** checkbox and change the size to `32` **dp X** `32` **dp**.

7. Click on **Next** and then on **Finish** to complete the import.
8. Repeat this process, and find the room service icon. Name this one `ic_food_black`, and don't forget to change its size to `32` **dp X** `32` **dp**.
9. Repeat this for the airport shuttle icon. This is `ic_transport_black`, and again, change its size to `32` **dp X** `32` **dp**.
10. Repeat and find the local movies icon; name this `ic_entertainment_black` and remember to change its size to `32` **dp X** `32` **dp**.
11. Find the "business center" icon and name it `ic_business_black`; again, change its size to `32` **dp X** `32` **dp**.
12. Finally, find the all inclusive icon, name it `ic_other_black`, and override its size to `32` **dp X** `32` **dp**.

Now you have a collection of black icons that will serve as the basis for your category selector.

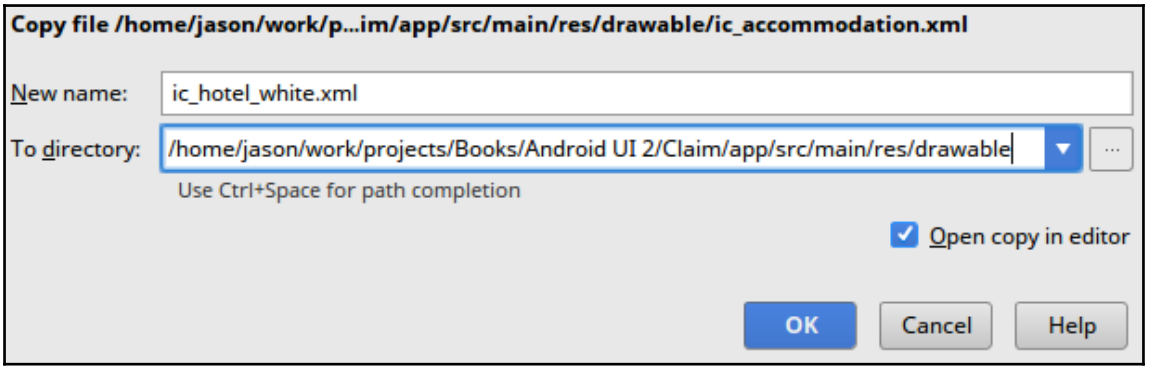
Making icons change with state

In Android, images have state; they can change how they look according to the widget that is using them. In fact, this is how a button works; it has a background image that changes state depending on whether it's pressed, released, enabled, disabled, focused, and so on. For us to show the user which of these categories they have actually selected, we need to provide them with a visual indication on the icon. This involves some editing:

1. Start by making a copy of the `ic_accommodation_black.xml` file that was generated, and name this one `ic_accommodation_white.xml`. Use copy, and then paste the file into the same directory to have Android Studio bring up a **Copy** dialog.

Vector graphics in Android are XML files representing the various shapes and colors that make up the graphic. A vector graphic doesn't contain the pixel data like a bitmap image (such as a `.png` or `.jpeg`), but contains instructions for how to render the image. This means that by adjusting the coordinates contained within the instructions, the image can be made larger or smaller with little or no loss of quality.

2. **Beware**, because by default, Android Studio might have selected the **drawable-xhdpi** directory as the target for the paste operation. If it has, you'll need to change this to `drawable`:



3. The editor will open with the new copy of the icon, which will still be black. The code for the file will look something like this:

```
<vector
  android:height="32dp"
  android:viewportHeight="24.0"
  android:viewportWidth="24.0"
  android:width="32dp"
  xmlns:android="http://schemas.android.com/apk/res/android">

    <path android:fillColor="#FF000000" android:pathData="..."/>
</vector>
```

4. Change the `android:fillColor` attribute from `#FF000000` to `#FFFFFFFF` to change the icon from black to white.

Colors in Android resources are specified using the standard Hexadecimal color notation. This is the same notation used on the web in CSS and HTML files. Each pair of two characters represents one part of the color component with values from 0 to 255 (inclusive). The components are always Alpha, Red, Green, and Blue, in that order. Alpha represents how transparent or opaque the color is, zero (00) being completely invisible, while 255 (FF) is completely opaque.

5. Now, repeat this operation for all the other icons you imported, ensuring that each one is copied to the **drawable** directory, and change its name from `_black` to `_white`.
6. You now have a black and white version of each icon; black is perfect to place against the white background of a `CardView`, while white is perfect to place against the accent color of your application, and shows how the icon has been selected by the user. For this, we need even more drawable resources. Right-click on the **drawable** directory and choose **New| Drawable resource file**.

6. Name this new file `ic_category_accommodation` and click on **OK**.
7. Android Studio will now open the new file, which will be an empty selector file:

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

A selector element corresponds to a `StateListDrawable` object from the `android.graphics.drawable` package. This class attempts to match its own state flags against a list of possible visual states (other `drawable` objects). The first item that matches is displayed, which means that it's important to consider the order you declare the states in.

9. First, tell the selector that it will always be the same size by setting its `constantSize` attribute, and then tell it that it should quickly animate between its state changes. This short animation gives the user an indication of these changes when choosing a category:

```
<selector
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:constantSize="true"
android:exitFadeDuration="@android:integer/config_shortAnimTime"
android:enterFadeDuration="@android:integer/config_shortAnimTime">
```

10. First, you'll need to create a state for when the category is selected; you'll use two layers: one will be a simple circle background filled with the accent color, and over that you'll have the white version of the accommodation icon:

```
<item android:state_checked="true">
  <layer-list>
    <item>
      <shape android:shape="oval">
        <solid android:color="@color/colorAccent"/>
      </shape>
```

```
        </item>
        <item
            android:width="28dp"
            android:height="28dp"
            android:gravity="center"
            android:drawable="@drawable/ic_accommodation_white"/>
    </layer-list>
</item>
```

11. Then, create another `item` that is the default state--the black-filled accommodation icon:

```
<item android:drawable="@drawable/ic_accommodation_black"/>
```

12. Repeat this process for each icon you imported so that each one has a stateful, drawable icon that you can use in the layout file.

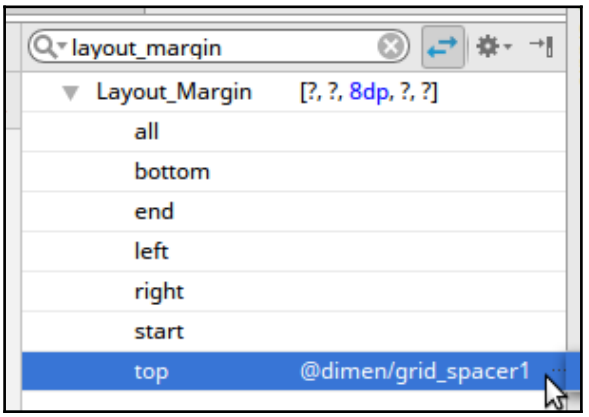
This process is often repeated, and there may even be more drawable resources involved for more varied state lists. Drawable elements are not always nested, as you did with the preceding `state_checked` item; they are often written into external drawable resources and then imported. This allows them to be reused without requiring the resource to be state-aware.

Creating the category picker layout

Now, it's time to go back to the layout editor and start creating the category selector box with these icons:

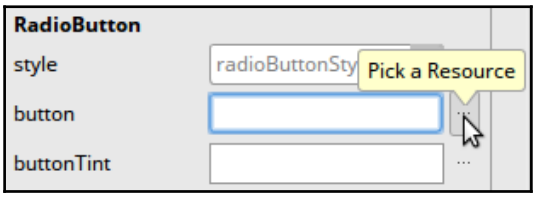
1. Reopen the `content_capture_claim.xml` layout file from the `res/layout` directory.
2. In the **Palette** panel, open the **AppCompat** section and drag another `CardView` into the layout editor. Drop it below the `CardView` for the description, amount, and date input fields.
3. In the **Attributes** panel, use the **View all attributes** toggle button and search box to find the layout margin.
4. Open the **Layout_Margins** attribute group.
5. Then, open the resource selector for the **top** attribute.

6. Select the `grid_spacer1` dimension resource you created earlier, and click on **OK** to close the resource selector:



7. Then, in the **Palette**, open the **Layouts** section and drag a **LinearLayout (vertical)** into the new `CardView`.
8. In the **Attributes** panel, use the resource selector to change the **all** margin attribute to `grid_spacer1` to create some padding from the edges of the `CardView`.
9. Clear the **Attributes** panel search box.
10. Open the **Containers** section of the **Palette**, and drag a `RadioGroup` into the new `LinearLayout` in the layout editor. A `RadioGroup` is a specialized `LinearLayout` that handles the toggling of its child `RadioButton` widgets, which you'll use to allow the user to select a category.
11. In the **Attributes** panel, change the **id** attribute to `categories`.
12. In the **Attributes** panel, use the search box to find the **orientation** attribute and change it to `horizontal`.
13. Clear the **Attributes** panel search box, and toggle it back to **View fewer attributes**.
14. Open the **Widgets** section of the **Palette** and drag a `RadioButton` into the new `RadioGroup`.
15. In the **Attributes** panel, change the **ID** to `accommodation`.
16. Clear the **layout_weight** attribute.

17. Use the resource editor for the button attribute to select the `ic_category_accommodation` you created earlier:

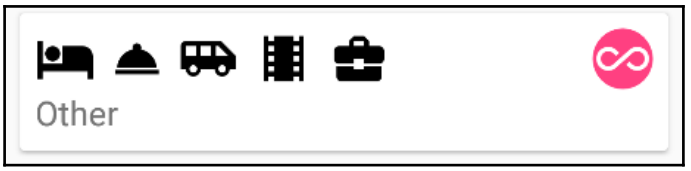


18. Clear the **text** attribute, since these radio buttons won't have any labels.
19. You'll then use the **contentDescription** attribute to store the human readable name of the category. Open the resource editor, create a new string resource named `description_accommodation`, and give it a value of `Accommodation`.

The `contentDescription` attribute is part of the accessibility system, and it's used by screen readers and similar aids to describe a component that might not have a text label. In this case, it's a perfect place for us to grab the human-readable description of a category. It's not an on-screen space, and it also serves users with accessibility enabled.

20. Toggle the **Attributes** panel to view all the attributes, and then find the layout margins.
21. Use the resource selector to change the **end** margin attribute to `grid_spacer1`.
22. Repeat the process of adding and populating radio buttons for the categories, giving each of them a suitable name in their **ID** and **contentDescription** attributes. Leave the "other" category until last so that it appears to the right of all the others.
23. In the **Component Tree** panel, select the **RadioGroup**.
24. In the **Attributes** panel, change its **layout_height** to `wrap_content`.
25. From the **Palette**, open the **Text** section and drag a `TextView` into place below the `RadioGroup`.
26. In the **Attributes** panel, change the **ID** to `selected_category`.
27. Clear the **text** attribute.
28. Use the drop-down on the **textAppearance** attribute to select `AppCompat.Medium`.
29. In the **Component Tree**, select the `CardView` containing the category selection components.
30. Now in the **Attributes** panel, change the **layout_height** to `wrap_content`.

The `CardView` will wrap upward, packing in the radio buttons and the label that you will use to display the currently selected category name. The `CardView`, again, serves to visually group the categories, and helps the user understand how they use this area of the screen:



The use of standard styles and theming, again, helps the user to quickly grasp how things work; even though the categories are just a row of icons, they are underscored by the selected category name.

Adding the attachment preview

After completing the category selector box, roughly half the available layout space should be left empty underneath. This is where the user will be able to preview the attachments they have added to the claim. We want the user to be able to swipe through these attachments left and right, and the easiest way to allow this is a `ViewPager`. A `ViewPager` is a special type of Android widget that links to an `Adapter` (other examples are `ListView`, `Spinner`, and `RecylerView`). An `Adapter` object turns data (such as rows from a database cursor, or objects from a `java.util.List`) into widgets that can be displayed on the screen.

Follow these steps to add it to the layout:

1. The `ViewPager` class is not available from the **Palette** panel, so at the bottom of the layout editor, change from **Design** mode to **Text** mode, so that you can edit the layout XML directly.
2. Go to the bottom of the file and find the space between where the last `CardView` element is closed and where the `LinearLayout` is closed.
3. Insert a `ViewPager` element into that space:

```
</android.support.v7.widget.CardView>

<android.support.v4.view.ViewPager
   android:id="@+id/attachments"
   android:clipChildren="false"
   android:clipToPadding="false"
   android:paddingBottom="@dimen/grid_spacer1"
```

```
        android:layout_weight="1"
        android:layout_width="wrap_content"
        android:layout_height="0dp"
        android:layout_marginTop="@dimen/grid_spacer1"/>
</LinearLayout>
```

4. Change back to the **Design** view, and you'll note that a new box has been added to the layout and blueprints where the empty space was.

The `clipChildren` and `clipToPadding` attributes in the preceding code change how the `ViewPager` and its children treat the space around them when rendering. The `CardView` class draws its shadows outside of its boundaries, and by default, these are clipped by the graphics system. Turning the clipping off allows the shadows and borders to be rendered completely.

A `ViewPager` doesn't look like anything on its own; its children are the only things that make it appear visually. So, until the user has added an attachment to a claim, nothing will appear in this space. This is not a problem, since the empty area provides a space for the software keyboard to appear when they enter the description and amount.

Try it yourself

Using the knowledge you have gained in this chapter, import the attachment icon as a vector graphic, change its fill color to white, and set it as the icon of the floating action button that appears at the bottom-right of your layout. Once you have the icon right, try increasing the size of the floating action button to make it more thumb-friendly for your users.

Test your knowledge

1. When designing a form screen, what is the first thing you should consider?
    - The colors and icons you want to use
    - The data you need from your user
    - The standard guidelines for Android

2. What is the standard spacing increment in Material design?
    - 8 Pixels
    - 8 Density Independent Pixels
    - 8 Device Pixels
3. The `ConstraintLayout`, `ViewPager`, and `CardView` are part of the support APIs. What does this mean?
    - Their bytecode must be included with your application if you use them
    - They are also used as part of the Android Studio code base
    - They can only contain other widgets from the support APIs
4. When building a new layout, your root widget should always be which of these?
    - A `ConstraintLayout`
    - A `LinearLayout`
    - The simplest widget that makes sense for your layout

Summary

In this chapter, we looked at how to design and then build a form screen in detail. These screens are an important part of applications, because they are where you users give you their details and as such they need to be especially intuitive and quick to use. Nobody likes to spend a lot of time filling in forms, and even less if they are using a mobile device. It's always good to remember that people normally use apps for relatively short periods of time; "what was that email?", is a more common action than "let me draft a letter to someone." This viewpoint helps when it comes to designing the user interfaces and overall experience you will build for your users.

It's always a good idea to sketch out your screen somewhere visually, and if you do, use software for it: ensure that it's something that lets you focus on layout and content rather than having to worry about colors, templates, or layout systems; always design first and then figure out how you'll build it. Pay attention to apps you enjoy using and that you find useful, look at how they do things--imitation is the sincerest form of flattery. Don't copy people too closely, but draw inspiration from good ideas; your users will thank you for it as well, because you'll be presenting them with something familiar, and hopefully more innovative at the same time.

Try to keep all text, colors, and dimensions as resources, and use generic names for these resources wherever possible. It's not uncommon to have an *ok* and *cancel* resource defined right under the application's name, because they are commonly used throughout applications. Keeping these values in the resource system allows for changes to be made far more easily, and keeps the application look and experience consistent for your users.

In the next chapter, we'll look at events, the Android event model, and how to best deal with events from your user interfaces in a way that provides the best user experience, while also being the more flexible to program with.

3
Taking Actions

Handling events is an essential part of any application; they are the raw input data for a user interface and how we interact with our users (rather than just presenting them with data). Android has an event model that will be instantly familiar to anyone who has programmed Java on their desktop--you attach listener objects to the widgets, and they deliver events to you.

Event listeners in Android take the form of interfaces that you need to implement in order to receive the events. Each possible event type is declared as a method on the relevant interface. To receive a notification that the user has *clicked* or *tapped* on a widget, you use the `OnClickListener` interface, which declares a method--`onClick(View)`--which will be invoked when the relevant widget receives what it considers a click gesture from the user.

In this chapter, we'll take a look at events on Android, and how best to implement them. Specifically, we'll be taking a closer look at the following:

- How Android dispatches events, and how it affects your program and user experience
- Different ways to implement event listeners and their pros and cons
- How to wrap groups of events into logical classes
- How to make events always happen quickly

Understanding Android event requirements

Android has a number of requirements around events delivered from the user interface that are important to be aware of, because they directly affect the user's experience and the perceived performance of your application. Android runs the **main** thread of an application as an event loop, rather than having a separate **event loop** or **event dispatcher** thread. This is an extremely important concept to understand, because this thread and event queue are shared between the following:

- All the events from the user interface
- The drawing requests from the widgets, where they paint themselves
- The layout system and all the calculations for positioning and sizing widget
- A variety of system-level events (such as network state changes)

This makes the *main* thread of the application a precious resource--every frame of an animation has to run through this event loop as a separate event, as does every layout pass, and every event from the user interface widget. On the other side of this contract, there are three other important factors to know and understand:

- All method calls to user interface elements must be done on the main thread
- No networking is allowed on the main thread
- Every event *slice* on the main thread is externally timed, and long-running events may cause your application to be terminated through an **Application Not Responding** dialog displayed to the user (which is as bad as crashing in most cases)

So, it's vital that we have models in place to avoid using the main thread more than is absolutely required. Every time you run something on the main thread, you're taking time away from vital systems such as graphics rendering and input events. This will cause your application to appear to stutter and become unresponsive. Fortunately, Android has many utilities to help, and there are a few extra steps that can be taken as a developer, which will reduce the complexity and help ensure the best possible user experience.

Listening for some events

When listening for user-interface events in Android, you'll typically hook up a listener object of some sort to the widgets you want to receive events on. However, how the listener object is defined may follow a number of different patterns, and listeners can take a number of different forms. You'll often see a simple anonymous class being defined as the listener, which is something like this:

```
closeButton.setOnClickListener(new View.OnClickListener() {
  @Override
  public void onClick(View v) {
    finish();
  }
});
```

However, while this pattern is common (especially because the much shorter lambda syntax was only introduced in Java 8, and Android didn't properly support it until 2017), it's not always your best choice for several reasons:

- This anonymous class is not reusable at all. It serves one purpose, for a single object in the entire application.
- You just allocated a new object that will also need to be garbage collected. This is not a big deal, but can sometimes be avoided or minimized by grouping listeners into classes that handle multiple related events.
- Any local variables from `onCreate` that are captured by the anonymous inner class must have references copied over into the new class as fields. You don't see this happen, but the compiler does it automatically (it's why the fields must be final).

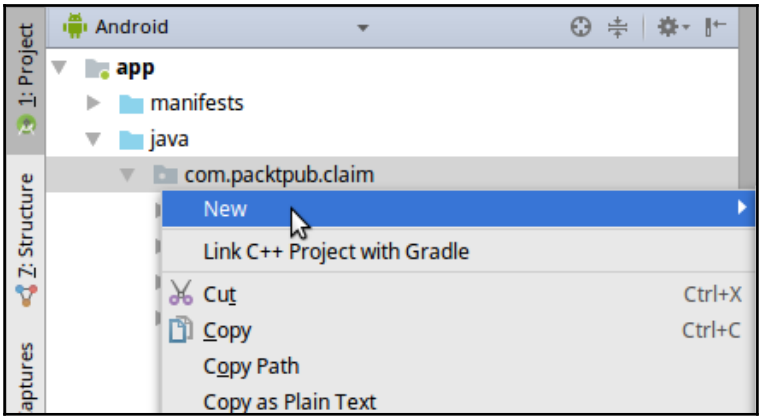
If Java 8 is available on your project, you can, of course, use lambdas, and shorten the syntax. However, this still results in an anonymous inner class being created. Another pattern for listening for events is to have the class that contains the layout (typically an `Activity` or `Fragment`) implement the listener interfaces, and use a `switch` statement to handle events from different widgets:

```
public class MyListenerActivity extends Activity implements
View.OnClickListener {
  @Override
  protected void onCreate(@Nullable Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.eventful_layout);

    findViewById(R.id.open).setOnClickListener(this);
    findViewById(R.id.find).setOnClickListener(this);
    findViewById(R.id.close).setOnClickListener(this);
  }

  // ...

  @Override
  public void onClick(View v) {
    switch (v.getId()){
      case R.id.open:
        onOpen();
        break;
      case R.id.find:
        onFind();
        break;
      case R.id.close:
        onClose();
        break;
    }
  }
}
```

This has two advantages: there are no new listener objects, and all the layout and event logic is now encapsulated within the `Activity` class. The `switch` statement carries a tiny overhead, but as the layout increases in size, it becomes a lot of boilerplate to maintain and somewhat encourages you to place simple event code directly into the `onClick` method, instead of always just dispatching to another method. This simple event code almost always leads to more complex event code, and eventually to a horrible mess in your code base.

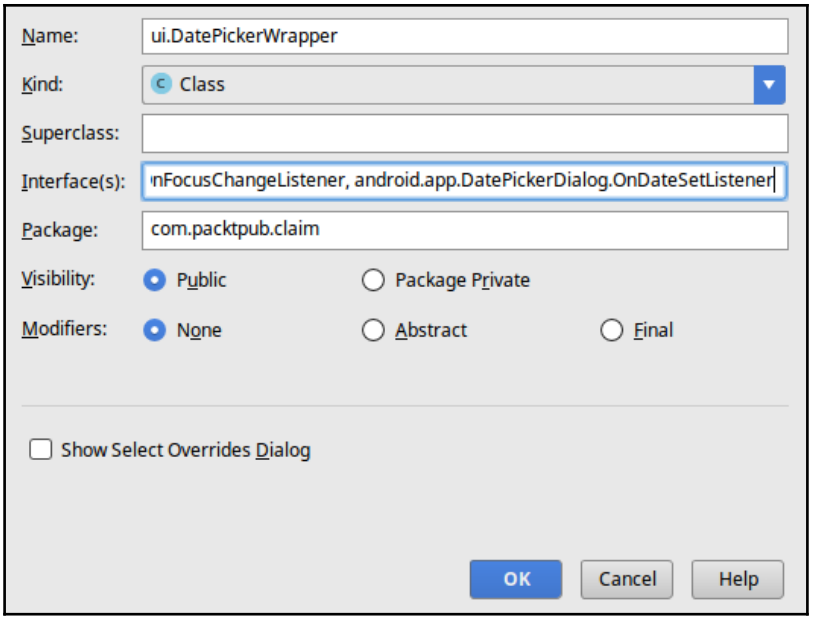
So, what is the right way to handle events? The answer is that there isn't one, but when deciding how to handle events, you should always consider how you will reuse the event handler code--don't repeat yourself. For the date selection widget from the last chapter, the expectation is that when the user taps on the date, they will see a calendar dialog open, allowing them to choose a new date. This will need an event handler, and such a handler should be reusable since you may want it elsewhere, so follow these steps to build the date-picker event listener:

1. Right-click on your default package (that is, `com.packtpub.claim`) and select **New** | **Java Class**:

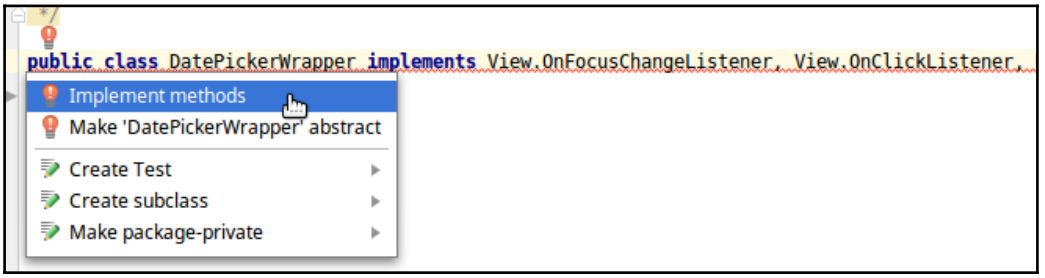


2. Name the new class `ui.DatePickerWrapper`; Android Studio will create a new package named `ui` automatically and place `DatePickerWrapper` inside it.
3. In the **Interfaces** list, add the following listener interfaces (use commas "," to separate the interfaces):
    - `android.view.View.OnClickListener`: To receive an event when the user taps on the date picker
    - `android.view.View.OnFocusChangeListener`: To receive an event if the date picker receives keyboard focus; this is important to handle if the user chooses to navigate the form using the "next" button on the keyboard, or has a physical keyboard attached to their device

- `android.app.DatePickerDialog.OnDateSetListener`: To receive an event when the user selects a new date from `DatePickerDialog`:



4. Click **OK** to create the new package and class.
5. If Android Studio has not created the skeleton methods for the listeners, select class name as `DatePickerWrapper` in the source, and use the code assistant to implement the methods:



6. Now you'll need a way to format the date string for the user, and it should be localized, so declare a `java.text.DateFormat` for this purpose:

```
private final DateFormat dateFormat =
DateFormat.getDateInstance(DateFormat.LONG);
```

7. This class is a wrapper, and will also need some fields to keep track of what it is wrapping, namely, the `TextView`, where it will display the date to the user (and where the user can tap to open the date picker dialog), an instance of `DatePickerDialog` to display to the user, and the currently selected/displayed `Date`:

```
private final TextView display;

private DatePickerDialog dialog = null;
private Date currentDate = null;
```

8. Then, we need a simple constructor that will capture `TextView` for display, and set it up as a date display and configure the events:

```
public DatePickerWrapper(final TextView display) {
  this.display = display;
  this.display.setFocusable(true);
  this.display.setClickable(true);
  this.display.setOnClickListener(this);
  this.display.setOnFocusChangeListener(this);

  this.setDate(new Date());
}
```

9. Now, we'll need getter and setter-like methods to change and retrieve the state of the date picker:

```
public void setDate(final Date date) {
  if(date == null) {
    throw new IllegalArgumentException("date may not be null");
  }

  this.currentDate = (Date) date.clone();
  this.display.setText(dateFormat.format(currentDate));

  if(this.dialog != null) {
    final GregorianCalendar calendar = new GregorianCalendar();
    calendar.setTime(currentDate);
    this.dialog.updateDate(
        calendar.get(Calendar.YEAR),
```

```
            calendar.get(Calendar.MONTH),
            calendar.get(Calendar.DAY_OF_MONTH)
        );
      }
    }

    public Date getDate() {
      return currentDate;
    }
```

10. Before we can actually handle the event, we need a method to display `DatePickerDialog` that will allow the user to change the date:

```
void openDatePickerDialog() {
  if (dialog == null) {
    final GregorianCalendar calendar = new GregorianCalendar();
    calendar.setTime(getDate());
    dialog = new DatePickerDialog(
        display.getContext(),
        this,
        calendar.get(Calendar.YEAR),
        calendar.get(Calendar.MONTH),
        calendar.get(Calendar.DAY_OF_MONTH)
    );
  }
  dialog.show();
}
```

11. Then, we need to complete the event listener methods so that when the user selects the displayed date, we open the `DatePickerDialog`, allowing them to change the selected date:

```
@Override
public void onClick(final View v) {
  openDatePickerDialog();
}

@Override
public void onFocusChange(final View v, final boolean hasFocus) {
  if (hasFocus) {
    openDatePickerDialog();
  }
}
```

12. Finally, we need to handle the event that comes back from the `DatePickerDialog`, indicating that the user has chosen a date:

```
@Override
public void onDateSet(
      final DatePicker view,
      final int year,
      final int month,
      final int dayOfMonth) {

  final Calendar calendar = new GregorianCalendar(
      year, month, dayOfMonth
  );

  setDate(calendar.getTime());
}
```

Now you have a class that can turn any `TextView` object into a space where the user can select a date via the standard `DatePickerDialog`. This is an ideal example of a good place to encapsulate events; you actually have three different event handlers that perform a related group of actions, and maintain user interface state in a single class that can be reused throughout your application.

Wiring the CaptureClaimActivity events

Now that we have a way for the user to pick a date for their travel expense claims, we need to actually wire it into the `CaptureClaimActivity`, which is where all the logic and wiring for the screen will live. To start wiring the events for the `CaptureClaimActivity`, follow these steps:

1. Open the `CaptureClaimActivity.java` file in Android Studio.
2. Now, declare a new field in the class (before the `onCreate` method) for the `DatePickerWrapper` that you wrote (Android Studio can help by writing the import for you):

```
private DatePickerWrapper selectedDate;
```

3. You'll note that (by default) the `FloatingActionButton` object is wired up with a simple anonymous event handler that will look something like this:

```
fab.setOnClickListener(new View.OnClickListener() {
  @Override
  public void onClick(View view) {
    Snackbar.make(
        view,
        "Replace with your own action",
        Snackbar.LENGTH_LONG
    ).setAction("Action", null).show();
  }
});
```

4. This is how many once off events get wired up (as discussed earlier in this chapter), but it's not what we want to do here, so remove that entire block of code.
5. At the end of the `onCreate` method instantiate the `DatePickerWrapper` object by searching for the `date TextView` you added to the layout:

```
selectedDate = new DatePickerWrapper((TextView)
findViewById(R.id.date));
```

6. You don't need to hold any other references to the `date TextView`, since you'll only ever need to access it through the `DatePickerWrapper` class. Try running your application now, and see how the date picker works.

In the application, you'll note that you can select the category icons, and they will work exactly as expected. However, the label following them isn't wired up at all, and no labels will be displayed, leaving the user confused as to what they are actually selecting. To fix this, you'll need another event listener that sets the content of the label when the state of the `RadioButton` widgets is changed. This is another case where a specialized listener class will make sense; since it'll be usable anytime, you have a group of icon `RadioButton` widgets and a single label for all of them:

1. Right-click on the `ui` package and select **New | Java Class**.
2. Name the new class as `IconPickerWrapper`.
3. Add `android.widget.RadioGroup.OnCheckedChangeListener` to the interfaces box.

4. Create a field for the `TextView` label, and a constructor to capture it:

```
private final TextView label;

public IconPickerWrapper(final TextView label) {
  this.label = label;
}
```

5. Add a method to set the label text content:

```
public void setLabelText(final CharSequence text) {
  label.setText(text);
}
```

6. Complete the `onCheckedChange` method to set the label text from the `contentDescription` field of the selected `RadioButton`:

```
@Override
public void onCheckedChanged(
    final RadioGroup group,
    final int checkedId) {

  final View selected = group.findViewById(checkedId);
  setLabelText(view.getContentDescription());
}
```

This is a very straightforward class, but it also potentially serves other purposes in your application, and it only makes two assumptions about the `RadioGroup` it will be connected to:

- Every `RadioButton` has a valid ID
- Every `RadioButton` has a `contentDescription` that will serve as a text label

Going back to `CaptureClaimActivity`, you'll want to wire this new listener into the layout through the following steps:

1. Before the `onCreate` method, create a new field to keep track of the `RadioGroup`, where the user can select the category icon:

```
private RadioGroup categories;
```

2. Then, at the end of the `onCreate` method, you'll need to find this `RadioGroup` in the layout, and instantiate its event handler:

```
categories = (RadioGroup) findViewById(R.id.categories);
categories.setOnCheckedChangeListener(
  new IconPickerWrapper(
      (TextView) findViewById(R.id.selected_category)
  )
);
```

3. Finally, set the default selection to `other`; this action will also trigger the event handler before the screen is presented to the user. This means the label will also be populated when the user first sees the **Capture Claim** screen:

```
categories.check(R.id.other);
```

Now if you run the application again, you'll see the labels defined to appear beneath the selected icons as you toggle through the category icons.

Handling events from other activities

On Android, you'll often find that you want to send your user to another `Activity` to do something, and then return them to your current `Activity` with the result of that action. Good examples are having the user pick a contact, or take a photo with the camera app. In these cases, Android uses a system of special events that are built into the `Activity` class. For capturing travel expense claims, your user needs to be able to go select a file to attach things such as photos or email attachments to their claim.

In order to present them with a familiar file chooser (and avoid writing a file chooser ourselves), you'll need to use this mechanism. However, to read files from outside of your application's private space, you'll need it to ask the user for permissions. Anytime an application needs access to potentially sensitive data (public directories, the device's camera or microphones, contact list, and so on), you need permission from the user. In versions of Android prior to 6.0, this was done during installation; the application declared what permissions it needed, and the user could choose to not install it. However, this mechanism isn't very flexible to users, and was changed in 6.0 so that applications must now ask for permission at runtime.

In order to access the user's files, the application will both declare that it requires the permission, and will also include the code to ask for permission while it's running (covering both cases):

1. Open the `CaptureClaimActivity` class, and make the class implement the `View.OnClickListener` interface:

```
public class CaptureClaimActivity extends AppCompatActivity
                                    implements View.OnClickListener {
```

2. Create two new constants to hold the request codes. Anytime your user leaves your current `Activity`, and you are expecting a result, you need a request code:

```
private static final int REQUEST_ATTACH_FILE = 1;
private static final int REQUEST_ATTACH_PERMISSION = 1001;
```

3. In the `onCreate` method, find the line where the Android Studio template captures the `FloatingActionButton`:

```
FloatingActionButton fab = (FloatingActionButton)
findViewById(R.id.fab);
```

4. Rename the button to `attach`, as follows (use the Android Studio refactoring to change the ID, and the ID in the layout file will be changed as well):

```
FloatingActionButton attach = (FloatingActionButton)
findViewById(R.id.attach);
```

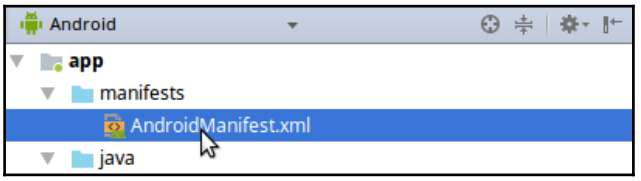
5. Now, set `OnClickListener` for the `FloatingActionButton` to `Activity`:

```
attach.setOnClickListener(this);
```

6. Now, at the end of the `CaptureClaimActivity`, implement the `onClick` method, and delegate the click from the `FloatingActionButton`:

```
@Override
public void onClick(View v) {
  switch (v.getId()){
    case R.id.attach:
      onAttachClick();
      break;
  }
}
```

7. Your application will need permission to read the content from outside its own private space. Open the `manifests` folder in the file browser, and open the `AndroidManifest.xml` file:



8. At the top of the file within the `manifest` element, but before the application element, add the following permission declaration:

```
<manifest
xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.packtpub.claim">

    <uses-permission
        android:name="android.permission.READ_EXTERNAL_STORAGE"
        android:maxSdkVersion="23" />

    <application
        android:name=".ClaimApplication"
```

9. The preceding permission only works for versions of Android where permissions were requested during installation; on Android 6.0 and newer, you need to check and request permissions at runtime. Doing this when the user taps on the `FloatingActionButton` to attach a file is the best time, as this is just before they actually choose a file that you will need permission to read. Implement the `onAttachClick` method, starting with a check for the permission, and request the permission if it's not already granted:

```
public void onAttachClick() {
  final int permissionStatus = ContextCompat.checkSelfPermission(
    this,
    Manifest.permission.READ_EXTERNAL_STORAGE);

  if (permissionStatus != PackageManager.PERMISSION_GRANTED) {
    ActivityCompat.requestPermissions(
      this,
      new String[]{Manifest.permission.READ_EXTERNAL_STORAGE},
      REQUEST_ATTACH_PERMISSION);
    return;
  }
```

10. Now the app can request that the system start an `Activity`, allowing the user to choose any openable file. This is where the `REQUEST_ATTACH_FILE` constant you defined earlier starts getting used:

```
    final Intent attach = new Intent(Intent.ACTION_GET_CONTENT)
          .addCategory(Intent.CATEGORY_OPENABLE)
          .setType("*/*");

    startActivityForResult(attach, REQUEST_ATTACH_FILE);
  }
```

11. If we failed the preceding permission check, the system will have launched a dialog asking whether the user would grant permission to access external files. When the user returns from this dialog, a method named `onRequestPermissionsResult` will be invoked. Here, you need to check whether they granted your request, and if so, you can simply trigger the `onAttachClick()` method to continue the process smoothly:

```
  @Override
  public void onRequestPermissionsResult(
        final int requestCode,
        final String[] permissions,
        final int[] grantResults) {

    switch (requestCode) {
      case REQUEST_ATTACH_PERMISSION:
        if (grantResults[0] == PackageManager.PERMISSION_GRANTED) {
          onAttachClick();
        }
        break;
    }
  }
```

12. Now when the system returns from the file chooser `Activity`, it will invoke a method called `onActivityResult`, which is very similar in structure to the `onRequestPermissionResult` method:

```
  @Override
  protected void onActivityResult(
        final int requestCode,
        final int resultCode,
        final Intent data) {

    switch (requestCode) {
      case REQUEST_ATTACH_FILE:
        onAttachFileResult(resultCode, data);
```

```
            break;
        }
    }
```

13. In the preceding `onActivityResult`, you simply check whether it's responding to your request to attach a file, and then delegate the rest to a method that will need to handle the resulting data:

```
public void onAttachFileResult(
    final int resultCode, final Intent data) {
```

14. Verify that the `resultCode` was okay and that the data is valid:

```
if (resultCode != RESULT_OK
    || data == null
    || data.getData() == null) {
  return;
}
```

15. For now, you'll just want a `Toast` to pop up showing that this code has run; later, you can build the complete logic to attach the selected file. A `Toast` is a small message that appears and then disappears with no user interaction, perfect for temporary messages or debugging:

```
Toast.makeText(this, data.getDataString(),
Toast.LENGTH_SHORT).show();
```

Now, if you run the application and tap on the floating action *attach* button, you'll be rewarded with a permission request (if you're running Android 6 or higher, on earlier versions the permission is granted as part of the installation), and then the option to select a file with whatever file selection systems you might have available on your emulator or device. Once you have chosen a file, you'll be returned to the `CaptureClaimActivity` and the selected `Uri` will be displayed in a `Toast` message on the screen:

This might not look like much, but it's all you need to access the file later on and attach it to the claim that the user is capturing. When you need to send your user to another Activity, you're hooking into Android's `Activity` to `Activity` messaging systems through methods such as `onActivityResult` and `onRequestPermissionsResult`.

Making events quick

Android places very strict limits on the use of threads in applications: every application has a main thread, where all user-interface related code must run, but any long-running code will cause an error. Any attempt at networking on the main thread will result in a `NetworkOnMainThreadException` immediately, as networking by its very nature will block the main thread for too long, making the application unresponsive.

This means most tasks that you will want to perform should take place on a background worker thread. This will also provide you with a form of isolation from the user interface, as typically you will capture the user interface state on the main thread, pass the state to the background thread, process the event and then, send the result back to the main thread where you will update the user interface. How do we know that the state we capture will be consistent? The answer is that because user interface code can only run on the main thread, while you read the state of the widgets, any events that would change their state are blocked until you are finished (because they must also occur on the main thread). The message queue and threading rules avoid the need for locks and other thread protection mechanisms by ensuring that only one unit of code (in the form of a message) is processed at a time.

Android tasks that require larger amounts of background processing time are usually written using the `AsyncTask` class provided by the Android platform (or one of its child classes). `AsyncTask` has methods for running code on a background worker, and publishing status updates to the main thread (and receiving these update messages), along with several other utility structures. This makes it ideally suited to tasks such as downloading large files, where the user needs to be kept informed of the download's progress. However, most event handlers you will implement won't need anywhere near to this level of complexity.

Most event handlers are relatively lightweight, but that doesn't mean that it will perform quickly on all devices in all situations. You can't control what else the user is busy doing with their device, and a simple database query can end up taking much longer than expected. As such, it's better to push event processing to background threads wherever the event is not purely a user-interface update (that is, showing a dialog or similar). Even fairly small tasks should be moved to a background thread so that the main thread can continue consuming the user's input; this will keep your application responsive. Here's the pattern you should try and follow when implementing event handlers:

- **On the Main Thread**: First, capture any required parameters
- **On a Background Worker**: Process the user's event and data
- **On the Main Thread**: End by updating the user interface with the new state

If you keep to this pattern, the application will always appear responsive to your users, since processing their data isn't stopping their events from being processed (events which may be them scrolling through a large list, for example). However, `AsyncTask` is not a great fit for these smaller events (such as attaching a file to a claim), so here's how to write a simple class (in a similar style to the command pattern) that will run first some code on the background and then pass the result of that code to another method on the main thread, perfect for carrying out smaller events:

1. Right-click on your root package (that is, `com.packtpub.claim`) and choose **New| Java Class**.
2. Name the class `util.ActionCommand`.
3. Change the **Modifiers** to make the new class `Abstract`.
4. Click **OK** to create the new package (`util`) and class.
5. Change the class definition to include generic parameters for a "parameter" and a "returned" type:

```
public abstract class ActionCommand<P, R> {
```

6. At the top of the new class, create a static constant that refers to the application main thread via an `android.os.Handler` object:

```
private static final Handler MAIN_HANDLER = new
Handler(Looper.getMainLooper());
```

A `Handler` object is how you gain access to another thread's message-queue in Android. In this case, any message or `Runnable` object posted to this `Handler` will be run on the main thread as soon as possible. You can also post tasks to be run at specific times or after a specified delay. This is the preferred method of creating timers on Android.

7. Create three method declarations for running code on the background worker, the main thread, and one for handling errors (with a default implementation):

```
public abstract R onBackground(final P value) throws Exception;
public abstract void onForeground(final R value);

public void onError(final Exception error) {
  Log.e(
      getClass().getSimpleName(),
      "Error while processing data",
      error
  );
}
```

8. Then, create two variations of an exec method that will be used to start the `ActionCommand` objects. The first one uses the standard `Executor` provided by `AsyncTask` that uses a single background thread to process tasks (this is the most common behavior you will want in an application):

```
public void exec(final P parameter) {
   exec(parameter, AsyncTask.SERIAL_EXECUTOR);
}

public void exec(final P parameter, final Executor background) {
   background.execute(new ActionCommandRunner(parameter, this));
}
```

9. In the preceding method, we submit an `ActionCommandRunner` object to the background `Executor` object; this is a `private` inner class that will carry the state between the background and main thread, which keeps the `ActionCommand` classes reusable and stateless:

```
private static class ActionCommandRunner implements Runnable {
```

10. `ActionCommandRunner` will be in one of the three possible states: background, foreground, or error. Declare three constants as names, and a field to keep track of which state the object is in:

```
private static final int STATE_BACKGROUND = 1;
private static final int STATE_FOREGROUND = 2;
private static final int STATE_ERROR = 3;
private int state = STATE_BACKGROUND;
```

11. Then, you'll need fields for the `ActionCommand` being run, and the current value. The `value` field is a catch-all in this class holding either the input parameter, the output from the background code, or the `Exception` thrown from the background code:

```
private final ActionCommand command;
private Object value;

ActionCommandRunner(
        final Object value,
        final ActionCommand command) {

   this.value = value;
   this.command = command;
}
```

12. Now, create methods to handle each of the `ActionCommandRunner` states:

```
void onBackground() {
   try {
       // our current "value" is the commands parameter
       this.value = command.onBackground(value);
       this.state = STATE_FOREGROUND;
   } catch (final Exception error) {
       this.value = error;
       this.state = STATE_ERROR;
   } finally {
       MAIN_HANDLER.post(this);
   }
}

void onForeground() {
   try {
       command.onForeground(value);
   } catch (final Exception error) {
       this.value = error;
       this.state = STATE_ERROR;
```

```
            // we go into an error state, and foreground to deliver it
            MAIN_HANDLER.post(this);
        }
    }

    void onError() {
        command.onError((Exception) value);
    }
```

13. Finally, create the `run` method that will call the preceding `onBackground`, `onForeground` or `onError` method depending on the current execution state of `ActionCommandRunner`:

```
    @Override
    public void run() {
        switch (state) {
            case STATE_BACKGROUND:
                onBackground();
                break;
            case STATE_FOREGROUND:
                onForeground();
                break;
            case STATE_ERROR:
                onError();
                break;
        }
    }
```

This class makes it very easy to create and reuse small tasks, which can be extended, composed, mocked, and tested in isolation. It's a good idea whenever creating a new event handler to consider a command pattern or something similar so that the event isn't coupled to the widget or even the screen that you are busy with. This allows for better code reuse, and keeps code easier to test since you can test the event handler without the screen that it will be part of later. You can also make these classes even more modular by writing them as `abstract` classes with only their `onBackground` methods implemented, allowing the result to be processed in different ways by subclasses.

Multiple event listeners

Unlike many other event systems, however, many Android components only allow a single event listener of certain types; this diverges from platforms such as Java desktop, or JavaScript in the browser, where any number of **click** listeners can be attached to a single element. In Android, click listeners are almost always **set** rather than **added**.

This is actually a clever tradeoff--having multiple listeners for each event means that you need at least an array of them; the array needs to be sized and copied when it runs out of space, while it's actually very seldom that multiple listeners are needed. Multiple listeners also means that the widgets must traverse the list every time they want to dispatch events, so sticking to a single listener simplifies the code, and reduces the amount of required memory.

If you ever find yourself needing more than one listener for an event and widget that only provides a single listener slot, simply write a simple delegate class, like this:

```
public class MultiOnClickListener implements View.OnClickListener {
  private final List<View.OnClickListener> listeners =
      new CopyOnWriteArrayList<>();

  public MultiOnClickListener(
      final View.OnClickListener... listeners) {
    this.listeners.addAll(Arrays.asList(listeners));
  }

  @Override
  public void onClick(View v) {
    for (final View.OnClickListener listener : listeners)
      listener.onClick(v);
  }

  public void addOnClickListener(
      final View.OnClickListener listener) {
    if (listener == null) return;
      listeners.add(listener);
  }

  public void removeOnClickListener(
      final View.OnClickListener listener) {
    if (listener == null) return;
    listeners.remove(listener);
  }
}
```

The preceding pattern allows compact and flexible multilistener delegation in the cases where you might need it. The `CopyOnWriteArrayList` class is an ideal listener container, as its internal array is only ever as large as the number of elements, so it remains compact (rather than having a buffer space like `ArrayList` and similar implementations).

Test your knowledge

1. What's the best way to implement event handlers?
    - As an anonymous inner class
    - By making the `Activity` a listener
    - As a class per listener
    - There isn't one

2. What are the conditions for any methods that change the state of a user-interface widget?
    - They must be called from a background thread
    - They must be thread-safe
    - They must be called from the main thread
    - They must be called from the graphics thread

3. Code running as part of an event handler should fulfill which of the following conditions?
    - Be surrounded by a synchronized block
    - Run as quickly as possible
    - Only interact with the user interface

4. When requesting data from another `Activity`, the data is returned through which of these?
    - An event listener you add to the `Activity` object
    - A callback on your `Activity` object
    - A message placed on your application's message queue

Summary

Android uses several different mechanisms when delivering events within its applications, each one tailored to the type of event being delivered and the intended recipient of the event. Most user-interface events are delivered to a single listener registered to each widget, but that doesn't stop the same listener from handling multiple event types from different widgets. This sort of design will reduce the load on the system and the amount of memory used, and will often help in producing more reusable code.

Event handlers are often written badly and become anonymous inner classes that, while starting life as a simple delegate to another method, eventually become bloated and unmaintainable blocks of code. It's often better to isolate the event handlers from their environment from the beginning as this encourages them to be reused, and makes them easier to test and maintain. Some event handler classes (such as `DatePickerWrapper`) handle different types of events in a way that is related, allowing for a single class to encapsulate a small, reusable bit of logic.

In the next chapter, we'll explore more of how to build reusable and more easily testable user interfaces, by breaking down user interfaces into smaller components.

4

Composing User Interfaces

Mobile apps seem like simple systems, but they are often actually quite deep and complex systems, with many different parts that help them keep the appearance of being simple. The user interfaces of applications are the same; they may appear simple, but they are often complex arrangements of screens and dialogs designed to hide the complexities of the application from the user and provide a smoother experience. The easiest way to think about this is that traditional desktop applications and websites tend to be *wide*, while mobile applications tend to be *deep*.

This comment applies (on the surface at least) to the navigation of applications. Desktop applications tend to have a central *control* area where most of the work is done. Think of a document editor--the application centers around the document being written, and you never really *leave* that area. Instead of navigating away, dialogs pop up to fulfill a single task, which alters the document before they disappear. There are many variations on this theme, but desktop applications tend to follow the same pattern.

Mobile applications, on the other hand, tend to start at an *overview* screen of some sort, or launch directly into an *action* screen. The user then navigates *downward* into a task or item before either returning to the overview screen, or completing their task and being presented with a *result* of some sort (for example, booking a flight). Achieving a goal by navigating away from the overview screen, rather than simply opening a dialog on top of the content, requires a different approach to your application's design. As there is no central screen always available to the user, you often need to remind them of where they are, and what they are doing. This sort of repetition will be unthinkable on a desktop application where the information is always available in another window or panel of the application; however, on the limited space offered by a mobile phone, this becomes an essential tool to keep the user on track, and help them complete the task they are trying to fulfill.

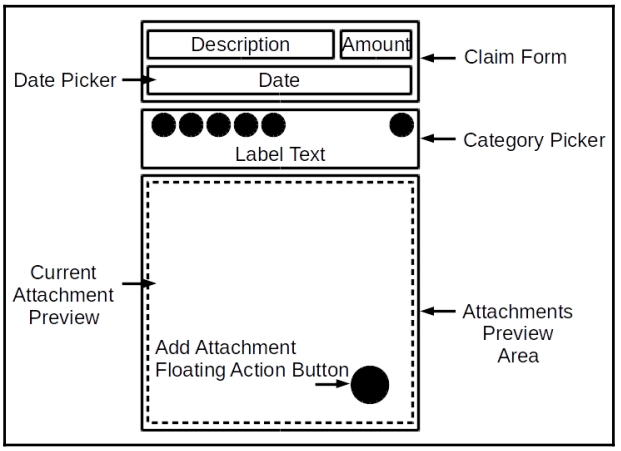
This change in how the user navigates, requires that screens often present the same data, or have elements that are repeated all over the application. These elements are best encapsulated so that they can be easily reused throughout the application. Android offers the perfect way to encapsulate these groups of widgets in a way that can be more aware than a simple layout component--fragments. A **Fragment** is like a *miniature* `Activity`; each Fragment has a full life cycle, just like `Activity`, except that they are always contained within an `Activity` (refer to `Appendix A` for details of the `Activity` life cycle). Using fragments allows your application to more easily adapt to various screen sizes. We'll be taking a closer look at fragments later in this chapter.

