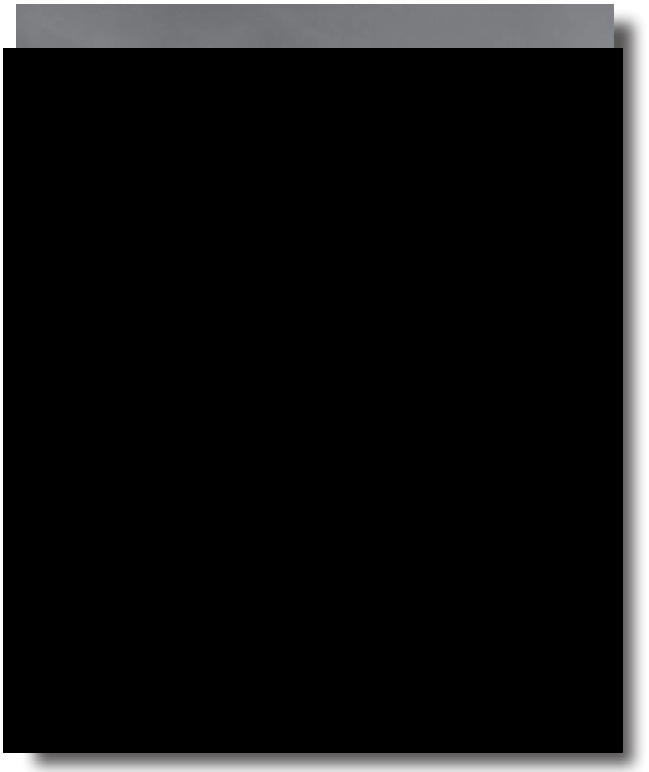
1 getting started

Diving In



Android has taken the world by storm.

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

Welcome to Androidville

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

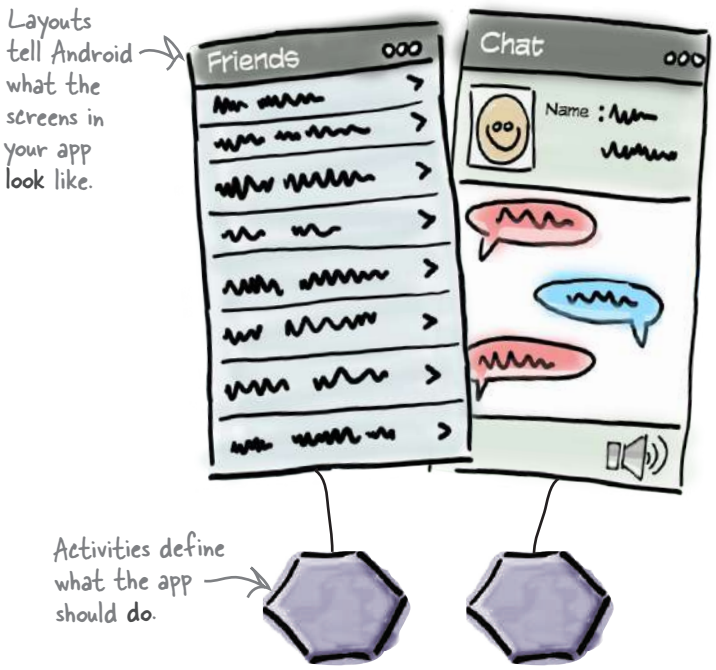
Android is a comprehensive open source platform based on Linux and championed by Google. It's a powerful development framework that includes everything you need to build great apps using a mix of Java and XML. What's more, it enables you to deploy those apps to a wide variety of devices—phones, tablets,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

So what makes up a typical Android app?

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

Layouts define what each screen looks like

A typical Android app is composed of one or more screens. You define what each screen looks like using a **layout** to define its appearance. Layouts are usually defined in XML, and can include GUI components such as buttons, text fields, and labels.



Activities define what the app does

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

Sometimes extra resources are needed too

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

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

The Android platform dissected

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

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

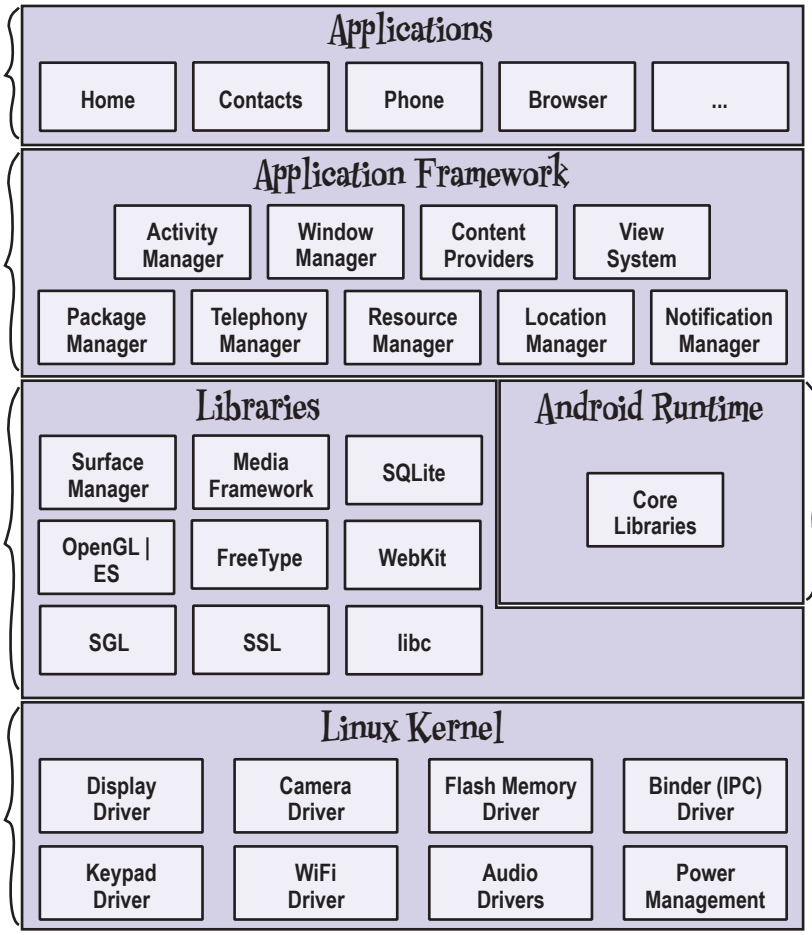
We're just giving you an overview of what's included in the Android platform. We'll explain the different components in more detail as and when we need to.

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

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

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

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



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

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

Here's what we're going to do

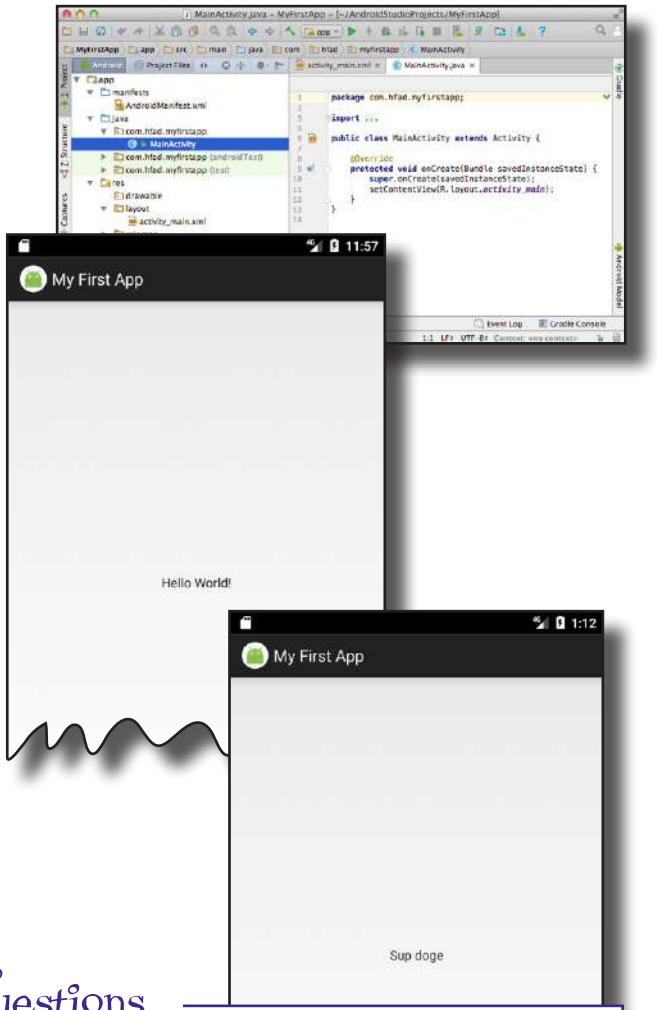
So let's dive in and create a basic Android app. There are just a few things we need to do:

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

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

3. **Run the app in the Android emulator.**
We'll use the built-in emulator to see the app up and running.

4. **Change the app.**
Finally, we'll make a few tweaks to the app we created in step 2, and run it again.



there are no Dumb Questions

Q: Are all Android apps developed in Java?

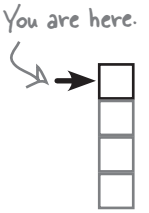
A: You can develop Android apps in other languages, too. Most developers use Java, so that's what we're covering in this book.

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

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

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

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



Your development environment

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

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

The Android SDK

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

SDK Platform
There's one of these for each version of Android.

SDK Tools
Tools for debugging and testing, plus other useful utilities. The SDK also features a set of platform dependent tools.

Sample apps
If you want practical code examples to help you understand how to use some of the APIs, the sample apps might help you.

Documentation
So you can access the latest API documentation offline.

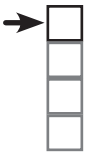
Android support
Extra APIs that aren't available in the standard platform.

Google Play Billing
Allows you to integrate billing services in your app.

Android Studio is a special version of IntelliJ IDEA

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

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



Set up environment
Build app
Run app
Change app

Install Android Studio

Before we go any further, you need to install Android Studio on your machine. We're not including the installation instructions in this book as they can get out of date pretty quickly, but you'll be fine if you follow the online instructions.

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

First, check the Android Studio system requirements here:

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

Then follow the Android Studio installation instructions here:

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

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

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

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

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

there are no Dumb Questions

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

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

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

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

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

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

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

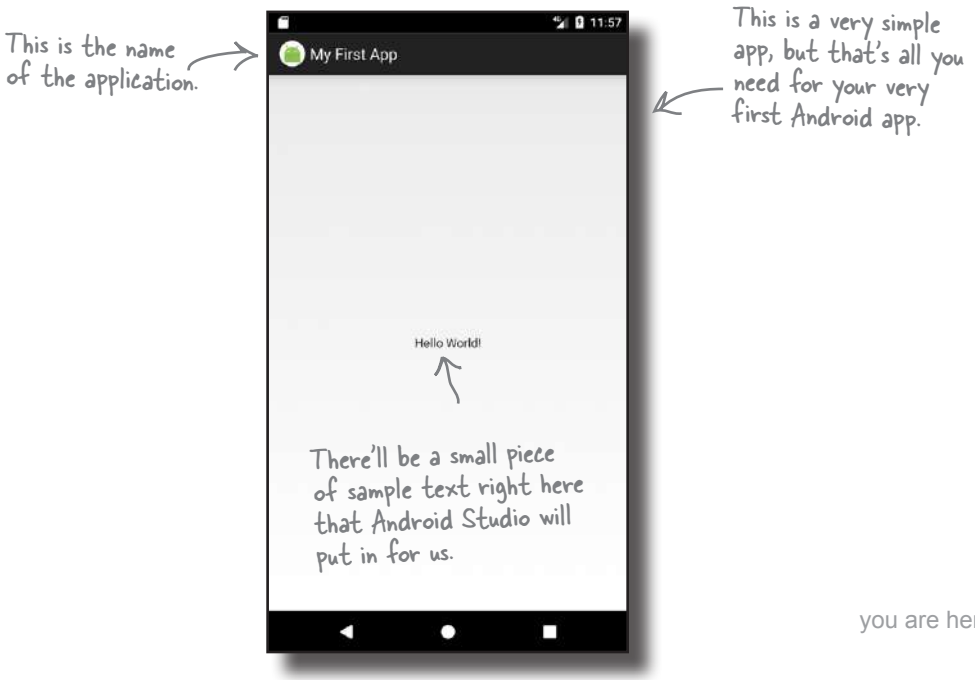
Q: Most apps are built using Gradle? I thought you said most developers use Android Studio.

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

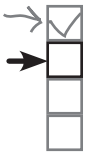
You can find out more about Gradle in Appendix II.