In this chapter, we'll look at various ways to break up a user interface, building modules that can be layered and reused to form complex behaviors without requiring complex code and wiring. We'll be looking at the following:

- How to build custom groups of widgets that can be directly included in layouts
- How to build Fragments that expose common functionality with a complete life cycle of their own
- How to use ViewPager to display pages or tabs of widgets

Designing a modular layout

So far, you've built a single `Activity` class with a layout composed out of two layout files containing widgets. This is a pretty normal state of affairs, but it's not the best situation. Like in most user interfaces, the claim capturing screen can be divided into a range of very logical areas:



In an Android user interface, you always have the individual widgets (such as `Button`, `TextView`, `ImageView`, and friends) at the bottom level, and `Activity` at the top level, but when you look at that screen mockup, you can instantly see that the screen can be divided into other layers in between `Activity` and widget. You can, of course, take each of the `CardView` layouts from this screen and place them in their own layout XML files, and then import them:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- card_claim_capture_info.xml -->
<android.support.v7.widget.CardView
   xmlns:android="http://schemas.android.com/apk/res/android"
   xmlns:app="http://schemas.android.com/apk/res-auto"
   android:layout_width="match_parent"
   android:layout_height="wrap_content">

  <android.support.constraint.ConstraintLayout
       android:layout_width="match_parent"
       android:layout_height="match_parent"
       android:layout_margin="@dimen/grid_spacer1">
       <android.support.design.widget.TextInputLayout
           android:id="@+id/description_layout"
           android:layout_width="0dp"
           android:layout_height="wrap_content"
           android:layout_marginEnd="8dp"
           app:layout_constraintEnd_toStartOf="@+id/amount_layout"
           app:layout_constraintStart_toStartOf="parent"
           app:layout_constraintTop_toTopOf="parent">
```

You can use the `<include>` element to include one layout file into another, like this:

```
<!-- content_capture_claim.xml -->
<include layout="@layout/card_claim_capture_info"/>
```

This nicely separates out the layout for this part of the screen, allowing you to reuse it in layouts for larger physical screens or even in other `Activity` classes in the application. The problem is that every screen that wants to use this layout will need to also duplicate all the logic associated with it. While the logical decoupling of the logic and the layout is mostly a good thing (especially when you can overlay the logic on multiple different layouts), their coupling is normally quite tight.

Creating the DatePickerLayout

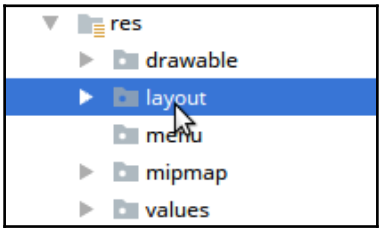
Each of these areas can easily be encapsulated in a Java class and reused elsewhere in your application. In `Chapter 3`, *Taking Actions*, you wrote the `DatePickerWrapper` class, which can turn any `TextView` widget into a date selection widget. However, `DatePickerWrapper` doesn't create the `TextView` label or change the styling of the widgets to look like `TextInputLayout`. This means that you need to copy that styling into each layout where you want a date-picker, which can quickly lead to inconsistencies in your user interface. While it's good to have the events and state decoupled from the display logic, it would also be nice to have them grouped together in a single structure that can be reused without every layout having to specify the date picker widgets by hand, and then bind them to the `DatePickerWrapper` in its code.

While it's not obvious at first, the Android layout XML files can reference any `View` class and not just those defined in the core and support packages, and it can do so without any special tricks. All you need to do is reference the `View` class by its fully-qualified name, much like you've already done for several widgets:

- `android.support.constraint.ConstraintLayout`
- `android.support.v7.widget.CardView`
- `android.support.design.widget.TextInputLayout`

All the preceding are classes that you can look up in Android Studio, and even read their code if you like. Let's get started by writing a `DatePickerLayout` to couple the layout XML with the `DatePickerWrapper`, and make the date picker reusable from any layout XML file in your application:

1. In the Android panel of Android Studio, right-click on the **layout** directory under **res**:



2. Select **New** | **Layout resource file**.
3. Name the new layout file as `widget_date_picker`.
4. Change the **Root element** field to `merge`:

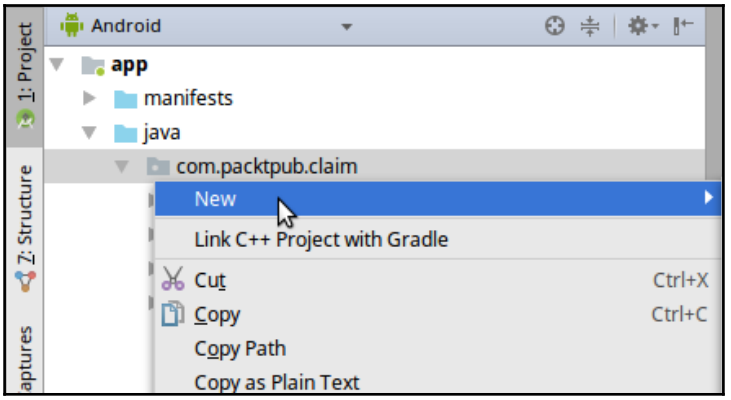
`merge` is a special root element for layout files. Normally, the root element of a layout file is a `View` class that results in the file having a root widget when it's inflated. The `merge` element doesn't create a root widget; instead, it's effectively skipped when the file is loaded, and its children are inflated directly. This makes it ideal for creating layout widgets, or reusable bits of layout, while also keeping the layout hierarchy flat and helping improve your application's performance.

5. Change the editor mode to **Text** instead of **Design**.
6. Remove the `layout_width` and `layout_height` attributes from the `merge` element.
7. Write the following two `TextView` widgets into the merge element:

```
<?xml version="1.0" encoding="utf-8"?>
<merge xmlns:android="http://schemas.android.com/apk/res/android">
   <TextView
       android:layout_width="match_parent"
       android:layout_height="wrap_content"
       android:text="@string/label_date"
android:textAppearance="@style/TextAppearance.AppCompat.Caption"
       android:textColor="@color/colorAccent" />
   <TextView
       style="@style/Widget.AppCompat.EditText"
       android:layout_width="match_parent"
       android:layout_height="wrap_content"
       android:layout_marginTop="@dimen/grid_spacer1" />
</merge>
```

In the preceding file, there are no `id` attributes set; this is because any layout the new widget is used in will be polluted with those `id` attributes, and `findViewById` can easily start returning unexpected results. When encapsulating parts of your layout, always consider what `id` values will appear in the layout, and where they might be used in code. `findViewById` simply finds the *first* matching `View` object in the layout and returns it, and doesn't consider where that `View` might have come from (that is: an `<include>`, or special `View` class).

8. In the **Android** panel of Android Studio, right-click on your `base` package (that is, `com.packtpub.claim`):



9. Select **New** and **Java Class**.
10. Name the new class `widget.DatePickerLayout`.
11. Change the **Superclass** to `android.widget.LinearLayout`.
12. Click **OK** to create the new package and class.
13. Declare fields in the `DatePickerLayout` for to reference a `TextView` label, and the `DatePickerWrapper` :

```
private TextView label;
private DatePickerWrapper wrapper;
```

14. Any class accessible from a layout XML requires several constructor overloads, so it's best to create a single `initialize` method that can be reused for all of them:

```
void initialize(final Context context) {
   setOrientation(VERTICAL);
}
```

15. Still within the `initialize` method, use `LayoutInflator` to load the layout XML file you wrote, adding its contents as elements to the `DatePickerLayout` object:

```
LayoutInflater.from(context).inflate(
  R.layout.widget_date_picker, this, true);
```

The parameters for the inflate method are the layout resource, the `ViewGroup` (in this case, `DatePickerLayout`) that will contain the layout, and whether or not to actually attach the elements of the layout resource to the `ViewGroup`. As you are using a merge element in the layout resource, the third parameter must be `true`, because otherwise the contents of the layout will be lost.

16. Use `getChildAt` to retrieve the new `TextView` elements that have been loaded by the `LayoutInflator`, and assign the fields of the `DatePickerLayout`:

```
label = (TextView) getChildAt(0);
wrapper = new DatePickerWrapper((TextView) getChildAt(1));
```

17. Overload the constructors and invoke the `initialize` method in each of them:

```
public DatePickerLayout(Context context) {
  super(context);
  initialize(context);
}

public DatePickerLayout(Context context, AttributeSet attrs) {
  super(context, attrs);
  initialize(context);
}

public DatePickerLayout(
    Context context, AttributeSet attrs, int defStyleAttr) {
  super(context, attrs, defStyleAttr);
  initialize(context);
}
```

18. Create getters and setters to make the `DatePickerLayout` usable from the `Activity` classes:

```
public void setDate(final Date date) {
   wrapper.setDate(date);
}

public Date getDate() {
   return wrapper.getDate();
}

public void setLabel(final CharSequence text) {
   label.setText(text);
}

public void setLabel(final int resid) {
   label.setText(resid);
}

public CharSequence getLabel() {
   return label.getText();
}
```

19. As the `DatePickerLayout` contains some of the user-interface state (the currently selected date), it's expected to keep track of this through possible `Activity` restarts, if required (an `Activity` is recreated every time the user changes between portrait and landscape, as these are considered *configuration* changes). This will involve saving its state to a `Parcel`, and restoring it from a `Parcel` when requested to (a `Parcel` is a bit like a `byte[]` of `Serialized` objects, except that all the marshaling work needs to be implemented). You'll need an inner class that can hold the state of the `DatePickerLayout` (and it's parent class--`LinearLayout`). For convenience, the `View` class provides a `BaseSavedState` abstract class to take care of some of the implementation for you, so extend `BaseSavedState` in a static inner class named `SavedState`:

```
private static class SavedState extends BaseSavedState {
   final long timestamp;
   final CharSequence label;

   public SavedState(
           final Parcelable superState,
           final long timestamp,
           final CharSequence label) {

       super(superState);
```

```
        this.timestamp = timestamp;
        this.label = label;
    }
}
```

Objects crossing between `Activity` instances need to be `Parcelable`, because Android may need to store the objects temporarily through the `Activity` life cycle. Being able to store just the important bits of data and state, instead of the entire widget tree, is very useful for conserving memory when the user has a lot of applications running. `BaseSavedState` implements `Parcelable` and will allow the `DatePickerLayout` to remember its state when the `Activity` is destroyed and recreated by the system.

20. `SavedState` will also need a constructor to load its fields from a `Parcel` object; a `CharSequence` cannot be read directly from a `Parcel`, but fortunately, `TextUtils` has a nice helper for reading `CharSequence` objects from `Parcel` objects for you:

```
SavedState(final Parcel in) {
  super(in);
  this.timestamp = in.readLong();
  this.label = TextUtils.CHAR_SEQUENCE_CREATOR
      .createFromParcel(in);
}
```

21. Then, `SavedState` needs the `writeToParcel` method implemented in order to actually write those fields to a `Parcel`; part of this is delegated to the `BaseSavedState` class:

```
@Override
public void writeToParcel(final Parcel out, final int flags) {
   super.writeToParcel(out, flags);
   out.writeLong(timestamp);
   TextUtils.writeToParcel(label, out, flags);
}
```

22. Every `Parcelable` implementation requires a special `public static final` field called `CREATOR`, which will be used by the `Parcel` system to create instances and arrays of the `Parcelable` objects. This also applies for every subclass, so write the following static final into the `SavedState` class:

```
public static final Parcelable.Creator<SavedState> CREATOR =
      new Parcelable.Creator<SavedState>() {
  @Override
```

```
  public SavedState createFromParcel(final Parcel source) {
    return new SavedState(source);
  }

  @Override
  public SavedState[] newArray(int size) {
    return new SavedState[size];
  }
};
```

When implementing a vanilla `Parcelable` class, Android Studio has a nice generator that can be triggered from the class declaration hints (look for "**Add Parcelable Implementation**") that will write a simple `writeToParcel` method, `Parcel` handling constructor, and the `CREATOR` field. Check whether it's working though; it skips any field it doesn't know how to handle.

23. In the `DatePickerLayout` class, you need to override the `onSaveInstanceState` method and create the `SavedState` object that will be recorded:

```
@Override
protected Parcelable onSaveInstanceState() {
  return new SavedState(
      super.onSaveInstanceState(),
      getDate().getTime(), getLabel());
}
```

24. You'll also need to restore the state from a `SavedState` object, which requires overriding `onRestoreInstanceState`:

```
@Override
protected void onRestoreInstanceState(final Parcelable state) {
   final SavedState savedState = (SavedState) state;
   super.onRestoreInstanceState(savedState.getSuperState());
   setDate(new Date(savedState.timestamp));
   setLabel(savedState.label);
}
```

25. Open the `content_capture_claim.xml` layout file in Android Studio.
26. Change to the **Text** editor, if required.
27. Find the two `TextView` elements that describe the date-picker, and replace them with the following snippet:

```
<com.packtpub.claim.widget.DatePickerLayout
   android:id="@+id/date"
   android:layout_width="0dp"
   android:layout_height="wrap_content"
   android:layout_marginTop="8dp"
   app:layout_constraintEnd_toEndOf="parent"
   app:layout_constraintStart_toStartOf="parent"
   app:layout_constraintTop_toBottomOf="@+id/description_layout" />
```

28. Open the `CaptureClaimActivity` class in Android Studio.
29. Replace the references to `DatePickerWrapper` with `DatePickerLayout`:

```
private DatePickerLayout selectedDate;

@Override
protected void onCreate(Bundle savedInstanceState) {
  // ...
  selectedDate = new DatePickerWrapper(    // remove this line
      (TextView) findViewById(R.id.date)); // remove this line
  selectedDate = (DatePickerLayout) findViewById(R.id.date);
  // ...
}
```

This new `DatePickerLayout` class allows you to reuse the same label and editor in any layout XML file in your application, while also coupling the required events in a single class. Any time you have a layout with `TextViewLayout` widgets, the new `DatePickerLayout` will fit right into the style and allow for safe date selection. It's also very important to implement the `onSaveInstanceState`/`onRestoreInstanceState` method on `View` subclasses if you intend to carry any state. These classes are marshaled, and new instances of the `View` are created every time a configuration state changes, which includes actions such as the user rotating the device (refer to `Appendix A` for more information on `Activity` life cycle).

Creating the data model

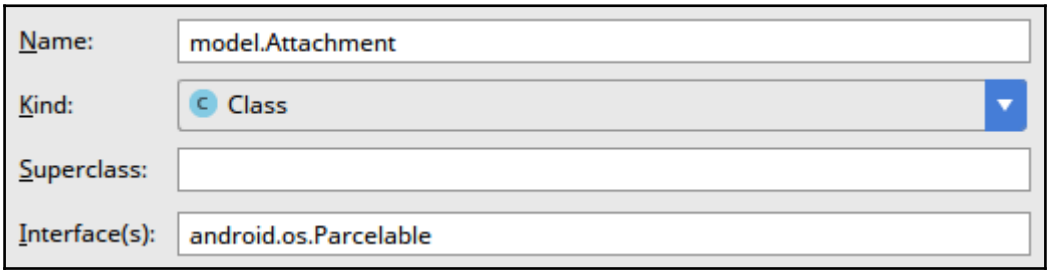
At this point in the application, it's time to build a simple data model that the user interface will back onto. Each claim will be represented by a `ClaimItem` object, and will contain any number of `Attachment` objects, each of which will reference the `File` that was attached, and have a marker to help decide how the attachment should be previewed. All these classes will need to be `Parcelable`, because they need to be saved in the `CaptureClaimActivity`. The `CaptureClaimActivity` will also use them as input and output parameters, and any time an object needs to be passed as a parameter to or from an `Activity`, it needs to be `Parcelable`.

You'll also be creating a `Category` enum that links the Android IDs to an internal model that can be stored without having to worry about the Android IDs changing their values as the application evolves.

Creating the Attachment class

The `Attachment` class represents files that have been attached to a `ClaimItem` by the user. These should always be files accessible by the application, and later on, we'll take steps to ensure this by copying all the attachments into a private space before attaching them to a claim item. For now, follow these steps to create the `Attachment` class:

1. In the **Android** panel, right-click on your default package (that is, `com.packtpub.claim`) and select **New | Java Class**.
2. Name the new class as `model.Attachment`, and in the **Interface(s)** box, add `Parcelable`:



3. Click **OK** to create the new package and class.
4. Attachments have different types, which can affect how they are previewed; for now, you'll just have images and unknown types. Inside the new `Attachment` class, create an `enum` to represent these types:

```
public enum Type {
  IMAGE,
  UNKNOWN;

  public static Type safe(final Type type) {
    // Use a ternary to replace null with UNKNOWN
    return type != null ? type : UNKNOWN;
  }
}
```

5. In the `Attachment` class, declare its fields, a constructor, and getters and setters:

```
File file;
Type type;

public Attachment(final File file, final Type type) {
 this.file = file;
 this.type = Type.safe(type);
}

public File getFile() { return file; }
public void setFile(final File file) {
  this.file = file;
}

public Type getType() { return type; }
public void setType(final Type type) {
  this.type = Type.safe(type);
}
```

6. Now, create the `Parcelable` implementation for the `Attachment` class. This is best done by hand in this case, as neither `File` nor the `Type enum` will be understood by Android Studio's `Parcelable` generator:

```
protected Attachment(final Parcel in) {
  file = new File(in.readString());
  type = Type.values()[in.readInt()];
}

@Override
public void writeToParcel(final Parcel dest, final int flags) {
```

```
    dest.writeString(file.getAbsolutePath());
    dest.writeInt(type.ordinal());
  }

  @Override
  public int describeContents() { return 0; }
```

7. Finally, at the top of the `Attachment` class, add its `Parcelable.Creator` instance:

```
public static final Creator<Attachment> CREATOR = new
Creator<Attachment>() {
  @Override
  public Attachment createFromParcel(final Parcel in) {
    return new Attachment(in);
  }

  @Override
  public Attachment[] newArray(final int size) {
    return new Attachment[size];
  }
};
```

Creating the Category enum

The next part of the model is the `Category` enumeration. This will serve a double purpose--when you change the list of available resources in your application, their IDs can all change. This makes these IDs unsuited to long-term identification of items; however, they are very useful as identifiers while the application is running:

- They are unique within the application
- They are integer types, which are very fast for comparisons
- They can be used to directly identify user-interface components

The `Category` enum will serve as a way to bind between a long-term stable identifier (the enum name), and the potentially unstable (but often much faster) Android resource ID. Follow these quick steps to create the `Category` enum:

1. Right-click on the `model` package, and select **New** | **Java Class**.
2. Name the class `Category`.
3. Change the **Kind** field to **Enum**.

4. Click **OK** to create the new enum file.
5. Declare the enum constants, and map them to their appropriate Android resource IDs:

```
ACCOMMODATION(R.id.accommodation),
FOOD(R.id.food),
TRANSPORT(R.id.transport),
ENTERTAINMENT(R.id.entertainment),
BUSINESS(R.id.business),
OTHER(R.id.other);
```

6. Declare the ID integer, private constructor, and getter method for the ID. Note the use of the `@IdRes` annotation, which indicates what should be used for these specific integers; attempting to pass anything other than an ID resource in here will result in a lint error in Android Studio:

```
@IdRes
private final int idResource;

Category(@IdRes final int idResource) {
  this.idResource = idResource;
}

@IdRes
public int getIdResource() {
  return idResource;
}
```

There are annotations similar to `@IdRes` for all of the different resource types available on Android. They are located in the `android.support.annotation` package. Use them wherever you expect an integer value to reference an Android resource of some type.

7. Finally, create a method to look up a `Category` enum constant from its Android ID resource:

```
public static Category forIdResource(@IdRes final int id) {
  for (final Category c : values()) {
    if (c.idResource == id) {
      return c;
    }
  }

  throw new IllegalArgumentException("No category for ID: " + id);
}
```

Creating the ClaimItem class

The **ClaimItem** is the heart of this application's object model. Each claim the user collects is represented in memory as a single `ClaimItem` instance. Here are the steps required to build the `ClaimItem` class:

1. Right-click on the `model` package, and select **New | Java Class**.
2. Name the class `ClaimItem`, and in the **Interface(s)** box, add `Parcelable`.
3. Click **OK** to create the new class file.
4. Declare the fields of the `ClaimItem` type, and a `public` default constructor:

```
String description;
double amount;
Date timestamp;
Category category;
List<Attachment> attachments = new ArrayList<>();

public ClaimItem() {}
```

5. Use Android Studio to generate getter and setter methods for all the fields, except the attachments field:

```
public String getDescription() { return description; }
public void setDescription(final String description) {
  this.description = description;
}
public double getAmount() { return amount; }
public void setAmount(final double amount) {
  this.amount = amount;
}
public Date getTimestamp() { return timestamp; }
public void setTimestamp(final Date timestamp) {
  this.timestamp = timestamp;
}
public Category getCategory() { return category; }
public void setCategory(final Category category) {
  this.category = category;
}
```

6. Create methods to add, remove, and list the `Attachment` objects for the `ClaimItem`:

```
public void addAttachment(final Attachment attachment) {
  if ((attachment != null) && !attachments.contains(attachment)) {
   attachments.add(attachment);
  }
}

public void removeAttachment(final Attachment attachment) {
  attachments.remove(attachment);
}

public List<Attachment> getAttachments() {
  return Collections.unmodifiableList(attachments);
}
```

7. Implement the `Parcelable` methods for the `ClaimItem` class; again, this is more complex than the Android Studio generator can typically handle:

```
protected ClaimItem(final Parcel in) {
  description = in.readString();
  amount = in.readDouble();

  final long time = in.readLong();
  timestamp = time != -1 ? new Date(time) : null;

  final int categoryOrd = in.readInt();
  category = categoryOrd != -1
      ? Category.values()[categoryOrd]
      : null;

  in.readTypedList(attachments, Attachment.CREATOR);
}

@Override
public void writeToParcel(final Parcel dest, final int flags) {
  dest.writeString(description);
  dest.writeDouble(amount);
  dest.writeLong(timestamp != null ? timestamp.getTime() : -1);
  dest.writeInt(category != null ? category.ordinal() : -1);
  dest.writeTypedList(attachments);
}

@Override
public int describeContents() { return 0; }
```

```
public static final Creator<ClaimItem> CREATOR = new
Creator<ClaimItem>() {
  @Override
  public ClaimItem createFromParcel(Parcel in) {
  return new ClaimItem(in);
  }

  @Override
  public ClaimItem[] newArray(int size) {
  return new ClaimItem[size];
  }
};
```

Excellent! With the formality of an object model out of the way, you can continue building the user interface. The next stage will involve building `Fragment` classes that will help modularize the capturing of the `ClaimItem` data.

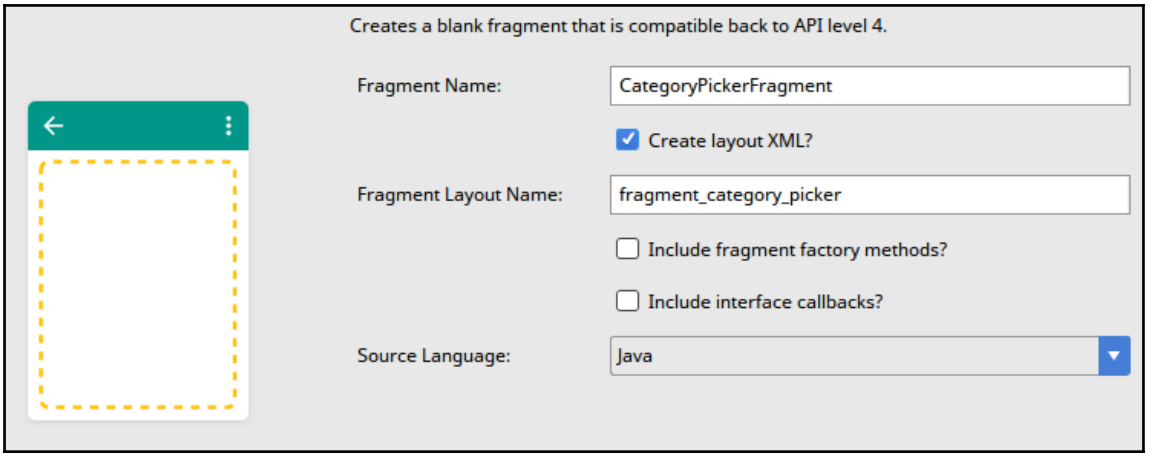
Wrapping up the category picker

The category picker you created for the `CaptureClaimActivity` is just a group of widgets in a card right now, and while it's one of the simplest cards to use on the screen, it's also one of the largest by the amount of code written for it. The best way to encapsulate this part of the screen is to move the layout that appears inside the `CardView` into a `Fragment` class.

However, why a `Fragment` class, and why not write another `Layout` class? `Fragment` classes are self-contained systems, and have their own life cycle within the context of their parent `Activity`. This means they can contain significantly more application logic, and can be reused more easily in other parts of the application. It's also because in this case, we rely on the IDs of the radio buttons to know what has been checked by the user, which means that we can very easily start polluting layouts with IDs specific to this specific widget. Fragment classes don't stop this from happening, but it's expected behavior. You don't expect ID pollution from `View` classes, but from a `Fragment`, it's okay. Follow these simple steps to encapsulate the category picker in a new `Fragment` class:

1. Right-click on the `ui` package in your project and select **New| Fragment | Fragment (Blank)**.
2. Name the new `Fragment` class `CategoryPickerFragment`.

3. Turn off **Include fragment factory methods?** and **Include interface callbacks**:



4. Click on **Finish** to create your new `Fragment` and its layout file.
5. Open the new `fragment_category_picker.xml` file, and change the editor view to **Text** mode.
6. Change the root node of the layout from `FrameLayout` to `LinearLayout`, and make it a `vertical` orientation:

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

7. Remove any contents of the `LinearLayout` placed there by the Android Studio template.
8. Open the `content_capture_claim.xml` layout file, and change the editor view to **Text** mode.
9. Cut the contents of the `LinearLayout` containing the existing category picker, the entire `RadioGroup` and `TextView` used as their label.
10. Paste this as the content of the `LinearLayout` in the `fragment_category_picker.xml` 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">

  <RadioGroup
      android:id="@+id/categories"
      android:layout_width="match_parent"
      android:layout_height="wrap_content"
      android:orientation="horizontal">

      <RadioButton
          android:id="@+id/accommodation"
          android:layout_width="wrap_content"
          android:layout_height="wrap_content"
          android:layout_marginEnd="@dimen/grid_spacer1"
          android:layout_marginRight="@dimen/grid_spacer1"
          android:button="@drawable/ic_category_hotel"
android:contentDescription="@string/description_accommodation" />

      <!-- ... -->
  </RadioGroup>

  <TextView
      android:id="@+id/selected_category"
      android:layout_width="match_parent"
      android:layout_height="wrap_content"
android:textAppearance="@style/TextAppearance.AppCompat.Medium" />
</LinearLayout>
```

11. In the `content_capture_claim.xml` layout file, you can now remove the `LinearLayout` for the category picker and replace it with a reference to the `Fragment` class:

```
<android.support.v7.widget.CardView
  android:layout_width="match_parent"
  android:layout_height="wrap_content"
  android:layout_marginTop="@dimen/grid_spacer1">

  <fragment
      class="com.packtpub.claim.ui.CategoryPickerFragment"
      android:id="@+id/categories"
      android:layout_width="match_parent"
      android:layout_height="match_parent"
      android:layout_margin="@dimen/grid_spacer1"/>

</android.support.v7.widget.CardView>
```

12. Now, open the `CategoryPickerFragment` class in Android Studio, and at the top of the class, declare the fields for the `RadioGroup` and `TextView` that you'll use to track and update the user's selection:

```
private RadioGroup categories;
private TextView categoryLabel;
```

13. Now in the `onCreateView`, you'll need to change how the `View` is inflated, because you need to capture the fields and set up the event listeners. Note the use of the `IconPickerWrapper` as the event listener:

```
public View onCreateView(
      final LayoutInflater inflater,
      final @Nullable ViewGroup container,
      final @Nullable Bundle savedInstanceState) {

  final View picker = inflater.inflate(
      R.layout.fragment_category_picker,
      container,
      false
  );

  categories = (RadioGroup) picker.findViewById(R.id.categories);
  categoryLabel = (TextView) picker.findViewById(
      R.id.selected_category);
  categories.setOnCheckedChangeListener(
      new IconPickerWrapper(categoryLabel));
  categories.check(R.id.other);
  return picker;
}
```

14. Now, create a simple getter and setter method to retrieve and alter state using the `Category` enum:

```
public Category getSelectedCategory() {
  return Category.forIdResource(
      categories.getCheckedRadioButtonId());
}

public void setSelectedCategory(final Category category){
  categories.check(category.getIdResource());
}
```

15. Open the `CaptureClaimActivity` in Android Studio.

16. Change the categories field to use `CategoryPickerFragment`, instead of a `RadioGroup`:

```
private CategoryPickerFragment categories;
```

17. In the `onCreate` method, remove the code that initialized the category picker:

```
categories = (RadioGroup) findViewById(R.id.categories);
categories.setOnCheckedChangeListener(
    new IconPickerWrapper(
        (TextView) findViewById(R.id.selected_category)
    )
);
categories.check(R.id.other);
```

18. Use `FragmentManager` to retrieve the new `CategoryPickerFragment` from the layout:

```
final FragmentManager fragmentManager =
getSupportFragmentManager();
categories = (CategoryPickerFragment)
    fragmentManager.findFragmentById(R.id.categories);
```

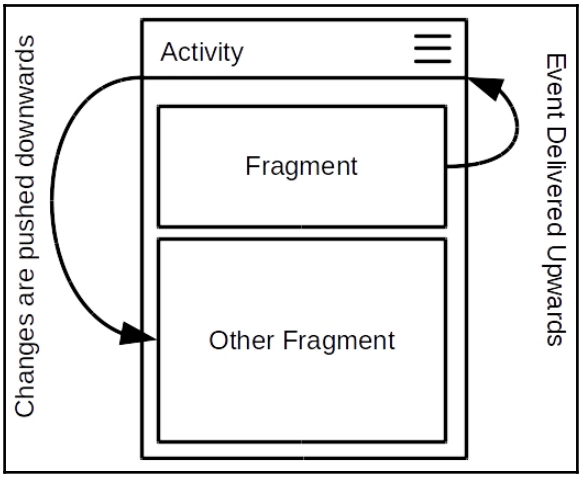
Note that you're using the `getSupportFragmentManager` method, and not `getFragmentManager`. This is because `CategoryPickerFragment` is built on top of the support APIs and is backwards compatible all the way to API level 4 (Android 1.6). Android Studio typically prefers the support APIs when generating code as it offers a very simple and stable target, since your application links against a static target, and you are in control of which version to link against and when to upgrade. You can reuse `CategoryPickerFragment` anywhere in the application, just as you would a custom `View` implementation.

Linking against the platform APIs (instead of the equivalent support) reduces backward compatibility and requires more testing since your application may behave slightly differently on different versions of the platform. However, platform versions may be slightly faster, and will result in smaller applications.

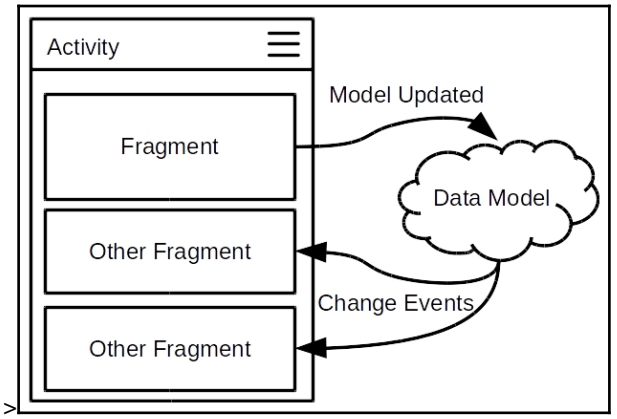
Creating the Attachment Pager

Having modularized the category picker, it's time to turn your attention to the attachments. When you implemented the file selection, you left a `Toast` in place to show where the code would normally attach the selected file to the `ClaimItem` being captured. This next stage will be to create a `Fragment` that will encapsulate the previewing of the `Attachment` objects. You'll also move much of the attachment logic into this `Fragment`. Although the code to connect to other applications and request permissions is commonly placed in an `Activity` class, `Fragment` classes are also capable of performing the same actions, and the attachment pager is a perfect opportunity to show this off.

This `Fragment` will show a pattern where the `Fragment` interacts with the `Activity` that it belongs to without directly sending events upward. The instinct of most developers when encountering a `Fragment` for the first time is to use the pattern in the template where the `Fragment` can send events upward to its `Activity`, as shown:



However, this is often not desirable. It's generally a much better idea to push changes through the data model, and have it deliver the events to areas that are interested in the changes. This is part of a unidirectional event flow, and serves to keep the application much easier to maintain and debug, because the data model always represents the *authority* for all information and state within the application, as illustrated here:



Creating the Attachment preview widget

The first part of the attachments will be a `View` implementation, to allow attachments to be previewed within the pager. This class will need an area where the attachment can be previewed if it's an image, or a placeholder icon can be displayed if it's a file the application cannot read. Follow these steps to create the new widget and it's layout XML file:

1. Right-click on the **layout** directory under **res**, and select **New | Layout resource file**.
2. Name the new file `widget_attachment_preview`.
3. Change the **Root** element field to `merge`.
4. Click **OK** to create the new layout file.

5. Within the `merge` element, create an `ImageView` that can carry the preview of the attachment file. The `ImageView` will need a margin to automatically scale the image to its size on the screen (while maintaining the image's proportions):

```
<?xml version="1.0" encoding="utf-8"?>
<merge xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="match_parent">
  <ImageView
      android:scaleType="fitCenter"
      android:layout_margin="@dimen/grid_spacer1"
      android:layout_width="match_parent"
      android:layout_height="match_parent"/>
</merge>
```

6. Right-click on the **drawable** resource directory, and select **New**, **Vector Asset**.
7. Using the **Icon** button, search for the `insert drive file` icon and select it.
8. Name the new resource `ic_unknown_file_type`.
9. Click on **Next** and then on **Finish** to create the new resource.
10. Open the `ic_unknown_file_type.xml` file in Android Studio.
11. Change the `fillColor` attribute of the path to `#FFBAB5AB`, and save and close the file.
12. Right-click on the `widget` package in your project, and select **New | Java Class**.
13. Name the new class `AttachmentPreview`.
14. Change the **Superclass** field to `android.support.v7.widget.CardView`.
15. Click **OK** to create the new class.
16. Create fields to reference the `Attachment` object and the `ImageView` that will render its preview onto the screen:

```
private Attachment attachment;
private ImageView preview;
```

17. Create the standard `View` subclass constructors and an `initialize` method that inflates the layout XML and captures the `ImageView`:

```
public AttachmentPreview(Context context) {
  super(context);
  initialize(context);
}

public AttachmentPreview(Context context, AttributeSet attrs) {
  super(context, attrs);
```

```
    initialize(context);
  }

  public AttachmentPreview(
      Context context,
      AttributeSet attrs,
      int defStyleAttr) {
    super(context, attrs, defStyleAttr);
    initialize(context);
  }

  void initialize(final Context context) {
    LayoutInflater.from(context).inflate(
        R.layout.widget_attachment_preview, this, true);
    preview = (ImageView) getChildAt(0);
  }
```

18. Create a simple getter for the `Attachment` field:

```
public Attachment getAttachment() { return attachment; }
```

19. Create a setter to update the `Attachment` field, and also start the update of the preview on-screen. You'll also create an inner class using the `ActionCommand` class you wrote in `Chapter 3`, *Taking Actions*, which will attempt to load the actual images on a background thread before updating the widget on the screen:

```
public void setAttachment(final Attachment attachment) {
  this.attachment = attachment;
  preview.setImageDrawable(null);

  if (attachment != null) {
    new UpdatePreviewCommand().exec(attachment);
  }
}

private class UpdatePreviewCommand
    extends ActionCommand<Attachment, Drawable> {

  @Override
  public Drawable onBackground(
      final Attachment attachment)
      throws Exception {

    switch (attachment.getType()) {
      case IMAGE:
        return new BitmapDrawable(
            getResources(),
```

```
                attachment.getFile().getAbsolutePath()
            );
        }

        return getResources().getDrawable(
            R.drawable.ic_unknown_file_type);
      }

      @Override
      public void onForeground(final Drawable value) {
        preview.setImageDrawable(value);
      }
    }
```

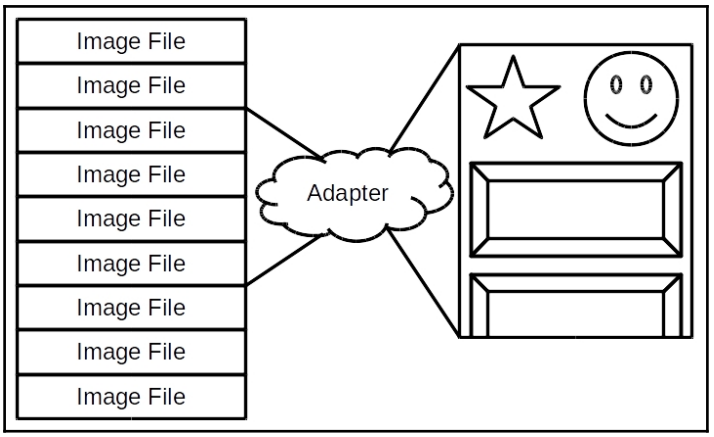
The preceding code is a great example of using an `ActionCommand` object to improve the user experience. When the `Attachment` is actually specified on the `AttachmentPreview` widget, the onscreen preview is instantly queued, and the loading of the actual preview (which may take a second or two on a slower device) takes place in the background. This frees the main thread to continue processing events from the user, or start the loading of other previews that may be required.

Creating the Attachment Pager Adapter

The `ViewPager` class is a special type of Android widget called an adapter view (although some, such as `ViewPager`, don't actually inherit from `AdapterView`, it shares much of the concepts). An `AdapterView` is used when it potentially needs to display more data than will fit on the screen at once, but maintain excellent performance. They do this by maintaining a small selection of widgets that will be displayed on the screen, and an `Adapter` that will populate the widgets with the data to display. Examples of `Adapter` widgets are as follows:

- `ListView`: A simple vertical scrolling list of similar items, such as phone contacts
- `GridView`: A vertical scrolling grid of similar items, such as photos
- `StackView`: A three-dimensional stack of items great for presenting media
- `RecyclerView`: A powerful all-purpose pooling view, originally added to replace `ListView`

If you wanted to display a scrolling list of images, for example, you would use a `RecyclerView` and provide it with an `Adapter` that could load previews of image files into `ImageView` widgets (much the same way as the `AttachmentPreview` class does):



`ViewPager` is a little different from the `AdapterView` classes described here; all of them only maintain as many widgets as what can fit on the screen at once. The normal `AdapterView` classes and `RecyclerView` all *recycle* their pool of widgets. When one widget is scrolled off the screen, it's resized, populated with new data, and scrolled into view looking like a new widget. `ViewPager` doesn't stop you from doing this, but it doesn't do it for you. This is because `ViewPager` often contains large and complex tab layouts, which would be too expensive to try and recycle (or simply don't repeat at all, in which case, recycling is useless).

For this application, the user is unlikely to have many attachments, so you can get away with simply creating an `AttachmentPreview` instance for each of the attachments when they are displayed, which keeps the steps for implementing the `Adapter` much simpler and to the point:

1. Right-click on your default package (that is, `com.packtpub.claim`) and select **New | Java Class**.
2. Name the new class `ui.attachments.AttachmentPreviewAdapter`.
3. Make its **Superclass** `android.support.v4.view.PagerAdapter`.
4. Click **OK** to create the new class.

5. This class will need a `List` of the `Attachment` objects it's expected to translate into widgets for previewing, and it'll need a setter to change what will be displayed:

```
private List<Attachment> attachments = Collections.emptyList();

public int getCount() {
  return attachments.size();
}

public void setAttachments(final List<Attachment> attachments) {
  this.attachments = attachments != null
          ? attachments
          : Collections.<Attachment>emptyList();
  notifyDataSetChanged();
}
```

After changing the `List` of attachments, `AttachmentPreviewAdapter` is wrapping; it invokes `notifyDataSetChanged()`, which informs the `ViewPager` it's attached to things that have changed, and some rerendering may be required. This sort of functionality can be found in all the `Adapter` classes, and allows for the reactive behavior that users expect from their apps. When a new email arrives, it can just appear on the list they're looking at. As a developer, this system is nice because the events can bubble up from the data model rather than being tied to the user interface.

6. The `ViewPager` maintains separate lists of the widgets used to display data on the screen and the object model being displayed. The `ViewPager` creates the widgets by invoking `instantiateItem` on the `PagerAdapter` object, which is expected to add the widget to the `ViewPager` and return the data model object that it's displaying:

```
public Object instantiateItem(final ViewGroup container, final int
position) {
  final AttachmentPreview preview =
      new AttachmentPreview(container.getContext());
  preview.setAttachment(attachments.get(position));
  container.addView(preview);
  return attachments.get(position);
}
```

7. The `ViewPager` may also ask `PagerAdapter` to remove the widgets that are not visible to the user. This is typically when a view is not visible, and cannot be directly scrolled into view by the user (that is, it's not directly to the left or right of the current view). The position argument passed to `destroyItem` is the position in the data model, not the index of the widget within the `ViewPager`, so you need a way to figure out which widget in the `ViewPager` actually needs to be removed. Here, we do it by simply iterating over all the child widgets in the `ViewPager` since there will never be many of them:

```
public void destroyItem(
    final ViewGroup container,
    final int position,
    final Object object) {
  for (int i = 0; i < container.getChildCount(); i++) {
    final AttachmentPreview preview =
        ((AttachmentPreview) container.getChildAt(i));
    if (preview.getAttachment() == object) {
      container.removeViewAt(i);
      break;
    }
  }
}
```

8. Finally, the `ViewPager` needs a way to know which widget child of its is associated with which part of the data model; this is really easy for you in this class, because the `AttachmentPreview` class directly references the `Attachment` objects:

```
public boolean isViewFromObject(final View view, final Object o) {
  return (view instanceof AttachmentPreview)
      && (((AttachmentPreview) view).getAttachment() == o);
}
```

This implementation of `PagerAdapter` is very naive and simple, but shows more of how the `Adapter` views work. They track their onscreen views completely independently of the dataset, and the order in which the child widgets appear on the screen doesn't have any direct relationship to the order in which the data model is presented.

The next step is to create another `ActionCommand` class that will create the `Attachment` objects when the user selects an external file to attach to the claims.

Creating the Create Attachment Command

When the user selects a file to attach to the claim, you'll need to ensure that your application can always access the file. This means copying the file into your application's private space, which can take a second or two. You also need to know what type of file it is, otherwise your application won't know if it can render a preview of the attachment. For both of these, you'll need an `ActionCommand` implementation that does the work:

1. Right-click on the `model` package, and select **New | Java Class**.
2. Name the new class `commands.CreateAttachmentCommand`.
3. Make the class **Abstract**.
4. Click **OK** to create the new package and class.
5. Change the class declaration to extend `ActionCommand<Uri, Attachment>`:

```
public abstract class CreateAttachmentCommand
    extends ActionCommand<Uri, Attachment> {
```

6. Declare a directory to write the local files to, and a `ContentResolver` that can be used to read the files that the user selects:

```
private final File dir;
private final ContentResolver resolver;

public CreateAttachmentCommand(
      final File dir,
      final ContentResolver resolver) {

  this.dir = dir;
  this.resolver = resolver;
}
```

A `ContentResolver` allows applications to read each other's data, if they choose to expose it. In this case, you'll be using `content://` URIs that are commonly used in Android when data needs to be exposed safely between applications. A `ContentResolver` counterpart is a `ContentProvider` that exposes the data for other applications to access.

7. Create a simple utility method to copy the file from a `Uri` into a new, randomly named file. The file is randomly named so that no two files are likely to collide in name:

```
File makeFile(final Uri value) throws IOException {
  final File outputFile =
      new File(dir, UUID.randomUUID().toString());
```

```
    final InputStream input = resolver.openInputStream(value);
    final FileOutputStream output = new FileOutputStream(outputFile);
    try {
        final byte[] buffer = new byte[10 * 1024];
        int bytesRead = 0;
        while ((bytesRead = input.read(buffer)) != -1) {
            output.write(buffer, 0, bytesRead);
        }
        output.flush();
    } finally {
        output.close();
        input.close();
    }
    return outputFile;
}
```

8. Override the `onBackground` method to copy the file using the preceding utility method:

```
public Attachment onBackground(final Uri value) throws Exception {
   final File file = makeFile(value);
```

9. Finally, check the type of the file your command just created, and if it looks like an image, ensure that you can read it before returning. This avoids the application having to do the same check every time it wants to preview the attachment. We check whether the image is readable by attempting to read it using the `BitmapFactory` class:

```
    final String type = resolver.getType(value);
    if (type != null
        && type.startsWith("image/")
        && BitmapFactory.decodeFile(file.getAbsolutePath()) != null)
    {
      return new Attachment(file, Attachment.Type.IMAGE);
    } else {
      return new Attachment(file, Attachment.Type.UNKNOWN);
    }
}
```

This simple command class doesn't have any foreground work, and has been left abstract. Instead, it assumes that the work of handling the `Attachment` will be done elsewhere. The next part is the `AttachmentPagerFragment` class that will handle the `Attachment` objects created here by attaching them to the `ClaimItem`, and notifying the `AttachmentPreviewAdapter` that there is a new attachment to render.

Creating the Attachment Pager Fragment

Now that you've assembled all of the parts required for the attachments to be created and previewed, you need to actually populate the area where they will be previewed. The `AttachmentPagerFragment` class will not only serve to encapsulate the `ViewPager` used to preview the attachments, but will also encapsulate the logic required to add new attachments to the user's claim. This will be done by moving the `onRequestPermissionsResult` and `onActivityResult` from the `CaptureClaimActivity` to the new `AttachmentPagerFragment` class. This process will require moving some of the code out of `CaptureClaimActivity` and into the `Fragment` class, so you'll be in for some cutting and pasting. Let's get started:

1. Create a new **layout** resource named `fragment_attachment_pager`.
2. Open the `content_capture_claim.xml` layout file.
3. Cut and paste the `ViewPager` from the bottom of `content_capture_claim.xml` into the `fragment_attachment_pager` layout file, overwriting all the content of the file. You'll need to define the XML namespaces (the `xmlns` attributes) on the `ViewPager` element, so that the `fragment_attachment_pager.xml` file reads like this:

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.v4.view.ViewPager
  xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:tools="http://schemas.android.com/tools"
  android:id="@+id/attachments"
  android:layout_width="wrap_content"
  android:layout_height="wrap_content"
  android:layout_weight="1"
  android:clipChildren="false"
  android:clipToPadding="false"
  tools:context=".ui.attachments.AttachmentPagerFragment"/>
```

4. Create a **New**, **Java Class** in the `ui.attachments` package.
5. Name the class `AttachmentPagerFragment`.
6. Make its **Superclass** `android.support.v4.app.Fragment`.
7. Open the `CaptureClaimActivity` class.
8. Cut the `REQUEST_ATTACH_FILE` and `REQUEST_ATTACH_PERMISSION` constants from the `CaptureClaimActivity` and paste them in `AttachmentPagerFragment`:

```
private static final int REQUEST_ATTACH_FILE = 1;
private static final int REQUEST_ATTACH_PERMISSION = 1001;
```

9. Create an instance of `AttachmentPagerAdapter` to help with rendering the attachment previews. As the `AttachmentPagerAdapter` can handle its list of `Attachment` object changing completely, you'll only ever need one in each `AttachmentPagerFragment`:

```
private final AttachmentPreviewAdapter adapter = new
AttachmentPreviewAdapter();
```

10. Create fields for the `ActionCommand` you'll use to attach the files, and another to hold a reference to the `ViewPager` object:

```
private ActionCommand<Uri, Attachment> attachFileCommand;
private ViewPager pager;
```

11. Your `AttachmentPagerFragment` needs a reference to the `ClaimItem` it is previewing `Attachment` for. This will also allow it to add new `Attachment` objects to the claim without invoking its `Activity` to do so. The `Fragment` will also expose a method that can be called to notify it that the list of attachments on the `ClaimItem` has changed. This can be invoked by the `ClaimItem` itself later on, or through an event-bus:

```
private ClaimItem claimItem;

public void setClaimItem(final ClaimItem claimItem) {
  this.claimItem = claimItem;
  onAttachmentsChanged();
}

public void onAttachmentsChanged() {
  adapter.setAttachments(
      claimItem != null
          ? claimItem.getAttachments()
          : null
  );
  pager.setCurrentItem(adapter.getCount() - 1);
}
```

12. Override the `Fragment` `onCreate` method. This looks just like the `onCreate` method of an `Activity`, and is called after your `Fragment` has been attached to its context (in this case, to its `Activity` object). `AttachmentPagerFragment` will use `onCreate` to instantiate the `attachFileCommand` for later use, and it'll do so using an anonymous inner class, inheriting from the `CreateAttachmentCommand` class that you just wrote:

```
public void onCreate(final @Nullable Bundle savedInstanceState) {
   super.onCreate(savedInstanceState);
   final File attachmentsDir =
      getContext().getDir("attachments", Context.MODE_PRIVATE);
   attachFileCommand = new CreateAttachmentCommand(
       attachmentsDir,
       getContext().getContentResolver()) {
       @Override
       public void onForeground(final Attachment value) {
           if (claimItem != null) {
               claimItem.addAttachment(value);
               onAttachmentsChanged();
           }
       }
   };
}
```

In any task that takes place in the background and then jumps back to the foreground, it's a good idea to check your context before running any code. In the preceding snippet, this takes the form of the `claimItem != null` check. If the command was started, and the user left the `Activity` (or similar), the foreground code could trigger errors by trying to alter variables that are invalid or `null`.

13. When the `Fragment` is released completely (with no chance of a later restart), its `onDestroy` method is called. Use this method to release the `claimItem`, stopping any background tasks from modifying it when they get back to the foreground:

```
public void onDestroy() {
   super.onDestroy();
   claimItem = null;
}
```

14. Just like the `CategoryPickerFragment` you wrote earlier, `AttachmentPagerFragment` needs a `View` that will display when it's inflated into a layout XML. In this case, you also need to adjust the `ViewPager` slightly, as the page margin is not exposed as an XML attribute:

```
public View onCreateView(
       final LayoutInflater inflater,
       final @Nullable ViewGroup container,
       final @Nullable Bundle savedInstanceState) {

  pager = (ViewPager) inflater.inflate(
      R.layout.fragment_attachment_pager, container, false);
  pager.setPageMargin(
      getResources().getDimensionPixelSize(R.dimen.grid_spacer1));
  pager.setAdapter(adapter);

  return pager;
}
```

15. Now, cut and paste the `onAttachClick` method from `CaptureClaimActivity` into `AttachmentPagerFragment`. This will immediately cause errors, because `onAttachClick` uses the fact that an `Activity` is also a `Context`; so, `ContextCompat.checkSelfPermission` can use the `CaptureClaimAcitvity` as the `Context` to check. `Fragment` doesn't inherit from `Context`, but it does expose the `getContext()` and `getActivity()` methods to retrieve the environment that it's attached to:

```
public void onAttachClick() {
  final int permissionStatus = ContextCompat.checkSelfPermission(
      getContext(),
      Manifest.permission.READ_EXTERNAL_STORAGE);

  if (permissionStatus != PackageManager.PERMISSION_GRANTED) {
    ActivityCompat.requestPermissions(
        getActivity(),
        new String[]{Manifest.permission.READ_EXTERNAL_STORAGE},
        REQUEST_ATTACH_PERMISSION);
    return;
  }

  final Intent attach = new Intent(Intent.ACTION_GET_CONTENT)
      .addCategory(Intent.CATEGORY_OPENABLE)
      .setType("*/*");

  startActivityForResult(attach, REQUEST_ATTACH_FILE);
}
```

16. Now cut and paste the `onRequestPermissionsResult`, `onAttachFileResult`, and `onActivityResult` methods over as well. These should copy over without any errors.
17. In the `onAttachFileResult` method, you can now remove the `Toast` you put in as a placeholder. Instead, invoke the `attachFileCommand` with the selected file; this will automatically result in the previews also being updated:

```
Toast.makeText(this, data.getDataString(),
Toast.LENGTH_SHORT).show();
attachFileCommand.exec(data.getData());
```

18. In the `content_capture_claim.xml` layout file, include the new `AttachmentPagerFragment` where the `ViewPager` used to be:

```
<fragment
   android:id="@+id/attachments"
class="com.packtpub.claim.ui.attachments.AttachmentPagerFragment"
   android:layout_width="match_parent"
   android:layout_height="0dp"
   android:layout_marginTop="@dimen/grid_spacer1"
   android:layout_weight="1" />
```

19. In the `CaptureClaimActivity`, create a new field for the `AttachmentPagerFragment`, and in `onCreate`, capture the field from the `FragmentManager`:

```
private AttachmentPagerFragment attachments;
// ...
protected void onCreate(Bundle savedInstanceState) {
   // ...
   final FragmentManager fragmentManager =
       getSupportFragmentManager();
   categories = (CategoryPickerFragment)
       fragmentManager.findFragmentById(R.id.categories);
   attachments = (AttachmentPagerFragment)
       fragmentManager.findFragmentById(R.id.attachments);
```

20. Finally, change the `onClick` method in `CaptureClaimActivity` to invoke `onAttachClick` on the `AttachmentPagerFragment`:

```
@Override
public void onClick(View v) {
  switch (v.getId()) {
    case R.id.attach:
      attachments.onAttachClick();
```

```
            break;
        }
    }
```

The `AttachmentPagerFragment` is a multipurpose `Fragment`. Although it has all the logic required to attach files to a `ClaimItem`, it doesn't attempt to hook them up to any event listeners itself. As a result, you can easily use it as a read-only preview of attachments if (for example) the current user was reviewing someone else's travel expenses (in which case, they shouldn't be editing the data).

It's always a good idea to consider how a `Fragment` might be reused in different situations, and to rather push data and events down into them rather than having them making call upward to their `Activity` to know what should be done (which forces every `Activity` wanting to use the `Fragment` to provide that information).

Capturing the ClaimItem data

While you've linked the new `Fragment` classes to the `CaptureClaimAcitvity`, things aren't quite finished yet. The `CaptureClaimActivity` doesn't actually have a `ClaimItem` to capture and modify. For this, you'll not only need to hold a reference to a `ClaimItem` in the `CaptureClaimActivity`, you'll need to ensure that it is saved and restored through life cycle changes for the `Activity` as well. Fortunately, your model is all `Parcelable`, which keeps this easy. It's time to capture a `ClaimItem`:

1. Open the `CaptureClaimActivity` class.
2. First, you'll need a way that a `ClaimItem` can be passed into the `CaptureClaimActivity` for editing. To keep this simple and flexible, you'll allow them to be passed as an "extra" field on the `Intent`. When you use `extras` in an `Intent`, it's a good idea to expose the name as a public constant so that they can be accessed by outside classes when creating the `Intent` objects:

   ```
   public static final String EXTRA_CLAIM_ITEM =
   "com.packtpub.claim.extras.CLAIM_ITEM";
   ```

3. You'll also need to save and restore the `ClaimItem` while it's being edited, and for this, you'll also need a key for the `Bundle`:

   ```
   private static final String KEY_CLAIM_ITEM =
   "com.packtpub.claim.ClaimItem";
   ```

4. Then, create a `private` field to reference the `ClaimItem` being edited, and you'll also need to reference all the inputs and `Fragment` objects on the screen; `CaptureClaimActivity` should have `private` fields looking like this:

```
private EditText description;
private EditText amount;

private DatePickerLayout selectedDate;
private CategoryPickerFragment categories;
private AttachmentPagerFragment attachments;

private ClaimItem claimItem;
```

5. In the `onCreate` method, ensure that you capture all the preceding fields after the call to `setContentView`:

```
description = (EditText) findViewById(R.id.description);
amount = (EditText) findViewById(R.id.amount);
selectedDate = (DatePickerLayout) findViewById(R.id.date);

final FragmentManager fragmentManager =
getSupportFragmentManager();
attachments = (AttachmentPagerFragment)
fragmentManager.findFragmentById(R.id.attachments);
categories = (CategoryPickerFragment)
fragmentManager.findFragmentById(R.id.categories);
```

6. Then, you'll need to check whether a `ClaimItem` has been passed in, either in the `savedInstanceState` `Bundle` (which will be populated if the `Activity` is being restarted due to a configuration change), or is being passed as an extra parameter on the `Intent` (a bit like a constructor argument):

```
if (savedInstanceState != null) {
   claimItem = savedInstanceState.getParcelable(KEY_CLAIM_ITEM);
} else if (getIntent().hasExtra(EXTRA_CLAIM_ITEM)) {
   claimItem = getIntent().getParcelableExtra(EXTRA_CLAIM_ITEM);
}
```

7. If a `ClaimItem` wasn't passed in through either of these mechanisms, you'll want to create a new, empty `ClaimItem` to be edited by the user. On the other hand, if one was passed in, you'll need to populate the user interface with its data:

```
if (claimItem == null) {
  claimItem = new ClaimItem();
} else {
  description.setText(claimItem.getDescription());
  amount.setText(String.format("%f", claimItem.getAmount()));
  selectedDate.setDate(claimItem.getTimestamp());
}

attachments.setClaimItem(claimItem);
```

8. Now, write a `utility` method to copy the data from the user interface widgets back into the `ClaimItem` object:

```
void captureClaimItem() {
  claimItem.setDescription(description.getText().toString());
  if (!TextUtils.isEmpty(amount.getText())) {
    claimItem.setAmount(
        Double.parseDouble(amount.getText().toString()));
  }
  claimItem.setTimestamp(selectedDate.getDate());
  claimItem.setCategory(categories.getSelectedCategory());
}
```

9. When the `Activity` is shutdown in ways that can result in it being restarted at a later time (and as a new instance), the `onSaveInstanceState` method is invoked with a `Bundle` where your `Activity` can save any state it needs to restore later (in this case, it'll be the `ClaimItem` being edited). This will happen if your `Activity` is in the background and the OS needs to reclaim memory, or if the `Activity` restarts due to a configuration change (such as the user changing between the portrait and landscape modes). This is where you set up the contents of the `Bundle` that gets passed into `onCreate`:

```
protected void onSaveInstanceState(final Bundle outState) {
  super.onSaveInstanceState(outState);
  captureClaimItem(); // make sure the ClaimItem is up-to-date
  outState.putParcelable(KEY_CLAIM_ITEM, claimItem);
}
```

10. We also want to ensure that when the `CaptureClaimActivity` is closed, it returns the edited `ClaimItem` to the `Activity` that started it. This can be done by overloading the `finish()` method, which is invoked to close an `Activity`:

```
public void finish() {
  captureClaimItem();
  setResult(
      RESULT_OK,
      new Intent().putExtra(EXTRA_CLAIM_ITEM, claimItem)
  );
  super.finish();
}
```

`CaptureClaimActivity` will always return a `ClaimItem` object; there is no notion of saving the `ClaimItem` or canceling its creation (although a calling `Activity` may choose to ignore the `ClaimItem` if it's empty). The idea is to assume that the user knows what they are doing, and rather offer them a way to undo their changes once they've been made. This is much less disruptive to the user than always asking them "are you sure" questions.

11. Finally, it's important that we give the user a visual method to exit the screen without using the Android back button. We'll do this by putting a *back* navigation arrow on the `Toolbar`. First, write a handler that listens for the *home* button being selected:

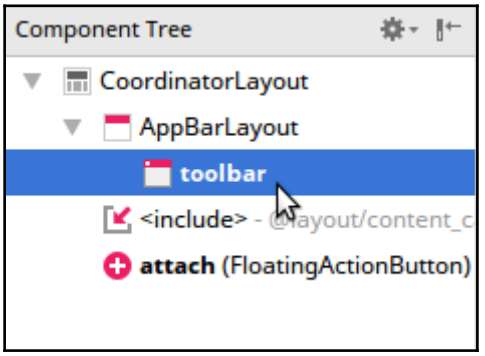
```
@Override
public boolean onOptionsItemSelected(final MenuItem item) {
  switch (item.getItemId()) {
    case android.R.id.home:
      finish();
      break;
    default:
      return false;
  }

  return true;
}
```

12. Now, right-click on the **drawable** resource directory and choose **New| Vector Asset**.
13. Using the **Icon** selector, search for `arrow back`.

14. Name the new Icon `ic_arrow_back_white_24dp`.
15. Click on **Next** and then on **Finish** to complete the wizard and create the new asset.
16. Open the `ic_arrow_back_white_24dp.xml` resource file.
17. Change the path `android:fillColor` attribute to white:

```
<path
   android:fillColor="#FFFFFFFF"
```

18. Open the `activity_capture_claim.xml` layout resource in **Design** mode.
19. Select the toolbar in the **Component Tree** panel:



20. In the **Attributes** panel, toggle to the **View all attributes** view.
21. Search for **navigationIcon**, and use the resource selector to pick the `ic_arrow_back_white_24dp` icon resource.

If you run the application now, you'll see that you can capture attachments for a claim, and when the device is rotated, or you navigate away using the home button, any data you've changed will remain the same when you return to the application. It's always important to consider what state you will need to maintain when navigating away from an `Activity`, as the `Activity` itself may need to be reclaimed.

It's also a good idea to separate the state of an `Activity` from the state of the application. While an `Activity` is busy editing a record, that record's data should remain encapsulated within the `Activity`.

Try it yourself

You've isolated the category picker and the attachment logic into `Fragment` classes in this chapter; now try writing a Fragment to encapsulate the contents of the first `CardView` on the screen. Remember to rather push the `ClaimItem` down into the `Fragment` instead of having the `Fragment` push the change events up to the `Activity`. Name the new `Fragment` class `CaptureClaimDetailsFragment` and name its layout resource `fragment_capture_claim_details.xml`.

You can also try pushing the logic to change the `Category` of the `ClaimItem` down into the `CategoryPickerFragment` in a way similar to how the `AttachmentPagerFragment` automatically adds new `Attachments` to the `ClaimItem`.

Test your knowledge

1. When developing a layout subclass, which of the following options is the best?
    - Programmatically instantiating its child widgets
    - Only having ID attributes in nested child widgets
    - Avoiding assigning ID attributes to child widgets
2. Which of these applies to the `Bundle` passed at an `Activity` in `onCreate`?
    - It is populated in the `onSaveInstanceState` method
    - It is populated automatically by the platform
    - It is never null
3. When the data for an `Adapter` changes, which of the mentioned happens?
    - It will be detected by the `View` automatically
    - It should be replaced by a new `Adapter` to reflect the changes
    - It should notify any attached listeners
4. Fragments and `View` classes should meet which of the following condition?
    - They should have their data and state pushed into them from the `Activity`
    - They should expose listener interfaces that their `Activity` implements to receive events
    - They should directly call event methods on their `Activity` by casting it to the correct class

Summary

In this chapter, you learned some of the practical techniques for breaking your user interface and application into modular components that can be reused. It's always a good idea to start with the finished user interface and break it up, preferably from the mockup stage. It's also good to identify where some parts of the system can serve multiple roles, for example, being both a read-only display and an editor. It's also a good idea to wrap components within other components, even if it's just conceptually. Keeping certain types of event handlers as their own modules makes them reusable over screens that don't share exactly the same widgets, but need to reuse the same logic.

When building user interfaces, it's a good idea to use an `Activity` to just wrap a collection of `Fragment` rather than nesting the screen logic in the `Activity`. This will allow the `Fragment` to take on specific responsibilities (such as attachments), making them more reusable elsewhere in your application. It also allows you far more flexibility when providing different layouts for different screen sizes. Devices with large screens might actually have more `Fragment` on the screen than smaller devices.

As a general best practice, always try and contain data and state by pushing it downward (as you do when you pass parameters to a method). This avoids `View` classes and `Fragment` from needing to be placed within specific parts of your application, just like a method doesn't need to know where it is being invoked from to do its job. This approach makes it much easier to move parts of the application around later on.

In the next chapter, we'll look at an Android system that makes this sort of modularization easier, and even more flexible. The data binding system is an incredibly powerful system that takes care of keeping user interfaces populated with data, and allows much of the presentation work to be bound directly to the layout XML files.

5
Binding Data to Widgets

So far, you've been copying the data from your data model into your presentation layer by hand, and then copying it back as well. This shifting data back and forth between stateful widgets is something that you always need to do at some level. Where and how the data is copied can change, but it has to be done to make applications work. In this chapter, we'll look at a system provided by Android called data binding. **Data binding** provides an alternative to the copying back and forth of data, but also opens several other design opportunities to allow more reuse of code.

Data binding offers you a way to dramatically reduce the amount of boiler-plate code in your application, while remaining type-safe and providing excellent performance. The data binding engine allows you to provide user interface logic that is clearly separated from the layout resources, and can be easily reused by many screens in the application, while reducing the complexity of both the application code and the layout resource files.

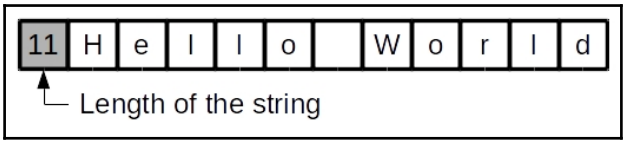
In this chapter, we'll look at the following topics:

- Why data binding exists
- How to write data-bound layouts
- How to use data binding in an MVP design
- Reactive programming and your data model
- How to use data binding in Activities, Fragments, and widgets

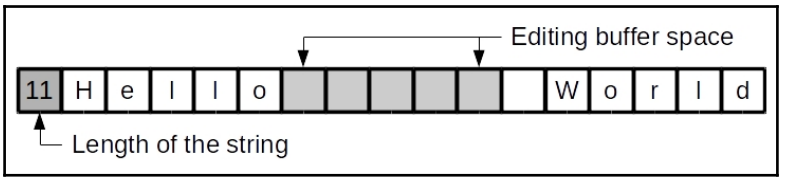
Exploring data models and widgets

In theory, the widgets can directly reference the memory that they are manipulating by holding points to the data, rather than copying the data back and forth, but more often than not, it doesn't make sense to use the same data format for storage and for editing.

Take strings of text for example; the best way to store a string is as a character array; whenever you need to send the text anywhere, over the network or to a display, you can simply read from the first character until the last one, and each one can be transmitted as-is. For example, "Hello World" can be stored as the string length followed by each of the characters:



This is not a good way to store a string that is being edited; however, for editing, it's best to have some buffer space around the cursor to avoid having to copy large amounts of data back and forth as the user types and corrects themselves. For example, if the user places their cursor right after the word "Hello", the same array might look like this:

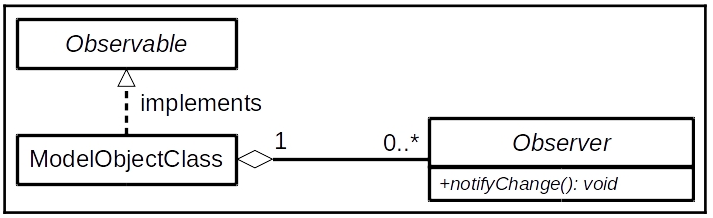


This tension between how data should be stored when it'll only be read, and how it should be edited, is an important part of why modern user-interface widgets tend to be complex pieces of machinery in their own right. They don't just need to look pretty; they need to be fast, and for that, they need to internally represent the data in a way that suits their implementation best. As a result, instead of being able to have an `EditText` widget just manipulate an array of characters, we're forced to copy the strings in and out of its internal structures, as you've been doing so by hand thus far.

The data binding system in Android allows you to reference your object model directly from your layout files, and then generate all the Java code required to wire the object model to the widgets. This system is called data-binding, and its core classes can be found in the `android.databinding` package. The data binding system also allows for *reactive programming*; when the data model is changed, it can directly reflect in the user interface widgets, allowing the application to keep what is on the screen up to date without having to explicitly update the widgets. The data binding system is also completely type-safe, because it generates all the code when your application is compiled, so any type errors are produced then and there, rather than possibly at runtime, where your users might see them.

The Observer pattern

The data binding framework in Android makes use of the **Observer pattern** to allow for *reactive programming*. Any object that is referenced by a layout file that implements the `Observable` interface is watched, and when it signals that it has changed, the user interface updates accordingly. As the data binding system can be used on any attribute or setter of any widget, this means that you can control far more than just the content or state of the user interface. You can control whether widgets are visible or invisible, and you can control which image is used for the background of a widget. At its core, the Observer pattern looks like this:



In the Android Observer pattern, the data model classes expose themselves as Observable by implementing the `android.databinding.Observable` interface and notifying a list of event-listeners (observers) of any changes to their state. Android provides several convenience classes that make implementing this pattern much easier. There are three ways in which you can implement this pattern for Android:

- Implement Observable in your object model
- Implement an Observable model on top of your object model
- Implement Observable in a presentation layer

Let's take a look at these three ways in detail:

- **Implementing Observable directly in your object model** is common, but has the side effect of polluting your object model with the Observable pattern and Android classes that will effectively stop you from using the same code base in other parts of the system (for example, on the server side). This is a good approach when your object model code will only ever be used by your Android application.
- **Implementing an Observable layer on top of the object model** is sometimes a better option, but can also lead to complications; every object referenced through the observable layer also needs to be wrapped in an Observable object. This leads to much greater complexity in the model implementation, and doesn't cover changes made outside of the Observable layer. This approach is useful when you are generating the code for your object model with a tool, or need an additional application-specific layer for your Android application code.
- **Implementing the Observer pattern at the presentation layer** means that the root references held by the data binding layer are themselves Observable, but the object model is not. This will technically allow you to have an immutable data model if you wanted to. The data binding engine will never see the changes to the individual fields in the data model, but instead is notified that the whole model has changed. This can also be a very expensive model, as the data binding layer will reevaluate every part of the data model for every change made to it. However, this is a good approach when your application tends to update several fields in a model simultaneously, or is heavily multithreaded.

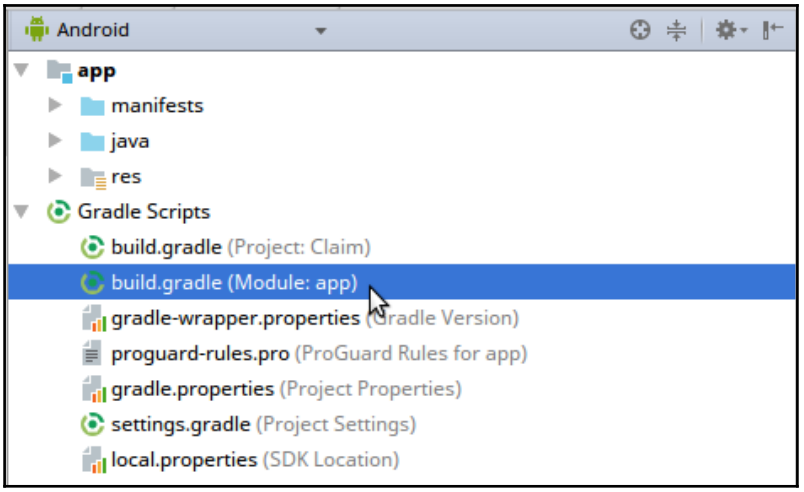
None of these options are always better than the others; rather, it's worth considering each of them when it comes to ensuring that your user interface will stay in-sync with the application's overall state. In some screens, this reactive behavior may even be undesirable, since it can easily disrupt the user. In these cases, it's worth using data binding just to populate the screen.

The data binding system is not bidirectional; changes in the model reflect in the user interface, but input in the user interface widgets are not pushed into the model automatically. This means that your application must still handle events and capture the changes in the user interface, as shown earlier.

Enabling data binding

By default in an Android project, the data binding capabilities are turned off. You'll need to enable them in your project's `build.gradle` file by hand. Follow these quick steps to enable the data binding system:

1. Start by locating the `build.gradle` file for your application module in the Android panel in Android Studio:



2. Open this file and locate the `android` block:

```
android {
  compileSdkVersion 26
  // ...
}
```

3. At the end of the `android` block, add the following snippet to enable data binding:

```
android {
 compileSdkVersion 26
   // ...
   dataBinding {
      enabled = true
   }
}
```

4. Once you save this file, Android Studio will open a banner at the top of the file, telling you that it needs to sync the project. Click on the **Sync Now** link on the right-hand side of the banner and wait for the sync to complete.

Congratulations! You've just enabled the data binding framework on your project. Now you can get started, making use of the data binding system in your layout files, which will simplify the application and open doors to new ways to reuse your code base.

Data binding a layout file

Data binding works primarily through code generation, and there is very little in the way of runtime overhead. It permits you to use a special expression language in your layout XML files, which gets converted into Java code before your application is compiled. These expressions can call methods, access properties, and are even useful for triggering events. They do have a few restrictions, however: they cannot directly reference the widgets in your user interface, and they cannot create any new objects (they have no `new` operator). As a result, you'll need to provide your layout files with some utility methods in order to keep things simple, and there are a few guidelines to follow when working with expressions:

- **Keep the expressions simple**: Don't write application logic into the expressions; rather, create a utility method that can be reused
- **Avoid manipulating data directly**: As tempting as it might be, ensure that your data is always ready for presentation before it's given to the layout binding. Keep default values in your model, `Activity`, or `Fragment` classes, and not in the layout XML files
- **Use presenter objects**: When you have simple transformations that need to be done on the data (such as formatting a date or number), put these into objects. The expression language can reference static methods, but presenter objects are much more powerful and flexible
- **Pass events in**: Avoid using the expressions language for more than a method call when you're writing events, and try to pass the events into the layout as objects, either as a presenter, or as command objects. This keeps the events flexible and reusable

By sticking to these guidelines, you'll find that working with the data binding system not only frees you from some of the most common user interface boilerplate code, but also improves the quality of your layouts and overall application. By using objects in your layout files rather than static methods, you'll end up with modular classes that can be easily reused throughout your application.

Now that your app can capture people's expenses as claims, it's time to start thinking about how the information will be displayed. There are two major components to this: the list of claim items the user has created, and their overall travel allowance that they're supposed to keep to. So far, you have the capture screen, and while in many ways it's the most important screen in the app, it's not the first one that the user will see--that will be the overview screen.

The overview screen's main job is to display the claim items in order, from most recent to the oldest. However, to keep the user's life simple, we'll also display a summary card at the top of the screen, which will help them stay on track with their spending. For this example, we'll assume that the allowance is specified as an amount per day that they are traveling.

Creating an Observable model

To get things started on this part of the project, you'll need a new model class to encapsulate the allowance and spending of the user. We'll call the new class `Allowance`, and build in some utility methods to fetch useful information (such as how much the user spent between two dates). Most importantly, this new model needs to tell us when it changes. This can technically be done in several ways: through an event-bus, or specialized listeners, but for this example, we'll go with an Observer pattern. To make this work, the `Allowance` class will extend from `BaseObservable`, a convenience class that is part of the data binding API. Whenever the `Allowance` class changes, it'll emit events notifying its observers of the change. Let's get started building the `Allowance` class:

1. Right-click on the `model` package and select **New| Java Class**.
2. Name the new class `Allowance`.
3. Change the **Superclass** to `android.databinding.BaseObservable`.
4. Add `android.os.Parcelable` to the **Interfaces** field.
5. Click **OK** to create the new class.

6. At the top of the class, declare the following fields and constructors, and a getter method for the `amountPerDay`, which represents the allowance the user is aiming for:

```
private int amountPerDay;
private final List<ClaimItem> items = new ArrayList<>();

public Allowance(final int amountPerDay) {
    this.amountPerDay = amountPerDay;
}

protected Allowance(final Parcel in) {
    amountPerDay = in.readInt();
    in.readTypedList(items, ClaimItem.CREATOR);
}

public int getAmountPerDay() { return amountPerDay; }
```

7. Now comes the first bit of the Observable implementation; when we change the `amountPerDay` field, we need to notify any Observers that the `Allowance` object has changed:

```
public void setAmountPerDay(final int amountPerDay) {
    this.amountPerDay = amountPerDay;
    notifyChange();
}
```

8. The `Allowance` class will always ensure that all the `ClaimItem` objects are sorted from the newest to the oldest; knowing this, we can add some convenience methods to find the *start* and *end* dates for the `Allowance` object:

```
public Date getStartDate() {
  return items.get(items.size() - 1).getTimestamp();
}
public Date getEndDate() {
  return items.get(0).getTimestamp();
}
```

9. Now, create a simple calculation method to determine how much has been spent in total for this `Allowance`. This method simply adds up all the amounts in all the `ClaimItem` objects:

```
public double getTotalSpent() {
    double total = 0;

    for (final ClaimItem item : items)
```

```
            total += item.getAmount();

        return total;
    }
```

10. Then, add another calculation method to calculate the amount spent between two dates. This can be used to find out how much was spent on a specific day, week, month, and so on:

```
public double getAmountSpent(final Date from, final Date to) {
   double spent = 0;
    for (int i = 0; i < items.size(); i++) {
        final ClaimItem item = items.get(i);
        if (item.getTimestamp().compareTo(from) >= 0
                && item.getTimestamp().compareTo(to) <= 0) {
            spent += item.getAmount();
        }
    }

    return spent;
}
```

11. Now, you'll need a method to add a `ClaimItem` to the `Allowance`. The `Allowance` always maintains the list of `ClaimItem` objects sorted from newest to oldest, so this method simply sorts the list each time an item is added, and then notifies observers that the `Allowance` has changed:

```
public void addClaimItem(final ClaimItem item) {
   items.add(item);
   Collections.sort(
     items,
     Collections.reverseOrder(new Comparator<ClaimItem>() {
       @Override
       public int compare(final ClaimItem o1, final ClaimItem o2) {
         return o1.getTimestamp().compareTo(o2.getTimestamp());
       }
     })
   );

   notifyChange();
}
```

Sorting a list like this is a very poor implementation, but very simple to write. In practice, you should binary-search for the correct position to add the `ClaimItem`. Android provides classes to help with this, which we'll explore later in the book.

12. We also need to be able to remove `ClaimItem` objects from the `Allowance`. This is also a mutative operation, so we notify any observers when it's done:

```
public void removeClaimItem(final ClaimItem item) {
  items.remove(item);
  notifyChange()
}
```

13. Add the accessor methods for the `ClaimItem` objects:

```
public int getClaimItemCount() {
  return items.size();
}
public ClaimItem getClaimItem(final int index) {
  return items.get(index);
}
public boolean isEmpty() {
  return items.isEmpty();
}
```

14. Finish the `Allowance` class by writing its `Parcelable` implementation:

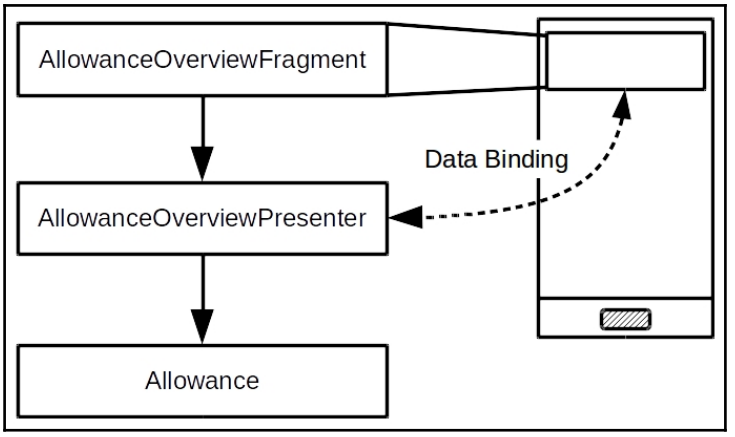
```
@Override
public void writeToParcel(Parcel dest, int flags) {
  dest.writeInt(amountPerDay);
  dest.writeTypedList(items);
}

@Override
public int describeContents() { return 0; }

public static final Creator<Allowance> CREATOR = new
Creator<Allowance>() {
  @Override
  public Allowance createFromParcel(Parcel in) {
    return new Allowance(in);
  }

  @Override
  public Allowance[] newArray(int size) {
    return new Allowance[size];
  }
};
```

The `Allowance` class is the first (and currently, the only) part of your object model that needs to be observed, as you can see; building an `Observable` model is not difficult, and being able to watch your model state for changes opens some fantastic opportunities, such as automatic network synchronization or statistics aggregation.

If you have an event bus in your application, pushing object model changes through there rather than direct observation is often a better option as it will offer better decoupling. There are a large number of event-bus APIs compatible with Android, and it's worth checking them out. A well-known API with an Event Bus implementation is Google's Guava API (`https://github.com/google/guava`).

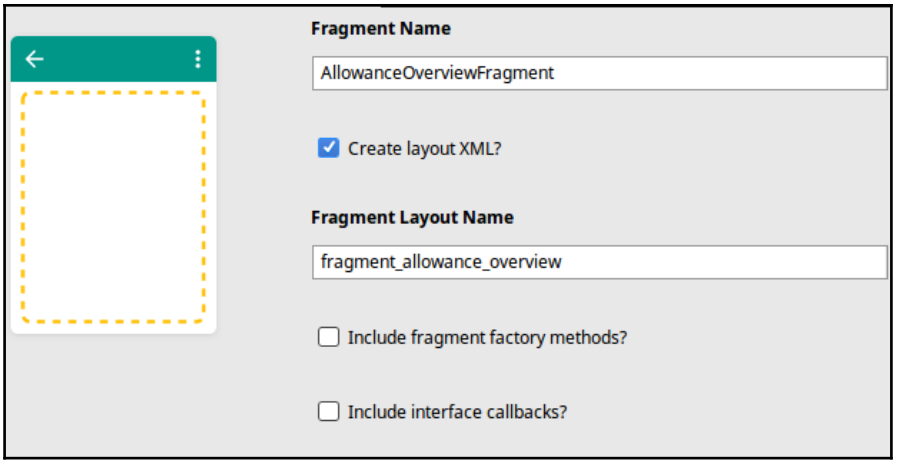
Establishing the AllowanceOverviewFragment

The allowance overview will be presented as a card at the top of an overview screen. The overview card will be populated by a new `Fragment` class that will encapsulate the first part of the data binding. The `AllowanceOverviewFragment` will depend on the data binding system to do most of the heavy lifting, and will provide the layout binding with a special `AllowanceOverviewPresenter` object that can be queried for statistics and data. The `AllowanceOverviewPresenter` will, in turn, reference the `Allowance` object, and listen for any changes on it in order to update and cache the statistics data. The relationship between these entities can be best explained with the following diagram:



Encapsulating the statistics in a `Fragment` means that it's easier to include in other layouts that might include different information to the overview screen. Follow these quick steps to create the `AllowanceOverviewFragment` and `AllowanceOverviewPresenter` skeleton:

1. Right-click on the `ui` package and select **New | Fragment |Fragment (Blank)**.
2. Name the Fragment `AllowanceOverviewFragment`.
3. Turn off the **Include fragment factory methods?** and **Include interface callbacks?** options:



4. Click on **Finish** to create the new `Fragment` and its default layout file.
5. Right-click on the `ui` package again and select **New| Java Class**.
6. Name the new class `presenters.AllowanceOverviewPresenter`.
7. Click **OK** to create the new package and class.
8. The first thing the `AllowanceOverviewPresenter` needs is an inner class to hold the cached spending statistics that will be displayed to the user. This will be an immutable structure; when the statistics change, we refresh all of them at the same time:

```
public static class SpendingStats {
   public final int total;
   public final int today;
   public final int thisWeek;
   SpendingStats(
           final int total,
           final int today,
           final int thisWeek) {
       this.total = total;
```

```
            this.today = today;
            this.thisWeek = thisWeek;
        }
    }
```

You'll note that the fields in the `SpendingStats` class are `public final`, and have no getters. When dealing with data binding, the coupling is typically very tight, so introducing getter methods can actually create more complexity. It's better to avoid getter methods until they're needed.

9. We need to expose the `SpendingStats` outside of the class in such a way that the data binding will watch for changes. Android data binding, again, has a helper class; when you have a field that needs to be observed, you can use the `ObservableField` class. When an expression in the data binding layout file references one of these, it will automatically listen for changes and reevaluate when the field is changed:

```
public final ObservableField<SpendingStats> spendingStats = new
ObservableField<>();
```

When using `ObservableField` (and its cousins: `OvservableString`, `ObservableInt`, and such), it's best to declare them as `final` and initialized. The data binding system can't watch for changes on the field itself, but instead, will attach a listener to the `ObservableField` object.

10. The `AllowanceOverviewPresenter` also requires an `Allowance` object that it will encapsulate, and a constructor:

```
public final Allowance allowance;
public AllowanceOverviewPresenter(final Allowance allowance) {
   this.allowance = allowance;
}
```

11. Finally, the `AllowanceOverviewPresenter` needs a method to allow the user to update the amount they're permitted to spend each day. In this case, the presenter acts as a helper to keep some of the logic out of the layout files; the `EditText` widget will provide a number as a `CharSequence`, so `AllowanceOverviewPresenter` needs to parse it and handle any errors if it's invalid in some way:

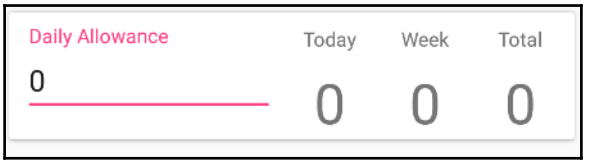
```
public void updateAllowance(final CharSequence newAllowance) {
  try {
    allowance.setAmountPerDay(
        Integer.parseInt(newAllowance.toString()));
```

```
    } catch (final RuntimeException ex) {
      //ignore
      allowance.setAmountPerDay(0);
    }
  }
```

The `AllowanceOverviewPresenter` class will serve as an intermediary system between the raw data-bound layout file, and the raw object model. This allows you to keep any rendering logic out of the object model, while also keeping the data model requirements out of the layout XML file.

Creating the AllowanceOverview layout

Now, it's time to create the layout file and bind it to the `AllowanceOverviewPresenter` class. A data-bound layout file is a little different from a normal Android layout. As each layout XML file results in its own binding class, they have a root element of `layout`, followed by a `data` section that declares the variables that they will bind to. Each variable is named and typed with its Java class, because during compilation, these are all turned into Java variables in a generated binding class. Ultimately, you want to create a layout that will look like this at the top of the overview screen:



The **Daily Allowance** field will allow the user to directly edit how much they are allocated per day, while the labels to the right will display their spending today, this week, and in total. Follow these steps to construct the preceding layout; unlike previous examples, these steps don't use the **Design** view for editing, and the layout is built from right to left:

1. Open the `fragment_allowance_overview.xml` layout file.
2. Change the editor to **Text** mode.
3. Change the root element from `FrameLayout` to layout, and remove the contents:

```
<layout xmlns:android="http://schemas.android.com/apk/res/android"
   xmlns:tools="http://schemas.android.com/tools"
   android:layout_width="match_parent"
   android:layout_height="match_parent"
   tools:context="com.packtpub.claim.ui.AllowanceOverviewFragment">
</layout>
```

4. Now, declare a data section within the `layout` and declare a presenter variable for the `AllowanceOverviewPresenter` class:

```
<layout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
tools:context="com.packtpub.claim.ui.AllowanceOverviewFragment">

  <data>
    <variable
        name="presenter"
        type="com.packtpub.claim.ui.
              presenters.AllowanceOverviewPresenter" />
  </data>
</layout>
```

5. Unlike the `data` section, the widget elements have no special root, so directly after the `data` section (and still nested within the `layout` element), declare the root element of this layout, which will be a `ConstraintLayout`:

```
<android.support.constraint.ConstraintLayout
   android:layout_width="match_parent"
   android:layout_height="match_parent">
</android.support.constraint.ConstraintLayout>
```

6. Within the `ConstraintLayout`, create a `TextView` that will serve as the label with the word `Total` in it:

```
<TextView
   android:id="@+id/totalLabel"
   android:layout_width="0dp"
   android:layout_height="wrap_content"
   android:layout_marginTop="@dimen/grid_spacer1"
   android:gravity="center"
   android:text="@string/label_total"
   android:minWidth="@dimen/allowance_overview_label_min_width"
   android:textAppearance="@style/TextAppearance.AppCompat.Caption"
   app:layout_constraintEnd_toEndOf="@+id/total"
   app:layout_constraintStart_toStartOf="@+id/total"
   app:layout_constraintTop_toTopOf="parent" />
```

7. On the line specifying the `android:text` attribute, Android Studio will complain that the `@string/label_total` resource doesn't exist. Use the code assistance (usually *Alt* + *Enter*), and select **Create string value resource** '**label_total**'.

8. A dialog will open, prompting you for the resource value; enter `Total` and click on the **OK** button.
9. Use the same code assistance to create a dimension resource on the following line, specifying the minimum width. Give the new `allowance_overview_label_min_width` resource a value of `50dp` and click **OK**.
10. Below the **Total** label widget, create a `TextView` that will contain the actual amount of money the user has spent in their `Allowance`:

```
<TextView
   android:id="@+id/total"
   android:layout_width="wrap_content"
   android:layout_height="wrap_content"
   android:layout_marginEnd="@dimen/grid_spacer1"
   android:layout_marginTop="@dimen/grid_spacer1"
   android:gravity="center"
   android:minWidth="@dimen/allowance_overview_label_min_width"
android:textAppearance="@style/TextAppearance.AppCompat.Display1"
   app:layout_constraintEnd_toEndOf="parent"
   app:layout_constraintTop_toBottomOf="@+id/totalLabel" />
```

11. Note that here, you haven't specified an `android:text` attribute. This will be the first data-bound attribute in the layout file, and we want to display the total field of the `SpendingStats` object from the presenter. Write this `android:text` attribute into the `TextView` above the `app:layout_constraintEnd_toEndOf` attribute:

```
android:text='@{Integer.toString(presenter.spendingStats.total) ??
"0"}'
```

Data-bound expressions are all wrapped in `@{..}` to signal their difference from normal attributes. The code looks like Java, but it's not. Note the `??` operator; it's a very useful "null-safe" operator. If any part of the left-hand side is null, the right-hand side (in this case, the `"0"` string) will be used instead (like a very specific ternary operator). Also, make note of the single quotes around the `android:text` attribute; data-bound layouts must still be a valid XML file, and the preceding code needs to specify a Java string that uses double quotes. Rather than escaping the Java string as `&quot;0&quot;`, it's cleaner to use single quotes for the XML attribute.

Another important factor is how you need to use `Integer.toString` to ensure that the correct method is invoked on the `TextView`. Leaving it as an `int` will cause `TextView.setText(int)` to be invoked, and this expects a string-resource identifier.

12. Next, you'll need to declare very similar `TextView` elements for the weekly label and amount display. These are virtually identical to the total `TextView` elements, except for their labels, IDs, and constraints. You'll also need to create a `label_week` string resource with a value of `Week`:

```
<TextView
   android:id="@+id/weekLabel"
   android:layout_width="wrap_content"
   android:layout_height="wrap_content"
   android:layout_marginEnd="0dp"
   android:layout_marginTop="@dimen/grid_spacer1"
   android:gravity="center"
   android:minWidth="@dimen/allowance_overview_label_min_width"
   android:text="@string/label_week"
   android:textAppearance="@style/TextAppearance.AppCompat.Caption"
   app:layout_constraintEnd_toEndOf="@+id/week"
   app:layout_constraintStart_toStartOf="@+id/week"
   app:layout_constraintTop_toTopOf="parent" />

<TextView
   android:id="@+id/week"
   android:layout_width="wrap_content"
   android:layout_height="wrap_content"
   android:layout_marginEnd="@dimen/grid_spacer1"
   android:layout_marginTop="@dimen/grid_spacer1"
   android:gravity="center"
   android:minWidth="@dimen/allowance_overview_label_min_width"
android:text='@{Integer.toString(presenter.spendingStats.thisWeek)
?? "0"}'
android:textAppearance="@style/TextAppearance.AppCompat.Display1"
   app:layout_constraintEnd_toStartOf="@+id/total"
   app:layout_constraintTop_toBottomOf="@+id/weekLabel" />
```

13. You'll need to repeat the same for the **Today** numbers. Again, you'll want to change the labels, IDs, and constraints, and create a `label_today` string resource with a value of `Today`:

```
<TextView
   android:id="@+id/todayLabel"
   android:layout_width="wrap_content"
   android:layout_height="wrap_content"
   android:layout_marginEnd="0dp"
   android:layout_marginTop="@dimen/grid_spacer1"
   android:gravity="center"
   android:minWidth="@dimen/allowance_overview_label_min_width"
   android:text="@string/label_today"
   android:textAppearance="@style/TextAppearance.AppCompat.Caption"
   app:layout_constraintEnd_toEndOf="@+id/today"
   app:layout_constraintStart_toStartOf="@+id/today"
   app:layout_constraintTop_toTopOf="parent" />

<TextView
   android:id="@+id/today"
   android:layout_width="wrap_content"
   android:layout_height="wrap_content"
   android:layout_marginEnd="@dimen/grid_spacer1"
   android:layout_marginTop="@dimen/grid_spacer1"
   android:gravity="center"
   android:minWidth="@dimen/allowance_overview_label_min_width"
   android:text='@{Integer.toString(presenter.spendingStats.today)
?? "0"}'
android:textAppearance="@style/TextAppearance.AppCompat.Display1"
   app:layout_constraintEnd_toStartOf="@+id/week"
   app:layout_constraintTop_toBottomOf="@+id/todayLabel" />
```

14. The last element of this card is the daily allowance input area, where the user can enter how much they are allowed to spend each day. It consists of a `TextInputLayout` and a `TextInputEditText` widget bound to the amount per day. In this element, you'll also be binding the `TextInputEditText` widget to an event handler, which looks a lot like a Java lambda, but like all the binding expressions, it's not. However, it is translated into Java:

```
<android.support.design.widget.TextInputLayout
  android:id="@+id/textInputLayout"
  android:layout_width="0dp"
  android:layout_height="0dp"
  android:layout_marginEnd="@dimen/grid_spacer1"
  android:layout_marginStart="@dimen/grid_spacer1"
  android:layout_marginTop="@dimen/grid_spacer1"
```

```
    app:layout_constraintBottom_toBottomOf="@+id/today"
    app:layout_constraintEnd_toStartOf="@+id/today"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent">

    <android.support.design.widget.TextInputEditText
      android:layout_width="match_parent"
      android:layout_height="wrap_content"
      android:hint="@string/label_daily_allowance"
      android:inputType="number"
      android:onTextChanged=
             "@{(text, start, before, end)
                -> presenter.updateAllowance(text)}"
      android:text='@{presenter.allowance.amountPerDay > 0 ?
             Integer.toString(presenter.allowance.amountPerDay) :
""}' />
    </android.support.design.widget.TextInputLayout>
</android.support.constraint.ConstraintLayout>
```

15. Use the Android Studio code assistant to create the `label_daily_allowance` string resource with a value of `Daily Allowance`.

Now, if you go back to the **Design** mode, you'll be able to see what your new fragment will look like on the screen of the user's device. The event-handler is hooked up and will be triggered every time the user changes any text in the **Daily Allowance** input box. The event trigger will invoke the `presenter.updateAllowance` method, which in turn will attempt to parse the value and set it on the `Allowance` object (assuming that it can be parsed as an integer).

Updating the SpendingStats class

You've already created the `SpendingStats` class and bound it to your layout, but it won't ever have any data in it because it's never actually created, and the `ObservableField<SpendingStats>` field in the `AllowanceOverviewPresenter` is never populated. There's a good reason for that--the stats take time to calculate. Even if we had a database to do the heavy lifting, there is potentially a substantial overhead to calculate these three numbers before you can put them on the screen, while you can directly invoke the `Allowance.getTotalSpent()` method as part of your layout XML that will block the main thread for the entire time it took to calculate that number. That's not a good idea, as that time delay can quickly add up and lead to degraded user experience or even **Application Not Responding** errors.

The answer is to listen for changes to the `Allowance` object, and recalculate the values on a worker thread before updating the `SpendingStats` field in the `AllowanceOverviewPresenter`. The data binding system will take care of the rest and populate the values on the screen. There are two structures that are needed for this part of the example: an observer to watch for any changes on the `Allowance` object, and an `ActionCommand` to calculate and update the `SpendingStats` in the `AllowanceOverviewPresenter`. Let's create them:

1. Open the `AllowanceOverviewPresenter` source file in Android Studio.
2. At the bottom of the `AllowanceOverviewPresenter` class, start a new `ActionCommand` inner class to update the `SpendingStats`, named `UpdateSpendingStatsCommand`:

```
private class UpdateSpendingStatsCommand
    extends ActionCommand<Allowance, SpendingStats> {
```

3. The `UpdateSpendingStatsCommand` will need two utility methods to calculate the date ranges for *this week*, and *today*. Unfortunately, Android doesn't support the new Java 8 time APIs; you'll need to use the `Calendar` class. On the other hand, Android provides a very useful utility class named `Pair`, that is perfect for defining a date range:

```
Pair<Date, Date> getThisWeek() {
  final GregorianCalendar today = new GregorianCalendar();
  today.set(
      Calendar.HOUR_OF_DAY,
      today.getActualMaximum(Calendar.HOUR_OF_DAY));
  today.set(
      Calendar.MINUTE,
      today.getActualMaximum(Calendar.MINUTE));
  today.set(
      Calendar.SECOND,
      today.getActualMaximum(Calendar.SECOND));
  today.set(
      Calendar.MILLISECOND,
      today.getActualMaximum(Calendar.MILLISECOND));

  final Date end = today.getTime();

  today.add(
      Calendar.DATE,
      -(today.get(Calendar.DAY_OF_WEEK) - Calendar.SUNDAY));

  today.set(Calendar.HOUR_OF_DAY, 0);
  today.set(Calendar.MINUTE, 0);
```

```
    today.set(Calendar.SECOND, 0);
    today.set(Calendar.MILLISECOND, 0);

    return new Pair<>(today.getTime(), end);
  }

  Pair<Date, Date> getToday() {
    final GregorianCalendar today = new GregorianCalendar();
    today.set(
        Calendar.HOUR_OF_DAY,
        today.getActualMaximum(Calendar.HOUR_OF_DAY));
    today.set(
        Calendar.MINUTE,
        today.getActualMaximum(Calendar.MINUTE));
    today.set(
        Calendar.SECOND,
        today.getActualMaximum(Calendar.SECOND));
    today.set(
        Calendar.MILLISECOND,
        today.getActualMaximum(Calendar.MILLISECOND));

    final Date end = today.getTime();

    today.add(Calendar.DATE, -1);
    today.set(Calendar.HOUR_OF_DAY, 0);
    today.set(Calendar.MINUTE, 0);
    today.set(Calendar.SECOND, 0);
    today.set(Calendar.MILLISECOND, 0);

    return new Pair<>(today.getTime(), end);
  }
```

You'll find that there are two different `Pair` implementations available to your application. One is part of the core Android platform (`android.util.Pair`), and the other is provided by the support packages (`android.support.v4.util.Pair`). The support implementation is intended for applications targeting API version 4 and lower, and your application is targeting API version 16 and higher; so, you should use the platform (`android.util.Pair`) implementation.

4. Then, you need to implement the `onBackground` method to process the data in the `Allowance` object into the `SpendingStats`:

```
public SpendingStats onBackground(final Allowance allowance)
      throws Exception {
  final Pair<Date, Date> today = getToday();
  final Pair<Date, Date> thisWeek = getThisWeek();
  // for stats we round everything to integers
  return new SpendingStats(
      (int) allowance.getTotalSpent(),
      (int) allowance.getAmountSpent(today.first, today.second),
      (int) allowance.getAmountSpent(thisWeek.first,
thisWeek.second)
  );
}
```

5. Then, the `UpdateSpendingStatsCommand` needs its `onForeground` to set the `SpendingStats` field on the `AllowanceOverviewPresenter`, which will cause the user interface to update with the new data:

```
public void onForeground(final SpendingStats newStats) {
   spendingStats.set(newStats);
}
```

6. That completes the `UpdateSpendingStatsCommand`; now, in the `AllowanceOverviewPresenter` class, you'll need an instance of the `UpdateSpendingStatsCommand` that you can invoke when the `Allowance` object changes:

```
private final UpdateSpendingStatsCommand updateSpendStatsCommand
                                  = new UpdateSpendingStatsCommand();
```

7. Then, you need `AllowanceOverviewPresenter` to be able to watch for changes to the `Allowance` object. This will involve an observer that Android's data binding API calls an `OnPropertyChangedCallback`. The problem is that `OnPropertyChangedCallback` is a class and not an interface, so for the `AllowanceOverviewPresenter`, use an anonymous-inner class for the `OnPropertyChangedCallback`:

```
private final Observable.OnPropertyChangedCallback
    allowanceObserver = new Observable.OnPropertyChangedCallback()
{

   public void onPropertyChanged(
       final Observable observable,
```

```
            final int propertyId) {
          updateSpendStatsCommand.exec(allowance);
        }
    };
```

8. The `AllowanceOverviewPresenter` needs to connect the observer to the `Allowance` object in its constructor:

```
public AllowanceOverviewPresenter(final Allowance allowance) {
   this.allowance = allowance;
   this.allowance.addOnPropertyChangedCallback(allowanceObserver);
}
```

9. References held by `Observable` objects to their observers are strong references, so if care isn't taken, you can find yourself with memory leaks. To avoid that, it's a good idea to detach the listener when the `AllowanceOverviewPresenter` will no longer be needed; however, this will need to be done from outside:

```
public void detach() {
   allowance.removeOnPropertyChangedCallback(allowanceObserver);
}
```

Most of the code for the `UpdateSpendingStatsCommand` is taken up by the date-range calculations; it's otherwise a very simple class. The important aspects are that it both encapsulates the calculations, and runs them on a background worker thread that keeps the user interface running smoothly while it adds up the numbers.

Data binding and fragments

When working with the data binding framework, it's important to put some additional thought into where to encapsulate the various parts of your user interface. As you can hook the logic directly into the layout files, it will often be a better idea to use classes similar to the `DatePickerWrapper` you wrote in `Chapter 3`, *Taking Actions*, with an `<include>` and `<merge>` tag, rather that wrapping groups of components in classes. Data-bound layouts that are included in other layouts still have variables, and it's the responsibility of the outer layout to pass those variables downward, into the included layout file. For example, a layout including a date picker might look something like this:

```
<include layout="@layout/merge_date_picker"
         app:date="@{user.dateOfBirth}"
         android:layout_width="match_parent"
         android:layout_height="wrap_content"/>
```

As long as either `user` or its `dateOfBirth` is `Observable`, the layout will automatically reflect any changes to it. This pattern allows you to not only modularize your layouts, but also ensure that they only receive the data that they actually require in order to work. The other advantage is that using it with `<merge>` elements plays very nicely with the `ConstraintLayout`, allowing you to build complex and reusable layout elements that nest in your code, but are flat (not nested) in the component hierarchy. Flat layouts using `ConstraintLayout` are often easier to build, are typically faster to render, and allow for more flexible animations than deeply nested layouts. They can be harder to modularize for reuse; data-binding makes this much easier.

If you're wondering whether it's a good idea to still introduce fragments and widget classes, look at logical boundaries, where you will be forced to nest your components. An excellent example of a good boundary is a `CardView`. A `CardView` requires a nested layout, so its contents are perfect candidates for a widget or fragment, which can further help with encapsulation of your layout and logic.

It's also important to consider your "presenter" classes and objects when you build them. A single layout can have any number of variables, and the presenter classes don't have to be shallow structures. It's common to build presenter classes in levels by inheritance, where you might build an application level presenter with global rules (how to format dates and numbers), with children for things like displaying dialogs; remember that some of the logic might not be used directly by the layout, but rather by an event handler method. Breaking the presenter classes up in this way allows you to further confine logic to where it's needed, and improve your code reusability.

Test your knowledge

1. Android's data binding framework follows what sort of binding?
    - Model-View View-Model (bidirectional) binding
    - Model-View-Presenter pattern
    - Model-View (unidirectional) binding
2. Data Bound Layouts have variables that must be which of the following?
    - Any Java Object
    - Observable by the data binding framework
    - Presenter objects
    - Model objects

3. Which of the following features belongs to data binding expressions?
    - They must be written in single quotes
    - They are Java expressions
    - They are a special expression language
    - They're only evaluated at runtime
4. To trigger an update of a data-bound user interface, you must do which of these?
    - Listen for object model changes with an event bus
    - Extend the `PropertyChangeCallback` class
    - Call refresh on the generated `Binding` object
    - Make a change that the `Binding` object can observe

Summary

Data binding can not only massively reduce the amount of boilerplate code required to write a user interface, but can actively improve your code base and increase how much code you can reuse. By avoiding complex binding expressions and encapsulating the display logic in your presenter classes, you can build highly modular layouts that are fast, type-safe, and reusable.

It's sometimes useful to think of the data-bound layout files as Java classes in their own right; after all, they will each result in a generated Binding class. It's useful to keep in mind that the Binding classes themselves are also observable, so any changes to them through their generated setter methods will automatically trigger an update in the user interface as well. Also, remember that when you include a data-bound layout in another, you need to pass all of its variables downward, which is just like specifying arguments on a constructor, and those variables don't need to be directly contained within the parent layout.

So far, you've been building an in-memory data model, but this means that when your application is terminated, all the data is lost. In the next chapter, we'll take a look at long-term data storage on Android, and find out how to integrate it with your user interface without degrading the user experience and perceived performance.

6
Storing and Retrieving Data

Data storage might not, at first glance, seem to be at all related to a user interface, but in the majority of applications, the user interface exists to manipulate long-lived data both on the device and on the network. This means that while it does not directly influence the look of the application, it does influence the user experience. Users expect the application to always reflect the latest data available to them, as we explored in `Chapter 5`, *Binding Data to Widgets*. Applications written using a reactive pattern ensure that the user interface is always up to date with the latest data available to the application, and the Android data binding system helps to make writing reactive applications easy. Even without the data binding framework, Android itself has always been built for reactive applications from the very bottom layers upward, but until recently, this behavior required huge amounts of boilerplate code.

When you develop any sort of application, it's important to establish a data container or authority within the application. In most web systems, this will be a database. The system may have many other layers to its data storage, such as caches and in-memory object models, but the *authority* in this case will be the database. Android applications may appear more complex at first; you typically have a server with some data, you often have a local database, and then there is also whatever is on screen and in-memory. Keeping all of these states in-sync might appear to be a nightmare, but it's actually well taken care of.

The Android team have built a collection of APIs known collectively as the Architecture Components. These collectively simplify the job of writing reactive applications by taking care of the most common problems when writing an application. They include APIs for storing and retrieving data, and for reacting to changes in state of the application.

In this chapter, we'll look at the following topics:

- How data and storage can influence the user experience
- The tools Android provides to store and retrieve structured data
- The best ways to keep the user interface up to date with the data store
- Building an SQLite database store using the Room persistence API

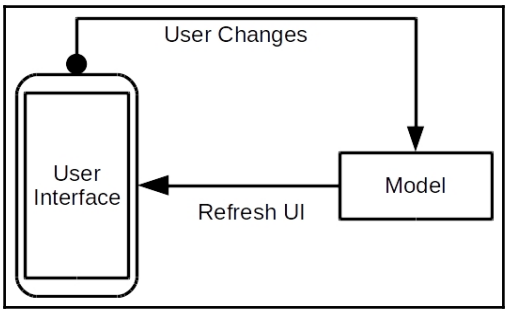
Data storage in Android

Almost every application needs to store data persistently at some point. Any data that needs to remain intact when your application is stopped must be placed in some kind of data storage system where you can retrieve it again later. You can store all the data on the server, but then your application will only function when the user has an active internet connection and will only ever be as fast as their available connection. You can also store data as files on the device's local filesystem, but this means you need to either load all the data into memory and save the whole application state every time it changes, or you need to write complicated logic to maintain integrity between the various files your application will write.

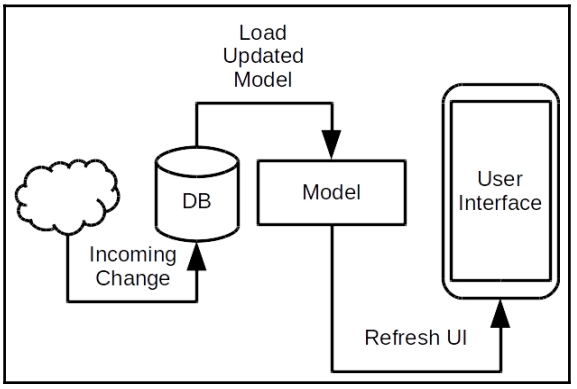
The Android ecosystem has a large number of database systems available, the most popular of which is probably SQLite. Data can be saved in SQLite tables and retrieved through structured queries. This provides an ideal way to store all of your application data on the device, while only ever retrieving what the application needs. An SQLite database can be instructed to exactly which fields on which records you need to retrieve, and you can use indices to make this process very fast.

Persistent data storage and object mapping does come at a significant cost--as fast as a database lookup might be, it takes significantly longer than is acceptable on the main thread, where it creates delays in graphics rendering and event dispatching. So once again, you want data to be loaded from a background thread. This can create some additional challenges: how do you ensure that the data is always up to date, and doesn't get stagnated when it's involved in `Activity` life cycles and is being persisted and loaded from multiple storage systems? This can quickly get out of hand, but again, Android has a whole ecosystem of structures that are designed to keep things under control.

When creating an Android application, it's best to design it so that anything being edited by the user on the current `Activity` remains in a mutable in-memory model, as in the following diagram:



This design pattern will provide your application with good performance, while also allowing you to easily *cancel* changes by simply discarding the in-memory model that the user has been changing. When the user is viewing rather than editing data, a different approach is needed. When a user is looking at a screen such as their email inbox, or a chat conversation, they expect it to update without their interaction. When this is the case, it's better to follow a unidirectional data flow design, such as the one in this diagram:



The **Incoming Change** in this diagram can be from absolutely anywhere. It can be from another part of the application, or it can be from the network, or it can even be from another part of the screen the user is looking at. The important thing is that the database (**DB**) is always updated first, which then triggers the **Model** to be reloaded or updated, and that in turn triggers the **User Interface** to update. This is in opposition to a **Model** where the **User Interface** receives the incoming event and fetches the new data. Here, the **User Interface** will always receive the latest data directly.

Using the SQLite database

SQLite is an excellent little SQL compatible database that is embedded into the core Android system. This allows you to leverage a complete SQL database without having to ship one with your application (which will raise your code size dramatically). This makes it the most common tool for storing structured data on Android, but it's by no means the only option.

For many applications that require real-time synchronization with a server, people use Firebase Database. **Firebase** is a Google cloud product that includes a powerful document database that synchronizes its data in real time, all the way to the client. This means that an event is triggered on a client when any of its data is modified from outside, making it suitable for chat and messaging applications. However, tools such as Firebase require a large additional client-side API, tie your application to a service, and are very difficult to port your application away for later. Applications built with them may also violate privacy laws in some countries if the application were to store private information without first encrypting it on the client side. In these cases, you'll either need to set up your own synchronization system, or use a database with filtered real-time synchronization, such as the *CouchDB project* from Apache.

Most typically though, SQLite serves as an excellent choice for storing structured data on the client. It's flexible, very powerful, and very fast, and because it's baked into the Android platform, it doesn't add any direct size overhead to your application. Most Java developers will be used to JDBC when accessing an SQL database, and while Android does have JDBC support, the `android.database` and `android.database.sqlite` packages are the preferred methods of accessing the database and are much faster. Android also offers an additional layer of abstraction above the direct use of SQLite, which we'll explore next.

For more information about SQLite and how to get the most out of it, it's worth browsing the project's excellent documentation at `https://sqlite.org/`.

Introducing Room

Direct use of SQLite requires a huge amount of code dedicated to converting the SQLite structured data into Java objects, and then preparing SQL statements to store those objects back into the database. Mapping an SQL record to a Java object normally takes the following form:

```
public Attachment selectById(final long id) {
   final Cursor cursor = db.query(
           "attachments",
           new String[]{"file", "type"},
           "_id=?",
           new String[]{Long.toString(id)},
           null, null, null);

   try {
       if (cursor.moveToFirst()) {
           return new Attachment(
                   new File(cursor.getString(0)),
                   Attachment.Type.valueOf(cursor.getString(1))
           );
       }
   } finally {
       cursor.close();
   }
   return null;
}
```

As you can immediately see, there's a lot of code there that you will need to repeat again and again for every one of your data-model objects.

Fortunately, Google has produced a solution to this boilerplate as part of their Architecture Components, and it's called **Room**. Room is an API and code generator that allows you to define your object model and the SQL queries you want to execute while it writes the boilerplate **Data Access Objects** (**DAO**) classes for you. Room is an excellent choice because all the heavy lifting is done at compile time by generating source code for your application. This also means that it requires much less additional code to be included in your application, which helps keep your application smaller on the end user's device.

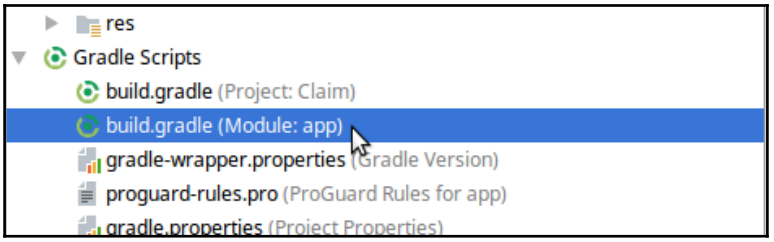
Room isn't a traditional **Object/Relational** (**O/R**) mapping layer, but rather allows you to define the `SELECT` statements, and copies whatever data they return to an object model that you specify. As a result, it doesn't directly handle inter-object relationships (such as `ClaimItem` containing an array of `Attachment` objects). While this may seem like a problem, it's a very important feature! Relationships like these are common in object models, but are expensive to implement in an Object/Relational layer, since every call to `ClaimItem.getAttachments` would require another database query, and on Android, those calls are likely to leak onto the main thread.

Instead, Room is designed so that you can create object models that are suitable for data-binding, and build SQL queries that can return them directly. This pushes the complexity back into the database, and helps encourage a single query to display programming behavior.

Adding Room to the project

Room is part of the Architecture Components, and is not imported into projects by default. Instead, you need to add them as a dependency to your project by following these simple steps:

1. In the Android panel, open the **Gradle Scripts** subsection and then open the `build.gradle` for the app module:



2. At the bottom of the file, you'll find a dependencies block; at the bottom of the block, add the following two lines of code:

```
implementation 'android.arch.persistence.room:runtime:+'
annotationProcessor 'android.arch.persistence.room:compiler:+'
```

3. Use the **Sync Now** link at the top of the editor to synchronize your project with its Gradle file. Android Studio will automatically download the new Room dependencies for your project.

4. Your project now has the Room API and its code generators integrated, and you can start creating a persistent object model and database schema.

Creating an Entity model

Room, much like an SQL database, is optionally asymmetric; what you write to it might not be in the exact same format as what you read from it. When you write to a `Room` database, you save `Entity` objects, but when you read, you can read virtually any Java object. This allows you to define object models that best suit your user interface, and load them with `JOIN` queries rather than resorting to one or more additional queries for each object you wish to present to the user. While `JOIN` queries might be overly expensive on a server, on a mobile device they are often significantly faster than a multiquery alternative. As such, when defining an entity model, it's worth considering what you will need to save in your database as well as what specific fields you will need on your user interface. The data you need to write to storage becomes your entity, while the fields on your user interface become fields in Java objects that can be queried through Room.

An `Entity` class in Room is annotated with `@Entity`, and is expected to follow certain rules:

- Fields must either be `public` or have Java Beans style getters and setters
- At least one field must be marked as a primary key using the `@PrimaryKey` annotation
- Room expects a single `public` constructor, so you may need to mark the other constructors with an `@Ignore` annotation in order for your code to compile. It's often best to leave only a default (no arguments) constructor for Room to use

In order to start storing claim data using Room, we will need to modify the existing `ClaimItem` and `Attachment` classes so that they are valid entities. This will involve making them usable as relational structures; `ClaimItem` and `Attachment` will both need an ID primary key, and attachment will need a foreign key identifier for the `ClaimItem` that it belongs to. Perform the following steps to modify these two data model classes so that they can be stored as entities using Room:

1. Start by opening the `ClaimItem` source file in Android Studio.
2. Annotate the class declaration with `@Entity`:

```
@Entity
public class ClaimItem implements Parcelable {
```

3. Add an ID field, annotate it with `@PrimaryKey`, and tell Room that you want it generated by the database rather than having to create IDs manually (you can also add getters and setters for this field if you like):

```
@PrimaryKey(autoGenerate = true)
public long id;
```

Leaving the fields `public` means that Room will directly access the fields in preference to using getters and setters. Field access can be much faster than method calls to the getters and setters.

4. Tell Room to ignore the `List` of `Attachment`. Room is unable to directly persist these sorts of relationships, and your application will fail to compile when it tries to generate mapping code for this field:

```
@Ignore List<Attachment> attachments = new ArrayList<>();
```

5. Modify the `Parcelable` implementation of `ClaimItem` to save and restore the ID field:

```
protected ClaimItem(final Parcel in) {
       id = in.readLong();
       description = in.readString();
       amount = in.readDouble();
       // ...
}

public void writeToParcel(final Parcel dest, final int flags) {
   dest.writeLong(id);
   dest.writeString(description);
   dest.writeDouble(amount);
   dest.writeLong(timestamp != null ? timestamp.getTime() : -1);
   dest.writeInt(category != null ? category.ordinal() : -1);
   dest.writeTypedList(attachments);
}
```

6. Open the `Attachment` source file.

7. Add the `Entity` annotation to the `Attachment` class; this time you'll need to include an `@Index` annotation as well to tell Room to generate a database index on a new field you'll be adding--`claimItemId`. The index will ensure that queries to fetch the attachments for a specific `ClaimItem` record are nice and fast:

```
@Entity(indices = @Index("claimItemId"))
public class Attachment implements Parcelable {
```

8. Add the database primary key field for the `Attachment`, and the new `claimItemId` field that will be used to indicate which `ClaimItem` the `Attachment` belongs to when it's stored in the database:

```
@PrimaryKey(autoGenerate = true)
public long id;
public long claimItemId;
```

9. Ensure that there is a `public` default constructor, and that any other `public` constructors are marked with `@Ignore`:

```
public Attachment() {}
@Ignore public Attachment(final File file, final Type type) {
    this.file = file;
    this.type = type;
}
```

10. Update the `Attachment` classes `Parcelable` implementation to include the new fields:

```
protected Attachment(final Parcel in) {
    id = in.readLong();
    claimItemId = in.readLong();
    file = new File(in.readString());
    type = Type.values()[in.readInt()];
}

public void writeToParcel(final Parcel dest, final int flags) {
    dest.writeLong(id);
    dest.writeLong(claimItemId);
    dest.writeString(file.getAbsolutePath());
    dest.writeInt(type.ordinal());
}
```

As you can see, modifying an existing object model to be stored in a `Room` database is very simple. Room will now be able to generate code to load and save these objects from tables in its database; it'll also be able to generate the database schema from these classes.

Creating the Data Access Layer

Now that you have something to write into the database, you need some way to actually write it, and a way to retrieve it again. The most common pattern is to have a dedicated class to deal with this for each class--a Data Access Object class, also known as a DAO. In Room, however, all you have to do is declare what they should look like using an interface; Room will write the implementation code for you. You define your queries using the `@Query` annotation on a method, like this:

```
@Query("SELECT * FROM users WHERE _id = :id")
public User selectById(long id);
```

This has a huge advantage over traditional O/R mapping layers in that you can still write any form of SQL query, and let Room figure out how to convert it into the object model you ask for. If it can't write the code, you get an error at compile time, rather than potentially having your app crash for your users. This also has an additional advantage: Room can bind your SQL queries to non-entity classes, allowing you to leverage the full power of your SQLite database without having to do all the column/field/object mapping by hand. For example, you can define a special `DisplayContact` class for displaying the summary data for a contact in a list, and then query them directly using a `join`:

```
@Query("SELECT contacts.firstname, contacts.lastname, emails.address FROM
contacts, emails WHERE emails._id = contacts.primaryEmailId ORDER BY
contacts.lastname")
public List<DisplayContact> selectDisplayContacts()
```

The preceding query doesn't return an object that can be directly saved in the database; it's the result of looking at two different tables and collecting fields from both of them. Room copes with this just fine though, and doesn't need any sort of annotations on the classes to be returned.

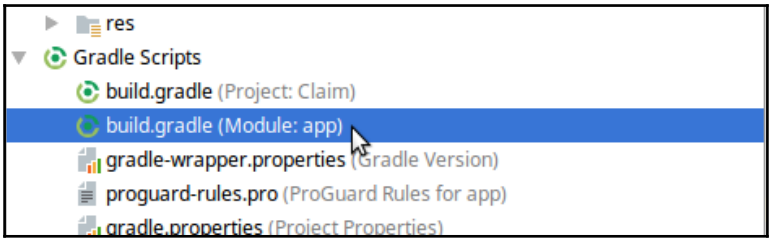
The LiveData class

Room does much more than binding your database structures to objects and back again; it offers you the ability to write reactive programs much more simply. As already mentioned, Room is one of the Android Architecture Components libraries. Architecture Components collectively provide generic infrastructure that can be used to more rapidly build reactive applications, while also maintaining excellent performance and safety. One of the most important classes in the architecture components is `LiveData`. `LiveData` is a generic encapsulation of data that is subject to external changes. `LiveData` can be observed, much like the classes used for data-bound layouts. The primary difference is that `LiveData` will always trigger a *first* event on any new observer, providing it with the current data state.

Room has built-in support for `LiveData`, meaning that you can return any object wrapped in a `LiveData` in order to receive any changes that occur to that object. At the time of writing, Room implements this by watching each of the tables for changes. This means you may receive updates to objects, even though they haven't actually changed. This should not be a problem for most applications, because the queries still run on a worker thread and only the notification happens on the main thread. This makes `LiveData` the preferred method for querying the database in most situations, because it takes care of running and processing the query on a worker, freeing your main thread to handle events and keep your application running smoothly.

`LiveData` is not part of Room directly, so you'll need to follow these steps to add `LiveData` and the other Architecture Components to your project:

1. In the Android panel, open the **Gradle Scripts** subsection and then open the **build.gradle** for the app module:



2. At the bottom of the file, you'll find a dependencies block; at the bottom of the block, add the following two lines of code:

```
implementation 'android.arch.lifecycle:runtime:+'
implementation 'android.arch.lifecycle:extensions:+'
annotationProcessor 'android.arch.lifecycle:compiler:+'
```

3. Use the **Sync Now** link at the top of the editor to synchronize your project with it's Gradle file, and download the new dependencies.

Implementing Data Access Objects in Room

You'll need to implement two different Data Access Object classes for the Claim application, one for each `Entity` object. Technically speaking, Room doesn't enforce one DAO per entity, and you can have a single DAO interface for your entire application or have one per screen. However, the most common pattern is to have one DAO class per entity type, even when some of its query methods return statistics or other views on the data. When working with more complex datasets, it's worth considering introducing additional DAO interfaces to cover queries that are either screen-specific or where the data overlaps several entities.

Here's how to implement the Data Access Object interfaces for the Claim example application, step by step:

1. Right-click on the `model` package in Android Studio and select **New | Java Class**.
2. Name the new class `db.ClaimItemDao`.
3. Change the **Kind** field to **Interface**. Room DAO types are normally interfaces, although this isn't a strict requirement, and they can also be abstract classes.
4. Click **OK** to create the new package and class.
5. Annotate the interface with `@Dao` to mark it as a data-access-object:

```
@Dao
public interface ClaimItemDao {
```

6. Declare a query method to fetch all the `ClaimItem` objects in order of the most recent first; ensure that it returns a `LiveData` so that the changes are reflected:

```
@Query("SELECT * FROM claimitem ORDER BY timestamp DESC")
LiveData<List<ClaimItem>> selectAll();
```

7. Next, you need methods to insert, update, and delete the `ClaimItem` objects in the database; these methods always take only the `Entity` object and instead of a query, are annotated with their operation. In the case of the insert method, it's useful to have it return the generated ID of the new record:

```
@Insert long insert(ClaimItem item);
@Update void update(ClaimItem item);
@Delete void delete(ClaimItem item);
```

8. Now, right-click on the `db` package again, and select **New | Java Class**.
9. Name the new class `AttachmentDao`, and make **Kind** to **Interface**.
10. Click **OK** to create the `AttachmentDao` class.
11. Declare the new interface as a `Dao`:

```
@Dao
public interface AttachmentDao {
```

12. Write a query method to fetch the `Attachment` objects for a single `ClaimItem`. This is where the index you declared on `Attachment` becomes important:

```
@Query("SELECT * FROM attachment WHERE claimItemId = :claimItemId")
LiveData<List<Attachment>> selectForClaimItemId(final long claimItemId);
```

13. Declare the insert, update, and delete methods for the `Attachment` classes, just as you did with the `ClaimItem` methods:

```
@Insert long insert(Attachment attachment);
@Update void update(Attachment attachment);
@Delete void delete(Attachment attachment);
```

Creating a database

When writing an application using Room, you'll need to define at least one *Database* class. Each of these corresponds to a specific database schema--a collection of Entity classes and the various ways in which they can be saved, and loaded from storage. It may also serve as a convenient place to write other database-related logic for your application. For example, the `ClaimItem` and `Attachment` classes need to save and load various types that Room will not understand; for example, `Date`, `File`, the `Category` enum, and `Attachment Type` enum. Each of these classes will need a `TypeConverter` method that can be used to convert it to and from primitives that are understood by Room.

Room Database classes are abstract. This is because they are extended by the Room annotation processor to produce the implementation you'll use at runtime. This allows you to define any number of concrete method implementations in a database class that might be useful for your application. Follow these steps to declare your new Room enabled database class:

1. Right-click on the `db` package in Android Studio, and select **New | Java Class**.
2. Name the new class `ClaimDatabase`, and change its **Superclass** to `RoomDatabase`.

3. Select the **Abstract** modifier.
4. Click **OK** to create the new class.
5. Annotate the class to indicate it as a database, and declare that it will store the `ClaimItem` and `Attachment` entities. You'll also need to specify the schema version, which will be `1` for the first version:

```
@Database(
        entities = {ClaimItem.class, Attachment.class},
        version = 1,
        exportSchema = false)
public abstract class ClaimDatabase extends RoomDatabase {
```

6. As mentioned earlier, you'll need to declare `TypeConverter` methods for all the non-primitive fields that `ClaimItem` and `Attachment` use. You need to tell the database where these methods can be found, and in this case, it'll be the `ClaimDatabase` class itself:

```
@Database(
        entities = {ClaimItem.class, Attachment.class},
        version = 1,
        exportSchema = false)
@TypeConverters(ClaimDatabase.class)
public abstract class ClaimDatabase extends RoomDatabase {
```

7. Now, define `abstract` methods to retrieve the Data Access Object implementations you created earlier; these methods will be implemented by the subclass generated by Room:

```
public abstract ClaimItemDao claimItemDao();
public abstract AttachmentDao attachmentDao();
```

8. Now, you'll need to tell Room how to convert the various fields to and from the primitives supported by the database. Start by implementing methods to convert `Date` objects into a timestamp long that can be stored in the database (SQLite has no `DATE` or `DATETIME` types):

```
@TypeConverter
public static Long fromDate(final Date date) {
    return date == null ? null : date.getTime();
}

@TypeConverter
public static Date toDate(final Long value) {
    return value == null ? null : new Date(value);
}
```

9. Now continue with this pattern for the other types that the `ClaimItem` and `Attachment` need:

```
@TypeConverter
public static String fromFile(final File value) {
    return value == null ? null : value.getAbsolutePath();
}

@TypeConverter
public static File toFile(final String path) {
    return path == null ? null : new File(path);
}

@TypeConverter
public static String fromCategory(final Category value) {
    return value == null ? null : value.name();
}

@TypeConverter
public static Category toCategory(final String name) {
    return name == null ? null : Category.valueOf(name);
}

@TypeConverter
public static String fromAttachmentType(final Attachment.Type value) {
    return value == null ? null : value.name();
}

@TypeConverter
public static Attachment.Type toAttachmentType(final String name) {
    return name == null ? null : Attachment.Type.valueOf(name);
}
```

The `TypeConverter` methods will be found and used by the Room annotation processor. They are invoked directly from the generated code, based on the types used in the Java classes being stored or retrieved. This means that they have almost no additional runtime overhead.

Accessing your Room database

So far, you've built all the components for a Room managed SQLite database, but you still don't actually have access to it. You can't instantiate the `ClaimDatabase` class directly because it's abstract, and you have the same problem with the DAO interfaces, so what's the best way to access the database? Room provides you with an entry class that will correctly instantiate the generated `ClaimDatabase` implementation, but that isn't the whole story; your entire application relies on this database, and it should be set up when the application starts and should be accessible by the entire application.

You can use a singleton `ClaimDatabase` object, but then where will the SQLite database file be placed? In order for it to be stored in your application's private space, you need a Context object. Enter the `Application` class, which when used, holds the first `onCreate` method that will be invoked in your application. Follow these quick steps to build a simple `Application` class that will instantiate and hold a reference to your `ClaimDatabase`:

1. Right-click on your root package (that is, `com.packtpub.claim`) and select **New | Java Class**.
2. Name the new class `ClaimApplication`.
3. Make its **Superclass** `android.app.Application`.
4. Click **OK** to create the application class.
5. Declare a static `ClaimDatabase` to be used by the application:

```
private static ClaimDatabase DATABASE;
```

6. Override the `onCreate` method and use it to instantiate the `ClaimDatabase` object using Room; this will happen before anything else in your application:

```
@Override
public void onCreate() {
    super.onCreate();
    DATABASE = Room.databaseBuilder(
            this,                   /* Context */
            ClaimDatabase.class,    /* Abstract Database Class */
            "Claims"                /* Filename */
    ).build();
}
```

7. Provide a `public static` method for other parts of the application to use, to access the singleton database instance:

```
public static ClaimDatabase getClaimDatabase() {
    return DATABASE;
}
```

8. You need to register the `ClaimApplication` with the Android platform so that it knows to initialize it when the application is started. You do this by opening the **manifests** directory and opening the `AndroidManifest.xml` file.
9. In the `<application>` element, you'll need to add an `android:name` attribute to tell the Android platform the name of the class that represents the root of your application:

```
<application
    android:name=".ClaimApplication"
    android:icon="@mipmap/ic_launcher"
    android:label="@string/app_name"
    android:roundIcon="@mipmap/ic_launcher_round"
    android:supportsRtl="true"
    android:theme="@style/AppTheme">
```

Now, whenever any part of your application needs the database, it can simply invoke `ClaimApplication.getClaimDatabase()` to retrieve a global instance, and because it's no longer tied to a specific context instance, it can be invoked from anywhere (even a presenter).

Test your knowledge

1. The Room API for Android provides which of the following?
    - A complete database solution
    - A lightweight API on top of SQLite
    - An object storage engine
2. Returning `LiveData` from a Room DAO requires that you do which of these?
    - You observe it for changes in order to retrieve data
    - You run the query on the main thread
    - You call the query method again when notified by the `LiveData` object

3. Database queries that don't return `LiveData` should do what?
    - Be avoided
    - Be run on a worker thread
    - Return `Cursor` objects
4. Writing an update method for Room requires which of the listed?
    - An `@Query("UPDATE` method on a DAO interface
    - An `@Update` method taking an `Entity` object on an interface
    - Will be added to your `Entity` implementations

Summary

The way you store and retrieve structured data in an Android application has a direct knock-on-effect on how your user will experience your application. When you choose to use a system like Room, CouchDB, or Firebase, where data changes are pushed through the application as updates, the user will naturally have a reactive application. What's more, the application will generally be responsive because these patterns naturally keep slow running queries and updates off the application main thread.

Room provides an excellent addition to the standard Android data storage ecosystem, not only dramatically reducing the need to write boilerplate data access code, but also providing a well-defined and excellently-written interface to run reactive queries for data. Of course, not all of your application needs to be reactive; once an object is delivered via a `LiveData` object, it's just an object and can be used as an in-memory snapshot or even edited if it's mutable.

When using Room, it's important to remember that you should avoid complex relationships between objects, because Room won't be able to save and resolve these for you. They're normally a sign that you might need to rethink how you are structuring your data; complex relationships will dramatically slow down the queries and therefore any user interface that depends on them. Typically, these relationships should be handled by creating presentation-specific object models, and then using a join in your query to fetch all the required data. For more information about SQL and how to use it in SQLite, take a look at the SQLite documentation and tutorials on the SQLite project's website, at `https://sqlite.org/`.

In the next chapter, we'll look at building overview screens. These are extremely common start screens in applications; they often form the central screen of an application, somewhere that the user is brought back to again and again during navigation. Android has a massively versatile widget for these screens--the `RecyclerView`. Also, we'll explore how to use the `RecyclerView`, by coupling it to `LiveData` and using Data Binding to keep it up to date with the rest of the application.

7
Creating Overview Screens

Overview screens, or dashboard screens, are layouts that allow the user to get a quick look at their data within an application. As such, they are also screens that the user will return to again and again. Most often, they are positioned as the first screen the user will normally see when they open the application, like the Inbox in an email application, or the list of files in your Google Drive. In apps, navigation is usually goal-oriented; the user starts with an overview, and then navigates to perform a specific action. Once they are finished with their action (for example, writing and sending an email), they are redirected to the overview screen.

Overview screens can be complex systems to build as they should be reactive, and they will often depend on large amounts of application data. As it's the screen your users will see the most often in your app, an overview screen needs special attention in the design process. It's important to present the user with the most important data, without overwhelming them. Placing too much information on the screen makes it harder for your user to find the information that they want.

In this chapter, we'll look at how to design Overview screens. We'll take a detailed look at the following:

- The `RecyclerView` class, which is the most commonly used component in overview lists
- How data-binding can make `RecyclerView` much easier to use
- Techniques that can be used when designing an Overview screen
- How to get data from a Room database into a `RecyclerView`

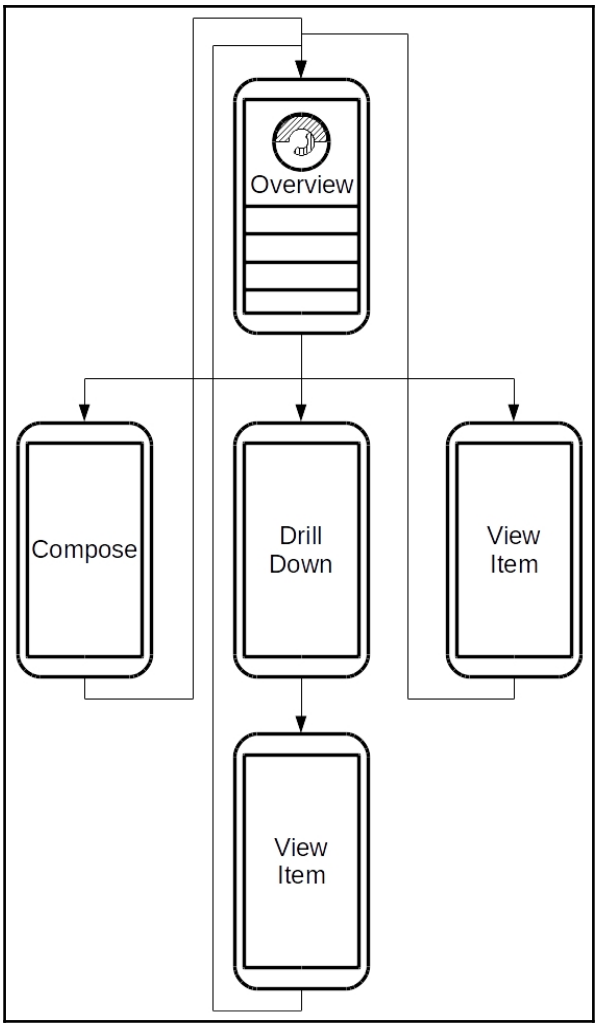
Designing an Overview screen

Overview screens and dashboard screens are not only the first thing your users will typically see, but they're also the most common point of contact with your user. They need to be functional, beautiful, and also very fast. An application that takes too long to load its first screen will only frustrate its users. If your application is frustrating to users, they will avoid using it. As such, it's very important to consider what information your user will need, and what are the most important actions they will take from the overview screen.

The *Material Design guideline*s have excellent recommendations to help you decide on these aspects of your application, which in turn will help you produce better applications. Remember that while it's fun (and important) to get creative with your designs, it's also very important to **stick to the rules**. Common patterns in design help your users understand what you're asking them to do, and how to use your application. This understanding between you and your users is why *Material Design is a Design Language*, and not just a look and feel. It's a language that you can speak to your users, and they can easily understand. For example, when you have a floating action button in the bottom-right of a screen, the user knows that it will generally start or create something new, such as create an empty document, or take a new photograph (depending on the application).

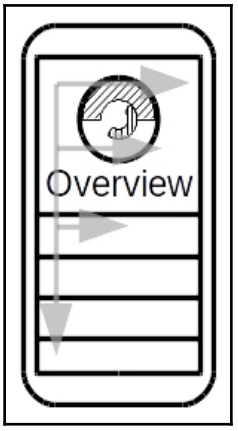
Overview screens need to allow a user to reach every part of your application, but unlike a website or desktop application, this might have some intermediary steps (although as few as possible). This means that while you might present them with data, it should never be for viewing only. Every element placed on an overview screen must earn its right to be there. They should all fulfill two roles: give the user information, and allow them to take some action with that information (even if only to find out more). One role is fulfilled by simply being on the screen, the other is fulfilled by allowing the user to tap on the widget. You can, of course, add more: swipe to dismiss, scroll, and so on. In these cases, the interactions must be consistent with interactions in Material Design (that is, swipe to dismiss should always be on list items, not on a button).

An example flow through an application should look like the following diagram. You'll note that all the processes eventually return the user to the overview screen. This is what is meant to be deep navigation; it's a goal-oriented structure designed to guide the user toward completing what they are trying to accomplish:



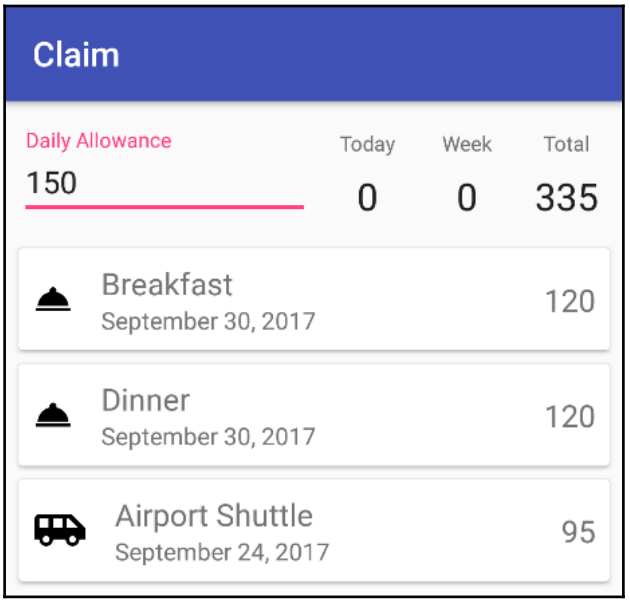
Elements of an Overview screen

Overview screens have certain common elements that let the user know what it is they're looking at, and how they're expected to use the screen. It's helpful to know how people look at a screen when they see it for the first time. Studies by groups such as Neilson show that most western people follow a sort of **F** shaped pattern when looking at the screen for the first time. Starting in the top-left corner, their eyes track right and downwards, as shown in this diagram:



This means that when designing an overview screen, the most important information should be at the top of the screen, with the second most important information to its right, and as you work down the screen, the information becomes less important. The preceding diagram uses a graph at the top of its screen; this is also an important element: favor using graphics and indicators over raw numbers where it's applicable. A user can get a much quicker overview from a graph than they can from a table of numbers, even though the latter is more powerful. An overview screen should be something the user can use in a few seconds; it's not a place where they will want to spend time understanding the details. As such, an overview screen should not need to be scrolled in order to be useful. It's not as important to avoid scrolling an overview screen as it is on a form/input screen, but any scrolling should only be applicable to access detailed information.

It's common for an overview screen to start with a graph, or some summary of the user's data, and then have a list of the most applicable details. Using the travel claim's app as an example, the overview should have the overview fragment you developed at the top of the screen, and this should be followed by a list of their travel claims with the most recent at the top:



The overview fragment allows them to see how much they have spent, while the list shows them instantly what they've been spending the money on. Another possibility would be a graph showing them the breakdown of how much they have spent in each category. However, this will typically be less useful on a day-to-day basis and more useful at the end of a business trip in the form of a report.

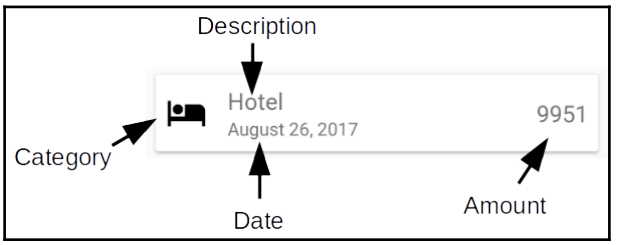
One of the most common elements on an overview screen is a list of some sort. Even when the overview doesn't include graphs and info-graphics, a list of the user's most current / most useful items is a very common structure, and Android provides the `RecylcerView` as the perfect system to build these sorts of lists. Unlike `ViewPager` or `ListView`, a `RecyclerView` is a generic system for displaying large amounts of scrolling data. Its child widgets don't need to be laid out in a strict way; they can be lists, they can be a grid, staggered unevenly, or anything you care to think of with a custom layout manager. However, they all share a common collection of structures--every `RecyclerView` needs the following components:

- An `Adapter` to provide the child `View` objects, and bind them to the data model
- `ViewHolder` classes that wrap the child `View` objects
- A `LayoutManager` to determine how to place the child `View` objects relative to each other

Let's explore how to build and use the components of a `RecyclerView` in some more detail, and how to build the overview screen for the travel claim app.

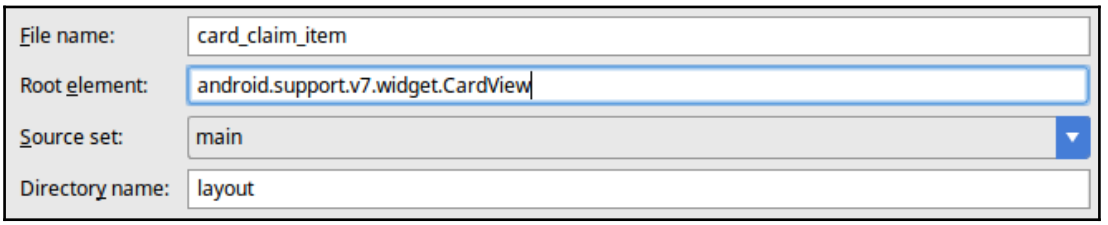
Creating layouts for ViewHolders

A `RecyclerView` does just what its name suggests--it recycles or reuses its children to present different data to the user. This means that while it appears to have a long list of child-widgets (such as cards or images), it actually has the ones that the user can actually see. When a widget is scrolled off the screen, the `RecyclerView` changes its data, and then scrolls it back into view. The `RecyclerView` doesn't directly bind the data to the child views; however, it instead goes through a `ViewHolder`. The job of the `ViewHolder` is to help speed up the data binding process. Think of the travel claim app again; if we want to display each claim item in a `RecyclerView`, each one will look something like the following:



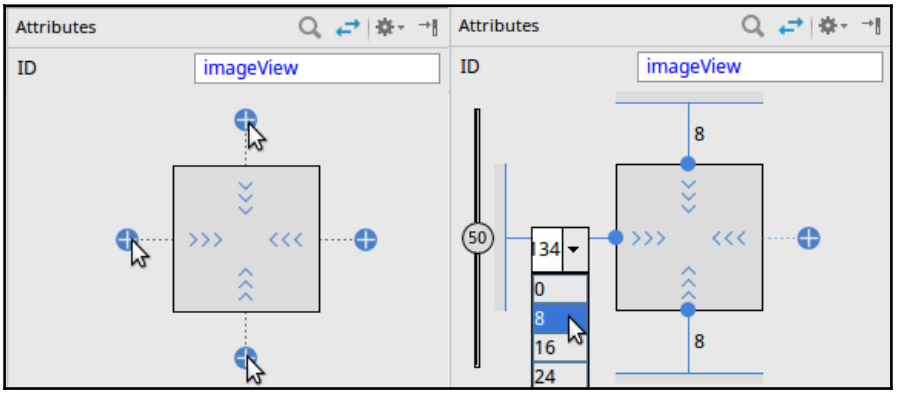
Each of the preceding items will require a different Android widget, and every time you want to populate them, they need to be looked up and bound to their new data. A `ViewHolder` implementation is a convenient place to look up, hold, and bind data for a specific data model type and display component. Let's go ahead and build a layout resource for the preceding diagram, and then we can create a `ViewHolder` to use it with a `RecyclerView`:

1. In Android Studio, under the application resources (**res**) directory, right-click on the **layout** directory and select **New | Layout resource file**.
2. Name the new layout resource `card_claim_item`.
3. Change the **Root element** to `CardView`:

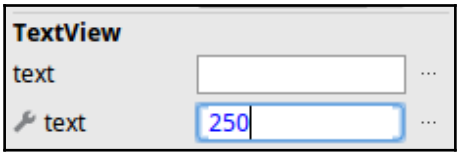


4. Click **OK** to create the new layout file.
5. In the **Palette**, open the **Layouts** section and drag a `ConstraintLayout` into the Design canvas.
6. In the **Palette**, open the **Images** section and drag an `ImageView` into the Design canvas.
7. Select the `ic_other_black` icon from the drawable resource selector that automatically opens.

8. Use the constraint editor on the right to add constraints to the top, left, and bottom of the new `ImageView`, and set all of them to **8**, as shown:

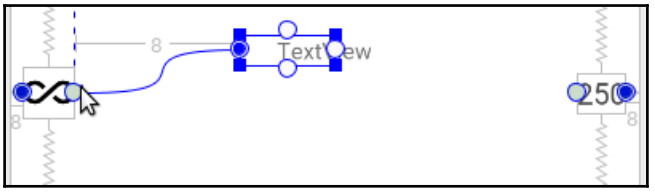


9. Change the **ID** of the `ImageView` to `item_category`.
10. In the **Palette**, open the **Text** section and drag a new `TextView` into the Design canvas to the right of the category icon `ImageView`.
11. Use the constraint editor to add constraints of `8dp` to the top, right, and bottom of the new `TextView` so that it centers and places itself to the right of the Design canvas (directly opposite the category icon `ImageView`).
12. Change the **ID** of the `TextView` to `item_amount`.
13. Remove the contents of the **text** attribute, and change the **text** attribute below it (the one with the spanner icon) to `250`. This value is only used by the Design canvas, and allows you to preview what the layout will look like with set values (even though the real values are populated at runtime):

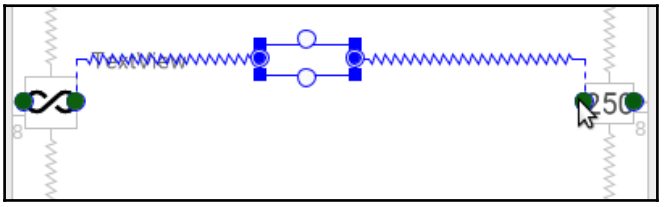


14. Change the **textAppearance** attribute to `@style/TextAppearance.AppCompat.Medium`, which will appear in the drop-down as `AppCompat.Medium`.
15. From the Palette, drag another `TextView` into the design view, roughly between the icon `ImageView` and the amount `TextView`.

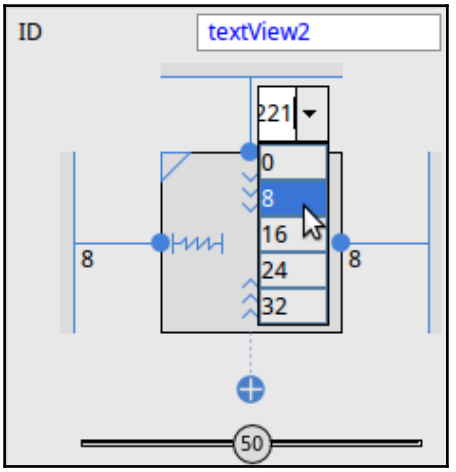
16. Drag a constraint from the left of the `TextView` to the right handle of the `ImageView`:



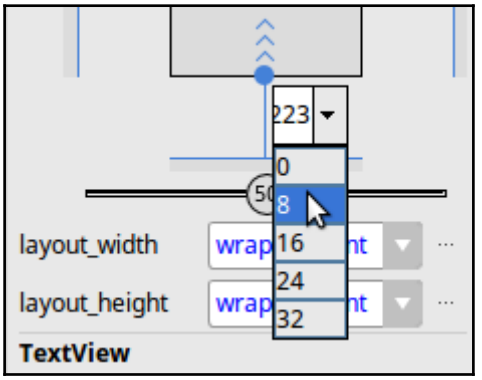
17. Drag another constraint from the right of the new `TextView` to the left of the amount `TextView`:



18. Use the constraint editor to add a constraint to the top of the new `TextView`.
19. Set the top constraint to `8`:



20. Change the `layout_width` attribute of the new `TextView` to `match_constraint` using the **Attributes** panel (just below the constraint editor).
21. Change the **ID** of the new `TextView` to `item_description`.
22. Clear the **text** attribute, and set the Design **text** attribute to `Airport Shuttle` so that you still have something visible in the Design canvas.
23. Change the **textAppearance** attribute to `@style/TextAppearance.AppCompat.Medium`, which will appear in the drop-down as `AppCompat.Medium`.
24. From the **Palette**, drag a third `TextView` into the Design canvas and drop it between the category icon `ImageView` and the amount `TextView`.
25. Just like the description `TextView`, constrain the new `TextView` to the right of the category icon and to the left of the amount `TextView`.
26. Using the constraint editor, add a constraint at the bottom of the new `TextView`, and set its bottom margin to `8`:



27. Set the **ID** of the new `TextView` to `item_timestamp`.
28. Change the **layout_width** of the new `TextView` to **match_constraint**.
29. Drag a constraint from the top of the new `TextView` to the bottom of the description `TextView`; this will ensure that they have at least 8dp between them.
30. Clear the **text** attribute, and set the Design tool text attribute to a date such as `27-December-2017`.
31. In the **Component Tree** panel, select the `CardView` at the root of your layout.
32. Switch over to the **View all attributes** panel.
33. Open the **Layout_Margin** group.

34. Set the top margin to `@dimen/grid_spacer1`.
35. Change the **layout_height** of the `CardView` to **wrap_content**; the layout will roll up looking something like this:

Creating a simple ViewHolder class

Creating a `ViewHolder` is very simple, and it's a good place to encapsulate any display specific logic for rendering the items for a `RecyclerView`. For the preceding layout, follow these steps to build a `ViewHolder`:

1. Right-click on the **ui** package in Android Studio, and select **New| Java Class**.
2. Name the new class `ClaimItemViewHolder`.
3. Set the new classes **Superclass** to `android.support.v7.widget.RecyclerView.ViewHolder`.
4. Click **OK** to create the new class.
5. The main job of a `ViewHolder` is to speed up the binding between the data model and the user interface widgets and for that to happen, the `ViewHolder` needs references to each of the `View` objects it will populate:

```
private final ImageView categoryIcon;
private final TextView description;
private final TextView amount;
private final TextView timestamp;
```

6. This `ViewHolder` will also need a way to format the timestamp, and the best way to do that is with a `java.text.DateFormat`, which is also something to keep a reference to as they're quite expensive to construct:

```
private final DateFormat dateFormat;
```

7. A `ViewHolder` is usually constructed with the `View` object it's expected to bind to. You can inflate the `View` object within the `ViewHolder` constructor, but to keep things flexible and avoid argument clutter on the constructor, this `ViewHolder` implementation will just take the `View` object it will wrap:

```
public ClaimItemViewHolder(final View claimItemCard) {
    super(claimItemCard);
    this.categoryIcon =
claimItemCard.findViewById(R.id.item_category);
    this.description =
claimItemCard.findViewById(R.id.item_description);
    this.amount = claimItemCard.findViewById(R.id.item_amount);
    this.timestamp =
claimItemCard.findViewById(R.id.item_timestamp);
```

8. You also need to create the `DateFormat` object and, for this, you want the long date format in the user's current locale:

```
this.dateFormat = DateFormat.getDateInstance(DateFormat.LONG);
```

9. This class will need a utility method to figure out which icon should be rendered for a `Category` that will involve manually referencing the application `Resources` to retrieve the black versions of the category icons:

```
public Drawable getCategoryIcon(final Category category) {
    final Resources resources = itemView.getResources();
    switch (category) {
        case ACCOMMODATION:
            return resources.getDrawable(R.drawable.ic_hotel_black);
        case FOOD:
            return resources.getDrawable(R.drawable.ic_food_black);
        case TRANSPORT:
            return resources.getDrawable(R.drawable.ic_transport_black);
        case ENTERTAINMENT:
            return
resources.getDrawable(R.drawable.ic_entertainment_black);
        case BUSINESS:
            return resources.getDrawable(R.drawable.ic_business_black);
        case OTHER:
        default:
            return resources.getDrawable(R.drawable.ic_other_black);
    }
}
```

10. You'll also need a utility method to format the amounts so that integer amounts don't have any decimal component, while non-integers only display two decimal places:

```
public String formatAmount(final double amount) {
    return amount == 0
            ? ""
            : amount == (int) amount
            ? Integer.toString((int) amount)
           : String.format("%.2f", amount);
}
```

11. Finally, you need a way for the adapter to populate all the `View` elements with data, and because this class is specific to the `ClaimItem` data objects, you can make this simple by having a setter-like method:

```
public void setClaimItem(final ClaimItem item) {
    categoryIcon.setImageDrawable(getCategoryIcon(item.getCategory()));
    description.setText(item.getDescription());
    amount.setText(formatAmount(item.getAmount()));
    timestamp.setText(dateFormat.format(item.getTimestamp()));
}
```

Creating a ViewHolder with data binding

As you can see from building a traditional `ViewHolder` implementation, there is quite a lot of work and boilerplate code required just to put the data from a single item onto the screen in a layout. Further, it's actually quite expensive in its own right, because every one of the `ViewHolder` instances creates and holds an instance of the `DateFormatter` where they can easily be shared between all the `ClaimItemViewHolder` instances for a `RecyclerView`.

In cases like this, data binding can make a huge difference. Using a few tricks, you can actually create a completely generic `ViewHolder` implementation that will work for any data object in your application (assuming that you can bind it to a layout file). First, you'll need to create a nice generic `ItemPresenter`, and then modify the layout, and then you're ready to create a generic data-binding `ViewHolder` implementation. Follow these instructions, and you'll only ever need one `ViewHolder` implementation:

1. Right-click on the **presenters package** in Android Studio, and select **New| Java Class**.
2. Name the class `ItemPresenter`.
3. Click **OK** to create the new class.

4. The `ItemPresenter` will need a `Context` to reference application `Resources` and files:

```
private final Context context;

public ItemPresenter(final Context context) {
    this.context = context;
}
```

5. Create a `formatAmount` utility method the same way as in the simple `ViewHolder` class:

```
public String formatAmount(final double amount) {
    return amount == 0
            ? ""
            : amount == (int) amount
            ? Integer.toString((int) amount)
            : String.format("%.2f", amount);
}
```

6. Write a `getCategoryIcon` utility method into the new `ItemPresenter` (this is almost exactly the same as the one in the `ClaimItemViewHolder`, except in how it accesses the `Resources` object):

```
public Drawable getCategoryIcon(final Category category) {
    final Resources resources = context.getResources();
    switch (category) {
        case ACCOMMODATION:
            return
resources.getDrawable(R.drawable.ic_hotel_black);
        case FOOD:
            return resources.getDrawable(R.drawable.ic_food_black);
        case TRANSPORT:
            return
resources.getDrawable(R.drawable.ic_transport_black);
        case ENTERTAINMENT:
            return
resources.getDrawable(R.drawable.ic_entertainment_black);
        case BUSINESS:
            return
resources.getDrawable(R.drawable.ic_business_black);
        case OTHER:
        default:
            return
resources.getDrawable(R.drawable.ic_other_black);
    }
}
```

7. Write a `formatDate` utility method that will convert the `Date` objects into text suitable for displaying on the screen. The conversion is done by a `DateFormat` object, which is only created the first time `formatDate` is called (it is lazy-initialized). The lazy-initialization is important, as this class is expected to be generic across all possible item presenters in the application and so, there will be cases where it's not used:

```
private DateFormat dateFormat;
public String formatDate(final Date date) {
    if (dateFormat == null) {
        dateFormat = DateFormat.getDateInstance(DateFormat.LONG);
    }

    return dateFormat.format(date);
}
```

8. Now, open the `card_claim_item.xml` layout resource.
9. Change to the **Text** view in the editor.
10. Create a new layout root element above the `CardView`, and ensure that you remove the namespace declarations from the `CardView` and close the layout element at the end of the file:

```
<layout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools">
```

11. Above the `CardView`, declare a data block with two variables. It's important to keep these names generic. One will be an instance of `ItemPresenter`, and the other will be the `ClaimItem` to be bound by the layout:

```
<data>
    <variable name="presenter"
type="com.packtpub.claim.ui.presenters.ItemPresenter" />
    <variable name="item" type="com.packtpub.claim.model.ClaimItem"
/>
</data>
```

12. Find the `ImageView` declaration for `item_category` and add a new data-bound attribute to use the `ItemPresenter` to find the correct icon:

```
<ImageView
    android:id="@+id/category_icon"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginBottom="8dp"
```

```
        android:layout_marginStart="8dp"
        android:layout_marginTop="8dp"
        app:imageDrawable="@{presenter.getCategoryIcon(item.category)}"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent" />
```

13. Find the amount `TextView` declaration and data bind its text attribute, using the presenter to format the amount from the `ClaimItem`:

```
<TextView
    android:id="@+id/item_amount"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginBottom="8dp"
    android:layout_marginEnd="8dp"
    android:layout_marginTop="8dp"
    android:text="@{presenter.formatAmount(item.amount)}"
    android:textAppearance="@style/TextAppearance.AppCompat.Medium"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintTop_toTopOf="parent"
    tools:text="150" />
```

14. Data bind the description from the `ClaimItem` to the `item_description TextView`:

```
<TextView
    android:id="@+id/item_description"
    android:layout_width="0dp"
    android:layout_height="wrap_content"
    android:layout_marginEnd="8dp"
    android:layout_marginStart="16dp"
    android:layout_marginTop="8dp"
    android:text="@{item.description}"
    android:textAppearance="@style/TextAppearance.AppCompat.Medium"
    app:layout_constraintEnd_toStartOf="@+id/item_amount"
    app:layout_constraintStart_toEndOf="@+id/category_icon"
    app:layout_constraintTop_toTopOf="parent"
    tools:text="Airport Shuttle" />
```

15. Use the presenter to data bind the timestamp from the `ClaimItem` to the timestamp `TextView`:

```
<TextView
    android:id="@+id/item_timestamp"
    android:layout_width="0dp"
```

```
android:layout_height="wrap_content"
android:layout_marginBottom="8dp"
android:layout_marginEnd="8dp"
android:text="@{presenter.formatDate(item.timestamp)}"
android:textAppearance="@style/TextAppearance.AppCompat.Small"
app:layout_constraintBottom_toBottomOf="parent"
app:layout_constraintEnd_toStartOf="@+id/item_amount"
app:layout_constraintStart_toStartOf="@+id/item_description"
app:layout_constraintTop_toBottomOf="@+id/item_description"
tools:text="16-December-2017" />
```

16. Now, it's time to start on a generic `ViewHolder` class that can be reused with any data-bound layout. Right-click on the **ui** package and select **New | Java Class**.
17. Name the new class `DataBoundViewHolder`.
18. Change the **Superclass** to `android.support.v7.widget.RecyclerView.ViewHolder`.
19. Click **OK** to create the new class.
20. Add a generic declaration to the class so that you have generic types for the Presenter, and Item (P, I) variables:

```
public class DataBoundViewHolder<P, I> extends
RecyclerView.ViewHolder {
```

21. Every Binding class generated by the data binding system extends `ViewDataBinding`; the `DataBoundViewHolder` will actually wrap one of these so that any data-bound layout can be wrapped:

```
private final ViewDataBinding binding;
```

22. Now, write a constructor that takes a `ViewDataBinding` object and a Presenter object for the data-bound layout to use. As `ViewDataBinding` is a generic abstract class, we can't directly call the `setPresenter` method that will be generated by the data binding system in the `CardClaimItemBinding` class. Instead, we can use a special generic data binding method that allows you to assign unknown variables based on a generated ID number; this is a bit like using Java reflection, except that the actual implementation is generated at compile time and is very fast:

```
public DataBoundViewHolder(final ViewDataBinding binding, final P
presenter) {
    super(binding.getRoot());
    this.binding = binding;
    this.binding.setVariable(BR.presenter, presenter);
}
```

If you are presented with a choice of multiple `BR` classes to import, use the one for your own project (`com.packtpub.claim`). Much like the normal Android resources (`R`), the data binding system generates a lookup class for each project.

23. Then, write in two setter methods so that the `Presenter` and `item` variables can be changed uniformly from the outside:

```
public void setItem(final I item) {
    binding.setVariable(BR.item, item);
}

public void setPresenter(final P presenter) {
    binding.setVariable(BR.presenter, presenter);
}
```

The `setVariable` method is generated at compile time, just like the getter and setter methods as a series of `if` statements. This makes it a little slower than the actual setter methods, but much faster than using reflection to invoke the setter methods. It's not the sort of area that should need optimization, especially as there are only two possible variables for these data-bound layouts. If your layout needs more than these two variables in a `RecyclerView`, you should consider composing or inheriting the logic and data required into more specific classes.

The generated `setVariable` implementation of the `card_claim_item` layout defined in this section will look something like this:

```
public boolean setVariable(int variableId, @Nullable Object variable) {
    boolean variableSet = true;
    if (BR.item == variableId) {
        setItem((com.packtpub.claim.model.ClaimItem) variable);
    }
    else if (BR.presenter == variableId) {
        setPresenter((ItemPresenter) variable);
    }
    else {
        variableSet = false;
    }
    return variableSet;
}
```

As you can see, this code will execute very quickly and won't throw exceptions if an unknown variable ID is given. However, it will throw a `ClassCastException` if you tried to pass in the wrong type for a data-bound variable.

Creating a RecyclerView adapter

In order to get data into a `RecyclerView`, you need an `Adapter` class, not unlike the `PagerAdapter` you wrote to display the attachment previews for the `CaptureClaimActivity`. However, `RecyclerView` does a lot more of the heavy lifting than `ViewPager` and as a result, what you can and can't do inside the adapter is far more restricted than with `PagerAdapter`. Also, unlike a `PagerAdapter`, a `RecylcerView` adapter has two actions that are involved in displaying each element: create and bind. When the `RecyclerView` needs a new child widget for an element, it will invoke `onCreateViewHolder`, which should return an unpopulated `ViewHolder`, which will then be passed to `onBindViewHolder` where the data should be mapped into the `View` from whatever data source the adapter uses.

First off, the `RecyclerView` maintains the list of its child views completely, so the adapter must never add or remove them directly. Secondly, the `RecyclerView` expects the adapter to be stable, that is, the data within the adapter must not change without telling the `RecyclerView` about the changes.

Unlike older recycling widget classes like `ListView` and `GridView`, `RecyclerView` does not assume that it's presenting the same object model over and over. Instead, each object returned from the `Adapter` can optionally have a view type indicator; when these are different, the `RecylcerView` maintains a separate pool for each of the view types and recycles all of them separately.

When using different view types, it's common for the adapter to use the layout resource ID as the view type; these are unique per application and avoid any need for a `switch` statement or similar mapping between internal view type IDs and the actual resources.

For the travel claim example, you will need an adapter to display all the `ClaimItems` on the overview screen. Fortunately, Room provides you with a prebuilt `LiveData`, which can be observed directly, which makes building the adapter much simpler. Follow these simple steps to build a `RecyclerView` adapter bound to a `LiveData` object, and use the `DataBoundViewHolder` to present the data to the user:

1. Right-click on the **ui** package and select **New| Java Class**.
2. Name the new class `ClaimItemAdapter`.
3. Click **OK** to create the new class.

4. Change the class declaration to extend from `RecyclerView.Adapter` and describe the `DataBoundViewHolder` generic you will be using:

```
public class ClaimItemAdapter
       extends
RecyclerView.Adapter<DataBoundViewHolder<ItemPresenter, ClaimItem>>
{
```

5. This adapter class will inflate the data-bound layout files as resources, so it'll need a `LayoutInflator` to do the work for it:

```
private final LayoutInflater layoutInflater;
```

6. The `ItemPresenter` instances can also be shared between all the claim item layouts that are on screen, so the `ClaimItemAdapter` should hold a reference to it:

```
private final ItemPresenter itemPresenter;
```

7. Most importantly, the `ClaimItemAdapter` needs data to display. Ensure that you instantiate this reference so that you don't need null-checks in your other methods:

```
private List<ClaimItem> items = Collections.emptyList();
```

8. Now, declare a constructor for `ClaimItemAdapter`; as `ClaimItemAdapter` will observe a `LiveData` object, it'll need a `LifecycleOwner` as well. The `LifecycleOwner` tells the `LiveData` when to notify you of changes, and when not to and also when to unregister any listeners. Typical `LifecycleOwners` are `Activity` or `Fragment` instances, but you can make almost any class a `LifecycleOwner`:

```
public ClaimItemAdapter(
        final Context context,
        final LifecycleOwner owner,
        final LiveData<List<ClaimItem>> liveItems) {

    this.layoutInflater = LayoutInflater.from(context);
    this.itemPresenter = new ItemPresenter(context);
```

For further flexibility, you can allow the `ItemPresenter` to be passed into the constructor. This will allow the `ItemPresenter` to be extended or configured outside of the `ClaimItemAdapter` objects, and each instance can be given different presentation rules.

9. Note that the `ClaimItemAdapter` doesn't keep a reference to the `LiveData` instance yet, and in fact, it won't directly hold a reference at all. Instead, you'll use an anonymous inner class (or lambda if it's available to you) to observe the `LiveData`. It's important to know that when you start observing a `LiveData` instance, you will automatically get an *initial* event with the current state of the data, if your `LifecycleOwner` is in the correct state. This means you should never need to attempt to fetch the data directly:

```
liveItems.observe(owner, new Observer<List<ClaimItem>>() {
    public void onChanged(final List<ClaimItem> claimItems) {
        ClaimItemAdapter.this.items = (claimItems != null)
                ? claimItems
                : Collections.<ClaimItem>emptyList();
        ClaimItemAdapter.this.notifyDataSetChanged();
    }
});
```

10. Now the constructor is complete, and it's time to implement the binding-related features. The first step is to implement `onCreateViewHolder`, which will use the `DataBindingUtil` to create the layout and the `ViewDataBinding` that the `DataBoundViewHolder` will wrap:

```
public DataBoundViewHolder<ItemPresenter, ClaimItem>
onCreateViewHolder(
        final ViewGroup parent,
        final int viewType) {

    return new DataBoundViewHolder<>(
            DataBindingUtil.inflate(
                    layoutInflater,
                    R.layout.card_claim_item,
                    parent,
                    false
            ),
            itemPresenter
    );
}
```

11. Due to the `DataBoundViewHolder` implementation, the `onBindViewHolder` method is trivial to implement:

```
public void onBindViewHolder(
        final DataBoundViewHolder<ItemPresenter, ClaimItem> holder,
        final int position) {

    holder.setItem(items.get(position));
}
```

12. The `RecyclerView` also needs to know how many items are in the data model:

```
public int getItemCount() {
    return items.size();
}
```

This adapter can be very easily adapted further in the same way as the `DataBoundViewHolder`, to allow you to present any `LiveData` list returned from a Room database with an arbitrary data-bound layout file. The combination of data binding and `LiveData` makes for an extremely powerful combination that dramatically simplifies your user-interface code, and avoids the need to write lots of boilerplate structure for every combination of views and models.

Data binding an adapter

If you want to use data binding on a layout with a `RecyclerView` in it, you can even data bind the adapter object to the `RecyclerView`. All you need to do is expose a method to access the desired adapter object in a presenter class:

```
private RecyclerView.Adapter<?> claimItemsAdapter;

public RecyclerView.Adapter<?> getClaimItemsAdapter() {
    if (claimItemsAdapter == null) {
        claimItemsAdapter = new ClaimItemAdapter(
                this, this,
                database.claimItemDao().selectAll()
        );
    }

    return claimItemsAdapter;
}
```

It's important that you preconstruct or cache the instances you create to avoid recreating the adapter object unnecessarily. It's also important to remember not to make the adapter an `ObservableField` or similar, since the adapter's content should be what changes, not the adapter itself. To bind the `RecyclerView` to its adapter, use the data binding system's *auto property* system:

```
<android.support.v7.widget.RecyclerView
    app:adapter="@{presenter.claimItemsAdapter}"
app:layoutManager="android.support.v7.widget.LinearLayoutManager"
    android:id="@+id/claim_items"
    android:clipChildren="false"
    android:layout_marginTop="@dimen/grid_spacer1"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
```

It's really important when using data binding and adapter views together to remember that they both update the user interface. As such, it's important to ensure that you keep the adapter reference in the presenter stable, and don't change it without being sure of yourself. Changing the adapter reference will cause the `AdapterView` (such as a `RecyclerView`) to completely rebuild its contents rather than just refreshing its contents. It's much better to use the adapter to notify the `AdapterView` of changes than it is to make the adapter observable.

Creating the Overview activity

The travel claim example app needs a nice overview activity to tie together the allowance overview, a list of the claim items, and a way for the user to create new claim items. As we have a Room database, things can become significantly more decoupled, and that's a really good thing. Having a central reactive source of data allows different parts of your application to always reflect the actual state of the application as it changes, without having to coordinate with each other.

The first part of building the `OverviewActivity` is creating the `Activity` class itself and populating it with the claim items that the user has entered. Follow these steps to create a skeleton `OverviewActivity` and register it as the main `Activity` for the application:

1. Start by right-clicking on your main package (that is, **com.packtpub.claim**) and selecting **New | Activity | Empty Activity** from the menu.

2. Name the new class `OverviewActivity`.
3. Leave all the other fields as their defaults and select **Finish** to create the new `Activity` and its layout file.
4. Open the new `activity_overview.xml` layout file and change to the **Text** editor.
5. Android Studio will have placed a `ConstraintLayout` as the root element; change it to a `FrameLayout` instead, because this layout is very simple and since the logic will be self-binding, there is no point in using a data-bound layout:

```
<FrameLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context="com.packtpub.claim.OverviewActivity">

</FrameLayout>
```

A **FrameLayout** is a very simple layout where its children are rendered on top of each other. The first child is painted first, and then the second child is painted on top of the first. This makes it ideally suited for building layered scenes, even when some layers will not always be visible.

6. The first child of the `FrameLayout` will be a simple `LinearLayout` to allow you to place the allowance overview above the scrolling list of claim items. `LinearLayout` is ideal here as it's a very simple and very fast layout to use, and we don't need the complexities of a `ConstraintLayout`:

```
<LinearLayout
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:clipToPadding="false"
    android:orientation="vertical"
    android:paddingTop="@dimen/grid_spacer1"
    android:paddingBottom="@dimen/grid_spacer1">
</LinearLayout>
```

7. The first child of the `LinearLayout` is the `AllowanceOverviewFragment`, which will allow the user to edit their daily allowance and see how much they're spending:

```
<fragment
    class="com.packtpub.claim.ui.AllowanceOverviewFragment"
    android:id="@+id/allowance_overview"
    android:layout_width="match_parent"
    android:layout_height="wrap_content" />
```

8. Next comes the `RecyclerView`, which will display a scrolling list of the claim items that the user has entered. Note the clipping and padding attributes here; they ensure that the claim item cards are inset, but that their full borders and shadows will be visible:

```
<android.support.v7.widget.RecyclerView
    android:id="@+id/claim_items"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:layout_marginTop="@dimen/grid_spacer1"
    android:clipToPadding="false"
    android:paddingLeft="@dimen/grid_spacer1"
    android:paddingRight="@dimen/grid_spacer1"
app:layoutManager="android.support.v7.widget.LinearLayoutManager"
/>
```

9. Now, open the `OverviewActivity` class that Android Studio generated; it's time to populate the layout with claim items.
10. We'll be rendering a list of `ClaimItem` objects using the `ClaimItemAdapter`, and it needs to watch for changes using the `LiveData` object produced by the database. This requires that the `Activity` report its life cycle, and this is done by extending one of the `Activity` implementations provided by the support packages (in this case, `AppCompatActivity`):

```
public class OverviewActivity
        extends AppCompatActivity {
```

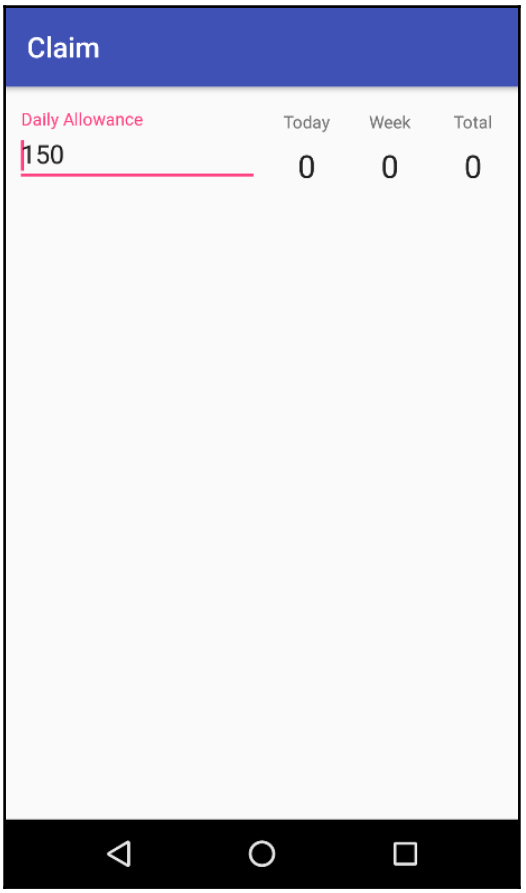
11. As all the behavior for this Activity is actually handled by its fragments and by the `LiveData` changes triggered by the `ClaimDatabase`, the `onCreate` implementation only needs to set the adapter for the `RecyclerView`. All other logic and behavior for the `OverviewActivity` will be handled by the fragments and adapter:

```
protected void onCreate(final Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_overview);

    final RecyclerView claimItems = findViewById(R.id.claim_items);
    claimItems.setAdapter(new ClaimItemAdapter(
            // both the Context, and LifecycleOwner are the
OverviewActivity
            this, this,
ClaimApplication.getClaimDatabase().claimItemDao().selectAll()
    ));
}
```

12. Finally, you'll need to change the `AndroidManifest.xml` file to tell the system that the main entry point for the application is now the `OverviewActivity`, and not the `CaptureClaimActivity`; open the `manifests` folder near the top of your projects file-tree and open the `AndroidManifest.xml` file.
13. Change the activity element declarations so that the MAIN / LAUNCHER intent-filter is in the `OverviewActivity` element instead of the `CaptureClaimActivity` element. It's also worth changing the `windowSoftInputMode` attribute so that the software keyboard doesn't automatically open when the `OverviewActivity` is started. The keyboard opens by default because the first widget on the screen is the `EditText` field, where the user can enter their daily allowance:

```
<activity
    android:name=".CaptureClaimActivity"
    android:label="@string/title_activity_capture_claim"
    android:theme="@style/AppTheme.NoActionBar" />
<activity
    android:name=".OverviewActivity"
    android:windowSoftInputMode="stateHidden">
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER"
/>
    </intent-filter>
</activity>
```

If you now run your application, you'll see that while the screen is technically complete, there are no claim items and no way to add them. As such, there is nothing in the `RecyclerView` to look at or scroll through:



You need to provide a way for your user to add new claim items. The best way is with a floating action button in the bottom-right corner of the screen, and we'll be doing that with a new `Fragment`. By using a `Fragment` for this task, you'll be able to place a "new item" floating action button on any screen in the application without having to implement any special code in the `Activity` class.

Creating new ClaimItems with a Fragment

An unusual property of using `LiveData` from a Room database is that various parts of your application can now interact without needing any direct knowledge of each other. In the case of your `OverviewActivity`, this will allow you to populate the database with new `ClaimItem` entities without dispatching any sort of "new item" or "item added" events to the `ClaimItemAdapter`. However, the Room database abstraction layer prevents you from running any query on the main thread unless it returns a `LiveData`. While the query to retrieve the `ClaimItem` entities returned a `LiveData`, inserting new `LiveData` entities will be required to run in the background. Follow these steps to build a `Fragment` that can allow the user to capture and record a new travel claim item:

1. You'll need a task to insert both a `ClaimItem` entity and any `Attachment` entities associated with it. This task will need to run on a background worker thread, so open the `ClaimDatabase` class in Android Studio.
2. After the abstract methods that return, the `ClaimItemDao` and `AttachmentDao` declare a new method to return a `Runnable` task to insert a new `ClaimItem`:

```
public Runnable createClaimItemTask(final ClaimItem claimItem) {
    return new Runnable() {
        @Override
        public void run() {
        }
    };
}
```

3. Within the new `Runnable` task, you'll want to use a transaction to save the contents of the `ClaimItem` object into the database; if any part of this method fails, the transaction will be rolled back, and the method will have had no effect:

```
beginTransaction();
try {
    final long claimId = claimItemDao().insert(claimItem);
    claimItem.id = claimId;

    for (final Attachment attachment : claimItem.getAttachments())
{
        attachment.claimItemId = claimId;
        attachment.id = attachmentDao().insert(attachment);
    }
    setTransactionSuccessful();
} finally {
    endTransaction();
}
```

4. You'll also need a method in `ClaimItem` to ensure that it has content and is considered valid, so open the `ClaimItem` class.
5. At the end of the `ClaimItem` class, create a new `isValid` method; this will be used when the `CaptureClaimActivity` returns a `ClaimItem` to check whether we should store the new `ClaimItem` in the database:

```
public boolean isValid() {
    return !TextUtils.isEmpty(description)
            && amount > 0
            && timestamp != null
            && category != null;
}
```

6. You'll need a new icon for adding claim items; right-click on the **drawable** resource directory and select **New| Vector Asset**.
7. Using the **Icon** selector, find and select the icon named `add`.
8. Name the new icon resource `ic_add_white_24dp`.
9. Click on **Next** and then on **Finish** to create the new resource.
10. Open the new icon resource in the Android Studio text editor.
11. Change the `fillColor` attribute of the path element to make it white:

```
<path
    android:fillColor="#FFFFFFFF"
    android:pathData="M19,13h-6v6h-2v-6H5v-2h6V5h2v6h6v2z"/>
```

12. Now, right-click on the **ui** package and select **New| Fragment| Fragment (Blank)**.
13. Name the new `Fragment` class `NewClaimItemFloatingActionButtonFragment`.
14. Turn off the **Include fragment factory methods** and **Include interface callbacks** options.
15. Click on the **Finish** button to create the new `Fragment` class.
16. Open the new layout file that should be called `fragment_new_claim_item_floating_action_button.xml`.
17. Replace the content of this file with just a `FloatingActionButton`:

```
<android.support.design.widget.FloatingActionButton
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    xmlns:app="http://schemas.android.com/apk/res-auto"
tools:context="com.packtpub.claim.ui.NewClaimItemFloatingActionButt
onFragment"
```

```
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        app:fabSize="normal"
        app:srcCompat="@drawable/ic_add_white_24dp" />
```

18. Now, open the new `NewClaimItemFloatingActionButtonFragment` class.
19. Change the class declaration to implement the `View.OnClickListener` interface:

```
public class NewClaimItemFloatingActionButtonFragment
        extends Fragment
        implements View.OnClickListener {
```

20. Declare a request-code to be used when sending the user to the `CaptureClaimActivity`:

```
private static final int REQUEST_CODE_CREATE_CLAIM_ITEM = 100;
```

21. Change the `onCreateView` method to also set the `OnClickListener` of the `FloatingActionButton`:

```
@Override
public View onCreateView(
        final LayoutInflater inflater,
        final ViewGroup container,
        final Bundle savedInstanceState) {

    final View button = inflater.inflate(
R.layout.fragment_new_claim_item_floating_action_button,
            container,
            false
    );

    button.setOnClickListener(this);
    return button;
}
```

22. Override the `onClick` method from the `View.OnClickListener` and start the `CaptureClaimActivity` for result:

```
@Override public void onClick(final View view) {
    startActivityForResult(
            new Intent(getContext(), CaptureClaimActivity.class),
            REQUEST_CODE_CREATE_CLAIM_ITEM);
}
```

23. Override the `onActivityResult` method to handle the incoming `ClaimItem`, and if it's valid, save it in the database using the `SERIAL_EXECUTOR` from `AsyncTask`:

```
public void onActivityResult(
        final int requestCode,
        final int resultCode,
        final Intent data) {

    if (requestCode != REQUEST_CODE_CREATE_CLAIM_ITEM
            || resultCode != Activity.RESULT_OK
            || data == null) {
        return;
    }

    final ClaimItem claimItem = data.getParcelableExtra(
            CaptureClaimActivity.EXTRA_CLAIM_ITEM
    );

    if (claimItem.isValid()) {
        final ClaimDatabase database =
ClaimApplication.getClaimDatabase();
        AsyncTask.SERIAL_EXECUTOR.execute(
                database.createClaimItemTask(claimItem)
        );
    }
}
```

24. Now, you'll need to add the new fragment to the `OverviewActivity`. Open the `activity_overview` layout file and change to **Text** mode.
25. At the bottom of the `FrameLayout` root element, include a fragment tag referencing the `NewClaimItemFloatingActionButtonFragment` and position it at the bottom-right of the screen:

```
<fragment
class="com.packtpub.claim.ui.NewClaimItemFloatingActionButtonFragme
nt"
    android:id="@+id/new_claim_item"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_gravity="bottom|end"
    android:layout_margin="@dimen/fab_margin" />
```

Now you should be able to run the application again; not only should you now have a floating action button at the bottom of the overview screen, but it will work! If you click on the button and capture some details on the `CaptureClaimActivity` and then select to navigate back to the `OverviewActivity`, the new claim item will appear in the list, sorted by date.

Unlike using `SQLiteDatabase` directly, Room will enforce only running queries on a worker thread. This makes it attractive to encapsulate your updates in a `Runnable` that can be run on a background thread such as you did with the `createClaimItemTask` in the `ClaimDatabase` class. Having these methods available on the `ClaimDatabase` makes them easily reusable, and keeps the logic consistent throughout your application. It also allows you to put them into queues or run them in parallel with other tasks if you choose to use a thread-pool instead of the `SERIAL_EXECUTOR` from `AsyncTask` (which will only run one task at a time).

Allowance overview with a Room database

If you run the overview screen and add a few claims to it, you'll notice that one piece of the code isn't reacting to the new items being added to the database: the allowance overview at the top of the screen. This is because while everything else is connected to the Room database, it's still watching the `Allowance` data model. Using a data model like this is a good idea when the data is just in memory, but now that you have a database in place, things can change and simplify. For example, the `Allowance` class only really keeps how much the user plans to spend each day; the claim items can actually be seen as an entirely separate structure in the database model.

As such, you can move the daily allowance into a different type of data storage--`SharedPreferences`. `SharedPreferences` are key-value stores for Android that have a shared in-memory representation and atomic updates. If you don't expect them to store too much data, this makes them ideal to keep track of data that doesn't really go in an SQLite database. Let's change the model of the `Allowance` overview to use the `ClaimDatabase` and `SharedPreferences`:

1. First, open the `AllowanceOverviewPresenter` class.
2. Change it from using the `Allowance` class to instead expose the daily allowance as an `ObservableInt`, and remove the `OnPropertyChangeCallback` so that the fields now look like this:

```
public final ObservableField<SpendingStats> spendingStats = new
ObservableField<>();
public final ObservableInt allowance = new ObservableInt();
```

```
private final UpdateSpendingStatsCommand updateSpendStatsCommand =
                    new UpdateSpendingStatsCommand();
```

3. Now, change the `UpdateSpendingStatsCommand` inner class to take a `List` of `ClaimItem` objects instead of an `Allowance` as its parameter:

```
private class UpdateSpendingStatsCommand extends
ActionCommand<List<ClaimItem>, SpendingStats> {
```

4. Now change the `onBackground` implementation to run a single scan through the given `List` of `ClaimItem` objects and calculate all the spending stats at once:

```
public SpendingStats onBackground(final List<ClaimItem> items)
throws Exception {
    final Pair<Date, Date> today = getToday();
    final Pair<Date, Date> thisWeek = getThisWeek();

    double spentTotal = 0;
    double spentToday = 0;
    double spentThisWeek = 0;

    for (int i = 0; i < items.size(); i++) {
        final ClaimItem item = items.get(i);
        spentTotal += item.getAmount();

        if (item.getTimestamp().compareTo(thisWeek.first) >= 0
                && item.getTimestamp().compareTo(thisWeek.second)
<= 0) {

            spentThisWeek += item.getAmount();
        }

        if (item.getTimestamp().compareTo(today.first) >= 0
                && item.getTimestamp().compareTo(today.second) <=
0) {

            spentToday += item.getAmount();
        }
    }

    // for stats we round everything to integers
    return new SpendingStats(
            (int) spentTotal,
            (int) spentToday,
            (int) spentThisWeek
    );
}
```

5. Now, change the constructor so that it takes a `LifecycleOwner` and the starting allowance to display to the user. Then, use the `ClaimDatabase` to update the spending statistics whenever there are new `ClaimItem` objects added:

```
public AllowanceOverviewPresenter(
        final LifecycleOwner lifecycleOwner,
        final int allowance) {
    ClaimApplication.getClaimDatabase()
            .claimItemDao()
            .selectAll()
            .observe(lifecycleOwner, new
Observer<List<ClaimItem>>() {
                @Override
                public void onChanged(final List<ClaimItem>
claimItems) {
                    updateSpendStatsCommand.exec(claimItems);
                }
            });

    this.allowance.set(allowance);
}
```

6. You'll also need to change the `updateAllowance` method to use the `ObservableInt` instead of the `Allowance` object:

```
public void updateAllowance(final CharSequence newAllowance) {
    try {
        allowance.set(Integer.parseInt(newAllowance.toString()));
    } catch (final RuntimeException ex) {
        //ignore
        allowance.set(0);
    }
}
```

7. Now, open the `AllowanceOverviewFragment` class.
8. Add a `SharedPreferences` field to the `AllowanceOverviewFragment`; we'll be using them more than once in this class:

```
private FragmentAllowanceOverviewBinding binding;
private SharedPreferences preferences;
```

9. Override the `onCreate` method of `Fragment` and retrieve the private `SharedPreferences` instance you'll be storing the daily allowance in. The first argument specifies the name of the `SharedPreferences` to retrieve, while the second specifies the scope as `private`, meaning only your application will be able to see or use this `SharedPreferences` instance:

```
@Override
public void onCreate(final Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    this.preferences = getContext().getSharedPreferences(
            "Allowance",
            Context.MODE_PRIVATE
    );
}
```

10. Create an `onCreateView` method to create an `AllowanceOverviewPresenter` and pass the `Fragment` instance as the `LifecycleOwner`, and to retrieve the current `allowancePerDay` from the `SharedPreferences`. The second argument passed to the `SharedPreferences.getInt` method is a default value that is returned if there is no existing value stored:

```
@Override
public View onCreateView(
        final LayoutInflater inflater,
        final ViewGroup container,
        final Bundle savedInstanceState) {

    this.binding = DataBindingUtil.inflate(
            inflater,
            R.layout.fragment_allowance_overview,
            container,
            false
    );

    this.binding.setPresenter(new AllowanceOverviewPresenter(
            this,
            preferences.getInt("allowancePerDay", 150)
    ));

    return this.binding.getRoot();
}
```

11. Finally, create an `onDestroy` method to store the allowance per day back in the `SharedPreferences` object. You do this by first requesting an `Editor` from the `SharedPreferences`, and then applying the changes. All the changes in an `Editor` are atomically applied at the same time (atomically):

```
@Override
public void onDestroy() {
    super.onDestroy();
    preferences.edit()
            .putInt("allowancePerDay",
this.binding.getPresenter().allowance.get())
            .apply();
}
```

Now, if you build and run the application, you'll notice that the allowance overview will correctly show you how much you've spent "today", "this week", and in total. Use the date selector in the `CaptureClaimActivity` to add a few claim items on different days and see how the user interface reacts and recalculates the amounts that you've spent.

Test your knowledge

1. An instance of `RecyclerView` will create one `View` instance for which of these?
    - Every item of data
    - Every item of data visible on the screen
    - Each type of data element that is also visible on the screen
2. When attaching an observer to `LiveData` you need to do which of the following?
    - Detach it when its `LifecycleOwner` is destroyed
    - Attach it on the main thread
    - Provide a valid `LifecycleOwner`
3. Overview / Dashboard screens should have which of these features?
    - They should only use graphs to display statistics
    - They should not scroll if it can be avoided
    - They should display an overview with the most important information first
4. The `ViewHolder` class is used by the `RecyclerView` to do what?
    - Improve the data binding performance
    - Reference the views that will be garbage collected
    - Store the `View` objects in a `Bundle`

5. When using `LiveData` objects to reference data used by multiple `Fragment` objects, which of these is true?
    - The `Fragment` instances must share the same `LiveData` reference to see changes
    - The `LiveData` will only update one of the `Fragment` instances
    - The `Fragment` classes must all extend `android.support.v4.app.Fragment`

Summary

Overview screens are the first thing a user will see and interact with in your application, and will be the area of the application they will spend most of their time in. It's important to keep the screen focused and opinionated on what data is displayed to the user, and how it's displayed. Always consider how long the user has to look at your screen, and what information they will need easy access to. Make use of the `RecyclerView` and `LiveData` classes to provide the user with detailed views arranged with the most important information first, and allow them to quickly scroll through their most important recent events.

It's also important to consider the navigation of your application, the various ways the user will leave your overview screen, and how they will get back to it. As far as possible, keep the `Overview` class responsible for just arranging the data on the screen. Any logic that takes the user away from the screen, for whatever reason, should be encapsulated in `Fragment` classes that also hold the logic to deal with their eventual return to the overview screen.

In this chapter, we looked at a very simple way to build an overview screen. There are many ways that these sorts of screens can be made more useful and powerful by reshaping the layout on the screen as the user scrolls and drags various elements of the user interface.

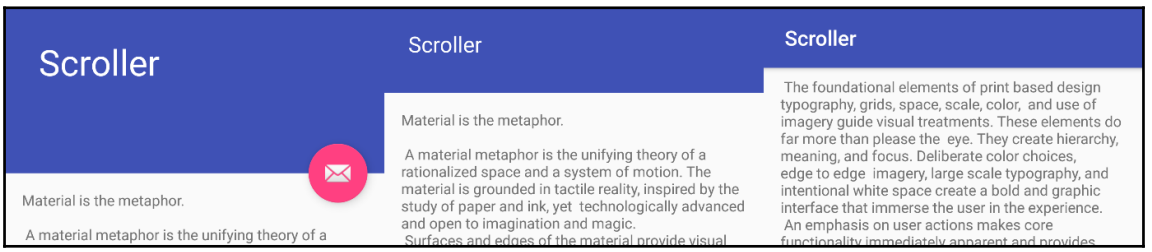
In the next chapter, we'll take a look at how to leverage some of the layout systems provided by the Material Design API to allow the user interface to change its shape and emphasis dynamically.

8

Designing Material Layouts

When designing and creating the layout for a screen, there are many different schools of thought about how it should be done. Modern layouts are often complex systems that change their shape dynamically as the user interacts with them. In the past, layouts tended to be quite rigid structures with only specialized areas such as windows or slit panels that could be adjusted by the user. However, a mobile application must make better use of their available space, since they're typically used on physically smaller devices. The direct interaction of a touch interface also changes how the user expects an application to behave; you need to not only react to the users gestures, but also be mindful of where their hand and fingers are likely to be on the screen as they might be obscuring some part of the screen, as they drag to scroll through your application.

The easiest way to see how a layout can change and adjust is with a jumbo collapsing toolbar. When the screen opens, the toolbar is full sized and takes up enough space to contain various additional widgets and information. As the screen is scrolled, the action button vanishes, and the toolbar collapses in size. Then, the toolbar pins itself to the top of the screen and remains visible as just the title and possibly some action buttons, as shown here:



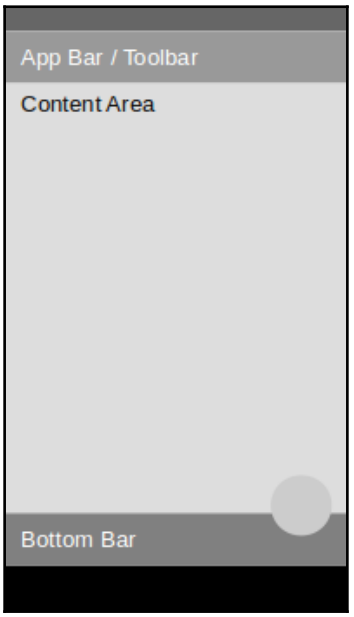
This collapsing behavior is something you see commonly in material applications--various parts of the user interface being shown or hidden as the user scrolls or changes what they are doing. These layouts often coordinate the moving, resizing, showing, and hiding of many different widgets at the same time, and there's a special class for that--the `CoordinatorLayout`.

In this chapter, we'll take a look at the `CoordinatorLayout` and some other specialist Android layout classes in order to do the following:

- Create layouts that change based on user actions
- Create layouts on flexible grids
- Allow the user to take actions using gestures
- Highlight some widgets above others using elevation

Looking at material structure

Material layouts have a selection of patterns that applications should follow for every screen they build. This sort of template is often called the **scaffolding**, and for mobile, it looks like this:



What is important about the scaffolding is that while it defines the basic layout of virtually every screen, it doesn't define how you should achieve this design, and even on Android, you'll find that there are several different ways of creating a screen with the preceding layout structure. Several elements are also optional: the **Bottom Bar** and floating action button are often left out because they aren't useful to a screen. The **App Bar** appears on most screens, but can be much larger and can also be folded away to provide the user with more reading space in the content area.

It's also important to understand that by default, the platform theming will put the **App Bar** (presented by the `ActionBar` class) into an `Activity` for you; it's also common to create your own **App Bar** using the `Toolbar` class and the `NoActionBar` theme on the `Activity`. In fact, in `Chapter 2`, *Designing Form Screens*, when you created the `CaptureClaimActivity`, the Android Studio template did exactly this:

```
<activity
    android:name=".CaptureClaimActivity"
    android:label="@string/title_activity_capture_claim"
    android:theme="@style/AppTheme.NoActionBar" />
```

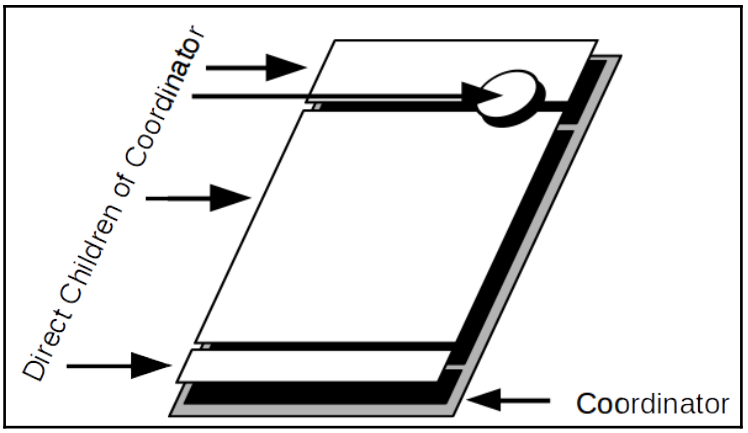
In the `CaptureClaimActivity` class, near the top of the `onCreate` method, you'll be able to find the following code snippet:

```
Toolbar toolbar = findViewById(R.id.toolbar);
setSupportActionBar(toolbar);
```

This code allows your application to take complete control of the `Toolbar`, from what it looks like, to what widgets it contains. Setting it as the `SupportActionBar` tells the `AppCompatActivity` to delegate any calls to `Activity.setTitle` and similar methods to the `Toolbar`, but in no other way changes how the `Toolbar` interacts with the layout system. This is still firmly under your control now.

Introducing CoordinatorLayout

Android has a small family of layouts designed to work together to achieve the dynamic movement effects when the user scrolls. At the core of this group is the `CoordinatorLayout` class, which allows complex behaviors to be attached to any number of floating sibling widgets that can depend on and react to each other's position and size. To illustrate how a `CoordinatorLayout` actually works, take a look at this diagram:



Even though the `FloatingActionButton` appears to be floating above the other widgets, it's a direct child of the `CoordinatorLayout`. It remains in place because it is anchored to the bottom of the toolbar. If the toolbar changes its size or position, the `CoordinatorLayout` will move the `FloatingActionButton` so that it appears to be attached to the bottom of the toolbar. These movements are all done together as part of the layout process, resulting in every frame being pixel perfect and everything appearing to move and resize together.

`CoordinatorLayout` defines two major ways to manipulate its child widgets--anchors and behaviors:

- **Anchors** are by far the simpler of these two; they simply attach one widget to another widget. The anchors respond to the `layout_gravity` attribute and a special `layout_anchorGravity` attribute to determine exactly where the anchored widget should appear relatively to the widget it's attached to.

- **Behaviors** are more complex; they are entire classes that can be used to manipulate the widget in any way based on other widgets (known as its **dependencies**). Several classes define their own behavior classes that should be used when they are declared within a `CoordinatorLayout`. For example, `FloatingActionButton` declares a `FloatingActionButton.Behavior` class that will hide the button if its anchor-point approaches too close to the end of the screen, and make it reappear when there is enough space again. This showing and hiding behavior is even accompanied by an animation.

Coordinating the Overview Screen

The Overview screen you built in `Chapter 7`, *Creating Overview Screens*, is a perfect candidate for a `CoordinatorLayout`. To start with, the allowance overview bar can be made to collapse, and unfold as the user scrolls. This allows more space for the claim items on the screen as they are scrolling, and by expanding the overview again when they scroll upward, the user doesn't have to scroll all the way to the top to get the overview back.

This behavior won't just use the `CoordinatorLayout`, but will also need the help of the `AppBarLayout` and `CollapsingToolbarLayout` classes as you'll need to take control of the Material Design scaffolding to make it work. Follow these steps to move the allowance overview into the header bar and make it collapse:

1. First, open the `AndroidManifest` file from the **manifests** folder in the project tree (use the **Android** perspective).
2. Find the `OverviewActivity` entry and add a theme attribute that will tell the system not to provide a system `ActionBar`, because you'll be adding your own:

```
<activity
    android:name=".OverviewActivity"
    android:theme="@style/AppTheme.NoActionBar"
    android:windowSoftInputMode="stateHidden">
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER"
/>
    </intent-filter>
</activity>
```

3. Now, open the `activity_overview` layout file and change to the text mode. Remove the `FrameLayout` and all of its contents; you'll need to completely rewrite this file.

4. Create the `CoordinatorLayout` root element with all the standard namespaces and context. Note that this time you'll tell the system that this widget will fit to the root window, and not act as *contents*:

```
<android.support.design.widget.CoordinatorLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:fitsSystemWindows="true"
    android:id="@+id/scaffolding"
    tools:context="com.packtpub.claim.OverviewActivity">
</android.support.design.widget.CoordinatorLayout>
```

5. Now, create the `AppBarLayout` element within the `CoordinatorLayout`; again, you'll tell the system that the `AppBarLayout` fits to the system window and is not to be treated as normal *content* widgets:

```
<android.support.design.widget.AppBarLayout
    android:id="@+id/app_bar"
    android:layout_width="match_parent"
    android:layout_height="@dimen/app_bar_height"
    android:fitsSystemWindows="true"
    android:theme="@style/AppTheme.AppBarOverlay">
</android.support.design.widget.AppBarLayout>
```

6. Use the code-assistance for the `layout_height` to create a new dimension resource named `app_bar_height`, and assign it a value of `180dp`:

```
<dimen name="app_bar_height">180dp</dimen>
```

7. Within the `AppBarLayout`, you need to declare the `CollapsingToolbarLayout`. This will handle the collapsing and expanding of the toolbar and other widgets as the user scrolls. You use the `layout_scrollFlags` to tell it how to collapse and expand, but it's important to note that it's actually the `AppBarLayout` that takes care of these, so any children of `AppBarLayout` can use these flags. In this particular case, we'll be telling it to collapse when the user scrolls down the list of items, but not to exit (vanish) completely, and to reenter as soon as the user starts to scroll up the list again:

```
<android.support.design.widget.CollapsingToolbarLayout
    android:id="@+id/toolbar_layout"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
```

```
    android:fitsSystemWindows="true"
    app:contentScrim="?attr/colorPrimary"
    app:expandedTitleGravity="top"
app:layout_scrollFlags="scroll|enterAlwaysCollapsed|snap|exitUntilC
ollapsed"
    app:toolbarId="@+id/toolbar">
</android.support.design.widget.CollapsingToolbarLayout>
```

In the preceding code, the `CollapsingToolbarLayout` declares its `contentScrim` as `?attr/colorPrimary`. The attributes `?` syntax is a type of lookup that is used with theming. It tells the resource-system to look up that attribute in the theme, rather than directly referencing the attribute.

8. Within the `CollapsingToolbarLayout`, you'll need to declare a `Toolbar` widget. This widget will take the place of the system `ActionBar`. We use the `layout_collapseMode` to tell the `CollapsingToolbarLayout` to *pin* the `Toolbar` to the top of the screen once it has been collapsed (rather than having it vanish completely):

```
<android.support.v7.widget.Toolbar
    android:id="@+id/toolbar"
    android:layout_width="match_parent"
    android:layout_height="?attr/actionBarSize"
    app:layout_collapseMode="pin"
    app:popupTheme="@style/AppTheme.PopupOverlay" />
```

9. After the `Toolbar` widget, you can declare the `AllowanceOverviewFragment`; it will use the `parallax` collapse mode and will disappear as the use scrolls down the list of claim items:

```
<fragment
    android:id="@+id/overview"
    class="com.packtpub.claim.ui.AllowanceOverviewFragment"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_gravity="bottom"
    android:layout_marginBottom="@dimen/grid_spacer1"
    app:layout_collapseMode="parallax"
    app:layout_collapseParallaxMultiplier="0.65" />
```

10. That concludes your new `AppBarLayout` structure; now you need to add the `RecyclerView` in after the `AppBarLayout` and tell the `CoordinatorLayout` that it is scrolling content using the `layout_behaviour` attribute. This will tell the `CoordinatorLayout` that when the `RecyclerView` is scrolled, the `AppBarLayout` should react to the scrolling:

```
<android.support.v7.widget.RecyclerView
    android:id="@+id/claim_items"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:layout_marginTop="@dimen/grid_spacer1"
    android:clipToPadding="false"
    android:clipChildren="false"
    android:paddingLeft="@dimen/grid_spacer1"
    android:paddingRight="@dimen/grid_spacer1"
app:layoutManager="android.support.v7.widget.LinearLayoutManager"
    app:layout_behavior="@string/appbar_scrolling_view_behavior" />
```

The `RecyclerView` declared that its behavior references a string resource by the name `appbar_scrolling_view_behavior`, but you haven't declared any such resource in your `strings.xml` file, so why doesn't the code assistant complain? This is a string resource declared by the `CoordinatorLayout` support library, and it gets merged into your application resources during the build. Its contents is the full class-name for the scrolling view `Behaviour` implementation (that is:).

11. The final element in your `CoordinatorLayout` should be the `NewClaimItemFloatingActionButtonFragment`, which will automatically gain special behavior within the `CoordinatorLayout` because of how the `FloatingActionButton` class is written:

```
<fragment
    android:id="@+id/new_claim_item"
class="com.packtpub.claim.ui.NewClaimItemFloatingActionButtonFragme
nt"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_gravity="bottom|end"
    android:layout_margin="@dimen/fab_margin" />
```

The `FloatingActionButton` class declares a default `Behaviour` class that the `CoordinatorLayout` looks for when any child is added to it. This defines how the `FloatingActionButton` is positioned on the screen, when it should disappear, reappear, and even move relative to panels that may appear at the bottom of the screen (such as snackbars). The declaration is made using a publicly accessible annotation:

```
@CoordinatorLayout.DefaultBehavior(FloatingActionButton.Behavior.cl
ass)
public class FloatingActionButton extends
VisibilityAwareImageButton {
```

Due to how your application has been structured, the `OverviewActivity` class doesn't need to be modified for this new layout to work. It will still automatically populate the `RecyclerView` with the `ClaimItem` objects, and the fragments will communicate through the database. It is, however, useful to make the new `Toolbar` widget act as the `ActionBar` for the `OverviewActivity`; you can do this by changing the `onCreate` method to call `setSupportActionBar`:

```
protected void onCreate(final Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_overview);

    setSupportActionBar(findViewById(R.id.toolbar));
    // ...
}
```

Swiping to delete

While you have a way for your users to create claim items, the users have no way to delete the claim items that they have created. A common pattern in lists on mobile apps is to allow the user to swipe right to dismiss or delete items. `RecyclerView` provides some excellent and easy-to-use structures to enable this sort of behavior; however, it's always important to ensure that users don't delete items by mistake.

In the past, most user interfaces used confirmation dialogs when executing destructive actions. However, these "are you sure" dialogs are a horrible distraction for most users, because such messages violate a key principle--the application assumes that the user probably doesn't want to perform an action that they just took. In reality, the user probably did mean to delete the item, but the application interrupts them to ask whether they're sure about their choice. A much better behavior is to assume that the user does want to take the action, but then to give them a way to undo their action if they have made a mistake. Material Design has a design pattern and widget dedicated specifically to this sort of task--the `Snackbar`.

In Material Design language, the `Snackbar` is a small bar that can appear at the bottom of the screen, providing the user with information and possible actions that they can take based on the information given. The most commonly seen use is when something is deleted, and the user is given the chance to undo the delete. Undo actions might appear to be challenging, but they're actually very simple to carry out if correctly wrapped in a `Command` class. Follow these steps to add a swipe-to-delete action and an undo option to the travel claims application:

1. Open the `DataBoundViewHolder` class in the **ui** package.
2. Your new classes will need a simple way to access the items from the `DataBoundViewHolder`, but the `ViewDataBinding` doesn't offer a `getVariable` method, so you'll need to keep it in a class field and provide a getter method:

```
private I item;
public I getItem() { return item; }
```

3. You'll also need to modify the `setItem` method to capture this field:

```
public void setItem(final I item) {
    this.item = item;
    binding.setVariable(BR.item, item);
}
```

4. Open the `OverviewActivity` source file in Android Studio.

5. At the bottom of the `OverviewActivity` class, you'll need to declare a new `ActionCommand` class that will encapsulate both the delete action and the undo operation. Unlike most other `ActionCommand` classes, this one is not reusable and takes no arguments:

```
class DeleteClaimItemCommand
        extends ActionCommand<Void, Void>
        implements View.OnClickListener {
}
```

6. The new `DeleteClaimItemCommand` class will need a reference to the `ClaimDatabase`, and will also have a `ClaimItem` field that it will delete and optionally restore:

```
private final ClaimDatabase database =
ClaimApplication.getClaimDatabase();
private final ClaimItem item;
public DeleteClaimItemCommand(final ClaimItem item) {
    this.item = item;
}
```

7. The `onBackground` implementation deletes the `ClaimItem` object from the database, but the `DeleteClaimItemCommand` keeps a reference to the in-memory implementation if the user decides to restore it:

```
public Void onBackground(final Void noArgs) {
    database.claimItemDao().delete(item);
    return null;
}
```

This code does not delete the `Attachments` related to the `ClaimItem`, which will cause the application to leak attachment files and database rows. In practice, you would also want to ensure that the attachments are cleaned up as well by copying the behavior used for `ClaimItem`, but that is beyond the scope of this example.

8. The `onForeground` implementation will need to display a `Snackbar` notification telling the user that the item was deleted; for that, you'll need a localizable message. The `Context` class offers a convenience `getString` method that will generate a formatted string from the application resources:

```
final String message = getString(
        R.string.msg_claim_item_deleted,
        item.getDescription());
```

9. Use the code assistance to create a new string resource named `msg_claim_item_deleted`:

```
<string name="msg_claim_item_deleted">%s Deleted</string>
```

These strings follow the formatting rules defined in `java.util.Formatter` or `String.format`, allowing you to create relatively complex formatting rules. By providing different `strings.xml` files for different languages and formats like this, you can very easily localize most strings in your application.

10. In the `onForeground` method, you'll need to grab a reference to the `CoordinatorLayout` as a base for the `Snackbar`:

```
final View scaffolding = findViewById(R.id.scaffolding);
```

11. Then, create the `Snackbar` object, specifying its undo action text, and use the `DeleteClaimItemCommand` as the action handler (`OnClickListener`):

```
Snackbar.make(scaffolding, message, Snackbar.LENGTH_LONG)
        .setAction(R.string.undo, this)
        .show();
```

12. Use the code assistance on the `R.string.undo` reference to create a new string resource for the text of the `undo` action:

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

13. If the user clicks on the undo action, the `DeleteClaimItemCommands` and `onClick` methods will be invoked. It can then use its cached reference to the deleted `ClaimItem` to restore it in the database:

```
public void onClick(final View view) {
AsyncTask.SERIAL_EXECUTOR.execute(database.createClaimItemTask(item
));
}
```

14. As another inner-class to the `OverviewActivity`, you'll need a class to provide the action definition and handling for the *swipe-to-delete* behavior. This new class will extend the `SimpleCallback` class of the `ItemTouchHelper` class, which provides the handling for the movement gesture recognition:

```
private class SwipeToDeleteCallback extends
ItemTouchHelper.SimpleCallback {
}
```

15. The `SimpleCallback` constructor takes two sets of "flags" in the form of `int` values. These are simply a number of numbers that you can binary "or" (using the `|` operator) together. These define the different gestures to allow and manage. The first of these are the flags for different types of "move" gestures that can be used to reorder the items in a `RecyclerView` (leaving this as zero indicates that no move gestures should be recognized). The second flag's argument is for "swipe" gestures, which are what we're interested in here:

```
SwipeToDeleteCallback() {
    super(0, ItemTouchHelper.RIGHT);
}
```

16. The `SimpleCallback` class requires that you declare handler methods for moving and swiping, even though the class will not deal with movement gestures. You'll need to declare `onMove`, but the class can just return `false` as its implementation:

```
public boolean onMove(
        final RecyclerView recyclerView,
        final RecyclerView.ViewHolder viewHolder,
        final RecyclerView.ViewHolder target) {

    return false;
}
```

17. Next, you can define the implementation of the `onSwipe` method, which will create a `DeleteClaimItemCommand` and execute it:

```
public void onSwiped(
        final RecyclerView.ViewHolder viewHolder,
        final int direction) {

    final DataBoundViewHolder<?, ClaimItem> holder
           = (DataBoundViewHolder<?, ClaimItem>) viewHolder;
    new DeleteClaimItemCommand(holder.getItem()).exec(null);
}
```

18. Now, to attach the `SwipeToDeleteCallback` to the `RecyclerView`, you need to wrap it using the `ItemTouchHelper` class, and attach that to your `RecyclerView` instance at the bottom of the `onCreate` method:

```
final RecyclerView claimItems = findViewById(R.id.claim_items);
    claimItems.setAdapter(new ClaimItemAdapter(
            this, this,
ClaimApplication.getClaimDatabase().claimItemDao().selectAll()
    ));

    new ItemTouchHelper(new SwipeToDeleteCallback())
            .attachToRecyclerView(claimItems);
}
```

Elevating widgets

An excellent way of highlighting one widget over the others on the screen is to make it appear over the others on the screen, not two-dimensionally, but floating above them as though in three-dimensions. This is already a clear pattern if you look at the `FloatingActionButton` classes; they don't just overlap other widgets, but they have a shadow and appear to float in space (hence the class name `FloatingActionButton`).

One of the great features in the Android widget library is that the `View` class defines the notion of elevation, which makes it usable by every widget in the toolkit. The elevation of a widget doesn't affect its two-dimensional position or size, but does cause it to produce a shadow that will be correctly shaded as though the widget is floating in three-dimensions. This can be used to create an amazing effect when you need to draw attention to a message, or when the user is repositioning a widget on the screen (for example, reorganizing a list of reminders). Given that most of a Material Design user interface is flat, adding a three-dimensional elevation instantly makes a widget stand out for the user.

Much like the borders and shadow of a `CardView` widget, when you use elevation, you need to ensure that the shadow is not clipped by the parent widget or the padding attributes. Use the `clipChildren` and `clipToPadding` attributes to control this.

Follow these steps to add an elevation effect to the swipe-to-delete behavior callback:

1. Open the `OverviewActivity` and find the `SwipeToDeleteCallback` inner class.
2. The class will need to be able to reset the elevation if the user "drops" an item after picking it up to delete it. For this, the `SwipeToDeleteCallback` class will need a field with the default card elevation:

```
final float defaultElevation =
getResources().getDimensionPixelSize(R.dimen.cardview_default_eleva
tion);
```

3. Every time a child of the `RecyclerView` is drawn after being picked up, the `ItemTouchHelper` allows you to override the drawing behavior. In your case, you want to adjust the elevation of the card relative to how far right the user has dragged it. In order to work on older versions of Android, this code uses the `ViewCompat` class to change the elevation:

```
public void onChildDraw(
        final Canvas c,
        final RecyclerView recyclerView,
        final RecyclerView.ViewHolder viewHolder,
        final float dX, final float dY,
        final int actionState,
        final boolean isCurrentlyActive) {

    if (isCurrentlyActive) {
        ViewCompat.setElevation(
                viewHolder.itemView,
                Math.min(
                        Math.max(dX / 4f, defaultElevation),
                        defaultElevation * 16f
                )
        );
    }

    super.onChildDraw(c, recyclerView, viewHolder, dX, dY,
actionState, isCurrentlyActive);
}
```

4. Once the card is released by the user, we need to clear the elevation value by setting it back to its default; the `ItemTouchHelper` will invoke the `clearView` callback when the user drops an item:

```
public void clearView(
        final RecyclerView recyclerView,
        final RecyclerView.ViewHolder viewHolder) {
    ViewCompat.setElevation(viewHolder.itemView, defaultElevation);
    super.clearView(recyclerView, viewHolder);
}
```

Once you've implemented this behavior, the user will receive secondary visual feedback on the swipe to delete gesture, as the card they drag will appear to rise above the others as they pull it to the right. It'll also do the reverse and appear to descend back to its normal elevation if they drag it to the left again. This elevation feedback can be even more useful on interfaces where the user can change the position of the cards in a list (for example, in a to-do list). Note that as the card's elevation increases, it casts a shadow on the cards both below and above it:

Building layouts using grids

When building screens, it's common to want specific widgets to appear the same size and shape as other widgets. This is often achieved using flexible grid models for the layout. By dividing the screen into a number of cells, and having each widget occupy one or more cells, you can create very complex layouts that will stretch to any screen size. However, this traditional model is completely outdated when faced with `ConstraintLayout`, which is capable of maintaining complex relationships between widgets without the need for a grid.

In most situations, `ConstraintLayout` should be more than capable of managing any complex layout you choose to design, and will be much more flexible than a grid/table layout manager. Unlike a grid-based layout engine, `ConstraintLayout` is much more flexible when dealing with widgets that are sized based on a font or images that can be various sizes, depending on the physical screen size and pixel-density. While `GridLayout` will adjust the size of the cells to accommodate such widgets, they are still confined by grid lines.

However, every now and then, you'll need to build a layout based on grid cells. For situations like this, you'll want to use the `GridLayout` class. `GridLayout` allows you to define layouts based on an invisible grid, where each widget can take up one or more cells, and the size of each row and column is flexible; that is, each column can be a different width, and each row can be a different height. It's important to remember that `GridLayout` is not intended for displaying large tables of data, but rather for laying out screens that favor a grid-like structure. If you need to present your user with a scrolling grid (for example, of icon images), then a better model to be used is the `RecyclerView` with a `GridLayoutManager`, as this will scale to virtually any number of child components.

There are two different implementations of `GridLayout` in Android: one is in the platform core APIs, and the other is in the support v7 APIs. For compatibility reasons, it's typically best to use the class from the support package as it includes many of the more recently added features that might not appear on the platform implementation.

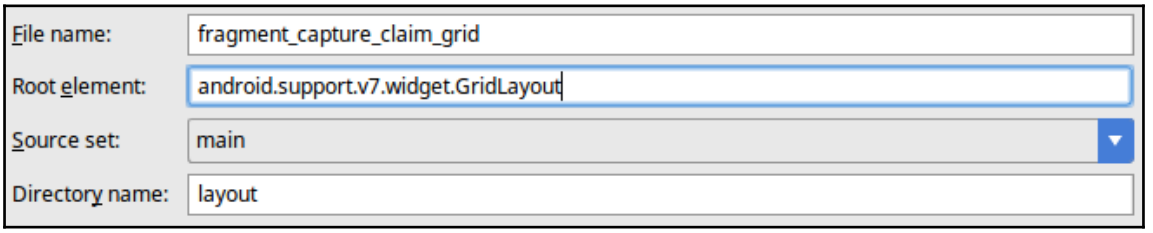
To explore the `GridLayout` a bit, let's take a look at how you will implement the *Capture Claim Details* card using `GridLayout` instead of `ConstraintLayout`:

1. First, you'll need to add the `GridLayout` implementation to your project. Open the **Gradle Scripts** in the project tree and open the **build.gradle** for the **app** module (use the **Android** perspective).
2. In the dependencies list, add a dependency for the grid-layout module:

```
implementation 'com.android.support:appcompat-v7:26.0.0'
implementation 'com.android.support:gridlayout-v7:26.0.0'
```

The version number at the end (in this case, `26.0.0`) must exactly match the version number of the `appcompat` module your application references. If these versions don't match exactly, it can lead to instabilities and in some cases, your application won't even compile. Change the version number to match the one declared on the `appcompat` reference in your `build.gradle` before continuing to the next step.

3. Save the file, and synchronize the project using the **Sync Now** link that appears at the top of the editor.
4. Right-click on the **res** | **layout** directory in the project file tree, and select **New** | **Layout resource file**.
5. Name the new file `fragment_capture_claim_grid`.
6. Change the root element to `android.support.v7.widget.GridLayout`:



7. Change to the **Text** mode editor.
8. As you're using the support libraries `GridLayout` implementation, several of the XML attributes will be in the `app` namespace rather than the platform (`android`) namespace. You'll need to add the `app` namespace to the `GridLayout` declaration:

```
<android.support.v7.widget.GridLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
</android.support.v7.widget.GridLayout>
```

9. `GridLayout` sets many of the layout attributes by default, and assumes, by default, that each child is in the cell, following the one before it (starting at the top-left cell). It allows you to specify the `columnWeight` and `rowWeight` attributes to define how much of the available space each cell should take up. Declare a `TextInputLayout` to take up 70% of the available space:

```
<android.support.design.widget.TextInputLayout
app:layout_columnWeight="0.7">
    <android.support.design.widget.TextInputEditText
        android:id="@+id/description"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:hint="@string/label_description" />
</android.support.design.widget.TextInputLayout>
```

The preceding `TextInputLayout` widget only occupies a single cell within the `GridLayout`, but that cell has been told to take up 70% of the available horizontal space when rendering.

10. Next, declare the amount `TextInputLayout`; this will also only occupy a single cell, but we'll want it to occupy the remaining 30% of the horizontal space:

```
<android.support.design.widget.TextInputLayout
app:layout_columnWeight="0.3">
    <android.support.design.widget.TextInputEditText
        android:id="@+id/amount"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:hint="@string/label_amount" />
</android.support.design.widget.TextInputLayout>
```

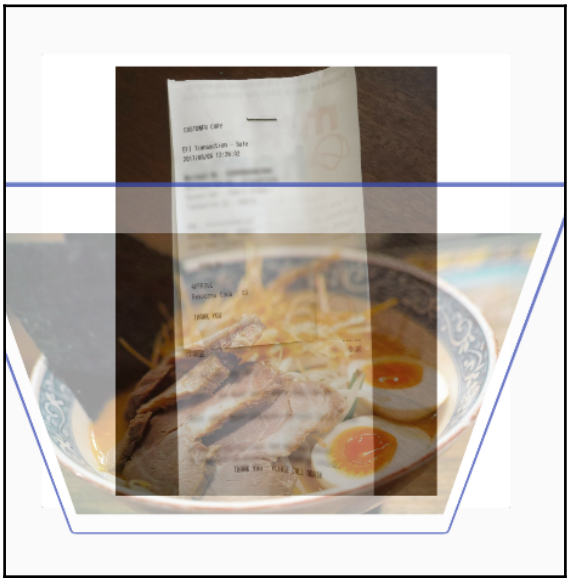
11. Now, we want to declare a `DatePickerLayout` for the user to select the date, but we need to tell the `GridLayout` to put it on the next row. You do this using the `row` and `column` attributes. This widget also needs to take up the full width of the `GridLayout`, which means it needs to occupy two columns, which is done using the `columnSpan` attribute:

```
<com.packtput.claim.widget.DatePickerLayout
    android:id="@+id/date"
    app:layout_row="1"
    app:layout_column="0"
    app:layout_columnSpan="2"
    app:layout_gravity="fill_horizontal" />
```

If you take a look in the **Design** view, you'll note that this layout looks almost identical to the one you wrote in `Chapter 2`, *Designing Form Screens*, for capturing the claims. The biggest difference is that the `ConstraintLayout` uses a fixed minimum size for the **Amount**, while this layout uses relative sizes by manipulating the weights of the grid cells. The resulting layout should look like this:

Stack view

Sometimes, it's useful to be able to display long lists of items with only one item visible at a time, for example, the list of attachments for a `ClaimItem`. In this case, you can use the side-to-side `ViewPager` as you've already done, but there is another option--the `StackView`. The `StackView` class presents its contents as a three-dimensional stack of cards, with the "top" card fully visible, and some of the cards "behind it," as shown:



This is often a very useful pattern, as it provides the user with plenty of screen space to view the top item, while also being able to see that there are other items that can be viewed. This makes it ideal for displaying photos or large cards of data. It's very similar to how Android displays the list of running applications when you tap on the "Recent Apps" button on a device.

The `StackView` is a classic `Adapter` view, and works using the same `Adapter` implementations as `ListView` or `GridView`. If done correctly, you can write code that can be used in any of these classes; follow these steps to build a simple `StackView` and `Adapter` implementation that can be used to preview the `Attachment` differently:

1. Right-click on the **ui.attachments** package in the project tree, and select **New|Java Class**.
2. Name the new class `AttachmentListAdapter`.
3. Change the **Superclass** to `android.widget.BaseAdapter`.
4. Click on **OK** to create the new class.
5. In the new `AttachmentListAdapter` class, declare a `List` of `Attachment` objects to present to the user:

```
private List<Attachment> attachments = Collections.emptyList();
```

6. Create a constructor to observe a `LiveData` and assign the `List` of attachments and notify the `StackView` when things change:

```
public AttachmentListAdapter(
        final LifecycleOwner lifecycleOwner,
        final LiveData<List<Attachment>> attachments) {

    attachments.observe(lifecycleOwner, new
Observer<List<Attachment>>() {
        @Override
        public void onChanged(final List<Attachment> attachments) {
            AttachmentListAdapter.this.attachments =
                    attachments != null
                            ? attachments
                            : Collections.<Attachment>emptyList();
            notifyDataSetChanged();
        }
    });
}
```

7. Much like a `RecyclerView.Adapter` implementation, the `BaseAdapter` needs a method to access the number of items it's expected to present:

```
public int getCount() { return attachments.size(); }
```

8. Unlike the `RecyclerView.Adapter` implementations, however, a `BaseAdapter` is expected to expose the underlying data directly. It's also required to expose a unique ID for each element of data:

```
public Object getItem(final int i) { return attachments.get(i); }
public long getItemId(final int i) { return attachments.get(i).id;
}
```

9. Also, unlike a `RecylcerView.Adapter`, there is only a single method to create and reuse the existing view items, as well as bind the data to them. In this method, the second argument may be `null`, but can also be an existing view that is expecting to be recycled:

```
public View getView(
        final int i,
        final View view,
        final ViewGroup viewGroup) {

    AttachmentPreview preview = (AttachmentPreview) view;
    if (preview == null) {
        preview = new AttachmentPreview(viewGroup.getContext());
    }

    preview.setAttachment(attachments.get(i));

    return preview;
}
```

10. To use a `StackView` from a layout XML file, you simply need to declare the `StackView` as you would a `RecyclerView` or `ViewPager`:

```
<StackView
    android:id="@+id/attachments"
    android:layout_width="match_parent"
    android:layout_height="match_parent" />
```

11. Then, from the enclosing `Activity` or `Fragment`, you will need to set its `Adapter`. Much like the `RecyclerView`, the `Adapter` can also be specified from a data-bound layout. Here's how the code might look in an `Activity`:

```
final StackView attachments = findViewById(R.id.attachments);
attachments.setAdapter(new AttachmentListAdapter(
        this,
        database.attachmentDao().selectForClaimItemId(claimItem.id)
));
```

The `StackView` class is an excellent way to present the user with a large number of larger and more visual items. It's excellent for browsing things such as photographs or preview graphics, and provides you with an easy-to-use three-dimensional transformation. Before using `StackView`, you should always consider whether the user will need to see the data from more than one item at a time. Sometimes, it's best to combine a `RecyclerView` as an "overview" with a `StackView` for viewing the individual items.

Test your knowledge

1. Elevation should be used for which of the following?
    - When the user selects an item in a list
    - To selectively highlight one item above a flat layout
    - When a user swipes to delete items
2. `CoordinatorLayout` can be used to coordinate movement and size between which of these?
    - The components nested in an `AppBarLayout`
    - Any of its direct child widgets
    - `Fragment` in different activities
3. To change elevation of a widget in a backward-compatible way, you need to do which of the mentioned?
    - Nest the widget in a `CardView`
    - Use the `ViewCompat` class
    - Use Java reflection to call `setElevation`

4. The `GridLayout` class should be used in which of the following conditions?
    - When `ConstraintLayout` is not available
    - To display large tables of data
    - To lay out screens along grid lines

Summary

Taking ownership of what is normally the system's decorations for your application provides you with a huge variety of additional flexibility and power. By using the `CoordinatorLayout` to host the scaffolding and content of your screens, you further extend your flexibility by allowing widgets to dynamically interact with each other, even when being animated. This provides you with a way to produce pixel-perfect screens with minimal additional work.

Using layouts that cannot only dynamically change their shape, but also change their content using gestures such as swipe-to-dismiss to further enhance the direct manipulation aspects of a touchscreen user interface have been covered. At the same time, it's important to always consider the interactions of your user and when to interrupt them, especially around destructive actions. While there are still times where you might want to use a confirmation dialog, it's generally better to give users a method to undo their actions. It's often a very simple thing to keep the deleted entity objects in-memory until the `Snackbar` vanishes and is released from memory. Inserting entities that you have deleted from Room will actually maintain their IDs, meaning that their rows are restored exactly as they were before they were deleted.

In the next chapter, we'll take a look at navigation in Android applications, and explore various user interface features that are provided for your user to navigate your application. We'll also take a look at some techniques that allow you to take more control of your application's navigation flow. Providing consistent and quality navigation makes a huge difference to your user's experience.

9
Navigating Effectively

Broadly speaking, navigation is how your user gets from one screen to another in your application. More specifically, however, it's what a user needs to do in order to reach a goal within your application. Navigation is an almost completely invisible part of your application's user interface design. It's an area that is often neglected, frequently done badly, and as a result, often leads users to frustration.

The problem is that the navigation design of an application is often a side effect of the user interface design, rather than something that has been planned. Navigation, just like a single screen, can and should be designed to center around the user rather than the designer or developer. Using the techniques you've already learned in this book, you should be able to make almost any navigation flow work easily, because the elements should not be tightly coupled to each other.

In this chapter, we'll look at navigation and navigation patterns within the Material Design language. You'll learn how to do the following:

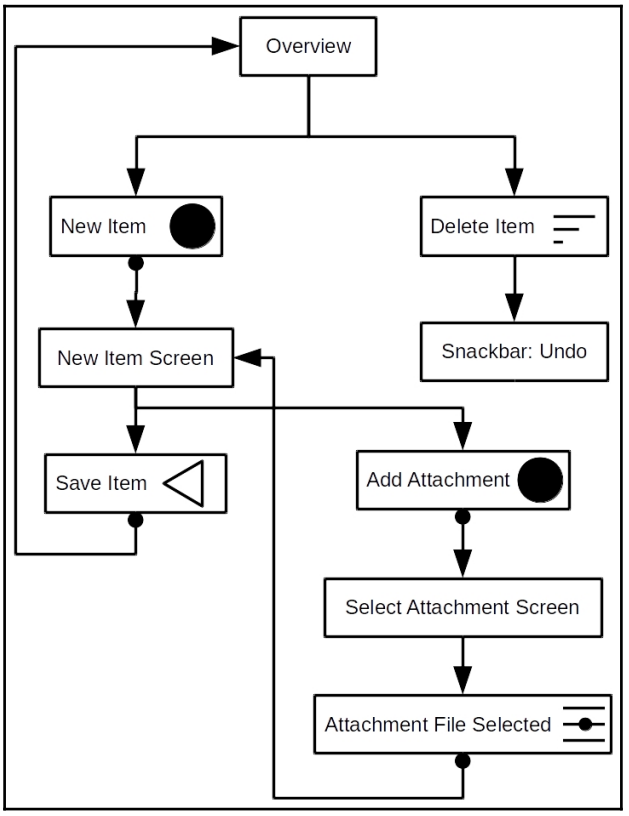
- Plan and design the navigation flow of an application
- Use the standard navigation menu component
- Build tabbed navigation applications
- Navigate using fragments instead of Activities

Planning navigation

Before leaping into your latest app idea, it's a good idea to stop and consider what you are trying to allow the user to do, and figure out how they will actually do it. One of the best ways to do this is with the decision tree or navigation tree. These can be easily drawn on paper, or if you're collaborating with other people, index cards on a magnetic whiteboard (or even a pin-board) are very effective.

The idea is to not just draw out the possible screens in your app, but how the user will get to each one. Navigation diagrams don't just help to define what screens your application actually needs, but will help ensure that the user will never be "lost" in your application. If the navigation lines become too complex, then you need to simplify the navigation (possibly by adding or removing some of the screens). Overly complex navigation is often hidden in the use of an application, but when drawn on a diagram, the complex relationships between screens become obvious, and often, a solution will become apparent.

To start drawing your diagram, create a box or card that represents the user's main entry into the application. Then, branch from each possible action that the user might take from that screen. For each action, draw a simple icon or describe the type of action the user is expected to take. For example, a circle can represent a floating action button, three staggered lines can represent a swipe gesture, and so on. These icons will also help by ensuring that the gestures and actions on a screen remain obvious to the user, and will help you avoid navigation techniques that hide the behaviors from the user. Here's an example navigation diagram for the current state of the travel claims app:



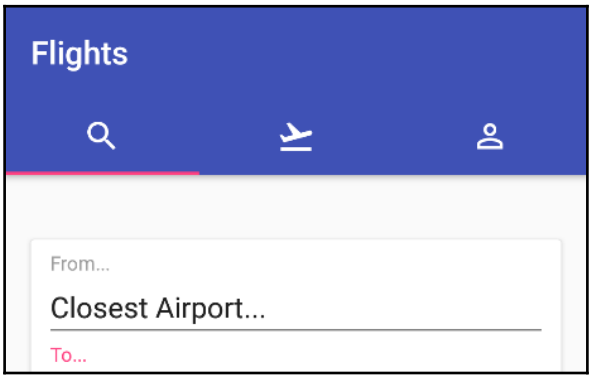
It's immediately apparent from the diagram that everything flows deeply into the application, and that there are currently three distinct action areas: **New Item**, **Delete Item**, and **Add Attachment**. Larger applications should still have these logical groupings of action areas, and should not have navigation lines that have to crossover too much of the diagram. If they do, it's a sign of an overcomplex navigation structure, and moving things around on the diagram will often help you produce a better and more intuitive application.

Now, let's take a look at various Android components built specifically for navigation.

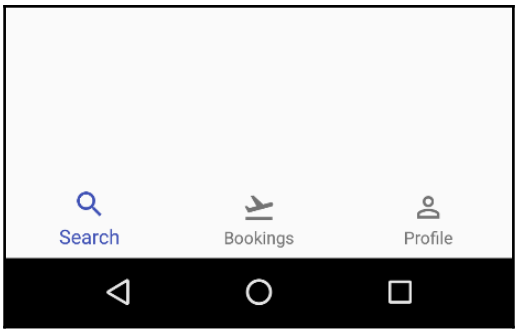
Tabbed navigation

When applications are broken down into a small number of logical areas, tabs often become the most obvious and simplest method of navigation. Most application's navigation is deeply hierarchical, and in these cases, tabs are not a good choice for a navigation mechanism. Tab navigation is best used when each tab will be used roughly as much as each of the other tabs (that is, they have roughly equal importance). There are two major types of tabbed layouts in Android: bottom tabs and top tabs (also known as action bar tabs or toolbar tabs).

Top tabs are the classic method of adding tabs to an Android application, and are perfect when the application areas are not switched between often. This is because they are at the top of the screen and typically far away from where the user's fingers are. Most typically, a user has their fingers near the bottom of the screen, close to the software keyboard and system navigation buttons:



Bottom tabs, on the other hand, are a far more subtle and challenging navigation technique to implement effectively. Bottom tabs take up significantly more vertical screen space than their top-bar cousins, and therefore need to *work* for the extra space they consume. Bottom tabs are good to implement if the user will switch between the spaces frequently and spend roughly the same amount of time in each. As they are at the bottom of the screen, they are typically more accessible to the user and therefore it's easier for them to switch between the screens they offer:



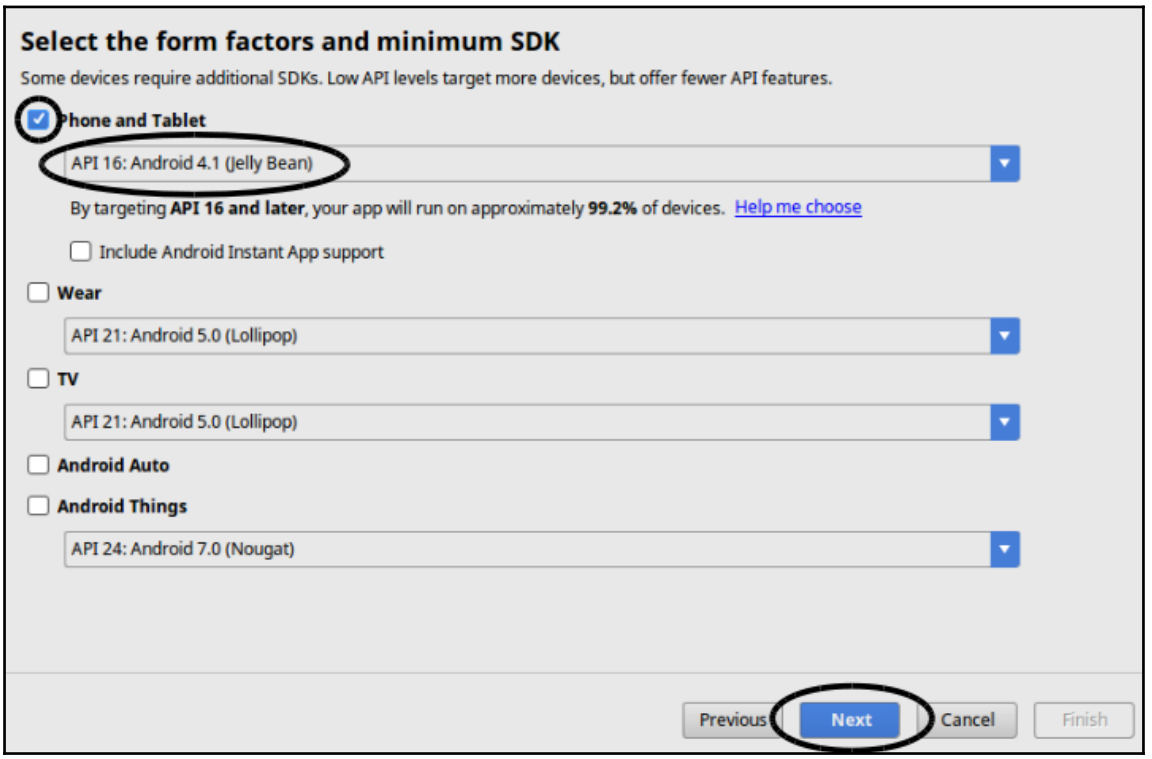
With either of these tab-based navigation options, it's important to consider that the tabs should always be visible in the application and therefore your application will have several **root** nodes in your navigation tree (one for each tab). You should also avoid navigating the user between tabs too much, as this can be confusing. Instead, each tab should represent a distinct part of the application process, almost like a mini-app.

The tab components provided by Android don't actually perform any of the navigation themselves; instead, it's assumed that you will encapsulate the actual navigation container and logic yourself. It's normal to use the `ViewPager` class to manage the switching between different tab screens, and a single `Fragment` for each of the tabs. Android Studio also includes some simple templates for both of these navigation patterns. Let's take a look at how you can build a simple Activity with tabs at the top:

1. Open the **File** menu, and select **New | New Project**.
2. Name the new project `Navigation`.

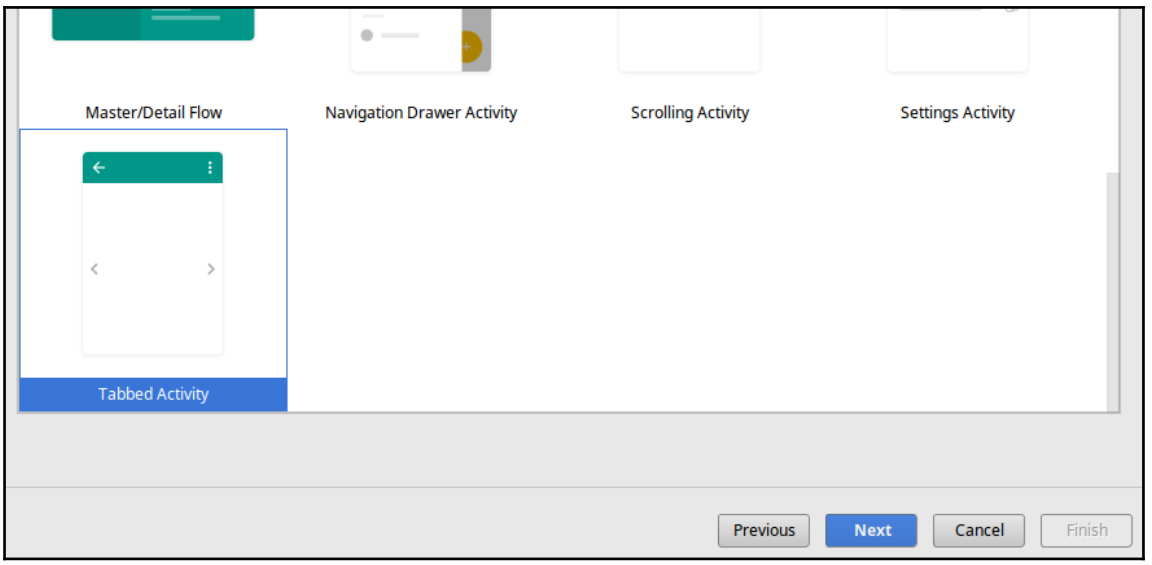
3. Select an appropriate **Company Domain** to determine the package:

4. Click on the **Next** button.
5. Select **Phone and Tablet** support, and at least **API 16** lever support:

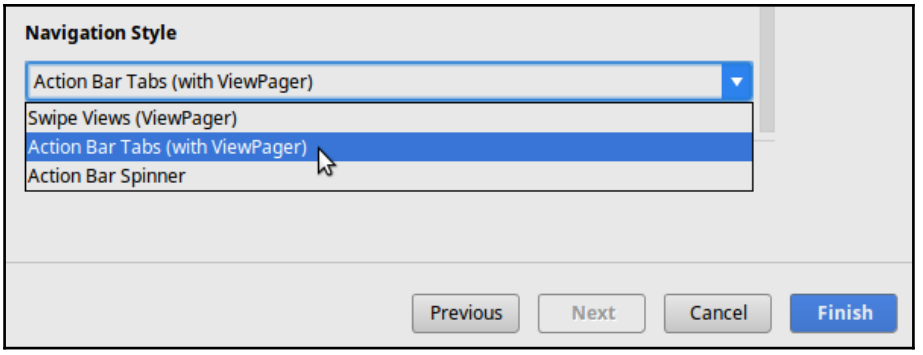


6. Then, click **Next**.

7. In the Activity Gallery, scroll right to the bottom and select **Tabbed Activity**:



8. Click on the **Next** button.
9. Name the new `Activity` `TopTabsActivity`.
10. Scroll down to the bottom of the wizard to **Navigation Style**.
11. Change the **Navigation Style** to **Action Bar Tabs (with ViewPager)**:



12. Click **Finish** to complete the wizard.
13. Wait for Android Studio to finish creating your project.

If your project has compile errors in the IDE, you might need to add the support library to the new project. Open the **build.gradle** for app module, and add `implementation 'com.android.support:support-v4:26.0.0'` (with the correct version number) to the `dependencies`.

14. Once the project has been created, Android Studio will have built a new `Activity` with three tabs in its `AppBarLayout`. Open the **res/layout** directory, and open the `activity_tob_tabs.xml` layout file to edit the number and appearance of the tabs:

```
<android.support.design.widget.TabLayout
    android:id="@+id/tabs"
    android:layout_width="match_parent"
    android:layout_height="wrap_content">

    <android.support.design.widget.TabItem
        android:id="@+id/tabItem"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/tab_text_1" />

    <android.support.design.widget.TabItem
        android:id="@+id/tabItem2"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/tab_text_2" />

    <android.support.design.widget.TabItem
        android:id="@+id/tabItem3"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/tab_text_3" />
</android.support.design.widget.TabLayout>
```

It's best to avoid having too many tabs in any sort of tabbed layout. If you are using text labels (as the template does), you should try and avoid having more than three tabs. If you require more than three, it's best to use the Material icons and remove the text descriptions.

15. To edit what appears in the tabs, you'll need to open the `TopTabsActivity` class.
16. Find the `SectionsPagerAdapter` inner class at the bottom of the file.

17. In this class, you can create a `switch` statement in the `getItem` method to create the `Fragment` instances for each tab. For example, the "Flight Search" images used earlier might have a `getItem` implementation looking like this:

```
public Fragment getItem(final int position) {
  switch (position) {
    case 0:
      return new FlightSearchFragment();
    case 1:
      return new BookingsFragment();
    case 2:
      return new ProfileFragment();
  }

  throw new IndexOutOfBoundsException(
      "no tab for position " + position);
}
```

Using a `switch` statement or similar structure instead of populating an array ensures that the `Fragment` objects are only allocated when they are actually needed. If the user doesn't change tabs, only one will need to be instantiated.

In the `TopTabsActivity`, you'll see in the `onCreate` method that Android Studio tethers the `TabLayout` in the `AppBarLayout` with the `ViewPager`, using two listener classes from the `TabLayout` class:

```
mViewPager.addOnPageChangeListener(
        new TabLayout.TabLayoutOnPageChangeListener(tabLayout));
tabLayout.addOnTabSelectedListener(
        new TabLayout.ViewPagerOnTabSelectedListener(mViewPager));
```

These listeners will keep the selected tab in the `TabLayout` and the current `Fragment` displayed by the `ViewPager` in sync. When a tab is selected, the appropriate page will be displayed and when the `ViewPager` is swiped, the appropriate tab will be selected.

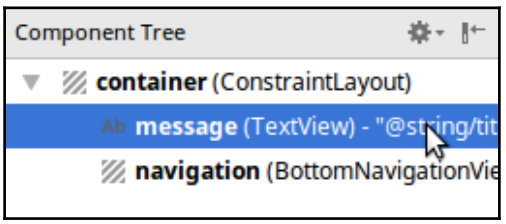
Bottom tabs navigation

Using the bottom navigation tabs is quite different in code structure to placing tabs in the toolbar of the application. Where the toolbar tabs use `TabItem` widgets to render their content, the `BottomNavigationView` uses a menu to decide how it should look. A menu, much like a layout file, is a specialized XML resource file in Android. They also compacted to binary XML during compilation of the project and can be inflated at runtime using a `MenuInflator` object. Unlike a layout resource, a menu specifies lists of menu items and submenus, and while these have text descriptions and optional icons, they have no other render logic of their own. As a result, they are perfect for representing navigation options to a variety of different widgets.

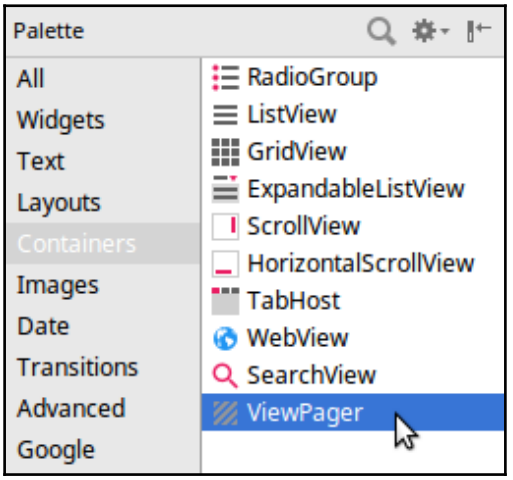
Bottom tabs are often used to present *alternative* views--different user interfaces on top of the same data; for example, search flights, upcoming bookings, and past reservations. All of them are flights for the user, but from different perspectives.

Let's build an `Activity` to use the `BottomNavigationView` to navigate between different areas of an application:

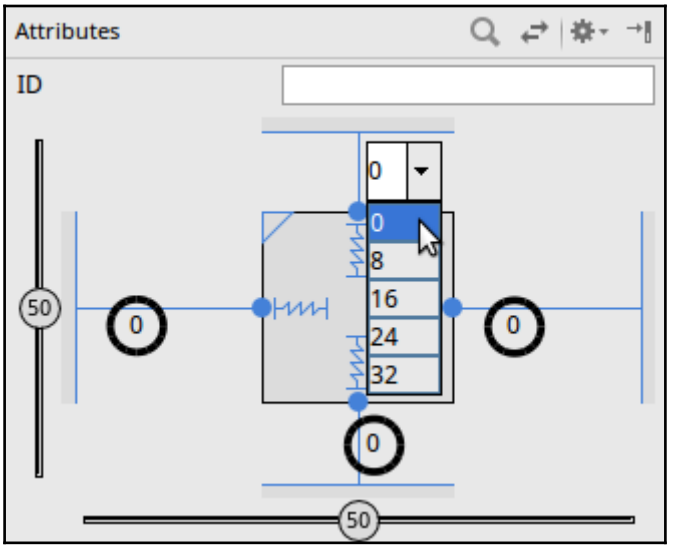
1. Right-click on the main package in the Navigation project and select **New | Activity | Bottom Navigation Activity**.
2. Name the new `Activity` as `BottomTabsActivity`.
3. Click **Finish** to create the new structure.
4. Android Studio will create several new files: the `Activity` class, the new layout XML file, several new icon files, and the navigation menu resource.
5. Open the new `res/layout/activity_bottom_tabs.xml` layout resource.
6. Ensure that the editor is in the **Design** mode.
7. In the **Component Tree** panel, select the **message (TextView)** item and delete it:



8. In the **Palette** panel, open the **Containers** and drag a `ViewPager` into the middle of the design canvas:

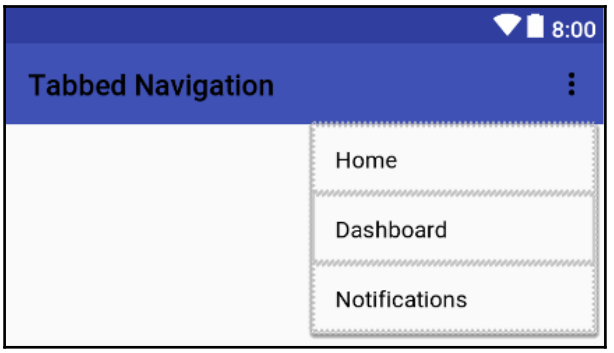


9. Using the **Attributes** panel, add constraints to all sides of the new `ViewPager` and make them `0`:



10. Change the **layout_width** and **layout_height** attributes to **match_constraint**.
11. Change the **ID** of the `ViewPager` widget to `container`.

12. Right-click on the **res/drawable** directory in the project view, and select **New | Vector Asset**.
13. Use the **Icon** selector to find the standard `search` icon, and leave the name as is (`ic_search_black_24dp`).
14. Select **Next**, and then **Finish** to import the icon into your project.
15. Import the `flight takeoff` and `bookmark` icons in the same way.
16. Open the **res/menu/navigation.xml** menu resource file. In the **Design** view, you should be presented with a menu editor, looking like this:



17. Select the **Home** menu item by clicking on it in the **Design** canvas.
18. In the **Attributes** panel, change the item's ID to `navigation_search`.
19. Use the string resource editor to change the **title** attribute to a new string resource called `title_search`, with the content `Search`.
20. Use the icon resource selector to change the icon to the `ic_search_black_24dp` icon you imported.
21. Select the **Dashboard** menu item in the **Design** canvas.
22. Change the **ID** attribute in the **Attributes** panel to `navigation_upcoming`.
23. Use the string resource editor to change the **title** attribute to a new string resource called `title_upcoming`, with the content `Upcoming Flights`.
24. Use the icon resource selector to change the icon to the `ic_flight_takeoff_black_24dp` icon you imported.
25. Select the **Notifications** menu item in the **Design** canvas.
26. Change the **ID** attribute in the **Attributes** panel to `navigation_flown`.
27. Use the string resource editor to change the **title** attribute to a new string resource called `title_flown`, with the content `Past Bookings`.

28. Use the icon resource selector to change the icon to the `ic_bookmark_black_24dp` icon you imported.
29. Now, open the `BottomTabsActivity` source file.
30. Remove the reference to the `TextView`, and replace it with a reference to the `ViewPager` and `BottomNavigationView`:

```
private TextView mTextMessage; // remove this line
private ViewPager container;
private BottomNavigationView navigation;
```

31. The `BottomNavigationView` (unlike the `TabLayout` used for top tabs) includes no listeners to automatically map between the selected tab and a `ViewPager`, so you'll need to map the `MenuItem` ID values to the index of the pages that should be displayed. Create an `int` array with the `MenuItem` ID values in the same order as the pages:

```
private final int[] pageIds = new int[]{
        R.id.navigation_search,
        R.id.navigation_upcoming,
        R.id.navigation_flown
};
```

32. The template created a `BottomNavigationView.OnNavigationItemSelectedListener` anonymous inner class to display the name of the selected tab in the `TextView`. You instead want the `ViewPager` to change to the selected tab `Fragment`, and you can do this using the array of ID values you just declared:

```
private BottomNavigationView.OnNavigationItemSelectedListener
onNavigationItemSelectedListener
        = new
BottomNavigationView.OnNavigationItemSelectedListener() {

    public boolean onNavigationItemSelected(final MenuItem item) {
        for (int i = 0; i < pageIds.length; i++) {
            if (pageIds[i] == item.getItemId()) {
                container.setCurrentItem(i);
                return true;
            }
        }

        return false;
    }
};
```

33. You'll also need a listener for when the user swipes between the tabs on the `ViewPager`, so that the `BottomNavigationView` also highlights the correct tab:

```
private ViewPager.OnPageChangeListener onPageChangeListener =
        new ViewPager.SimpleOnPageChangeListener() {
            public void onPageSelected(final int position) {
                navigation.setSelectedItemId(pageIds[position]);
            }
        };
```

34. In the `onCreate` method, remove the assignment to the `TextView`, and assign the new `ViewPager` field:

```
mTextMessage = findViewById(R.id.message); // remove this line
container = findViewById(R.id.container);
```

35. Change the `BottomNavigationView` assignment and listeners to assign to the field in your `Activity`, and then correctly assign both the listeners:

```
navigation = findViewById(R.id.navigation);
navigation.setOnNavigationItemSelectedListener(
    onNavigationItemSelectedListener);
container.addOnPageChangeListener(onPageChangeListener);
```

36. You can now assign a `ViewPagerAdapter` to the `ViewPager` with three tabs (such as the `SectionsPagerAdapter` generated in the `TopTabsActivity`):

```
container.setAdapter(
    new SectionsPagerAdapter(getSupportFragmentManager()));
```

If the preceding line complains about `TopTabsActivity` not being an enclosing class, then change the `SectionsPagerAdapter` to be a static inner class--`public static class SectionsPagerAdapter extends FragmentPagerAdapter`.

The listeners in this example can be reused in any number of applications that require bottom-tabbed navigation. The only thing you'll need to change is the list of `pageIds` that are displayed to the user. You should avoid having more than three or four tabs in the `BottomNavigationView`; it normally implies that another form of navigation is more appropriate to your application.

Navigation menus

Sometimes, you need to provide your user with a broad set of navigation options that won't fit into a set of tabs. This is where a hidden navigation menu, sometimes called a hamburger menu, becomes useful. This menu pattern was once popular to put as a sort of main menu, available on every screen in your application. However, navigation menus hide options, and they often encourage sloppy navigation design, because they provide a space where any navigation items can be *dumped*. It's better to try and avoid any form of hidden navigation until you're absolutely sure that you need it.

They can be useful when they augment other navigation patterns (such as tabs), and are used to offer seldom used or advanced functionality that the user is unlikely to access every day. For example, on a photo gallery screen, a hidden menu might be used to access the ability to create new labels, access photos that have been deleted, and to access settings and help.

Let's add a navigation menu to the example with the bottom tabs, to allow the user access to some other functionality that they might need:

1. Right-click on the `res/menu` directory and select **New | Menu resource file**.
2. Name the new file `nav_menu`, and click **OK** to create the new resource file.
3. Open the **Text** editor for the new file.
4. Copy the following menu structure into the new file:

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

  <item
    android:id="@+id/loyalty_programs"
    android:title="Frequent Flyer" />
  <item
    android:id="@+id/deals"
    android:title="Special Deals" />
  <item
    android:id="@+id/guides"
    android:title="Travel Guide" />
  <item
    android:id="@+id/settings"
    android:title="Settings">
    <!-- nesting a menu produces a "group" in the navigation menu -
->
    <menu>
      <item android:id="@+id/profile"
```

```
            android:title="Profile"/>
        <item android:id="@+id/about"
            android:title="About"/>
    </menu>
  </item>
</menu>
```

5. Now, open the `activity_bottom_tabs.xml` layout file.
6. Change to the **Text** editor.
7. The root element should currently be a `ConstraintLayout`; you'll need to wrap it inside a `DrawerLayout` widget that will manage the showing and hiding of the navigation drawer. You'll also need to give the `ConstraintLayout` a top-margin the same size as the `ActionBar`; otherwise, it will be hidden behind the system `ActionBar` (another way around this is to use the `AppBarLayout` and `CoordinatorLayout` with no system `ActionBar`):

```
<android.support.v4.widget.DrawerLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/drawer_layout"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:fitsSystemWindows="true"
    tools:openDrawer="start"
    tools:context="com.packtpub.navigation.BottomTabActivity">
    <!-- This ConstraintLayout is your old root layout widget -->
    <android.support.constraint.ConstraintLayout
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:layout_marginTop="?attr/actionBarSize">
```

8. After the `ConstraintLayout` element is closed, you'll need to add the `NavigationView`, which will contain the navigation menu you just wrote:

```
</android.support.constraint.ConstraintLayout>

    <android.support.design.widget.NavigationView
        android:id="@+id/nav_view"
        android:layout_width="wrap_content"
        android:layout_height="match_parent"
        android:layout_gravity="start"
        android:fitsSystemWindows="true"
        app:menu="@menu/nav_menu" />
</android.support.v4.widget.DrawerLayout>
```

9. Open the `BottomTabActivity` source file.
10. By default, the `NavigationView` won't respond to any form of tapping menu items, and won't even close the navigation drawer when you select a menu item. You'll need to add a listener and tell it what to do yourself. At the bottom of the `onCreate` method, look up the `NavigationView` and add a listener to at least close the navigation drawer:

```
final NavigationView navigationView = findViewById(R.id.nav_view);
navigationView.setNavigationItemSelectedListener(new
NavigationView.OnNavigationItemSelectedListener() {
  public boolean onNavigationItemSelected(final MenuItem item) {
    // your normal click handling would go here
    final DrawerLayout drawer = findViewById(R.id.drawer_layout);
    drawer.closeDrawer(GravityCompat.START);
    return true;
  }
});
```

11. Users also expect to be able to close the navigation drawer using the back button. This requires you to override the default back-button behavior:

```
public void onBackPressed() {
  final DrawerLayout drawer = findViewById(R.id.drawer_layout);
  if (drawer.isDrawerOpen(GravityCompat.START)) {
    drawer.closeDrawer(GravityCompat.START);
  } else {
    super.onBackPressed();
  }
}
```

Overriding the back button behavior like this is something you should be very careful with. The default behavior is strongly consistent across the entire platform and all well-behaved applications. Applications that have inconsistent back-button behavior are obvious to users, and are often very frustrating.

The navigation drawer here is a very good example of its use within an application context. The bottom tabs allow the user quick access to the most commonly used areas of the application, while the navigation drawer can be used to access the less frequently used features. Remember that navigation drawers hide features of your application and should only be used for features that are not required for the user, to make effective use of your application. It's sometimes worth forcibly opening the navigation drawer the first time the screen is opened by the user (you can use a `SharedPreferences` to remember that they have seen it). You can do this using the `DrawerLayout.openDrawer` method in `Activity.onCreate`.

Also, remember that while overriding the default back-button behavior is important for the user experience in this specific case, it's normally not a good idea. Inconsistent back-button behavior is something that users pick up on very easily, and it's one of the most common irritations. For some behaviors such as closing the navigation drawer, it's important because it's the most common pattern, but using the ability to ask whether the user is "Sure they want to exit" (and similar additional behaviors) is a waste of the user's time, and should be avoided.

Navigating using Fragments

So far in the book, you've mostly been navigating users from one `Activity` to another `Activity`, and this is in fact how most applications are built. However, there is another option, which is often much more flexible and allows you to build even more modular applications--navigation using `Fragment` instances. So far, we've only really looked at Fragments as little blocks of your application that can be assembled to form parts of a screen, but they can be so much more than that.

The tabbed `Activity` classes both provide a sort of navigation using the `ViewPager` class and the `FragmentPagerAdapter` class. In these cases, each of the pages that the user can swipe to is a complete `Fragment`, with its life cycle that is paused and resumed, stopped, and started as the user swipes the `Fragment` in or out of view.

If you look into the `FragmentPagerAdapter` class, you'll find that it doesn't add and remove the `Fragment` view instances directly to the `ViewPager` object. Instead, it uses a `FragmentTransaction` to add and remove the `Fragment` to the `ViewPager` using the `ViewPager` ID attribute:

```
mCurTransaction = mFragmentManager.beginTransaction();
// ...
fragment = getItem(position);
mCurTransaction.add(container.getId(), fragment,
                    makeFragmentName(container.getId(), itemId));
```

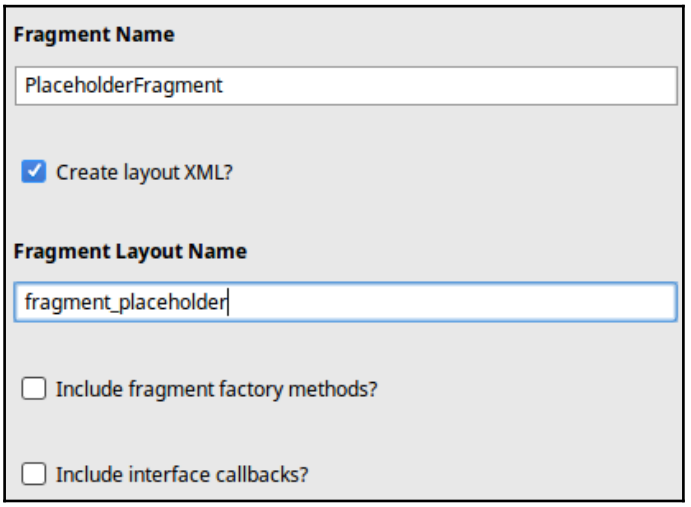
The `FragmentTransaction` class allows you to define any number of actions that will all appear to happen at once. You can add, remove, attach, detach, and replace any number of `Fragment` instances on your user interface, and then trigger them all at once. The best part is that you can also add the transaction to the "back stack". This means that the user can undo the transaction by pressing the *back* button on their device.

So, by creating your main `Activity` with a content space (like the `ViewPager` in the tabs examples), and populating it with `Fragment` objects, you can simulate `Activity` to `Activity` navigation. This also means that your primary navigation controls, such as tabs or a hidden navigation menu, only need to be defined in your activity layout rather than on the layouts of each screen in the application. This also makes the navigation within your application slightly quicker, because the heavy-weight components of the screen are reused for each navigation.

Let's add some navigation behaviors to the bottom tabs example that we've started, so that the navigation menu options actually do something:

1. First, you'll need a `Fragment` class that you can use for the various navigation actions you'll have in the example. Right-click on your default package (that is, `com.packtpub.navigation`) and select **New | Fragment | Fragment (blank)**.
2. Name the new `Fragment` class `PlaceholderFragment`.

3. Deselect the **Include fragment factory methods?** and **Include interface callbacks?** checkboxes:



4. Click on **Finish** to create the new fragment class and layout file.
5. Open the `fragment_placeholder.xml` layout file in **Design** mode.
6. Select the `FrameLayout` in the **Component Tree** panel.
7. In the **Attributes** panel, toggle to **View all attributes.**
8. Find the `background` attribute, and set it to `#ffffff` (white) so that the background of this `Fragment` is opaque.
9. Select the `TextView` in the **Component Tree** panel.
10. In the **Attributes** panel, change the ID attribute to `placeholder_text`.
11. Change the `textAppearance` attribute to `@style/TextAppearance.AppCompat.Display1`, which will appear in the drop-down as **AppCompat.Display1**.
12. Now, open the new `PlaceholderFragment` Java source file.
13. Declare a `static String` constant to allow the `PlaceholderFragment` to hold its placeholder-text argument:

```
private static final String ARG_TEXT = "text";
```

14. Change the `onCreateView` method so that it sets the text of the `TextView` to the placeholder-text:

```
public View onCreateView(
    final LayoutInflater inflater,
    final ViewGroup container,
    final Bundle savedInstanceState) {

  final View rootView = inflater.inflate(
      R.layout.fragment_placeholder,
      container,
      false
  );

  final TextView textView =
      rootView.findViewById(R.id.placeholder_text);
  textView.setText(getArguments().getString(ARG_TEXT));</strong>
  return rootView;
}
```

15. Create a convenience factory method to create the `PlaceholderFragment` with the placeholder-text specified as a method argument:

```
public static PlaceholderFragment newInstance(final String text) {
  final PlaceholderFragment fragment = new PlaceholderFragment();
  final Bundle args = new Bundle();
  args.putString(ARG_TEXT, text);
  fragment.setArguments(args);
  return fragment;
}
```

16. Open the `activity_bottom_tabs.xml` layout resource in the **Text** editor.
17. Find the `ViewPager` below the `BottomNavigationView` widget.
18. Change the `ViewPager` so that it is wrapped in a full-size `FrameLayout` with an ID of `host`; this will be used to contain the `Fragment` instances used to navigate the user around the app:

```
<FrameLayout
    android:id="@+id/host"
    android:layout_width="match_parent"
    android:layout_height="0dp"
    app:layout_constraintBottom_toTopOf="@+id/navigation"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent"
    tools:layout_editor_absoluteX="8dp"
```

```
    tools:layout_editor_absoluteY="8dp">

    <android.support.v4.view.ViewPager
        android:id="@+id/container"
        android:layout_width="match_parent"
        android:layout_height="match_parent" />
</FrameLayout>
```

19. Open the `BottomTabsActivity` source file.
20. When the user taps on one of the bottom navigation items, you want to ensure that any navigation they have done is cleared so that the back button doesn't navigate them back into their previous stack, and ensure that there are no residual `Fragment` instances on the screen. In the `OnNavigationItemSelectedListener.onNavigationItemSelected` method in your anonymous class, you'll want to pop the backstack, before telling the `ViewPager` to change tab:

```
private BottomNavigationView.OnNavigationItemSelectedListener
    onNavigationItemSelectedListener
    = new BottomNavigationView.OnNavigationItemSelectedListener() {

  @Override
  public boolean onNavigationItemSelected(final MenuItem item) {
    final FragmentManager fragmentManager =
        getSupportFragmentManager();
    if (fragmentManager.getBackStackEntryCount() > 0) {
      fragmentManager.popBackStack(
            fragmentManager.getBackStackEntryAt(0).getId(),
            FragmentManager.POP_BACK_STACK_INCLUSIVE);
   }
      for (int i = 0; i < pageIds.length; i++) {
      // ...
```

21. At the bottom of the `onCreate` method, you need to add a new listener to the `NavigationView` to listen for the taps in the menu. These will trigger the navigation using the `FragmentManager`, and will also close the navigation drawer:

```
final NavigationView navigationView = findViewById(R.id.nav_view);
navigationView.setNavigationItemSelectedListener(new
NavigationView.OnNavigationItemSelectedListener() {
  @Override
  public boolean onNavigationItemSelected(final MenuItem item) {
    final String location = item.getTitle().toString();

    getSupportFragmentManager()
```

```
                .beginTransaction()
                .replace(
                    R.id.host,
                    PlaceholderFragment.newInstance(location)
                )
                .setTransition(FragmentTransaction.TRANSIT_FRAGMENT_OPEN)
                .addToBackStack(location)
                .commit();

            final DrawerLayout drawer = findViewById(R.id.drawer_layout);
            drawer.closeDrawer(GravityCompat.START);
            return true;
        }
    });
```

As a bonus, the preceding code will also produce a lovely transition navigation between each of the navigation actions. It's also possible that you might want to clear the backstack each time the user takes one of these navigation actions. On top of that, you may also want to select specific tabs in the `BottomNavigationView` to indicate which part of the app the user is in, or you may want the `FrameLayout` to wrap the entire `ConstaintLayout` so that the bottom tabs vanish when the user is navigated using the `FragmentManager`.

It's important to note that in this structure, the other layouts and `Fragment` instances are still in the layout. They're just hidden by the `Fragment` instances being placed over them as the user navigates using the menu. To avoid this, you can wrap the `ViewPager` in a dedicated `Fragment` class, but it's important to add it to the layout through the `FragmentManager` in the `Activity.onCreate` method and not by using the `<fragment>` tag in the layout XML. The `FragmentManager` will only remove a `Fragment` from the layout if it was added via a `FragmentTransaction` in the first place.

Test your knowledge

1. When using bottom tabs for navigation, which of these is important?
    - They all have single color icons
    - The tabs are of roughly equal importance
    - There are always exactly three tabs

2. Top tabs are preferred to bottom tabs in which of these situations?
    - When the user won't need to navigate as frequently
    - When the tabs don't have icons
    - When there are more than three tabs
3. Fragments can be used for navigation in which of these cases?
    - Only when a navigation drawer is used as well
    - Any time the user navigates within the application
    - When they can be nested in a `FrameLayout`
4. When the user selects an item in a navigation drawer, which of these is true?
    - The drawer needs to be closed by the user
    - The drawer should be closed programmatically
    - The drawer is closed automatically after a short delay

Summary

Navigation is a critical part of a user's experience, and should be carefully thought out and designed. Material design has various different design structures and widgets to help you implement more effective navigation, but it's important to use them carefully and in the right place. As with any screen design, it's important to consider what the user will want to do most often, and to rank each possible action and navigation from the most important to the least on every screen they are available in.

In many applications, dedicated navigation components won't even be needed, and navigation will be achieved purely through goal-oriented actions from an overview screen or dashboard. In all instances, it's a good idea to draw up a navigation map ahead of time (even if it's incomplete or overly simplified). They will often tell you what sort of navigation structure and components your application will require.

Navigation achieved using the `FragmentManager` instead of always launching a new `Activity` is an extremely powerful pattern. It offers a large number of additional options and significantly more control over the backstack, and even the animations played during each transition. It's also possible to change more than one onscreen `Fragment` in a single `FragmentTransaction`, which can be used to produce some amazing effects.