Build a basic app

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



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



Set up environment
Build app
Run app
Change app

How to build the app

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

1. Create a new project

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

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

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

How to build the app (continued)

2. Configure the project

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

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

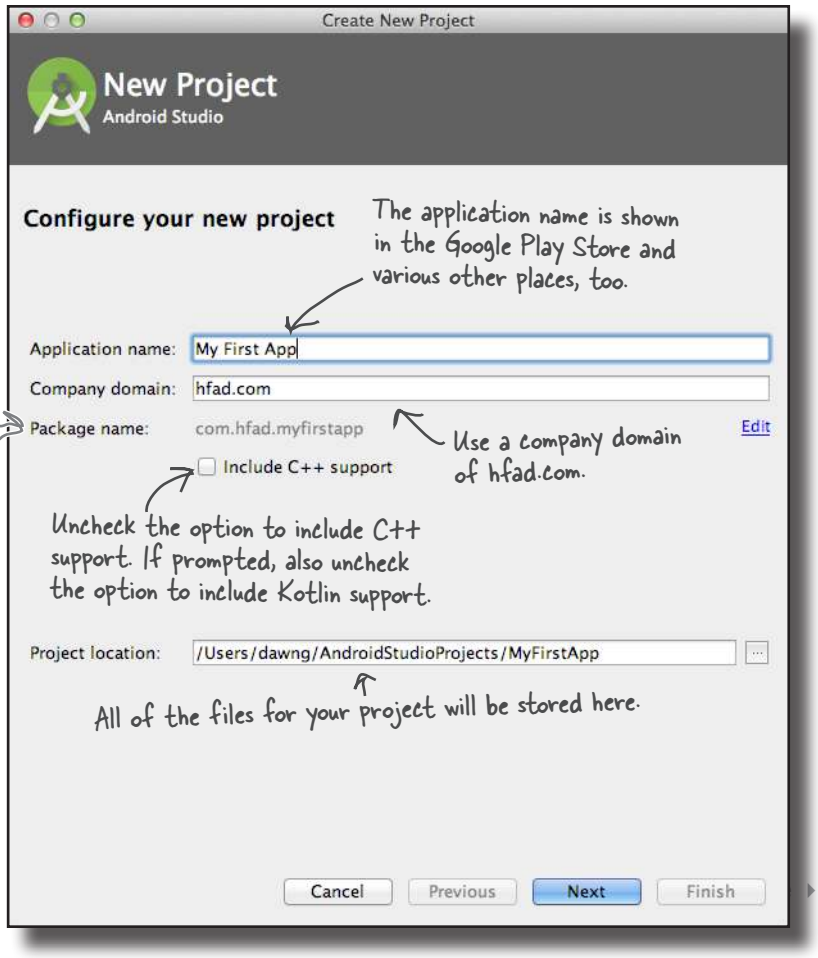
Enter an application name of "My First App", enter a company domain of "hfad.com", uncheck the option to include C++ support, and accept the default project location. Then click on the Next button.

Watch it!

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

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

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



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

How to build the app (continued)

Set up environment
Build app
Run app
Change app

3. Specify the minimum SDK

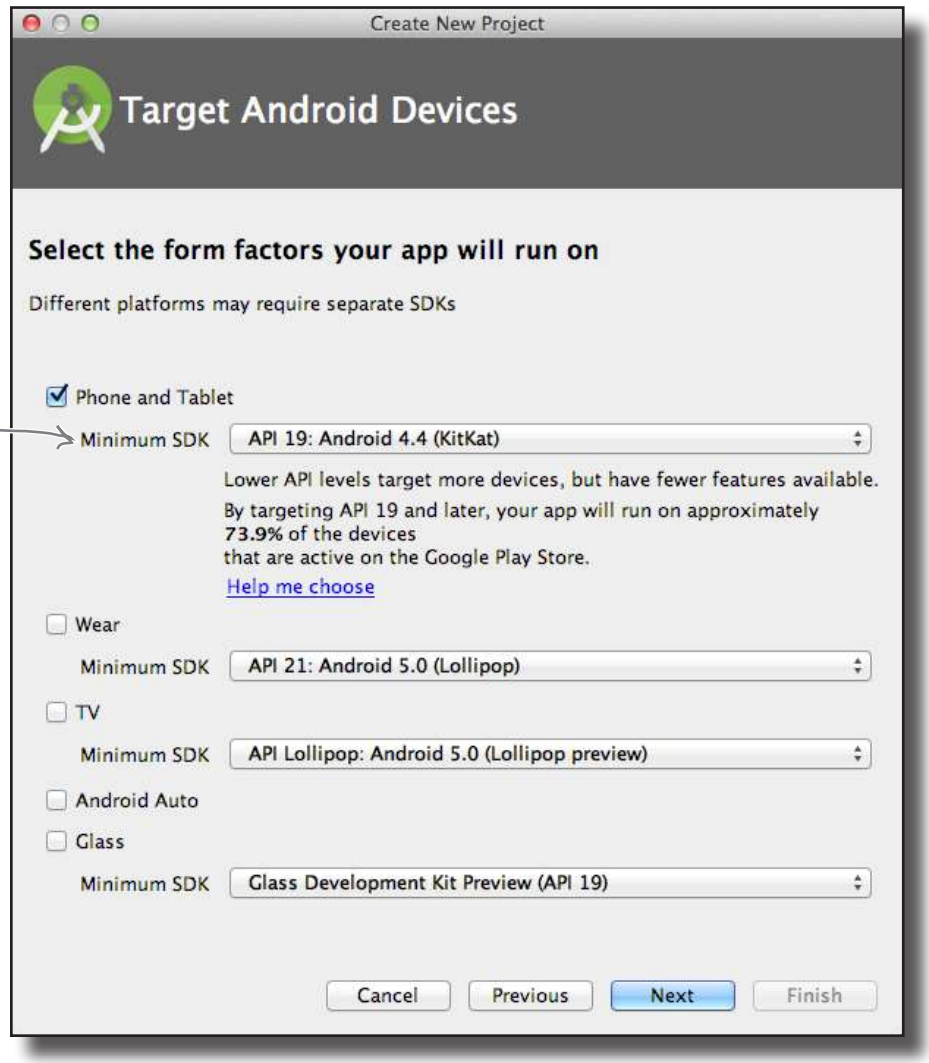
You now need to indicate the minimum SDK of Android your app will use. API levels increase with every new version of Android. Unless you only want your app to run on the very newest devices, you'll probably want to specify one of the older APIs.

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

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

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

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



Android Versions Up Close

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

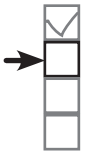
Android versions have a version number and a codename. The version number gives the precise version of Android (e.g., 7.0), while the codename is a more generic "friendly" name that may cover several versions of Android (e.g., Nougat). The API level refers to the version of the APIs used by applications.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

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

Hardly anyone uses these versions anymore.

Most devices use one of these APIs.

When you develop Android apps, you really need to consider which versions of Android you want your app to be compatible with. If you specify that your app is only compatible with the very latest version of the SDK, you might find that it can't be run on many devices. You can find out the percentage of devices running particular versions here: *https://developer.android.com/about/dashboards/index.html.*



Set up environment
Build app
Run app
Change app

Activities and layouts from 50,000 feet

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

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

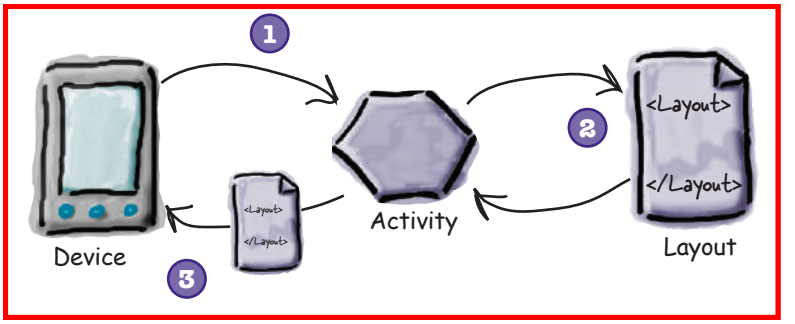
A **layout** describes **the appearance of the screen.** Layouts are written as XML files and they tell Android how the different screen elements are arranged.

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

Layouts define how the user interface is presented.

Activities define actions.

1. The device launches your app and creates an activity object.
2. The activity object specifies a layout.
3. The activity tells Android to display the layout onscreen.
4. The user interacts with the layout that's displayed on the device.
5. The activity responds to these interactions by running application code.
6. The activity updates the display...
7. ...which the user sees on the device.

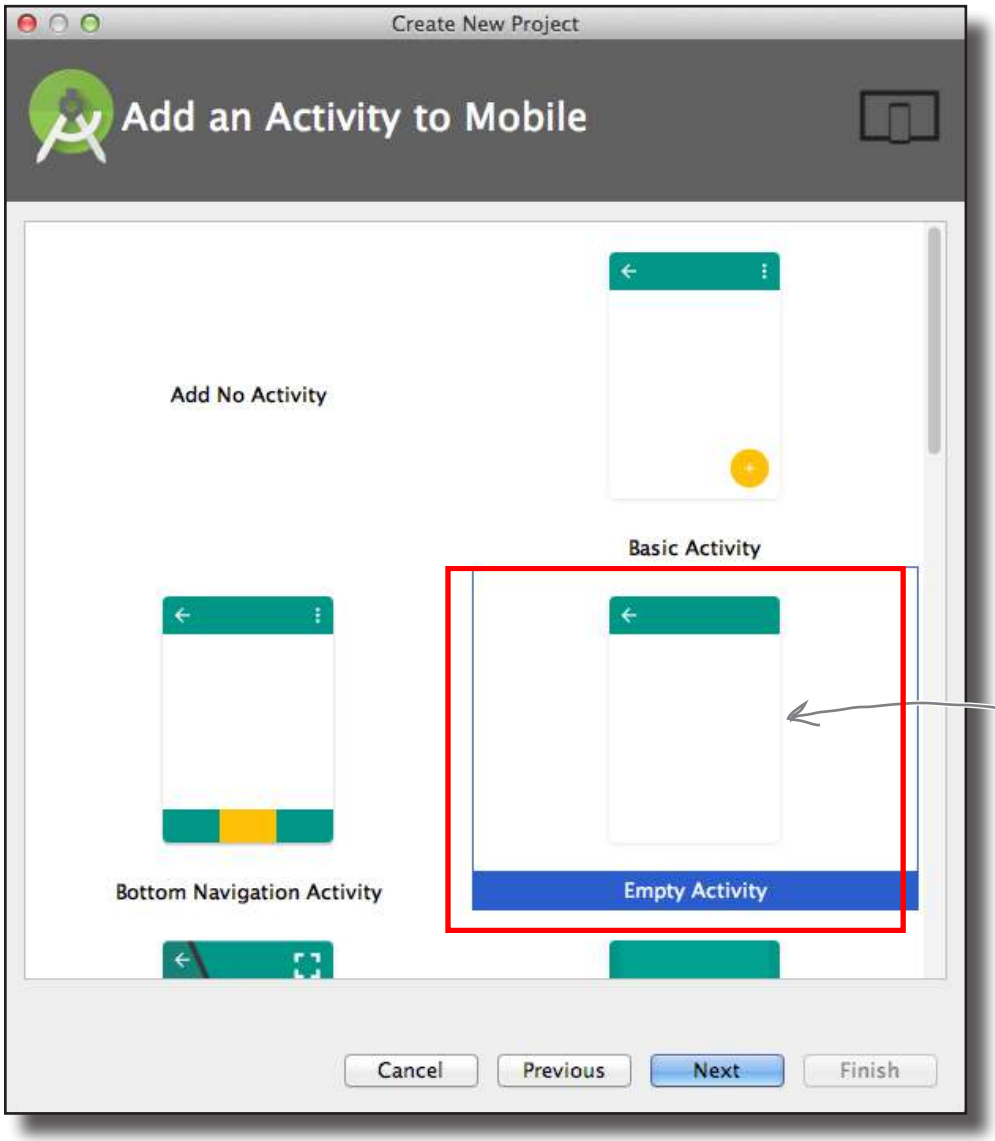


Now that you know a bit more about what activities and layouts are, let's go through the last couple of steps in the Create New Project wizard and get it to create an activity and layout.

How to build the app (continued)

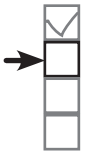
4. Add an activity

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



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

How to build the app (continued)