In the next chapter, we'll go back to the travel claim example and explore some more of the `RecyclerView`. The chapter will take a look at some of the more advanced capabilities of the `RecyclerView` and how to integrate it with the `LiveData` class and Room using some powerful classes from the support API to achieve some exciting effects.

10
Making Overviews Even Better

When you built the overview/dashboard screen in `Chapter 7`, *Creating Overview Screens*, you used a `RecyclerView` and used Room and data binding to retrieve the list of records from the database and display them to the user, and it worked fantastically well. However, it can be done even better. `RecyclerView` is an incredibly powerful engine for data display, and we've only really scratched the surface of what it's capable of. In this chapter, we'll take a deeper look at some of the ecosystem surrounding the `RecyclerView` and integrate some big improvements into the claim example. Specifically, we'll explore the following:

- Different ways to lay out a `RecyclerView` with more than one view type
- Ways to improve the `RecyclerView` performance
- Animating changes to `RecyclerView`
- Keeping the complexity off the main thread

Multiple view types

`RecyclerView` is capable of handling almost any number of different types of widgets for display on the screen, and recycling them all independently. This is an amazingly powerful and useful technique, not just for being able to display different types of data on the screen, but also to adjust the layout of the `RecyclerView` in a way that is mostly transparent. However, you'll need to look at how exactly you want to break the layout up.

There are generally two main reasons you will want to use different view types in a `RecyclerView`:

- To break up a long list of items with a divider
- As you have different types of data you want to render together

Let's start with creating and adding dividers; you can just adjust the margin of each of the widgets when the data is bound to them, but that doesn't help the user understand why the divider is there. Often, you'll want a divider to carry details of what it actually represents, such as a date label. In these cases, you need widgets to render the label.

You can create a special variant of your normal item layout that includes the divider, by embedding it in a `LinearLayout`. For example, if you wanted to add a divider label to the claim items displayed in the overview of the travel claim application, you could add a special layout named `card_claim_item_with_divider`, and have it look something like this:

```
<?xml version="1.0" encoding="utf-8"?>
<layout xmlns:android="http://schemas.android.com/apk/res/android">
  <data>
    <variable
        name="presenter"
        type="com.packtpub.claim.ui.presenters.ItemPresenter" />

    <variable
        name="item"
        type="com.packtpub.claim.model.ClaimItem" />
  </data>

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

    <TextView
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_marginTop="@dimen/grid_spacer1"
        android:layout_marginBottom="@dimen/grid_spacer1"
        android:text="@{presenter.dividerLabel(item)}"
        android:textAppearance="@style/TextAppearance.AppCompat.Caption" />

    <include
        item="@{item}"
        layout="@layout/card_claim_item"
        presenter="@{presenter}" />
```

```
    </LinearLayout>
</layout>
```

This approach is very simple to implement, because the divider is made part of the item that appears below it. This, in turn, means that your `Adapter` implementation only needs to decide whether or not an item needs a divider or not, as opposed to keeping track of the dividers as their own object type.

However, the approach also has several large disadvantages; each divider now carries an entire card with it, and if there are no dividers on the screen, the `RecyclerView` will still maintain a pool of them off-screen. This means that the entire unused card is unusable, and takes up much more memory and data than it should. The other problem with this approach is that you have nested the widgets another layer deep, within a `LinearLayout`. The `LinearLayout` renders the contained widgets in exactly the same way as the `LinearLayoutManager` attached to the `RecyclerView`. So, this layout introduces a widget into the layout system that doesn't really add any value and will negatively impact the application's performance.

So, what is the alternative? It's quite simple, really; treat dividers as special items in your `RecyclerView`. When you break up a `RecyclerView`, each view type is given an integer identifier that allows the `RecyclerView` to keep track of them independently in different recycling pools, and ensure that each one is only used in the right place. The easiest way to introduce dividers as special items, is to introduce them as special items in your dataset. This can be as simple as adding null values into the `List` of `ClaimItem` objects where dividers are required, but that won't play very nicely with data binding layers, and doesn't scale well.

A better way is to use a `wrapper` object in the dataset that tells the `Adapter` implementation how to render each of the items. This list can be calculated upfront and reduces the complexity of the layout and rendering. This also allows for very complex choices to be made for every item in the dataset, without affecting the user's perceived performance of the application. Let's build a `DisplayItem` class that can be used along with the `DataBoundViewHolder` class, to allow for any number of different item types to be used together in a single `Adapter`:

1. Right-click on the `com.packtpub.claim.ui` package in the travel claim example project and select **New | Java Class**.
2. Name the new class `DisplayItem` and click **OK**.

3. Declare an integer field to represent the layout resource for each `DisplayItem` object. These will be used by the `Adapter` class to figure out which layout to load and render:

```
public class DisplayItem {
  public final int layout;
```

4. This class is expected to be used as part of a mixed list, making generics inappropriate at this level. Declare a plain `Object` field to hold the data (if there is any) for the `DisplayItem` to bind to its layout:

```
public final Object value;
```

5. Now, you'll need a constructor to assign these two fields:

```
public DisplayItem(
    final int layout,
    final Object value) {
  this.layout = layout;
  this.value = value;
}
```

6. As a convenience to the `Adapter` classes, the `DisplayItem` will offer a `bindItem` method to help with the `DataBoundViewHolder` class:

```
public <I> void bindItem(final DataBoundViewHolder<?, I> holder) {
  @SuppressWarnings("unchecked") final I item = (I) value;
  holder.setItem(item);
}
```

This is a very simple class to implement, but makes a very big difference in how the `Adapter` implementation works. As the dataset in the `Adapter` is no longer the raw dataset read from the database or network, you're free to mix various data sources without having to do any of the work in the `onBindViewHolder` method. The `DisplayItem` is a bit like a `ViewHolder` that doesn't actually hold the user-interface widgets; instead, it just signals which layout needs to be used for the data it carries.

Introducing dividers

In order to introduce dividers into the claim overview screen, you'll need to run a second pass over the data being delivered from the Room database layer, and figure which items require a divider. This should be done on a background worker thread, so that larger datasets won't impact the user experience. Let's get to work and add some simple dividers to the travel claim app to appear between claim items made on different days; this will require some major changes to how the `ClaimItemAdapter` class works. The most obvious change is that it will now have a `List` of `DisplayItem` objects instead of directly containing a `List` of `ClaimItem` objects.

Follow these steps to restructure the `ClaimItemAdapter` to use `DisplayItem` objects to mix both claim items and dividers in the `RecyclerView`:

1. First, you'll need a nice line that you can use as a divider. This will be a drawable that can be rendered using an `ImageView` widget. Right-click on the `res/drawable` directory and select **New| Drawable resource file**.
2. Name the new file `horizontal_divider`, and click **OK** to create the new resource.
3. Change to the **Text** editor.
4. By default, Android Studio will have created a `selector` drawable, but you want to declare a `shape` drawable. Replace the generated template code with the following XML drawable:

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

    <stroke
        android:width="1dp"
        android:color="#e0e0e0" />
</shape>
```

5. You'll also need a layout to use for the dividers in the `RecyclerView`. Right-click on the `res/layout` directory and select **New| Layout resource file**.
6. Name the new layout file `widget_divider`.
7. Change the **Root element** to `layout`.
8. Click **OK** to create the new layout resource file.

9. The new layout doesn't actually need any variables to be bound, so you can leave the `data` section empty. Use an `ImageView` to render the new `horizontal_divider` at full width:

```
<layout xmlns:android="http://schemas.android.com/apk/res/android">
    <data></data>

    <ImageView
        android:layout_width="match_parent"
        android:layout_height="@dimen/grid_spacer1"
        android:layout_marginTop="@dimen/grid_spacer1"
        android:src="@drawable/horizontal_divider" />
</layout>
```

10. Now, open the `ClaimItemAdapter` source file.
11. Change the `List` of `ClaimItem` objects to a `List` of `DisplayItem` objects:

```
private List<DisplayItem> items = Collections.emptyList();
```

12. Declare a new override method--`getItemViewType`--and use the `DisplayItem.layout` value to identify the differences between the layouts that will be used in the `RecyclerView`. This method will delegate to the `DisplayItem` objects and use the layout resource ID as an identifier:

```
@Override
public int getItemViewType(final int position) {
    return items.get(position).layout;
}
```

It's a common trick to directly use the layout resource ID to determine the different view types in a `RecyclerView`. This avoids any mapping between internal ID numbers and the layout resources.

13. Now, change the `onCreateViewHolder` method to use the `viewType` to decide which layout resource to load. The `viewType` will be passed in by the `RecyclerView`, and will be the same value that was returned by `getItemViewType`:

```
@Override
public DataBoundViewHolder<ItemPresenter, ClaimItem>
    onCreateViewHolder(
      final ViewGroup parent,
      final int viewType) {
```

```
    return new DataBoundViewHolder<>(
        DataBindingUtil.inflate(
            layoutInflater,
            viewType,
            parent,
            false
        ),
        itemPresenter
    );
  }
```

14. Change the `onBindViewHolder` method to use the `DisplayItem.bindItem` method, rather than directly invoking `DataBoundViewHolder.setItem`:

```
@Override
public void onBindViewHolder(
    final DataBoundViewHolder<ItemPresenter, ClaimItem> holder,
    final int position) {

  items.get(position).bindItem(holder);
}
```

15. At the bottom of the `ClaimItemAdapter` class, you'll need a new `ActionCommand` inner class to do the work of calculating where the dividers should be placed, and wrapping all the `ClaimItem` objects in `DisplayItem` objects:

```
private class CreateDisplayListCommand
        extends ActionCommand<List<ClaimItem>, List<DisplayItem>> {
```

16. The `CreateDisplayListCommand` will need a utility method to decide whether or not to insert a divider between two items. This utility method will simply check whether both items have timestamps on the same day:

```
boolean isDividerRequired(
    final ClaimItem item1, final ClaimItem item2) {
  final Calendar c1 = Calendar.getInstance();
  final Calendar c2 = Calendar.getInstance();

  c1.setTime(item1.getTimestamp());
  c2.setTime(item2.getTimestamp());

  return c1.get(Calendar.DAY_OF_YEAR)
           != c2.get(Calendar.DAY_OF_YEAR)
       || c1.get(Calendar.YEAR)
           != c2.get(Calendar.YEAR);
}
```

17. Then, you'll need to implement the `onBackground` method of `ActionCommand`, and process the list of `ClaimItem` objects into a `List` of `DisplayItem` objects:

```
@Override
public List<DisplayItem> onBackground(
    final List<ClaimItem> claimItems)
    throws Exception {

  final List<DisplayItem> output = new ArrayList<>();

  for (int i = 0; i < claimItems.size(); i++) {
    final ClaimItem item = claimItems.get(i);
    output.add(new DisplayItem(R.layout.card_claim_item, item));

    if (i + 1 < claimItems.size() // not the last item
        && isDividerRequired(item, claimItems.get(i + 1))) {

      output.add(new DisplayItem(R.layout.widget_divider, null));
    }
  }

  return output;
}
```

18. To complete the `CreateDisplayListCommand` implementation, you'll need to implement the `onForeground` method. This will assign the new `List` of `DisplayItem` objects up to the `ClaimItemAdapter` and notify the `RecyclerView` of the changes:

```
@Override
public void onForeground(final List<DisplayItem> value) {
  ClaimItemAdapter.this.items = value;
  notifyDataSetChanged();
}
```

19. You'll need an instance of `CreateDisplayListCommand` for the `ClaimItemAdapter` to use each time the `LiveData` is updated. Create a new field at the top of the `ClaimItemAdapter` class, and instantiate it:

```
private final CreateDisplayListCommand createDisplayListCommand
        = new CreateDisplayListCommand();
private final LayoutInflater layoutInflater;
private final ItemPresenter itemPresenter;
private List<DisplayItem> items = Collections.emptyList();
```

20. Now, you can change the constructor to use the `CreateDisplayListCommand` instead of directly referencing the `List` of `ClaimItem` objects returned by the Room database:

```
public ClaimItemAdapter(
        final Context context,
        final LifecycleOwner owner,
        final LiveData<List<ClaimItem>> liveItems) {

  this.layoutInflater = LayoutInflater.from(context);
  this.itemPresenter = new ItemPresenter(context);

  liveItems.observe(owner, new Observer<List<ClaimItem>>() {
    @Override
    public void onChanged(final List<ClaimItem> claimItems) {
      createDisplayListCommand.exec(claimItems);
    }
  });
}
```

If you run the travel claim application now, you'll have a lovely thin and light divider between any claim items that were captured for different dates. Try creating a few using the datepicker to shift the days and force the user interface to produce different groupings of cards. You'll also find that because all the data still comes from the database, you can add and remove the data, and the user interface will remain up to date.

Updating by Delta Events

Up until this point, when the data changes in the database, the `ClaimItemAdapter` simply tells the `RecyclerView` that the data has changed. This is not the most efficient use of resources, because the `RecyclerView` doesn't actually know what in the model has changed, and it's forced to relayout the entire scene as though the entire model has changed (although it will reuse the widgets it has already pooled).

`RecyclerView` actually has a secondary mechanism that allows you to tell it what has changed, rather than just saying that the data has changed. This is provided through a series of notifications the signal single items, or ranges being added, removed, and moved. The problem is that in order to use these methods, you need to know what has actually changed.

Most developer's first instincts here will be to use more events and signal from the DAO or a delegate layer what is changing, and then catch those events in the `Adapter` and forward them to the `RecyclerView`. This can be made to work, and in fact, it can work quite well when done through an event bus rather than having the `Adapter` directly attach itself to the DAO layer. The problem is that it also forces you to generate these events, translate them, and they can become unwieldy as the possible list of simultaneous changes becomes more complex.

Another way is to leave the events up to Room. When new data is provided via `LiveData`, you can compare the dataset currently displayed to the user with the new dataset and calculate what has changed. This is the same way that version control software such as Git or Mercurial work; they compare what you have done with what you started with to create a delta, or diff of the changes. This can be complicated and hard work, but Android support library has you covered; it provides a class named `DiffUtil` that can not only be used to compute the differences between virtually any two datasets, but also produce the correct set of events to deliver to the `RecyclerView`. Let's use `DiffUtil` in the `ClaimItemAdapter`, to only apply the changes to the `RecyclerView`:

1. Before searching for differences, it's important to be able to determine whether two `ClaimItem` objects refer to the same database record, but are different. For that, you'll need a complete `equals` method, which can be generated by Android Studio. Open the `ClaimItem` source file in Android Studio.
2. Click on the class name in the editor, and then select **Code|Generate** from the main menu bar.

3. Select **equals() and hashCode()** from the pop-up menu.
4. Use all the defaults that are provided by the IDE, clicking on **Next** and **Finish** until the wizard is complete.
5. Open the `ClaimItemAdapter` source file.
6. At the bottom of the `ClaimItemAdapter` class below the `CreateDisplayListCommand` inner class, declare a new `ActionCommand` inner class to deal with updating the existing list of `DisplayItem` objects, and triggering the required change notifications:

```
private class UpdateDisplayListCommand
        extends ActionCommand<
            Pair<List<DisplayItem>, List<ClaimItem>>,
            Pair<List<DisplayItem>, DiffUtil.DiffResult>
        > {
```

This class takes and processes `Pair` objects containing two parameters each. As input, we'll be passing the old `List` of `DisplayItem` objects, and the new `List` of `ClaimItem` objects that it needs to process. As output, it will produce the new `List` of `DisplayItem` objects, and a `DiffResult` that can be used to trigger the update events.

7. The first thing you'll need in the `UpdateDisplayListCommand` is the `onBackground` method. This method will use the `List` of `DisplayItem` objects passed in through the `Pair` as the "old" `List` of items, and will generate a "new" `List` of `DisplayItem` objects by invoking the `CreateDisplayListCommand` directly:

```
@Override
public Pair<List<DisplayItem>, DiffUtil.DiffResult> onBackground(
        final Pair<List<DisplayItem>, List<ClaimItem>> args)
        throws Exception {

    final List<DisplayItem> oldDisplay = args.first;
    final List<DisplayItem> newDisplay =
            createDisplayListCommand.onBackground(args.second);
```

8. Now that you have the `List` that is currently being displayed to the user and the `List` that needs to be displayed, it's time to calculate the differences between them. In order to keep things completely generic, the `DiffUtil` defines a callback interface that is queried to find the details of the two lists. In the `UpdateDisplayListCommand` class, we'll simply use an anonymous inner class:

```
final DiffUtil.DiffResult result =
      DiffUtil.calculateDiff(new DiffUtil.Callback() {
  @Override
  public int getOldListSize() {
    return oldDisplay.size();
  }

  @Override
  public int getNewListSize() {
    return newDisplay.size();
  }
```

9. The `Callback` implementation also requires a method to compare items at two different positions, to see whether they appear to be the same item. The first thing to do is to check whether their layouts appear to be the same. If the layouts aren't the same, we can be sure that they are not the same object. If the layouts are the same, then we can look at the layout integer as an indicator of what type of data is in the `DisplayItem` object. If it's a `ClaimItem`, we use the database ID of the objects to see whether they represent the same record in the database:

```
@Override
public boolean areItemsTheSame(
    final int oldItemPosition,
    final int newItemPosition) {
  final DisplayItem oldItem = oldDisplay.get(oldItemPosition);
  final DisplayItem newItem = newDisplay.get(newItemPosition);

  if (oldItem.layout != newItem.layout) {
    return false;
  }

  switch (newItem.layout) {
    case R.layout.card_claim_item:
      final ClaimItem oldClaimItem = (ClaimItem) oldItem.value;
      final ClaimItem newClaimItem = (ClaimItem) newItem.value;
      return oldClaimItem != null
          && newClaimItem != null
          && oldClaimItem.id == newClaimItem.id;
    case R.layout.widget_divider:
      return true;
```

```
    }

    return false;
  }
```

10. The `Callback` also needs another method to test whether the contents of the two objects has actually changed. This method will only be invoked by the `DiffUtil` if the `areItemsTheSame` method returned true, which permits you to take some shortcuts through the implementation by assuming that both sides represent the same record:

```
@Override
public boolean areContentsTheSame(
    final int oldItemPosition,
    final int newItemPosition) {
  final DisplayItem oldItem = oldDisplay.get(oldItemPosition);
  final DisplayItem newItem = newDisplay.get(newItemPosition);

  switch (newItem.layout){
    case R.layout.card_claim_item:
      final ClaimItem oldClaimItem = (ClaimItem) oldItem.value;
      final ClaimItem newClaimItem = (ClaimItem) newItem.value;
      return oldClaimItem != null
          && newClaimItem != null
          && oldClaimItem.equals(newClaimItem);
    case R.layout.widget_divider:
      return true;
  }

  return false;
}
```

11. That concludes the `Callback` implementation. Now, you'll need to close the `onBackground` method by returning a `Pair` containing the new list of `DisplayItem` objects and the `DiffResult`:

```
    }); // end of the DiffUtil.Callback implementation

    return Pair.create(newDisplay, result);
  } // end of the onBackground implementation
```

12. In the `onForeground` method of the `UpdateDisplayListCommand`, you'll need to assign the new list of `DisplayItem` objects up to the `ClaimItemAdapter`, just as before. However, instead of notifying the `RecyclerView` that the whole model has changed, you can use the `DiffResult` to relay the differences you found as a series of events:

```
@Override
public void onForeground(
    final Pair<List<DisplayItem>,
    DiffUtil.DiffResult> value) {
  ClaimItemAdapter.this.items = value.first;
  value.second.dispatchUpdatesTo(ClaimItemAdapter.this);
}
```

13. Back at the top of the `ClaimItemAdapter`, you'll now need a field with an instance of the new `UpdateDisplayListCommand` class:

```
public class ClaimItemAdapter extends
        RecyclerView.Adapter<DataBoundViewHolder<ItemPresenter,
ClaimItem>> {

  private final UpdateDisplayListCommand updateCommand
        = new UpdateDisplayListCommand();
  private final CreateDisplayListCommand createDisplayListCommand
        = new CreateDisplayListCommand();
  private final LayoutInflater layoutInflater;
  private final ItemPresenter itemPresenter;
```

14. Now, in the `ClaimItemAdapter` class's constructor, the `LiveData` observer changes again. If the data is from the first notification, there is no point in calculating the differences between the two lists, but if it's any invocation after that, you can now run it through the `UpdateDisplayListCommand`:

```
public ClaimItemAdapter(
        final Context context,
        final LifecycleOwner owner,
        final LiveData<List<ClaimItem>> liveItems) {

  this.layoutInflater = LayoutInflater.from(context);
  this.itemPresenter = new ItemPresenter(context);

  liveItems.observe(owner, new Observer<List<ClaimItem>>() {
    @Override
    public void onChanged(final List<ClaimItem> claimItems) {
      if (!items.isEmpty()) {
        updateCommand.exec(Pair.create(items, claimItems));
```

```
            } else {
              createDisplayListCommand.exec(claimItems);
            }
          }
        });
    }
```

These changes might seem like a lot of work just to change the events produced by the `Adapter` implementation, but they can be made relatively generic with some work so that they can be reused in various different list implementations. The changes also bring a much nice user experience to the table: animations. Because you are now telling the `RecyclerView` exactly what is changing, it will automatically animate the changes for you.

The `DiffUtil` is also an excellent tool in this case because of the dividers. Although the model changes one `ClaimItem` at a time, the `DiffUtil` also takes care of adding and removing any dividers that are affected by these changes. If you triggered each of these events from the database layer, you would need to handle these extra changes manually, and while `DiffUtil` might not be the most efficient tool, it keeps the data absolutely consistent.

Test your knowledge

1. How many different view types can you use in a single `RecyclerView` instance?
    - One or two
    - Any number
    - 256
2. When using a `DiffUtil`, which of the following applies to the data you are comparing?
    - It must be a database entity
    - It must be comparable
    - It is exposed through a Callback
3. When adding dividers to a `RecyclerView`, you should do which of these?
    - Make them part of the item above the divider
    - Add them to the display in the `onBindViewHolder` method
    - Make them distinct items in the dataset

Summary

In this chapter, we largely focused on the `RecyclerView` and how to make it work even better within your application, and especially for overview/dashboard screens. Changes such as adding the dividers and animations don't change the functionality of an application, but they do change the user experience. In this case, they make it easier for the user to understand the screen and easier for them to understand what happened when things changed.

These sorts of changes can be seen as "polishing" the application. You can build the application without them to ensure that everything works, and then add them in afterward. It's a good idea to slowly build a list of generic structures that can be used to quickly polish any application. A good example will be a generic `ActionCommand` to use the `DiffUtil` and apply the changes to an `Adapter`.

In the next chapter, we'll spend some more time on polishing applications. We'll look at animations, colors, and styling, and explore how to define and use them throughout an application to apply a consistent theme.

11
Polishing Your Design

Application polish is one of the more subtle areas of the user experience. The mix of colors, fonts, and animations are generally not something that users register on a conscious level, but this doesn't mean they're not important. While the choice of colors doesn't directly affect the application's functionality, it does affect the usability of the application. These choices can also be the difference between a user completing a sale through your application, or the same user uninstalling it.

Android has a massive array of tools that you can use to polish your application. Applying branding, colors, and extensive theming to your application can be done in ways that allow you to maintain a distinct look and feel, while still following Material Design guidelines and without building any custom widgets. In fact, most graphical effects for widgets on Android can be achieved purely through styling. In this chapter, we'll explore the given topics:

- How to choose and apply colors to an application
- How and when to generate color palettes dynamically
- Creating and applying animations, and when to do so
- Defining and using custom styles for widgets

Choosing colors and theming

Colors are one of the least understood and most important aspects of your user interface design. Text colors must stand out from the background colors to keep text legible, but not too much either. Color choices should follow a palette throughout the application and should reflect the application's branding, but should also help convey meaning to the user. Choosing the right mix of colors will maximize the usability of your application, while helping reduce the user's cognitive load. The wrong color combinations will make text more difficult to read, cause eye strain, and increase the user's levels of cognitive fatigue.

When you apply custom colors to your application, it's important to ensure that you don't have too many colors, and that they are applied consistently within the application. Color conveys meaning; it can be used to tell the user that the *new* button is the opposite of the *delete* button. These styles should be defined as resources and applied consistently throughout your application. Consistent styling helps the user understand each screen in the application more quickly, by telling them what they are looking at. Typically, style information is defined in the `res/values/styles.xml` file of your project. This is an excellent starting point in our exploration of colors and in polishing your application. If you open the `res/values/styles.xml` file of the travel claim example app, you'll see something like this near the top of the file:

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

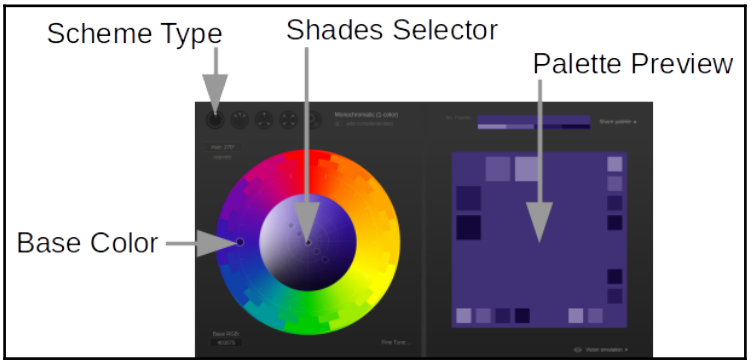
This defines a style named `AppTheme`, which is applied to your entire application from the `AndroidManifest.xml` file. The style declares that its parent is the `Theme.AppCompat.Light.DarkActionBar` style, which is imported from the `app-compat` library (in your `build.gradle` dependencies). The parent of the style is a bit like the parent of a class; it defines all the defaults, and you can override them in the child style. In the default `AppTheme` style, there are three colors that are overridden with references to color resources: primary, primary-dark, and accent. These colors are used all over the `AppTheme` for the backgrounds of the `Toolbar` objects, buttons, floating action buttons, and so on. Primary is used for the background of a `Toolbar` and `FloatingActionButton` by default, primary-dark is used for the status bar background, and accent is used for the foreground of `FloatingActionButton` and the labels above the `TextInputLayout` widgets.

Producing an application palette

The first thing to do when applying colors to your application is to decide on your application color scheme, or palette. A **palette** is a small group of colors that form the basis of your theme, and can be adjusted (typically by making them brighter or darker) to produce a wide range of colors that will all look similar enough to be seen as part of the same theme.

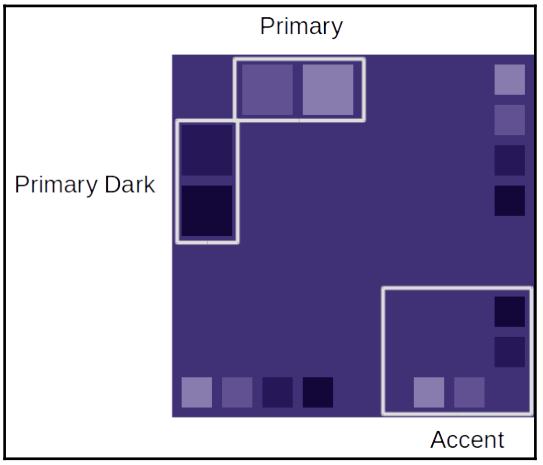
It's best to use a good color designing or palette construction tool. An excellent one is **Paletton**, which is available at `http://paletton.com` for free (another good tool is `https://www.materialpalette.com/`). For this section, we'll use Paletton to define a basic color palette for the travel claims application example; let's get started:

1. Navigate to `http://paletton.com` in your web browser of choice.
2. There are two major parts to the Paletton application; on the left is a color-wheel with draggable handles that allow you to select a primary color (the secondary colors are derived automatically using various available algorithms). On the right of the application is the palette sample for the application:

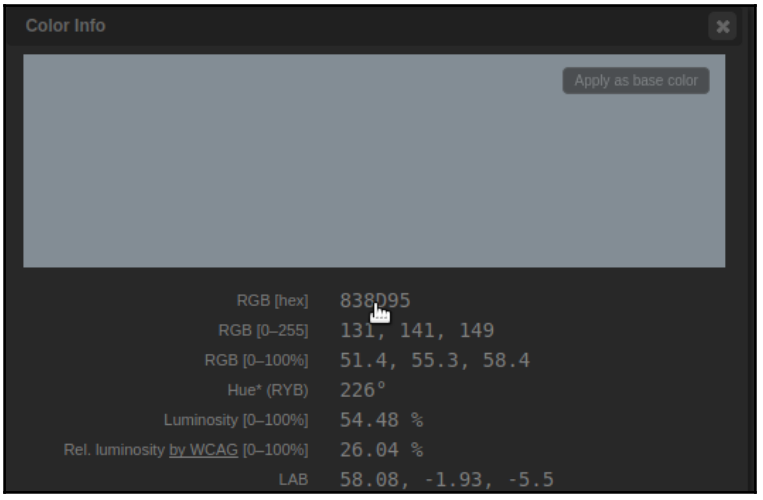


3. Use the **Scheme Type** selector to choose the second type of color scheme: **Adjacent colors (3-colors)**.
4. To the right of the Scheme Type selector, use the small toggle button to turn on **add complementary**. This will add a complimentary color to your palette. A complementary color will be on the opposite side of the color-wheel to your main color, and will serve as *accent* color.

5. Adjust the base color and shades until the palette preview on the right is a combination you are happy with:

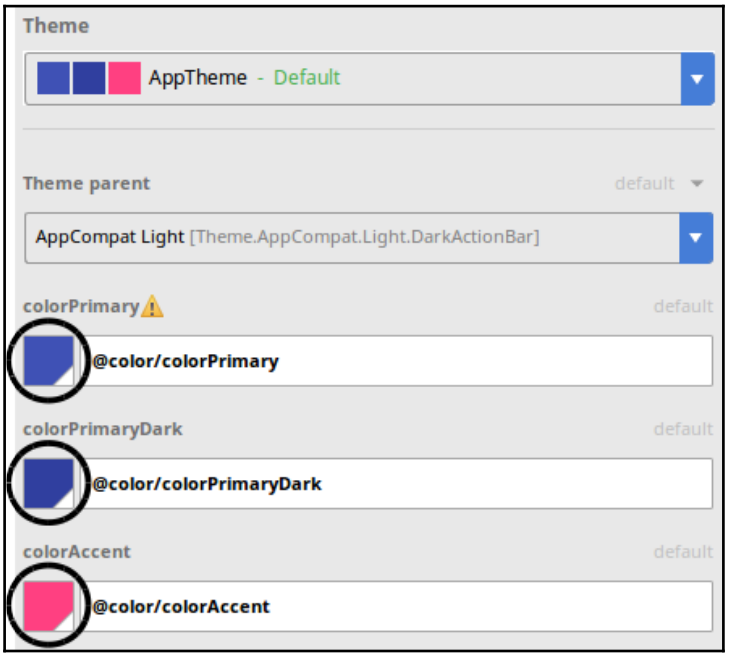


6. By clicking on any of the boxes in the palette preview and then their hex codes, you can copy the RGB hex code to the clipboard and paste it into Android Studio:



7. Ensure that you use colors from the top-left box for your primary and primaryDark colors, while you use a color from the bottom-right box for the accent color.

8. In Android Studio, use **Tools | Android | Theme Editor** to open the Android Theme Editor.
9. In the **Theme** panel on the right-hand side, you'll find a list of colors that define your theme:



10. Click on the color buttons in the Theme Editor to open a color editor. Copy the colors from Paletton into the theme editor for primary, primary-dark, and accent.

If you run the travel claim example application now, you'll see that the entire application has a completely new theme. The floating action button will be the same color as the underline on the `EditText` widget. This will be your accent color, while the background of your `Toolbar` will be your primary color.

It' generally best to use your primary color's complementary color as your accent color. This is the color that is on the opposite side of color wheel, and will normally have an excellent contrast against your primary color. The contrast will help readability and reduce eye strain. It's important to ensure legibility for everyone, and Paletton includes a **Vision simulation** option below the palette preview that can be used to test your palette for various types of colorblindness.

Generating palettes dynamically

There are times when you're not sure what your palette should be ahead of time. There are also times that you would like the color scheme to match some user content, such as a photo or the album art for the music they're listening to. In cases like these, it's nice to be able to grab the key colors from an image and produce a palette that will match them. The problem is that the palette still needs to not be too jarring, and your text still needs to be legible against the background color. These are pretty hard problems to solve in pure code, but Android support libraries have an excellent little tool that does exactly this--the `Palette` API.

A very useful way of using generated palettes is to colorize cards with different icons according to the colors in the icon. Let's write a `CardView` implementation that can colorize its contents according to a generated palette:

1. You'll first need to add the `Palette` API to your project. In the travel claim app, open the **app** module's `build.gradle` file in Android Studio.
2. In the `dependencies` at the bottom of the file, include the `Palette` API by declaring this:

```
implementation 'com.android.support:palette-v7:+'
```

3. Click on the **Sync Now** link that appears at the top of the editor panel.
4. Right-click on the widget's package and select **New| Java Class**.
5. Name the new class `ColorizedCardView`.
6. Change the `Superclass` to `android.support.v7.widget.CardView`.
7. Add `android.support.v7.graphics.Palette.PaletteAsyncListener` to the **Interface(s)**.
8. Click **OK** to create the new class.
9. Add the required `View` constructors so that the class can be used from XML files:

```
public ColorizedCardView(final Context context) {
  super(context);
}

public ColorizedCardView(
      final Context context,
      final AttributeSet attrs) {
  super(context, attrs);
}

public ColorizedCardView(
```

```
        final Context context,
        final AttributeSet attrs,
        final int defStyleAttr) {
    super(context, attrs, defStyleAttr);
}
```

10. The `ColorizedCardView` doesn't just change its own background, it needs to change the color of any text as well so that the text remains legible to the user. This means that the `ColorizedCardView` needs to find all the `TextView` instances that don't have their background `Drawable` set (a `Button` is just a `TextView` with a specialized background, and we want to leave that as is). This method will traverse (depth-first) into the `ColorizedCardView`, and add any `TextView` objects it finds to a `Collection`:

```
static Collection<TextView> findTextViews(
    final ViewGroup viewGroup,
    final Collection<TextView> textViews) {

  final int childCount = viewGroup.getChildCount();
  for (int i = 0; i < childCount; i++) {
    final View child = viewGroup.getChildAt(i);

    if (child instanceof ViewGroup) {
      // recurse downwards
      findTextViews((ViewGroup) child, textViews);
    } else if (child instanceof TextView
        && child.getBackground() == null) {
      textViews.add((TextView) child);
    }
  }

  return textViews;
}
```

11. Each `Palette` is actually a list of `Swatch` objects, each one containing a base color and colors suitable for heading text and body text. The `ColorizedCardView` allows you to specify a `Swatch` directly to colorize the background and all the text:

```
public void setSwatch(final Palette.Swatch swatch) {
  setCardBackgroundColor(swatch.getRgb());

  final Collection<TextView> textViews = findTextViews(
      this, new ArrayList<TextView>()
  );
```

```
    if (!textViews.isEmpty()) {
      for (final TextView textView : textViews) {
        textView.setTextColor(swatch.getBodyTextColor());
      }
    }
  }
```

12. When a `Palette` is generated, it can have any number of `Swatch` objects. There are a selection of *standard* swatches that are normally generated when you create a `Palette` from a `Bitmap`, but any number of them may remain unpopulated (`null`). When you colorize the card by a `Palette` object, you'll need to look for an available `Swatch`; in the `ColorizedCardView` implementation. We'll favor *light* swatches over *dark* swatches, and *muted* swatches over *vibrant*:

```
public void setPalette(final Palette palette) {
  if (palette.getLightMutedSwatch() != null) {
    setSwatch(palette.getLightMutedSwatch());
  } else if (palette.getLightVibrantSwatch() != null) {
    setSwatch(palette.getLightVibrantSwatch());
  } else if (palette.getDarkMutedSwatch() != null) {
    setSwatch(palette.getDarkMutedSwatch());
  } else if (palette.getDarkVibrantSwatch() != null) {
    setSwatch(palette.getDarkVibrantSwatch());
  }
}
```

You may need to adjust the ordering of this method in your application, depending on the colors chosen for the rest of the application. Typically, muted colors cause less eye strain for your users, but you may want to colorize action buttons using vibrant colors to draw attention to them.

13. Now, we need a way to specify a `Bitmap` to colorize the entire `ColorizedCardView` with. The `Palette` uses a `Builder` object to generate its swatches, and has its own built-in `AsyncTask` to handle generating the `Palette` on a background thread (which can take a few seconds on larger images or slower devices). The `setColorizeBitmap` method is defined so that it's easy to invoke from a data-bound layout XML file. The `Palette.Builder` needs a callback to handle the generated `Palette`, which will be the `ColorizedCardView` instance (remember that you've implemented the `PaletteAsyncListener` interface):

```
public void setColorizeBitmap(final Bitmap image) {
  new Palette.Builder(image).generate(this);
}
```

```
        @Override
        public void onGenerated(final Palette palette) {
          setPalette(palette);
        }
```

14. You'll also need a way to colorize the `ColorizedCardView` based on a `Drawable` object, which will offer better interoperability with application `Resources`. The following `renderDrawable` method has a shortcut if the `Drawable` object is a `BitmapDrawable` (which simply wraps a `Bitmap`); otherwise, it'll try and render the `Drawable` to a `Bitmap` object. As a `Drawable` has bounds that include its position (and not just its size), you'll need to translate the `Canvas` that it's going to draw on so that it renders in the top-left corner of the `Bitmap`:

```
private Bitmap renderDrawable(final Drawable drawable) {
  if (drawable instanceof BitmapDrawable) {
    return ((BitmapDrawable) drawable).getBitmap();
  }

  final Rect bounds = drawable.getBounds();
  final Bitmap bitmap = Bitmap.createBitmap(
      bounds.width(),
      bounds.height(),
      Bitmap.Config.ARGB_8888
  );

  final Canvas canvas = new Canvas(bitmap);
  canvas.translate(-bounds.left, -bounds.top);
  drawable.draw(canvas);

   return bitmap;
}

public void setColorizeDrawable(final Drawable drawable) {
  setColorizeBitmap(renderDrawable(drawable));
}
```

To use the `ColorizedCardView` in the travel claim application, you can find and download colored icons for all the categories and change the `ItemPresenter` to use them, instead of the standard black icons we imported from the Material Icons set. An excellent resource for finding icons and sets of icons is Iconfinder--`https://www.iconfinder.com/`. Iconfinder allows you to search for and filter icon sets according to your criteria, and purchase or download the icons you need for your application.

To change the overview screen to use your favorite colorful icons, follow these steps:

1. Place your new icons in the `res/drawable` directory of your application; ensure that you download PNG icons so that they can be read by Android.
2. Open the `card_claim_item` layout resource in Android Studio.
3. Change to the **Text** editor.
4. Change the declaration of the `CardView` to a `ColorizedCardView`, and use the `app:colorizeDrawable` data binding attribute to invoke `setColorizeDrawable` with the same `Drawable` that will be rendered as the icon:

```
<com.packtpub.claim.widget.ColorizedCardView
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_marginTop="@dimen/grid_spacer1"
    android:foreground="?attr/selectableItemBackground"
    android:onClick="@{() -> presenter.viewClaimItem(item)}"
app:colorizeDrawable="@{presenter.getCategoryIcon(item.category)}"
```

5. Open the `ItemPresenter` Java source file.
6. Change the icons returned by the `getCategoryIcon` method to return your new icons, instead of those used by the category selector:

```
public Drawable getCategoryIcon(final Category category) {
  final Resources resources = context.getResources();
  switch (category) {
    case ACCOMMODATION:
        return resources.getDrawable(R.drawable.hotel);
    case FOOD:
        return resources.getDrawable(R.drawable.dinner);
    case TRANSPORT:
        return resources.getDrawable(R.drawable.airplane);
    case ENTERTAINMENT:
        return resources.getDrawable(R.drawable.clapboard);
    case BUSINESS:
        return resources.getDrawable(R.drawable.briefcase);
    case OTHER:
    default:
        return resources.getDrawable(R.drawable.misc);
  }
}
```

The icon names used earlier are just an example; you'll need to use the names of the icon files you downloaded and placed in the `drawable` directory.

The `ColorizedCardView` is a very useful and generic implementation of colorization using the `Palette` class. Using the bold background colors on each card makes them quickly recognizable to the user, and allows the user to more quickly find what they are looking for in a long scrolling list of items. As it can be automatically colorized using data binding, the `ColorizedCardView` can be populated with virtually any content.

Adding animations

Animations may appear to just be a nice bit of polishing on top of your user interface, but they can also serve an important purpose. In any design, whether it's a building, an API, or a user interface, it's good to try and follow a principle of *least surprise*. Try and offer your user things that make sense without them having to try and understand the details of how it works. A good example of violating this principle is when a button is wired incorrectly. If you were to press the *copy* button on a printer, and instead of making a copy it printed a test page, this would be a surprise. You expected the machine to do one thing because of the label, but it did something unexpected.

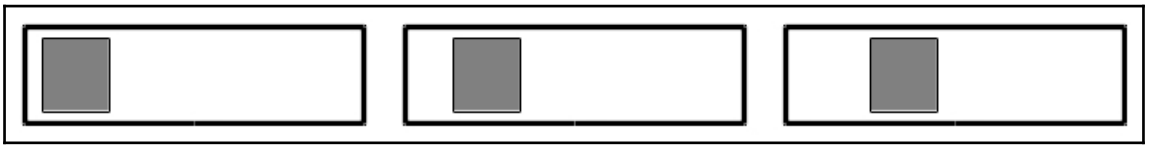
It's always important to consider what your user will expect to happen when they look at or use your application's user interface. Using well-known names and icons for the elements of a user interface help make it instantly understood by your user, but sometimes your application will change what is on the screen without being entirely obvious as to *what has changed*. In cases like this, animations become essential to tell users what has happened. A good example of using animation to express a change is the automatic animations you added to the `RecyclerView` using the `DiffUtil` class. When the user adds a new claim item, it appears in the list at the correct position, but an animation will draw the user's attention to where it has appeared and let them know that it's the new item.

Animations have to strike a careful balance. However, if everything is animated, then the user can become frustrated by the extra time being taken by the animations. This leads to another important factor--animations should be quick. The Android platform defines a *short* animation as just *200 milliseconds*, just one-fifth of a second.

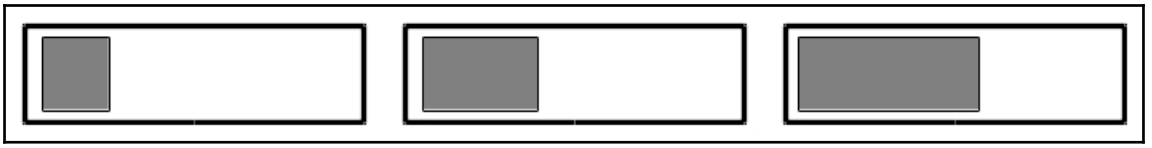
You've already added implicit animations to the travel claim application using the `RecyclerView` and `DiffUtil`. **Implicit animations** are all over in the Android platform and cover a wide range of everyday cases, such as the changes in the `RecyclerView` contents. There are also ways to add your own animations to layouts and widgets, and there are several widgets that are specifically designed to render animations and transitions.

Within layout animations, there are four basic actions that an animation can perform on a widget or group of widgets that are being animated.

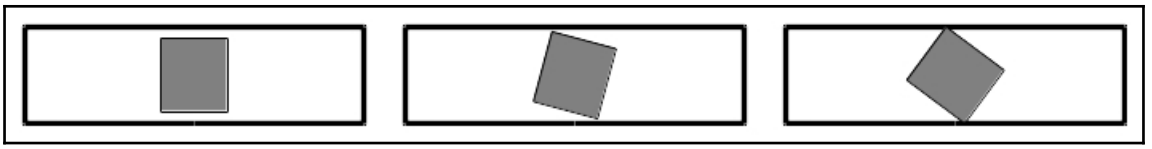
A widget can be translated, which involves moving it left or right, up or down (or any combination of those), as shown here:



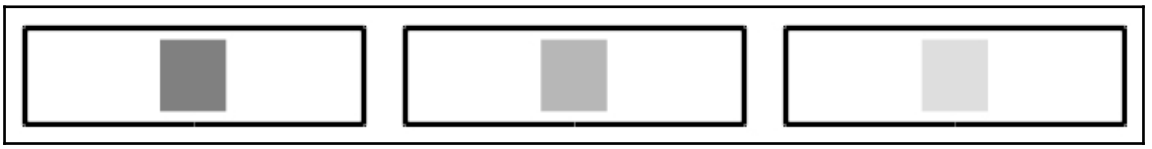
An animation can also scale a widget. This involves changing its size to make it appear larger or smaller. Scaling, like translation, can be applied on either the horizontal (x) axis, or the vertical (y) axis, or both of them together:



You can also have an animation rotate the widget. Rotation is not a natural change for a user interface widget, as typically, all widgets are layouts out of a box-like grid. Rotation can be very useful, and can produce a pleasing effect when applied to widgets that appear to be round (such as a `FloatingActionButton` or a circular avatar):



While all the first three transformations are concerned with the physical structure of the widget being animated, the fourth one changes how opaque it is. The alpha transformation allows you to produce animations where widgets appear to fade-in or fade-out:



These four animation actions can be combined into what Android calls a **set**. A *set* is a group of animation actions that will all appear to happen at the same time.

Creating custom animations

Android animations are in fact resource files, much like an icon or a layout. The animations that apply to layouts and widgets are XML files that define the various transformations and are placed in the `res/anim` directory. Android provides a small selection of simple animations that you can use in your application, without needing to build your own:

- `android.R.anim.fade_in` - `@android:anim/fade_in`
- `android.R.anim.fade_out` - `@android:anim/fade_out`
- `android.R.anim.slide_in_left` - `@android:anim/slide_in_left`
- `android.R.anim.slide_out_right` - `@android:anim/slide_out_right`

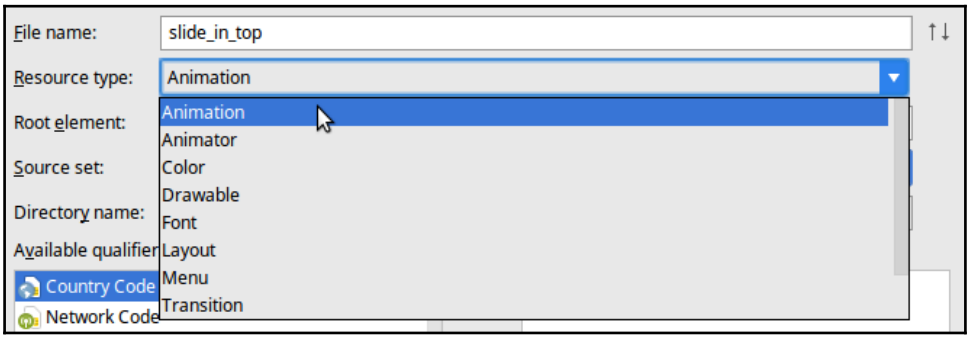
These four animations cover two different types of transition: fading out and in, or sliding the widgets from left to right. Nothing stops you from mixing them together, for example, fading the widget out and then sliding a new one in from the left.

To do these sorts of transitions, there are a family of Android widgets that will manage the animation for you. These can either focus on animating the content (that is, text or an image), or they transition through a list of child widgets. The base of these classes is `android.widget.ViewAnimator`, and the best-known implementations are these:

- `TextSwitcher`: Behaves like an animated `TextView`; each time its text is changed, it animates between the old text and the new text
- `ImageSwitcher`: Just like `TextSwitcher`, but for images
- `ViewFlipper`: It is used like a `FrameLayout`, but only one of its children is shown at a time, and you can have it animate between them

Let's create two new animation sets to animate some text, and change the category label in the `CategoryPickerFragment` to use a `TextSwitcher`:

1. Right-click on the **res** directory in the travel claim example app and select **New | Android resource file**.
2. Name the new file `slide_in_top`.
3. Change the **Resource type** to **Animation** (not **Animator**):



An `Animator` allows the direct manipulation of any property in any Java Object; while this is a very powerful system to use, it does not work with classes such as `TextSwitcher`. `Animation` refers to the *view animation* system, which is designed specifically for animating widgets, and has various performance enhancements in the layout system to avoid the user interface stuttering during animations.

4. Click **OK** to create the new animation XML resource.
5. On the `<set>` element, we need to define how long the animation will take, and the interpolator. The **interpolator** defines the relative motion of the animation. Does it happen with linear smoothness (which often appears artificial, but is the easiest), or does the animation appear to *bounce*, or something else entirely? In this case, we will use the standard `anticipate_overshoot_interpolator`, which includes a small *bounce* effect at the end of the animation:

```
<?xml version="1.0" encoding="utf-8"?>
<set xmlns:android="http://schemas.android.com/apk/res/android"
android:interpolator="@android:anim/anticipate_overshoot_interpolat
or"
    android:shareInterpolator="true"
    android:duration="@android:integer/config_shortAnimTime">
</set>
```

6. This animation will consist of two parts. The first part is a translation from offscreen, downward to where the text should normally appear. The second part is a fade in from fully transparent to opaque. Each action taken by a view animation is defined in terms of what the value should be when the animation starts, and what the value should be when it stops (from and to). The values in between are defined by the time of each frame, and the interpolator. Inside the `<set>` element, add a `translation` to bring the view along the y-axis from above where it ends:

```
<translate
    android:fromYDelta="-50%p"
    android:toYDelta="0" />
```

7. Now, add the fade-in using an `alpha` action. A zero alpha value indicates that the widget should be invisible, while a one indicates that it should be fully opaque. The alpha is a floating-point number, so you can define any value between zero and one for partial transparency:

```
<alpha
    android:fromAlpha="0.0"
    android:toAlpha="1.0" />
```

8. While a single animation is nice, you need two animations running together to create a *transition*. Right-click on the new `res/anim` directory and select **New | Animation resource file**.
9. Name the new animation `slide_out_bottom`.
10. Click **OK** to create the new resource file.
11. This animation works the same way as `slide_in_top`, but pushes the view downward and makes it transparent:

```
<?xml version="1.0" encoding="utf-8"?>
<set xmlns:android="http://schemas.android.com/apk/res/android"
android:interpolator="@android:anim/anticipate_overshoot_interpolat
or"
    android:shareInterpolator="true"
    android:duration="@android:integer/config_shortAnimTime">

    <translate
        android:fromYDelta="0"
        android:toYDelta="50%p" />

    <alpha
        android:fromAlpha="1.0"
        android:toAlpha="0.0" />
```

```
</set>
```

12. Now, you'll need to change the `CategoryPickerFragment` to use a `TextSwitcher` instead of a `TextView`. Start by opening the `fragment_category_picker` layout resource file and change to the **Text** editor.
13. Locate the `TextView` at the bottom of the file, and change it to be a `TextSwitcher`. A `TextSwitcher` needs two `TextView` child elements to animate between. Each time you change the text on the `TextSwitcher`, it puts the new text on the invisible `TextView` and then animates between the visible `TextView` and the invisible one (that is, it switches them around and hence its name). You'll need to tell the `TextSwitcher` to use the animation resources you just created as its *in* and *out* animations:

```
<TextSwitcher
    android:id="@+id/selected_category"
    android:inAnimation="@anim/slide_in_top"
    android:outAnimation="@anim/slide_out_bottom"
    android:layout_width="match_parent"
    android:layout_height="wrap_content">

    <TextView
        android:layout_width="match_parent"
        android:layout_height="match_parent"
android:textAppearance="@style/TextAppearance.AppCompat.Medium" />

    <TextView
        android:layout_width="match_parent"
        android:layout_height="match_parent"
android:textAppearance="@style/TextAppearance.AppCompat.Medium" />
</TextSwitcher>
```

14. Open the `CategoryPickerFragment` source file and change the references to the `TextView` to a `TextSwitcher`. There will be one as a field, and the other should be in the `onCreateView` method:

```
private RadioGroup categories;
private TextSwitcher categoryLabel;

// …

categories = (RadioGroup) picker.findViewById(R.id.categories);
categoryLabel = (TextSwitcher)
picker.findViewById(R.id.selected_category);
```

15. Open the `IconPickerWrapper` source file. This currently wraps a `TextView`, but will now need to wrap a `TextSwitcher`. Like the `CategoryPickerFragment`, change the references from `TextView` to `TextSwitcher`:

```
private final TextSwitcher label;
public IconPickerWrapper(final TextSwitcher label) {
    this.label = label;
}
```

That's all you need to do in this case; the `CaptureClaimActivity` will now have a very pleasing animation on the text in the category chooser, which indicates that the icons are used to change the category. While a `TextSwitcher` doesn't subclass `TextView`, it does expose the same critical method for these cases--`setText(CharSequence)`. Unfortunately, this means that you can't substitute the classes directly one for the other. Instead, you'll need to treat each as a separate type (as earlier). You can, however, create `abstract wrapper` class to wrap these two and allow your layout to define whether there should be animations or not:

```
public abstract class TextWrapper<V extends View> {
  public final V view;

  public TextWrapper(final V view) {
    this.view = view;
  }

  public abstract void setText(CharSequence text);

  public abstract CharSequence getText();

  public static TextWrapper<TextView> wrap(final TextView tv) {
    return new TextWrapper<TextView>(tv) {
        @Override
        public void setText(final CharSequence text) {
          view.setText(text);
        }

        @Override
        public CharSequence getText() {
          return view.getText();
        }
      };
  }

  public static TextWrapper<TextSwitcher> wrap(final TextSwitcher ts) {
    return new TextWrapper<TextSwitcher>(ts) {
        @Override
```

```
            public void setText(final CharSequence text) {
              view.setText(text);
            }

            @Override
            public CharSequence getText() {
              return ((TextView) view.getCurrentView()).getText();
            }
          };
        }

        public static TextWrapper<?> wrap(final View v) {
          if (v instanceof TextView) {
            return wrap((TextView) v);
          } else if (v instanceof TextSwitcher) {
            return wrap((TextSwitcher) v);
          } else {
            throw new IllegalArgumentException("unknown text view: " + v);
          }
        }
    }
```

This class can be used to wrap references to widgets that can be either `TextView` or `TextSwitcher`, depending on the context. This allows you to reuse more of your Java code when dealing with cases where some screens require a simple layout, while others require animations. It's generally a useful pattern to remember, because it reduces the coupling between your user interface and your code when you can't use the class inheritance, and want to avoid casting.

Data binding can also be used to solve this problem. By having the `CategoryPickerFragment`, use a data-bound layout; the `TextSwitcher` will automatically animate when the model was changed by the user clicking on the `RadioButton` widgets.

Activating more animations

There are some other small ways in which Android can provide your application with animations that let the user know what is going on. For example, you can tell any `ViewGroup` implementation (any of the `Layout` classes: `FrameLayout`, `LinearLayout`, or `ConstraintLayout`) to animate the changes to the layout. You do this by simply turning on `animateLayoutChanges` in your layout resource:

```
<android.support.v7.widget.CardView
    android:animateLayoutChanges="true"
```

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

This is especially useful when you offer the ability to *unfold* a card to expose more functionality or more information. Coupling the `animateLayoutChanges` attribute with the `ViewGroup` class is a very powerful combination. `ViewStub` is a special type of widget that can be used like an `<include>`, that only loads when you tell it to. When it's loaded, it doesn't act as a container, but *replaces itself* with the layout it has loaded. Using `animateLayoutChanges` inflating a `ViewStub` can automatically trigger a nice animation to reveal the new content to the user. The following code snippet is a `CardView` that will animate the inflation of a menu that can be made to appear at the bottom of the card:

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.v7.widget.CardView
    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="wrap_content"
    app:contentPadding="@dimen/grid_spacer1">

    <android.support.constraint.ConstraintLayout
        android:animateLayoutChanges="true"
        android:layout_width="match_parent"
        android:layout_height="match_parent">

        <ImageView
            android:id="@+id/imageView"
            android:layout_width="48dp"
            android:layout_height="48dp"
            app:layout_constraintStart_toStartOf="parent"
            app:layout_constraintTop_toTopOf="parent"
            app:srcCompat="@drawable/ic_category_food" />

        <TextView
            android:id="@+id/heading"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_marginStart="8dp"
            android:textAppearance="@style/TextAppearance.AppCompat.Large"
            app:layout_constraintStart_toEndOf="@+id/imageView"
            app:layout_constraintTop_toTopOf="parent"
            tools:text="Dinner a the Hotel" />

        <TextView
            android:id="@+id/date"
            android:layout_width="wrap_content"
```

```
            android:layout_height="wrap_content"
            android:layout_marginStart="8dp"
            app:layout_constraintStart_toEndOf="@+id/imageView"
            app:layout_constraintTop_toBottomOf="@+id/heading"
            tools:text="22-September-2017" />

        <ViewStub
            android:id="@+id/menu"
            android:layout_width="0dp"
            android:layout_height="wrap_content"
            android:layout_marginEnd="8dp"
            android:layout_marginStart="8dp"
            android:layout_marginTop="8dp"
            android:layout="@layout/card_menu"
            app:layout_constraintBottom_toBottomOf="parent"
            app:layout_constraintEnd_toEndOf="parent"
            app:layout_constraintStart_toStartOf="parent"
            app:layout_constraintTop_toBottomOf="@+id/date" />

    </android.support.constraint.ConstraintLayout>
</android.support.v7.widget.CardView>
```

When you inflate the preceding `ViewStub`, it will replace itself with the contents of the `card_menu` layout resource, and the `ConstraintLayout` will animate the change, making the `card_menu` appear to unfold. You can use the following code snippet to inflate the `ViewStub` when the `CardView` is tapped on by the user:

```
cardView.setOnClickListener(new View.OnClickListener() {
  @Override public void onClick(final View view) {
    final ViewStub menu = (ViewStub) findViewById(R.id.menu);
    menu.inflate();
    view.setOnClickListener(null);
  }
});
```

The preceding code is a single-use `OnClickListener` that removes itself when it's been triggered. This is important because once a `ViewStub` has been inflated, it no longer exists in the layout. After the preceding listener is triggered, `findViewById(R.id.menu)` will return the root element of the `card_menu` layout resource, not the `ViewStub`.