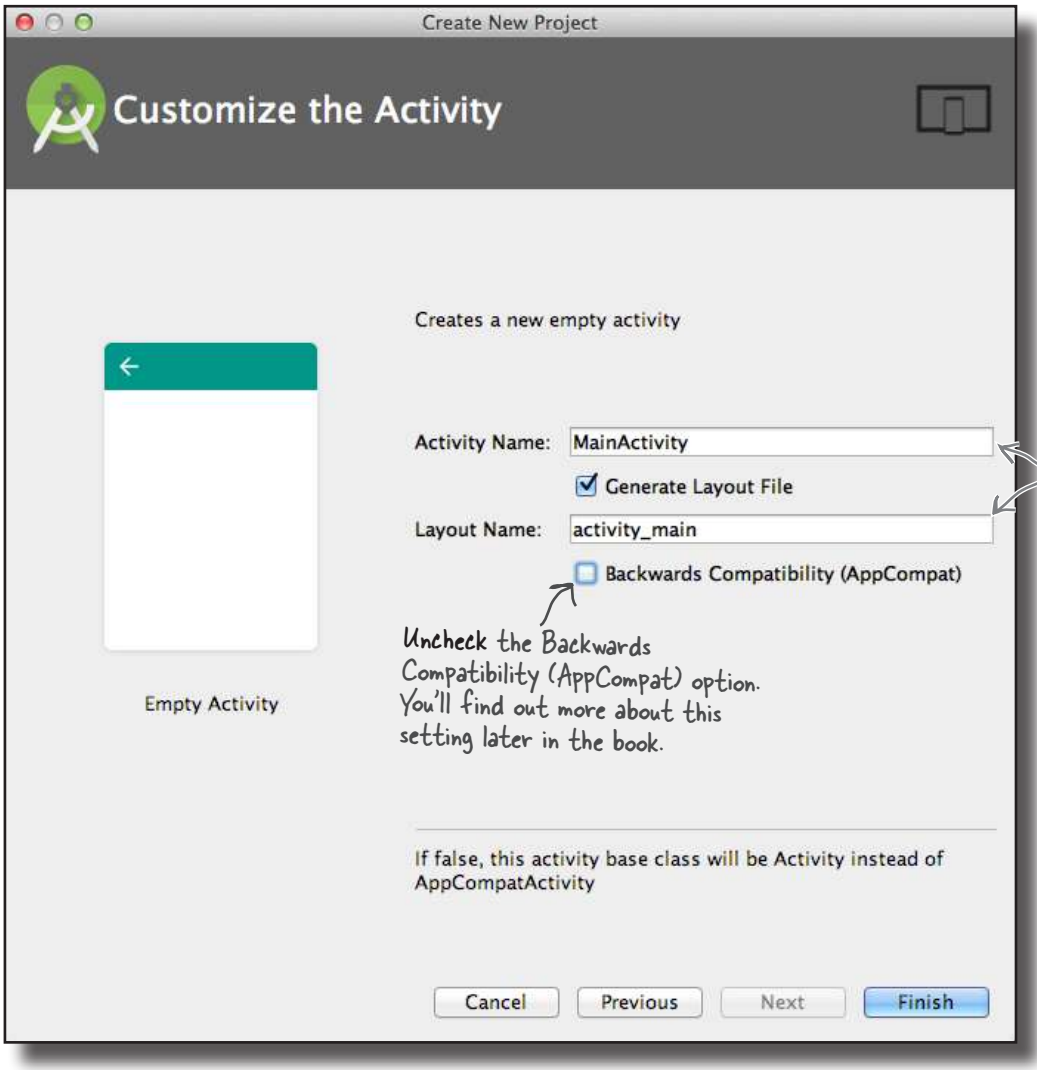


Set up environment
Build app
Run app
Change app

5. Customize the activity

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

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



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

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

You've just created your first Android app

So what just happened?

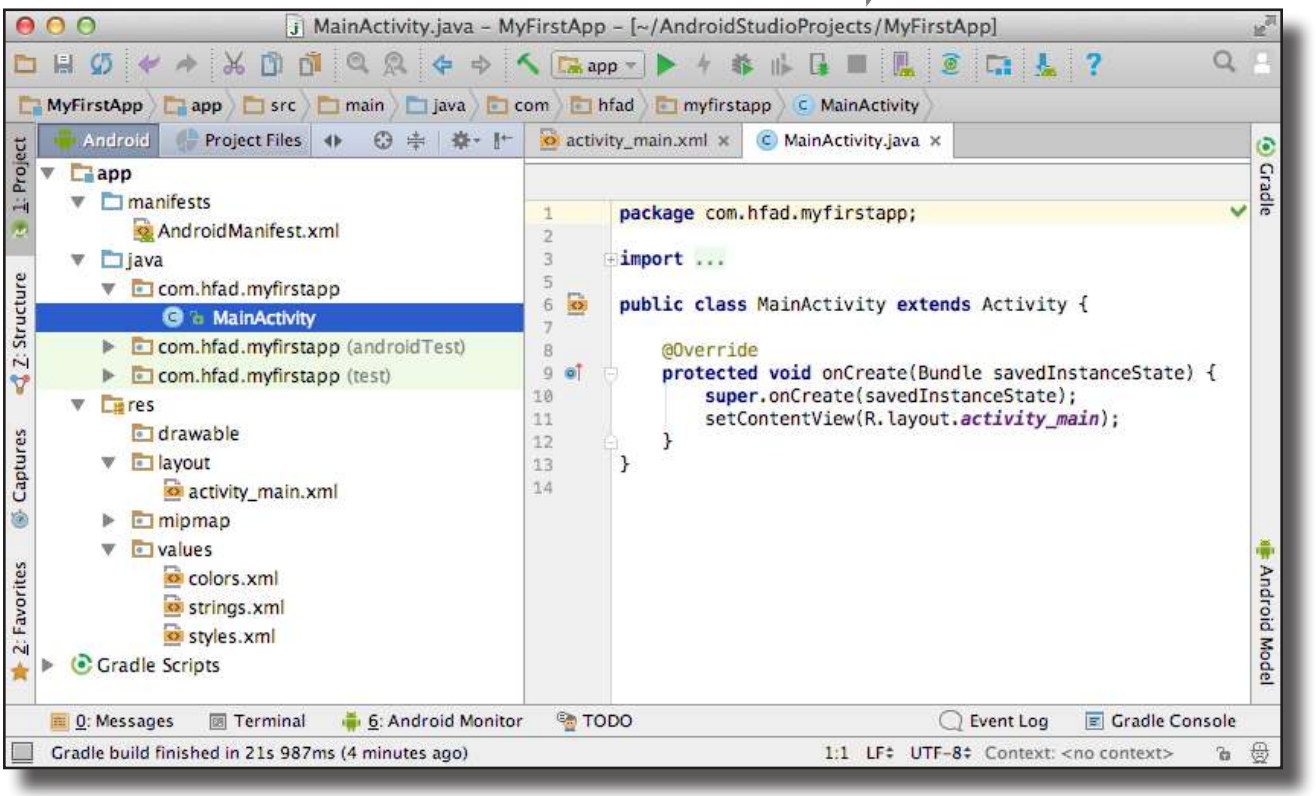
The Create New Project wizard created a project for your app, configured to your specifications.
You defined which versions of Android the app should be compatible with, and the wizard created all of the files and folders needed for a basic valid app.

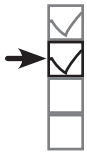
The wizard created an activity and layout with template code.
The template code includes layout XML and activity Java code, with sample "Hello World!" text in the layout.

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

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

This is the project in Android Studio.





Set up environment
Build app
Run app
Change app

Android Studio creates a complete folder structure for you

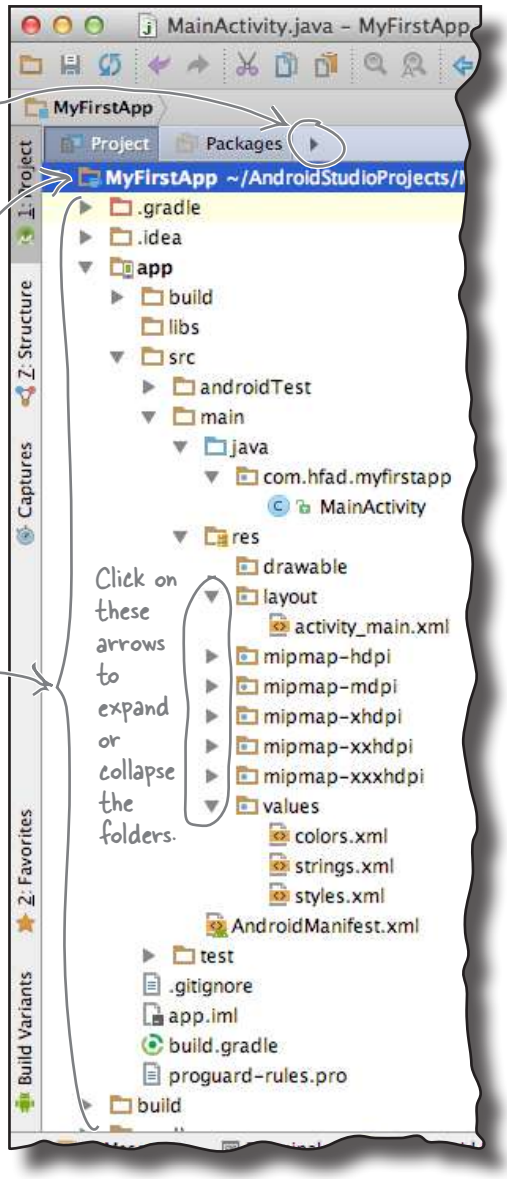
An Android app is really just a bunch of valid files in a particular folder structure, and Android Studio sets all of this up for you when you create a new app. The easiest way of looking at this folder structure is with the explorer in the leftmost column of Android Studio.

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

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

This is the name of the project.

These files and folders are all included in your project.



The folder structure includes different types of files

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

Java and XML source files
These are the activity and layout files for your app.

Android-generated Java files
There are some extra Java files you don't need to touch that Android Studio generates for you automatically.

Resource files
These include default image files for icons, styles your app might use, and any common String values your app might want to look up.

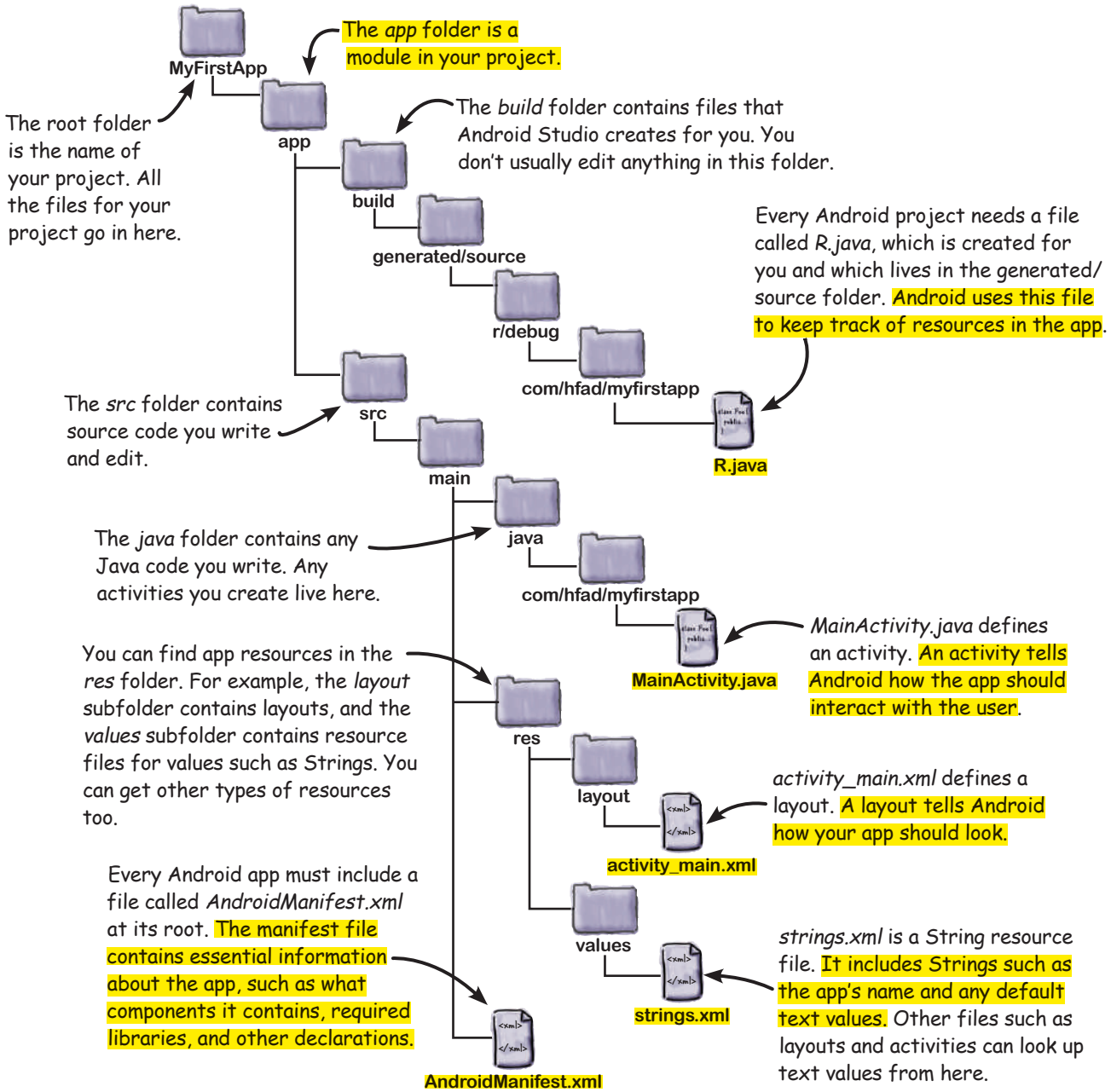
Android libraries
In the wizard, you specified the minimum SDK version you want your app to be compatible with. Android Studio makes sure your app includes the relevant Android libraries for that version.

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

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

Useful files in your project

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



Set up environment
Build app
Run app
Change app

Edit code with the Android Studio editors

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

The code editor

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

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

The design editor

If you're editing a layout, you have an extra option. Rather than edit the XML (such as that shown on the next page), you can use the design editor, which allows you to drag GUI components onto your layout, and arrange them how you want. The code editor and design editor give different views of the same file, so you can switch back and forth between the two.

You dictate which editor you're using with these tabs.

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

Here's the code from an example layout file (**not the one Android Studio generated for us**). We know you've not seen layout code before, but just see if you can match each of the descriptions at the bottom of the page to the correct lines of code. We've done one to get you started.

activity_main.xml

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

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

Add padding to the screen margins.

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

Make the GUI component just large enough for its content.

Display the String "Hello World!"

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

WHAT'S MY PURPOSE? SOLUTION

Here's the code from an example layout file (**not the one Android Studio generated for us**). We know you've not seen layout code before, but just see if you can match each of the descriptions at the bottom of the page to the correct lines of code. We've done one to get you started.

activity_main.xml

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

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

Add padding to the screen margins.

Display the String "Hello World!"

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

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

Make the GUI component just large enough for its content.