Creating custom styles

When polishing your application, you'll find that certain styling requirements become common over the entire application, but in specific places. For example, the *positive* / *go* buttons should have a specific background color that highlights them from the other buttons in the application, or that the *negative* / *delete* buttons should have a color that highlights them as destructive for the user.

Android offers you the ability to define your own styles apart from those defined by the system. The theming system in Android is built entirely on top of the styling system. Styles have some very simple attributes:

- Styles can be named
- A style can change any attribute exposed in the layout XML file
- A style can inherit from another style and override its attributes (a bit like classes extending each other)
- Styles are defined as value resources (a bit like dimensions, strings, and colors)

Let's jump right in and create a new style for the travel claim application for the amount-input; we want to create a style that can be reused whenever the user needs to type a monetary amount into the application:

1. Open the `styles.xml` resource file in the **res/values** project folder.
2. You'll note in this file that you already have several styles defined by the Android Studio templates. These are mostly theme-related styles, and will apply to the entire application. We want to define a new style that can be applied to specific widgets. Declare a new style element named `AmountInput`:

```
<style name="AmountInput">
</style>
```

3. The first thing we want this style to do is to align the text to the right of the input box. This is normally done by changing the `android:gravity` attribute on the `EditText` box. In the `style` element, you need to declare this as an `item` you wish to override:

```
<item name="android:gravity">right</item>
```

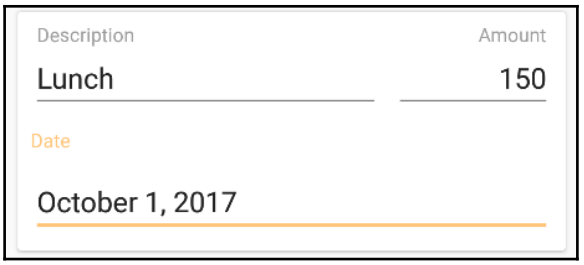
4. You also want to change the focus behavior so that when the user taps to edit the amount, the existing value is selected. This allows them to more easily enter a new number, which is more common than wanting to edit an existing number. The `TextView` class defines an attribute named `selectAllOnFocus` that is perfect for this purpose:

```
<item name="android:selectAllOnFocus">true</item>
```

5. To apply the style to the amount input, open the `fragment_claim_capture_details.xml` layout resource in **Text** mode (this is from the Try it Yourself section of `Chapter 4`, *Composing User Interfaces*).
6. Find the `EditText` entry for the amount, and apply the style. It's important to note that the style attribute is not in the android XML namespace:

```
<EditText
    style="@style/AmountInput"
    android:id="@+id/amount"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:hint="@string/label_amount"
    android:inputType="number|numberDecimal" />
```

When you run the application or change to the **Design** view, you'll find that the amount field is now right aligned, and if you tap on it, the entire content will be selected. This style can now be applied to any number of fields in your application:



Styles themselves can be overridden at each layer. When you inherit from another style, the child can override any of its parent items. When a widget has a style applied, any attributes specified on the widget's XML element take precedence over the style being applied. For example, if you wanted to create an `AmountInput` styled widget that left aligned its text content (instead of the style's right alignment), you might use the following:

```
<EditText
    style="@style/AmountInput"
    android:id="@+id/amount"
```

```
android:gravity="left"
android:layout_width="match_parent"
android:layout_height="wrap_content"
android:hint="@string/label_amount"
android:inputType="number|numberDecimal" />
```

Although it's not commonly done, you can also use styles to apply attributes, such as labels and hints, to widgets. This allows for two screens to easily replicate a widget exactly, without requiring an `include`. Any time you find that your layout code appears to repeat itself, consider using a style if an include doesn't look suitable.

Test your knowledge

1. When choosing a color scheme, it's important that the accent color has which of these features?
    - It is the same hue as the primary color
    - It is complementary to the primary color
    - It is not black and not white
2. Dynamically generating a palette should meet which of these conditions?
    - It should be used in preference to defining the color scheme up front
    - It should be done on a background thread
    - It should only be used in media applications
3. Which of these is to be kept in mind while animating layouts in your application?
    - They should not block or distract the user from achieving their goals
    - They should be done whenever the user interface changes
    - They should be kept as simple as possible to conserve battery
4. Custom styles can be used to define which of these?
    - Common groups of attributes for widgets based on their class
    - Common groups of attributes to be applied through the `style` attribute
    - Default values for any attribute in a layout resource file

Summary

Polishing an application (much like optimizing an application) shouldn't be taken on too early in its development, as it can distract getting the application working and making the user experience smooth. It is, however, a vital part of an application's development, and the careful application of colors, fonts, and animations can sometimes be the difference between success and failure.

Using color tools such as Paletton make the selection of a color-scheme much easier. It's important to also consider how color-blind people will see your application, and to ensure that the application is still usable by this sizeable portion of the population. If you know someone who has any form of color blindness, ask them to help test your choice in colors. Alternatively, use the color blindness simulations that palette design tools such as Paletton provide.

When adding animations to the application, it's a good idea to leverage the default animation systems provided by the platform. Avoid adding animations to widgets that don't already provide some form of animation capability. If you find yourself animating things by hand, there is probably something wrong. Try and stick to using the animations built into classes like `RecyclerView` and `ViewPager`, while using animating widgets such as `TextSwitcher` where appropriate. It's also important to keep animations short. While you might think your animation looks lovely, your users will become frustrated if they slow down the use of the application.

In this chapter, we looked at various ways in which your application can be styled to fit a color-scheme, and polished with animations and styles for certain components. In the next chapter, we'll look at how to create your own completely customized widget classes and how to repurpose the existing widget classes for new or specialized use cases.

12
Customizing Widgets and Layouts

In everyday development on Android, you'll find that the core platform and support libraries offer a wide range of widgets and layouts for you to build your application. There are also a wealth of open source and third-party widgets available on the internet. The *Android Arsenal* website (`https://android-arsenal.com/`) is a well-cataloged list of APIs available for Android, and it is an excellent starting point when you need some functionality that is not available in the platform or support libraries. Even with this massive wealth of available widgets and libraries, you'll sometimes find yourself wanting a widget that hasn't already been built.

Creating your own widgets on any platform is a sizable undertaking. Widgets need to be able to render themselves to look as native as possible using graphics primitives such as lines, arcs, circles, and polygons. Many Android widgets (such as `Button`) avoid having to do this using the excellent `Drawable` class and resources. This enables you to customize the look of widgets simply by changing the drawable resources they use to a stateful drawable (as you did with the `RadioButton` widgets in `Chapter 2`, *Designing Form Screens*).

In this chapter, we'll take a look at how to build custom widgets and layout components. We'll take a look at the best practices to use when building your own `View` implementations, and how to render 2D graphics using the Android graphics APIs. Specifically, we'll explore the following:

- Creating a completely custom `View` class
- Rendering 2D graphics using graphics primitives
- How to create a custom `ViewGroup` to produce custom layout effects
- Rendering animations using `Drawable` objects
- Creating `View` classes that self-animate

Creating custom view implementations

Sometimes, the existing widgets just aren't enough, no matter how much you customize them. Sometimes, you need to display something that simply isn't supported by the platform. In these cases, you might find yourself needing to implement your own custom widget. The `View` class can be easily extended to produce many different effects, but there are a few things that are worth knowing before you tackle it:

- The rendering for a `View` is expected to happen in the `onDraw` method.
- When rendering the graphics for the `View`, you'll use a `Canvas` to send the drawing instructions.
- Each `View` is responsible for calculating the offsets for its padding, and by default, the graphics will be clipped to these dimensions.
- You should avoid any object allocation (including arrays, if possible) in the `onDraw` method. The `onDraw` methods are probably the most time-sensitive method calls in any application, and need to produce as little garbage as possible. Any object allocations should be done in other methods and just used in the `onDraw` implementation.

In the travel claims example, it will be really nice if the user can see a simple overview graph of their spending for the last few days. To do this, we'll need to write a class that can draw this graph for them. It's useful to be able to change some of the `View` attributes (specifically, the size and color of the line graph) using the layout XML file. For this, you'll need to specify the attribute names and their type information for the layout resource compiler. Follow these instructions to write a simple line-graph `View` implementation:

1. Right-click on the `res/values` resource directory in the travel claim app and select **New| Values resource file**.
2. Name the new file `attrs_spending_graph_view`.
3. Click **OK** to create the new resource file.
4. You'll use this file to declare some new XML attributes for the resource compiler that can be used in your layout XML files when dealing with your new graph `View` class. These XML attributes are given type information (in the form of a `format` attribute), which affects how the resource compiler handles them in the layout XML:

```
<resources>
  <declare-styleable name="SpendingGraphView">
    <attr name="strokeColor" format="color" />
    <attr name="strokeWidth" format="dimension" />
  </declare-styleable>
</resources>
```

5. Now, right-click on the widget package and select **New| Java Class**.
6. Name the new class `SpendingGraphView`.
7. Change the `Superclass` to `android.view.View`.
8. Click **OK** to create the new class.
9. In the `SpendingGraphView`, declare variables to hold the values that can be specified in the layout XML files. These should typically reflect the names used in the XML file, and should be initialized with sensible default values:

```
private int strokeColor = Color.GREEN;
private int strokeWidth = 2;
```

10. Next, declare an array for the data points to be rendered into the graph. In this implementation, we'll assume that each data point is the amount spent on an unspecified day:

```
private double[] spendingPerDay;
```

11. As mentioned earlier, the `onDraw` implementation should have to do as little work as possible. In this graph implementation, it means that the entire graph is actually calculated ahead of time, and cached in local variables to be drawn in the `onDraw` method. The Android graphics APIs provide a `Path` class to define any abstract group of connected lines, and the `Paint` class that defines the colors, stroke size (pen), fill-style, and so on. You'll need to declare one of each of these to be calculated and rendered:

```
private Path path = null;
private Paint paint = null;
```

Storing the widget's rendering state on a field in the class might seem to go against everything you know about where you should store and pass state, but a widget is a form of state container. Its job is to present its state to the user and to capture events to trigger state changes. Keeping the construction of the graphics primitives out of the `onDraw` implementation means that the graphics pipeline isn't slowed by recalculating the graphics state from the graph data for every frame.

12. Now, implement the standard constructors for a `View` class. You'll want all of these constructors to invoke a single `init()` method to handle the actual initialization of the widget, which in this case, will also need to fetch and read the attributes given in the layout XML:

```
public SpendingGraphView(final Context context) {
  super(context);
  init(null, 0);
}

public SpendingGraphView(
    final Context context,
    final AttributeSet attrs) {
  super(context, attrs);
  init(attrs, 0);
}

public SpendingGraphView(
    final Context context,
    final AttributeSet attrs,
    final int defStyle) {
  super(context, attrs, defStyle);
  init(attrs, defStyle);
}
```

13. Now, implement the `init` method and use the `Context` to convert the `AttributeSet` object and its data into a `TypedArray` object. This is where all the style information is merged in from the current `Theme` of the application. When you are finished with a `TypedArray`, you need to recycle them, handing them back to the platform to be reused. This helps the performance of the `obtainStyledAttributes` method:

```
private void init(final AttributeSet attrs, final int defStyle) {
  final TypedArray a = getContext().obtainStyledAttributes(
          attrs, R.styleable.SpendingGraphView, defStyle, 0);

  strokeColor = a.getColor(
          R.styleable.SpendingGraphView_strokeColor,
          strokeColor);

  strokeWidth = a.getDimensionPixelSize(
          R.styleable.SpendingGraphView_strokeWidth,
          strokeWidth
  );

  a.recycle();
}
```

14. In order to paint the graph correctly, you'll need a utility method to help find the scale of the vertical axis. This involves finding the maximum value that the graph will have, and unfortunately, the Android platform doesn't expose a method to do this directly, so you'll need to implement it yourself:

```
protected static double getMaximum(final double[] numbers) {
  double max = 0;

  for (final double n : numbers) {
    max = Math.max(max, n);
  }

  return max;
}
```

15. The next step is to implement the actual rendering method for the graph data. This method will be invoked on the main thread, but won't be invoked as part of the rendering loop. Instead, you'll calculate all the values and plot the graph using a `Path` object (a vector graphics primitive). Then, this method will store the plotted line and the `Paint` to use in the path and paint fields you declared, and signal that the `View` is *invalid* and needs to have its `onDraw` method called as soon as possible:

```
protected void invalidateGraph() {
  if (spendingPerDay == null || spendingPerDay.length <= 1) {
    path = null;
    paint = null;
    invalidate();

    return;
  }

  final int paddingLeft = getPaddingLeft();
  final int paddingTop = getPaddingTop();
  final int paddingRight = getPaddingRight();
  final int paddingBottom = getPaddingBottom();

  final int contentWidth =
        getWidth() - paddingLeft - paddingRight;
  final int contentHeight =
        getHeight() - paddingTop - paddingBottom;
  final int graphHeight =
        contentHeight - strokeWidth * 2;

  final double graphMaximum = getMaximum(spendingPerDay);

  final double stepSize = (double) contentWidth / (double)
(spendingPerDay.length - 1);
  final double scale = (double) graphHeight / graphMaximum;

  path = new Path();
  path.moveTo(paddingLeft, paddingTop);

  paint = new Paint();
  paint.setStrokeWidth(strokeWidth);
  paint.setColor(strokeColor);
  paint.setFlags(Paint.ANTI_ALIAS_FLAG);
  paint.setStyle(Paint.Style.STROKE);

  path.moveTo(
      paddingLeft,
      contentHeight - (float) (scale * spendingPerDay[0]));
```

```
    for (int i = 1; i < spendingPerDay.length; i++) {
      path.lineTo(
            (float) (i * stepSize) + paddingLeft,
            contentHeight - (float) (scale * spendingPerDay[i]));
    }

    invalidate();
  }
```

It would actually be quite possible to encapsulate this code in an `ActionCommand` or `AsyncTask`, so that these calculations don't block the main thread. You will need to invoke the `invalidate()` method in `onForeground()`, or use the `postInvalidate()` method instead (which posts the `invalidate()` signal to the main thread). Moving such complexity to a background thread is good practice if the amount of data the graph is expected to present became very large.

16. Now, you're ready to override the `onDraw` method and actually paint the graph onto the `Canvas` provided by the platform. This `onDraw` implementation simply verifies that the graph has been rendered, and then paints the fields onto the screen:

```
@Override
protected void onDraw(final Canvas canvas) {
  if (path == null || paint == null) {
    return;
  }

  canvas.drawPath(path, paint);
}
```

It's useful to know that you can construct a `Canvas` yourself by having it paint to an offscreen `Bitmap` object, allowing you to capture *screenshots* of widgets by invoking their `onDraw` method yourself.

17. Now, you just need a few getter and setter methods to allow your application to specify the data to be rendered, and a programmatic way to change and fetch the XML attribute values. The setter methods also need to invoke the `invalidateGraph()` method to cause the data to be recalculated and rendered:

```
public void setSpendingPerDay(final double[] spendingPerDay) {
  this.spendingPerDay = spendingPerDay;
  invalidateGraph();
}

public int getStrokeColor() {
  return strokeColor;
}

public void setStrokeColor(final int strokeColor) {
  this.strokeColor = strokeColor;
  invalidateGraph();
}

public int getStrokeWidth() {
  return strokeWidth;
}

public void setStrokeWidth(final int strokeWidth) {
  this.strokeWidth = strokeWidth;
  invalidateGraph();
}
```

The `SpendingGraphView` requires that its actual data is delivered programmatically using the `setSpendingPerDay` method. This can, fortunately, be done easily using the data-binding system, which will also keep the data up to date when the data changes.

Integrating the SpendingGraphView

Integrating the `SpendingGraphView` into an application is as simple as declaring it in your layout XML file, and providing it with some data points to render:

```
<com.packtpub.claim.widget.SpendingGraphView
    android:id="@+id/spendingGraphView"
    android:layout_width="match_parent"
    android:layout_height="?attr/actionBarSize"
    app:strokeWidth="2dp"
    app:strokeColor="@color/colorAccent"
    app:spendingPerDay="@{dataSource.spendingPerDay}"/>
```

You can also programmatically find the `SpendingGraphView` using `findViewById`, and invoke the `setSpendingPerDay` method from your Java code. Integrating the `SpendingGraphView` into the travel claim example is a little more complex. The graph belongs on the overview screen, since it gives the user a quick visual indication of what their last few days of spending have looked like. If the user starts scrolling, the graph needs to scroll off the screen so that there is more screen space for the claim items. A nice way to do this is to leverage the `DisplayItem` class you wrote to handle the spacers, and simply add one at the beginning of the overview. Let's integrate the new `SpendingGraphView` with the overview screen:

1. Right-click on the **res/layout** directory and select **New| Layout resource file**.
2. Name the new resource file `card_spending_graph`.
3. Change the **Root element** to be `layout`.
4. Click **OK** to create the new resource file.
5. This layout resource will be used with the `DataBoundViewHolder`, and we'll be passing the spending-per-day indirectly as the `item` variable. It's also worth noting that the XML namespace for your custom attributes on the `SpendingGraphView` (`strokeColor` and `strokeWidth`) is the `app` namespace. This is what the layout resource should look like:

```
<?xml version="1.0" encoding="utf-8"?>
<layout>
    <data>
        <variable
            name="item"
            type="double[]" />
    </data>

    <android.support.v7.widget.CardView
        xmlns:android="http://schemas.android.com/apk/res/android"
        xmlns:app="http://schemas.android.com/apk/res-auto"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        app:contentPadding="@dimen/grid_spacer1">

        <com.packtpub.claim.widget.SpendingGraphView
            android:id="@+id/spendingGraphView"
            android:layout_width="match_parent"
            android:layout_height="@dimen/spending_graph_height"
            app:spendingPerDay="@{item}"
            app:strokeColor="@color/colorAccent"
            app:strokeWidth="2dp" />
    </android.support.v7.widget.CardView>
</layout>
```

6. Use the Code Assistant on the `layout_height` attribute to create a new dimension value named `spending_graph_height` (as was just highlighted) in your `dimens.xml` values resource file:

```
    <dimen name="app_bar_height">180dp</dimen>
    <dimen name="spending_graph_height">80dp</dimen>
</resources>
```

7. Open the `ClaimItemAdapter` source file.
8. Most of the changes will be in the `CreateDisplayListCommand` inner class. You'll need to calculate the user's spending for a selection of recent days. To do this, you'll need to know how many days ago each claim was, so that you can add its amount to the correct day. This method simply counts backward one day at a time, until it reaches the given timestamp:

```
int countDays(final Date timestamp) {
  final Calendar calendar = Calendar.getInstance();
  calendar.setTime(timestamp);

  final Calendar counterCalendar = Calendar.getInstance();
  counterCalendar.clear(Calendar.HOUR_OF_DAY);
  counterCalendar.clear(Calendar.MINUTE);
  counterCalendar.clear(Calendar.SECOND);
  counterCalendar.clear(Calendar.MILLISECOND);

  int days = 0;
  while (calendar.before(counterCalendar)) {
    days++;
    counterCalendar.add(Calendar.DAY_OF_YEAR, -1);
  }

  return days;
}
```

Time APIs such as JODA time (`http://www.joda.org/joda-time/`) and the Java 8 time APIs offer methods specifically for calculating the difference between two instants in time (in various different time units). However, the use of these APIs is beyond the scope of this book.

9. Next, you'll need another method in the `CreateDisplayListCommand` to create the array of amounts representing the user's spending over the last few days. To keep the implementation simple and quick, we limit this to ten days by default. The `getSpendingPerDay` method creates a double for each of these days, and adds the amounts to each double for the `ClaimItem` objects filed on each day:

```
double[] getSpendingPerDay(final List<ClaimItem> claimItems) {
  final double[] daysSpending = new double[10];
  final int lastItem = daysSpending.length - 1;
  Arrays.fill(daysSpending, 0);

  for (final ClaimItem item : claimItems) {
    final int distance = countDays(item.getTimestamp());

    // the ClaimItems are in timestamp order
    if (distance > lastItem) {
      break;
    }

    daysSpending[lastItem - distance] += item.getAmount();
  }

  return daysSpending;
}
```

10. The final thing to do in the `CreateDisplayListCommand`, is to create a `DisplayItem` as the first item in the list:

```
public List<DisplayItem> onBackground(
    final List<ClaimItem> claimItems)
    throws Exception {

  final List<DisplayItem> output = new ArrayList<>();
  output.add(new DisplayItem(
        R.layout.card_spending_graph,
        getSpendingPerDay(claimItems)));

  for (int i = 0; i < claimItems.size(); i++) {
```

11. You'll also need to add some new code to the `UpdateDisplayListCommand` inner class, because it doesn't know how to compare the spending graph for the `DiffUtil`. In the implementation of the `areItemsTheSame` method in the `DiffUtil.Callback`, you can treat the `card_spending_graph` layout resources exactly the same as the separators, because there is only one of them in the list:

```
@Override
public boolean areItemsTheSame(
    final int oldItemPosition,
    final int newItemPosition) {
  // ...

  switch (newItem.layout) {
    case R.layout.card_claim_item:
        final ClaimItem oldClaimItem = (ClaimItem) oldItem.value;
        final ClaimItem newClaimItem = (ClaimItem) newItem.value;
        return oldClaimItem != null
             && newClaimItem != null
             && oldClaimItem.id == newClaimItem.id;
    case R.layout.widget_divider:
    case R.layout.card_spending_graph:
        return true;
  }

  return false;
}
```

12. However, you also need the `DiffUtil` to detect that the graph data may have changed. In this case, we simply assume that the data has changed, and force the `RecyclerView` to bind the new data points to the existing `SpendingGraphView`:

```
@Override
public boolean areContentsTheSame(
    final int oldItemPosition,
    final int newItemPosition) {
  final DisplayItem oldItem = oldDisplay.get(oldItemPosition);
  final DisplayItem newItem = newDisplay.get(newItemPosition);

  switch (newItem.layout) {
    case R.layout.card_claim_item:
        final ClaimItem oldClaimItem = (ClaimItem) oldItem.value;
        final ClaimItem newClaimItem = (ClaimItem) newItem.value;
        return oldClaimItem != null
             && newClaimItem != null
             && oldClaimItem.equals(newClaimItem);
```

```
            case R.layout.widget_divider:
                return true;
            case R.layout.card_spending_graph:
                return false;
        }

        return false;
    }
```

13. The one last thing to ensure is that the user can't swipe the `SpendingGraphView` to delete it from the `RecyclerView`, as this will be a huge surprise for the user. Open the `OverviewActivity` source file and locate the `SwipeToDeleteCallback` inner class.
14. We need to tell the `ItemTouchHelper` that the first item in the list cannot be swiped or moved. We do this by overriding the default `getMovementFlags` method. This method usually just returns the flags that you passed into the constructor, but you now want these flags to change for just one item:

```
@Override
public int getMovementFlags(
    final RecyclerView recyclerView,
    final RecyclerView.ViewHolder viewHolder) {

  if (viewHolder.getAdapterPosition() == 0) {
    return 0;
  }

  return super.getMovementFlags(recyclerView, viewHolder);
}
```

Creating a layout implementation

In most applications, you'll find that a combination of the `ConstraintLayout`, `CoordinatorLayout`, and some of the more primitive layout classes (such as `LinearLayout` and `FrameLayout`)) are more than enough to achieve any layout requirements you can dream up for your user interface. Every now and again though, you'll find yourself needing a custom layout manager to achieve an effect required for the application.

Layout classes extend from the `ViewGroup` class, and their job is to tell their child widgets where to position themselves, and how large they should be. They do this in two phases: the measurement phase and the layout phase.

All `View` implementations are expected to provide measurements for their actual size according to specifications. These measurements are then used by the `View` widget's parent `ViewGroup` to allocate the amount of space the widget will consume on the screen. For example, a `View` might be told to consume, at most, the screen width. The `View` must then determine how much of that space it actually requires, and records that size in its **measured dimensions**. The measured dimensions are then used by the parent `ViewGroup` during the layout process.

The second phase is the layout phase, and it is conducted by the `ViewGroup` parent of each `View` widget. This phase positions the `View` on the screen, relative to its parent `ViewGroup` location, and specifies the actual size that the widget will consume on the screen (typically based on the measured size calculated in the measurement phase).

When you implement your own `ViewGroup`, you'll need to ensure that all of your child `View` widgets are given a chance to measure themselves before you perform the actual layout operations.

Let's build a layout class to arrange its children in a circle. To keep the implementation simple, we'll assume that all the child widgets are the same size (for example, if they were all icons):

1. Right-click on the `widget` package in the travel claim example app, and select **New | Java Class**.
2. Name the new class `CircleLayout`.
3. Change the **Superclass** to `android.view.ViewGroup`.
4. Click **OK** to create the new class.
5. Declare the standard `ViewGroup` constructors:

```
public CircleLayout(final Context context) {
  super(context);
}

public CircleLayout(
    final Context context,
    final AttributeSet attrs) {
  super(context, attrs);
}

public CircleLayout(
```

```
        final Context context,
        final AttributeSet attrs,
        final int defStyleAttr) {

    super(context, attrs, defStyleAttr);
  }
```

6. Override the `onMeasure` method to calculate the size of the `CircleLayout` and all of its child `View` widgets. The measurement specifications are passed in as `int` values, which are interpreted using the `static` methods in the `MeaureSpec` class. Measurement specifications come in two flavors: *at most* and *exactly*, and each has a *size* value attached. In this particular layout, we always measure the `CircleLayout` as the size given in the specification. This means that the `CircleLayout` will always consume the maximum amount of space available. It also expects all of its children to be able to specify sizes without the `match_parent` attribute (as this will cause each child to take up all the available space):

```
@Override
protected void onMeasure(
    final int widthMeasureSpec,
    final int heightMeasureSpec) {
  super.onMeasure(widthMeasureSpec, heightMeasureSpec);
  measureChildren(widthMeasureSpec, heightMeasureSpec);

  setMeasuredDimension(
        MeasureSpec.getSize(widthMeasureSpec),
        MeasureSpec.getSize(heightMeasureSpec));
}
```

7. The next method to implement is the `onLayout` method. This performs the actual arrangement of the child `View` widget within the `CircleLayout`, by invoking their `layout` method. The `layout` method should never be overridden, because it's closely tied to the platform and performs several other important actions (such as notifying layout listeners). Instead, you should override `onLayout`, but invoking `layout.CircleLayout` assumes that all the child `View` widgets are of the same size (and forces this as part of the `onLayout` implementation). This `onLayout` method simply calculates the available space, and then positions the child `View` widgets in a circle around the outside edge:

```
protected void onLayout(
    final boolean changed,
    final int left,
    final int top,
```

```
      final int right,
      final int bottom) {

    final int childCount = getChildCount();

    if (childCount == 0) {
      return;
    }

    final int width = right - left;
    final int height = bottom - top;

    // if we have children, we assume they're all the same size
    final int childrenWidth = getChildAt(0).getMeasuredWidth();
    final int childrenHeight = getChildAt(0).getMeasuredHeight();

    final int boxSize = Math.min(
        width - childrenWidth,
        height - childrenHeight);

    for (int i = 0; i < childCount; i++) {
      final View child = getChildAt(i);
      final int childWidth = child.getMeasuredWidth();
      final int childHeight = child.getMeasuredHeight();

      final double x = Math.sin((Math.PI * 2.0)
            * ((double) i / (double) childCount));
      final double y = -Math.cos((Math.PI * 2.0)
            * ((double) i / (double) childCount));

      final int childLeft = (int) (x * (boxSize / 2))
            + (width / 2) - (childWidth / 2);
      final int childTop = (int) (y * (boxSize / 2))
            + (height / 2) - (childHeight / 2);
      final int childRight = childLeft + childWidth;
      final int childBottom = childTop + childHeight;

      child.layout(childLeft, childTop, childRight, childBottom);
    }
  }
```

Although the implementation of the `onLayout` method is quite long, it's also relatively simple. Most of the code is concerned with determining the desired position of the child `View` widgets. Layout code needs to execute as quickly as possible, and should avoid allocating any objects during the `onMeasure` and `onLayout` methods (similar to the rules of `onDraw`). Layout is a critical part of building the screen from a performance standpoint, because no rendering can actually occur without the layout being completed. The layout will also be rerun every time the layout changes its structure. For example, if you add or remove any child `View` widgets, or change the size or position of the `ViewGroup`. Changing the size of a `ViewGroup` might happen on every frame if you use a `CoordinatorLayout`, where the `ViewGroup` is being collapsed (or if you change its size as part of a property-animation).

Creating animated views

Most widget animation can be taken care of using the animation APIs in Android. The standard animation APIs are designed to take care of animations with a defined start and end, or animations that form a simple loop. Some animations, however, don't fit into this mold; a good example would be a game. A game has many animations running continuously, and you can even think about the entire game screen as a single, continuous animation.

There are a number of widgets that need to be continuously animated, and your standard Android animation API won't work. In these cases, you'll need a `View` that can continuously animate and update itself as long as it's visible to the user. In these cases, a slightly different design is called for, as the widget will always be changing.

To illustrate how to write a widget that has a continuous animation, let's write a `View` class that animates some number of bouncing `Drawable` objects. Each `Drawable` will be tracked separately, and when it reaches a side, it will "bounce off", and head in the other direction. This class is unrelated to the travel claim example code, so you can add it to a new project if you like. Follow these steps to write a `BouncingDrawablesView`:

1. On your default package, select **New| Java Class**.
2. Name the class `widget.BouncingDrawablesView`.
3. Make the **Superclass** `android.view.View`.
4. Click on **OK** to create the new class.

5. You'll have some number of bouncing objects in the scene, and you'll need to track both their position and speed vector. For this, you'll want to encapsulate each bouncing `Drawable` in a `Bouncer` object; we'll write this as an inner class:

```
public static class Bouncer {
  final Drawable drawable;
  final Rect bounds;
  int speedX;
  int speedY;

  public Bouncer(
        final Drawable drawable,
        final int speedX,
        final int speedY) {

    this.drawable = drawable;
    this.bounds = drawable.copyBounds();
    this.speedX = speedX;
    this.speedY = speedY;
  }
```

6. The next thing to do in the `Bouncer` inner class is to create a single `step` method, which will set up the `Bouncer` for the next animation frame to be rendered. This method will take a parameter that represents the boundaries of the *field* it's being rendered on. If the next position collides with any of the edges of the field, the `Bouncer` will avoid crossing the edge and will reverse direction on the axis it would have collided with:

```
void step(final Rect boundary) {
  final int width = bounds.width();
  final int height = bounds.height();

  int nextLeft = bounds.left + speedX;
  int nextTop = bounds.top + speedY;

  if (nextLeft + width >= boundary.right) {
    speedX = -speedX;
    nextLeft = boundary.right - width;
  } else if (nextLeft < boundary.left) {
    speedX = -speedX;
    nextLeft = boundary.left;
  }

  if (nextTop + height >= boundary.bottom) {
    speedY = -speedY;
    nextTop = boundary.bottom - height;
```

```
    } else if (nextTop < boundary.top) {
      speedY = -speedY;
      nextTop = boundary.top;
    }

    bounds.set(
         nextLeft,
         nextTop,
         nextLeft + width,
         nextTop + height
    );
  }
```

7. The `Bouncer` class also needs a convenient draw method that will update the boundaries of the `Drawable`, before rendering it to a given `Canvas` object. The `Bouncer` keeps track of its own boundaries, so that all the `Bouncer` instances can actually share the same `Drawable` instance, and simply paint it at different locations on the field:

```
    void draw(final Canvas canvas) {
      drawable.setBounds(bounds);
      drawable.draw(canvas);
    }
  } // end of Bouncer inner class
```

8. Now, in the `BouncingDrawablesView`, declare an array of `Bouncer` objects that will be contained and animated by the `View` implementation:

```
private Bouncer[] bouncers = null;
```

9. The `BouncingDrawableView` also needs a status field to track whether it should be animating or not:

```
private boolean running = false;
```

10. Next, declare the standard `View` implementation constructors:

```
public BouncingDrawablesView(
    final Context context) {
  super(context);
}

public BouncingDrawablesView(
    final Context context,
    final AttributeSet attrs) {
  super(context, attrs);
```

```
}

public BouncingDrawablesView(
    final Context context,
    final AttributeSet attrs,
    final int defStyleAttr) {
  super(context, attrs, defStyleAttr);
}
```

11. Implement the `onDraw` method by simply telling each of the `Bouncer` objects to paint themselves:

```
@Override
protected void onDraw(final Canvas canvas) {
  super.onDraw(canvas);

  if (bouncers == null) {
    return;
  }

  for (final Bouncer bouncer : bouncers) {
    bouncer.draw(canvas);
  }
}
```

12. Next, you'll need to implement that actual logic to animate each frame. Do this by creating an `onNextFrame` method that first checks whether the animation should still be running (if it's not running, we stop the animation), and then tells each `Bouncer` to move one step in the animation. After you've set up the next animation frame, you'll need to tell the platform to repaint the `BouncingDrawablesView`, by calling the `invalidate()` method. Once the `onNextFrame()` method is complete, we schedule it to be invoked again in 16 milliseconds time (scheduling just under 60 frames per second):

```
private final Runnable postNextFrame = new Runnable() {
  @Override
  public void run() {
    onNextFrame();
  }
};


void onNextFrame() {
  if (bouncers == null || !running) {
    return;
  }
```

```
    final Rect boundary = new Rect(
         getPaddingLeft(),
         getPaddingTop(),
         getWidth() - getPaddingLeft() - getPaddingRight(),
         getHeight() - getPaddingTop() - getPaddingBottom()
    );

    for (final Bouncer bouncer : bouncers) {
      bouncer.step(boundary);
    }

    invalidate();
    getHandler().postDelayed(postNextFrame, 16);
  }
```

13. In order to automatically start the animation when the `BouncingDrawablesView` becomes visible and make it stop when it's invisible, you need to know when the `BouncingDrawablesView` is attached to the `Window` (when it's attached to the screen components). To do this, you'll need to override `onAttachedToWindow` and invoke `onNextFrame()`. However, `onAttachedToWindow` is invoked before the layout is executed, so you'll schedule `onNextFrame()` to be run at the end of the current event queue:

```
@Override
protected void onAttachedToWindow() {
  super.onAttachedToWindow();
  running = true;

  post(postNextFrame);
}

@Override
protected void onDetachedFromWindow() {
  super.onDetachedFromWindow();
  running = false;
}
```

14. Finally, write a setter and getter for the `Bouncer` objects:

```
public void setBouncers(final Bouncer[] bouncers) {
  this.bouncers = bouncers;
}

public Bouncer[] getBouncers() {
  return bouncers;
}
```

Setting up the `BouncingDrawablesView` is a very simple process. An `Activity` will need to create an array of `Bouncer` objects with some random positions and speeds, and then hand them over to the `BouncingDrawablesView` instance to take care of them. As soon as the `BouncingDrawablesView` becomes visible on the screen, it will start animating the `Drawable` objects around the screen. A simple example setup of the `BouncingDrawableView` might look something like this:

```
final BouncingDrawablesView bouncingDrawablesView =
(BouncingDrawablesView) findViewById(R.id.bouncing_view);
final BouncingDrawablesView.Bouncer[] bouncers = new
BouncingDrawablesView.Bouncer[10];
final Random random = new Random();
final Resources res = getResources();

final Drawable icon = res.getDrawable(R.drawable.ic_other_black);
final int iconSize =
res.getDimensionPixelSize(R.dimen.bouncing_icon_size);
for (int i = 0; i < bouncers.length; i++) {
  final Rect bounds = new Rect();
  bounds.top = random.nextInt(400);
  bounds.left = random.nextInt(600);
  bounds.right = bounds.left + iconSize;
  bounds.bottom = bounds.top + iconSize;
  icon.setBounds(bounds);

  bouncers[i] = new BouncingDrawablesView.Bouncer(
        icon,
        random.nextBoolean() ? 6 : -6,
        random.nextBoolean() ? 6 : -6
  );
}

bouncingDrawablesView.setBouncers(bouncers);
```

Test your knowledge

1. When rendering specialized graphics for a custom widget, you need to do which of these?
    - Buffer all the rendering in an offscreen `Bitmap`
    - Set a custom background `Drawable`
    - Override the `onDraw` method

2. Where should you create instances of graphics primitives such as `Drawable`, `Paint` and `Path` for rendering in `onDraw`?
    - On the main thread
    - In the `onDraw` method
    - Anywhere that doesn't affect `onDraw` directly
3. What are the two phases involved in the layout process?
    - Layout and then measurement
    - Measurement and then layout
    - Measurement and then rendering
4. When painting a `Drawable` object, you need to do which of the following?
    - Pass it a valid `Canvas` object
    - Use `Canvas.paintDrawable`
    - Invoke its `onDraw` method
5. To tell the platform that a widget needs to repaint itself (from the main thread), you use which of these?
    - `View.redraw()`
    - `View.invalidate()`
    - `View.repaint()`

Apply your knowledge

Most of what has been covered in this book is a mix of theory (how to go about designing a screen) and hard practical knowledge (writing the code to produce that screen). When you combine a good theoretical base with practical knowledge of the platform you're working on, you have a powerful combination. Being about to write great applications isn't just about being able to write code (very little in programming is about being able to *just* write code). It's about having an eye for detail in the user interface, and always thinking about your users.

Android is an amazingly powerful platform when used correctly. In this book, you've learned to use APIs such as data binding, the Room data storage system, and `LiveData`. This mix of APIs on the Android Platform doesn't just allow you to rapidly develop excellent applications, but it also provides an excellent separation between different areas of your code base. They also don't, in any way, reduce the power you can leverage from the underlying platforms and systems (such as SQLite).

The Android community is massive, and there is plenty to find and work with outside of the core platform that can make development even easier. Here are a few links to resources, documentation, and APIs, that are especially useful:

- The official Android platform reference:
  `https://developer.android.com/reference/packages.html`
- Firebase (handles hosting, push-notifications, database synchronization, authentication, and much more):
  `https://firebase.google.com/`
- The Android Arsenal, an unofficial list of third-party APIs and widgets:
  `https://android-arsenal.com/`
- Joda-Time API, the de-facto standard time API before Java 8 on the core Java platform, still useful on Android, though:
  `http://www.joda.org/joda-time/`
- The official SQLite website:
  `https://sqlite.org/`

Finally, here are some fun ideas for projects that you might want to try implementing once you've finished this book:

- Try expanding the travel claim example to allow for multiple trips.
- Write a simple expense tracker to allow your user to enter and track their spending.
- A packing/moving organizer app, allowing the user to photograph the contents of boxes and record their contents for when they are moving their house.
- A to-do list application, allowing the user to create various lists of things they need to do and check them off when done. To make this more interesting, you can add reminders and deadlines (items that must be completed by a certain date and time).
- A real-time chat application, which is a bit more complex; use the Firebase Real-time database to store and synchronize the chat messages.

Summary

Building your own custom components can be a lot of work, but can also be extremely rewarding. Having complete control over the measurement, layout, and rendering cycle provides you with an amazing amount of power to virtually build any widget that you can imagine. Android also has some excellent defaults defined, allowing you to focus on how your widget should look and work, rather than getting stuck on the intricacies of the rendering pipeline.

The `Drawable` class is one of the most powerful graphics primitives Android has. It's difficult to call it a primitive due to how powerful they actually are. Wherever possible, use them instead of a `Bitmap` or `Path`, as they make future improvements much simpler, and easily integrate with the resources system.

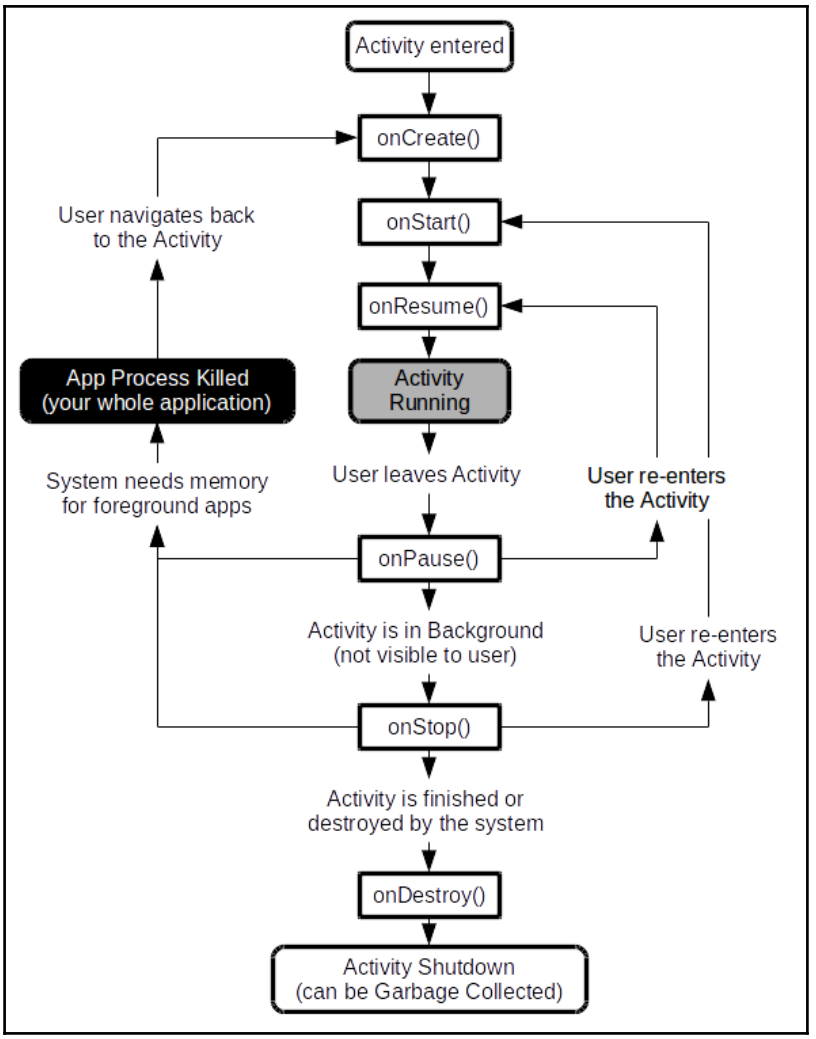
Using the `Handler` class to animate a widget is also a very powerful and low-level mechanism. It's often a good idea to introduce a sense of real time into these sorts of animations so that frames that take slightly longer, or shorter, to render don't affect the overall feel of the application. This can be done simply by using the timestamp in each frame and moving values according to that, instead of having fixed values. In this case, the speed of the `Bouncer` will become the number of `pixels/time` instead of a fixed number of pixels per frame.

Before building your own widgets or layouts, you should always look around on the internet to see whether there is an existing project that does what you are looking for. Knowing how widgets are actually built and fit together is useful knowledge, and should give you the confidence to not just create your own, but also help others build theirs.

A
Activity Lifecycle

Every `Activity` within an app runs within the App Process, but each `Activity` is also subject to its own lifecycle. These are methods that are triggered by the platform when the `Activity` is about to change states, for example, when the user exits the `Activity` temporarily to use another `Activity` (whether it's in the same application, or another one). `Fragment` objects are also subject to a lifecycle, and while it mostly follows the same pattern as the `Activity` lifecycle, it also has the notion of being "attached" and "detached" from its parent `Activity`.

The following workflow explains the `Activity` lifecycle in detail:



B

Test Your Knowledge Answers

This appendix contains answers to all the Test Your Knowledge quizzes that appear in the chapters.

Chapter 2 - Designing Form Screens

1. When designing a form screen, what is the first thing you should consider?
    - The data you need from your user - *this will determine what fields are required and will inform your other decisions*
2. What is the standard spacing increment in Material design?
    - 8 Density Independent Pixels
3. The `ConstraintLayout`, `ViewPager`, and `CardView` are part of the support APIs. What does this mean?
    - Their bytecode must be included with your application if you use them
4. When building a new layout, your root widget should always be which of these?
    - The simplest widget that makes sense for your layout - *other options may provide more power, but will consume more system resources*

Chapter 3 - Taking Actions

1. What's the best way to implement event handlers?
    - There isn't one - *you need to consider each case, and choose the most appropriate pattern*
2. What are the conditions for any methods that changes the state of a user-interface widget?
    - They must be called from the main thread
3. Code running as part of an event handler should fulfill which of the following conditions?
    - Only interact with the user interface - *anything else should be done on a worker thread*
4. When requesting data from another `Activity`, the data is returned through which of these?
    - A callback on your `Activity` object

Chapter 4 - Composing User Interface

1. When developing a layout subclass, which of the following options is the best?
    - Avoiding assigning ID attributes to child widgets - *the ID attributes can cause unexpected side-effects in the application*
2. Which of these applies to the `Bundle` passed at an `Activity` in `onCreate`?
    - It is populated in the `onSaveInstanceState` method
3. When the data for an `Adapter` changes, which of the mentioned happens?
    - It should notify any attached listeners
4. Fragments and `View` classes should meet which of the following conditions?
    - They should have their data and state pushed into them from the `Activity`

Chapter 5 - Binding Data to Widgets

1. Android's data binding framework follows what sort of binding?
    - Model-View (unidirectional) binding
2. Data Bound Layouts have variables that must be which of the following?
    - Any Java Object
3. Which of the following features belongs to data binding expressions?
    - They are a special expression language
4. To trigger an update of a data-bound user interface, you must do which of these?
    - Make a change that the `Binding` object can observe

Chapter 6 - Storing and Retrieving Data

1. The Room API for Android provides which of the following?
    - A lightweight API on top of SQLite
2. Returning `LiveData` from a Room DAO requires that you do which of these?
    - You observe it for changes in order to retrieve data
3. Database queries that don't return `LiveData` should do what?
    - Be run on a worker thread
4. Writing an update method for Room requires which of the listed?
    - An `@Update` method taking an `Entity` object on an interface

Chapter 7 - Creating Overview Screens

1. An instance of `RecyclerView` will create one `View` instance for which of these?
    - Every item of data visible on the screen
2. When attaching an observer to `LiveData`, you need to do which of the following?
    - Provide a valid `LifecycleOwner`

3. Overview/Dashboard screens should have which of these features?
    - They should display an overview with the most important information first
4. The `ViewHolder` class is used by the `RecyclerView` to do what?
    - Improve the data binding performance
5. When using `LiveData` objects to reference data used by multiple `Fragment` objects, which of these is true?
    - The `Fragment` instances must share the same `LiveData` reference to see changes

Chapter 8 - Designing Material Layouts

1. Elevation should be used for which of the following?
    - To selectively highlight one item above a flat layout
2. `CoordinatorLayout` can be used to coordinate movement and size between which of these?
    - Any of its direct child widgets
3. To change elevation of a widget in a backward-compatible way, you need to do which of the mentioned?
    - Use the `ViewCompat` class
4. The `GridLayout` class should be used in which of the following conditions?
    - To lay out screens along grid lines

Chapter 9 - Navigating Effectively

1. When using bottom tabs for navigation, which of these is important?
    - The tabs are of roughly equal importance
2. Top tabs are preferred to bottom tabs in which of these situations?
    - When the user won't need to navigate as frequently
3. Fragments can be used for navigation in which of these cases?
    - Any time the user navigates within the application
4. When the user selects an item in a navigation drawer, which of these is true?
    - The drawer should be closed programmatically

Chapter 10 - Making Overviews Even Better

1. How many different view types can you use in a single `RecyclerView` instance?
    - Any number
2. When using a `DiffUtil`, which of the following applies to the data you are comparing?
    - It is exposed through a Callback
3. When adding dividers to a `RecyclerView`, you should do which of these?
    - Make them distinct items in the dataset

Chapter 11 - Polishing Your Design

1. When choosing a color scheme, it's important that the accent color has which of these features?
    - It is complementary to the primary color
2. Dynamically generating a palette should meet which of these conditions?
    - It should be done on a background thread
3. Which of these is to be kept in mind while animating layouts in your application?
    - They should not block or distract the user from achieving their goals
4. Custom styles can be used to define which of these?
    - Default values for any attribute in a layout resource file

Chapter 12 - Customizing Widgets and Layouts

1. When rendering specialized graphics for a custom widget, you need to do which of these?
    - Override the `onDraw` method
2. Where should you create instances of graphics primitives such as `Drawable`, `Paint`, and `Path` for rendering in `onDraw`?
    - Anywhere that doesn't affect `onDraw` directly

3. What are the two phases involved in the layout process?
    - Measurement and then layout
4. When painting a `Drawable` object, you need to do which of the following?
    - Pass it a valid `Canvas` object
5. To tell the platform that a widget needs to repaint itself (from the main thread), you use which of these?
    - `View.invalidate()`

Index

D

E

F

G

H

I

J

L

M

N

O

P

R

S

T

U

V

W