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

MainActivity.java

```
package com.hfad.myfirstapp;

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

public class MainActivity extends Activity {

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

This is the package name.

These are Android classes used in `MainActivity`.

Specify which layout to use.

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

`MainActivity` extends the Android class `android.app.Activity`.

WHAT'S MY PURPOSE? SOLUTION

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

MainActivity.java

```
package com.hfad.myfirstapp;

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

public class MainActivity extends Activity {

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

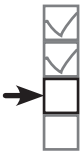
This is the package name.

These are Android classes used in `MainActivity`.

Specify which layout to use.

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

`MainActivity` extends the Android class `android.app.Activity`.



Set up environment
Build app
Run app
Change app

Run the app in the Android emulator

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

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

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

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

So what does the emulator look like?

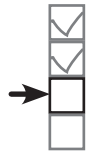
Here's an AVD running in the Android emulator. It looks just like a phone running on your computer.

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

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

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

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



Set up environment
Build app
Run app
Change app

Create an Android Virtual Device

There are a few steps you need to go through in order to set up an AVD within Android Studio. We'll set up a Nexus 5X AVD running API level 25 so that you can see how your app looks and behaves running on this type of device. The steps are pretty much identical no matter what type of virtual device you want to set up.

Open the Android Virtual Device Manager

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

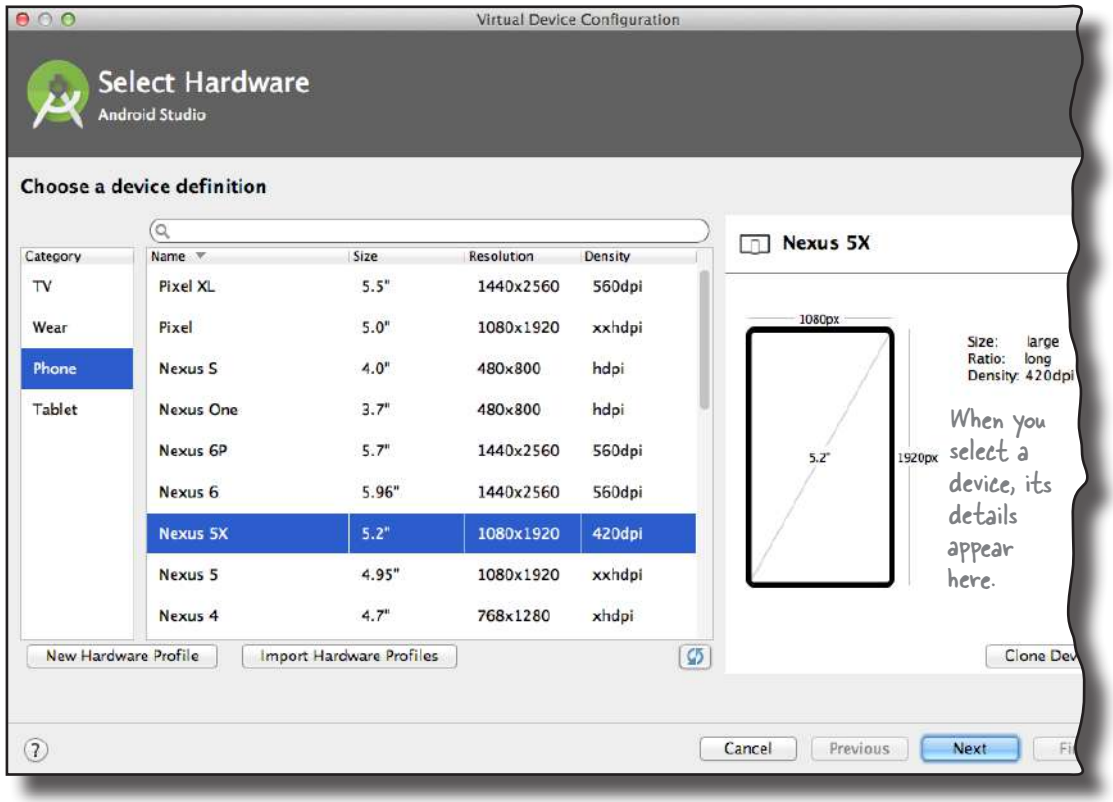
If you have no AVDs set up already, you'll be presented with a screen prompting you to create one. Click on the "Create Virtual Device" button.

Click on this button to create an AVD.

Select the hardware

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

We're going to see what our app looks like running on a Nexus 5X phone. Choose Phone from the Category menu and Nexus 5X from the list. Then click the Next button.



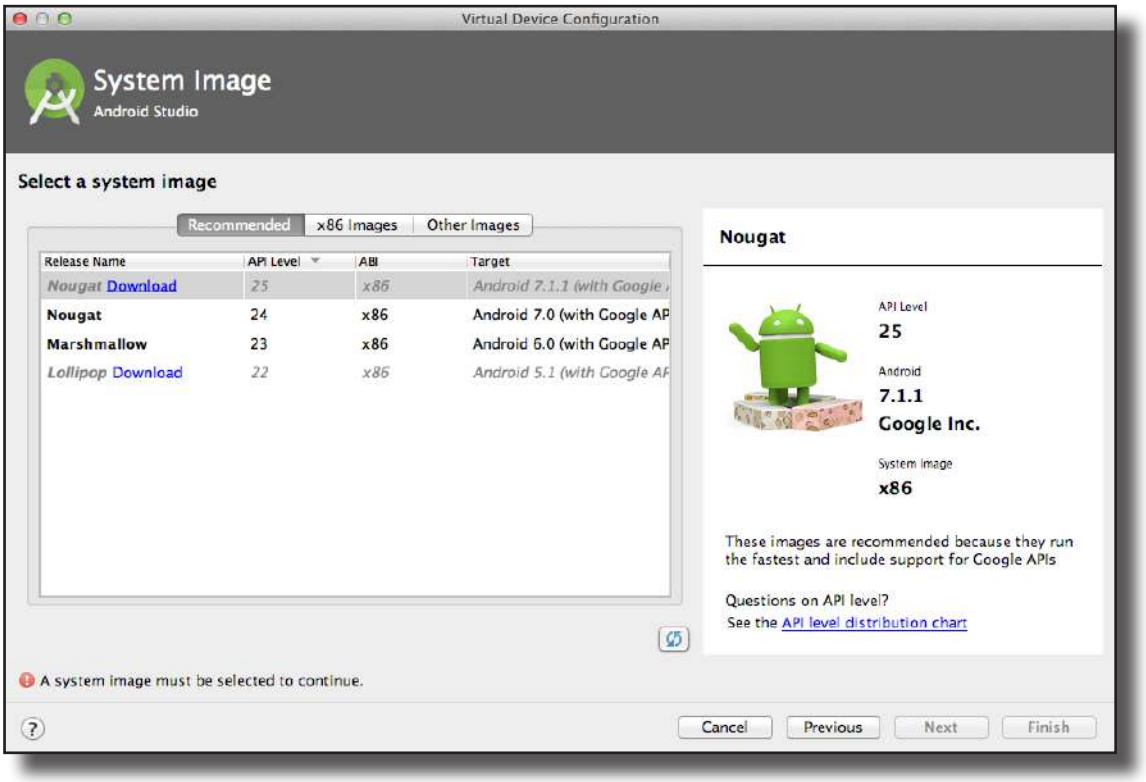
Creating an AVD (continued)

Select a system image

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

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

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



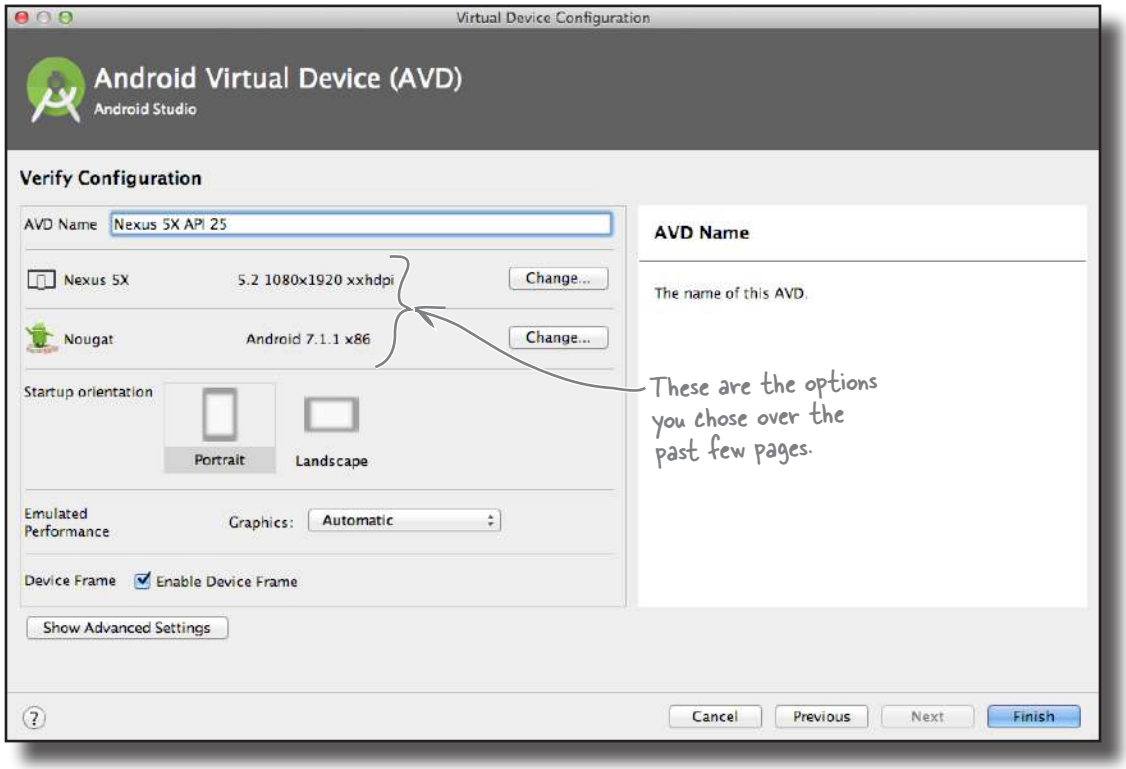
We'll continue setting up the AVD on the next page.

Set up environment
Build app
Run app
Change app

Creating an AVD (continued)

Verify the AVD configuration

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

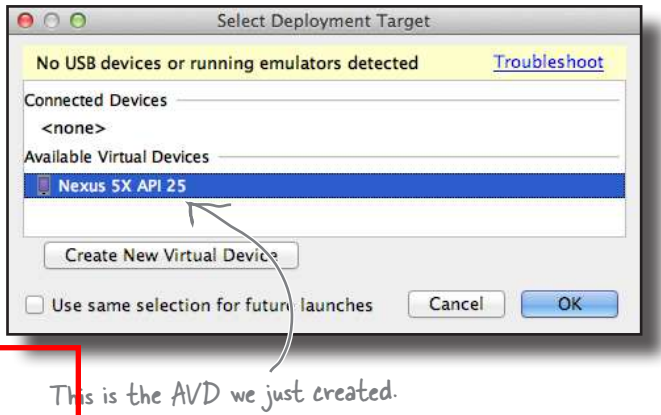


The AVD Manager will create the AVD for you, and when it's done, display it in the AVD Manager list of devices. You may now close the AVD Manager.

Run the app in the emulator

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

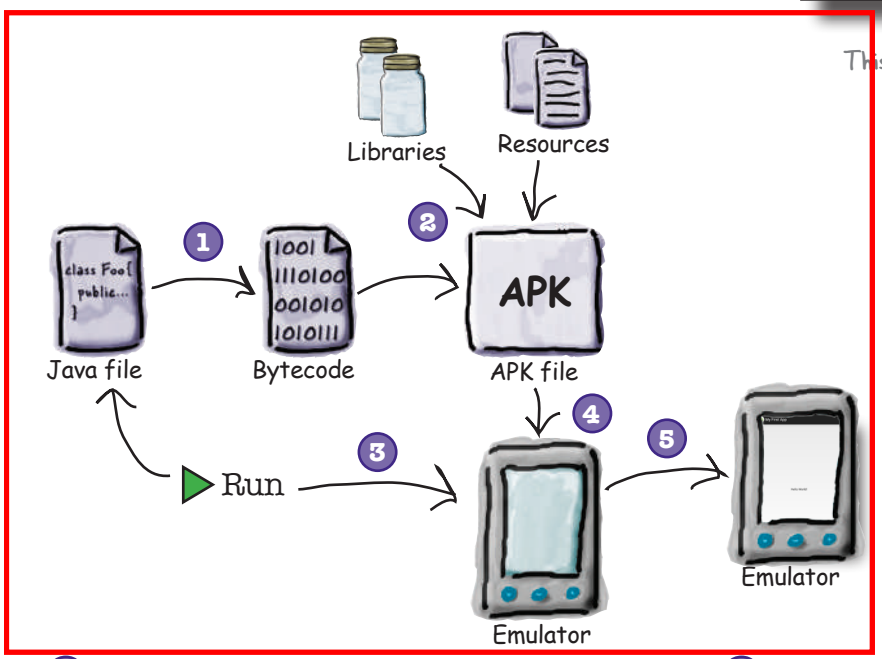
The AVD can take a few minutes to appear, so while we wait, let's take a look at what happens when you choose Run.



This is the AVD we just created.

Compile, package, deploy, and run

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



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

1. The Java source files get compiled to bytecode.

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

3. Assuming there's not one already running, the emulator gets launched and then runs the AVD.

4. Once the emulator has been launched and the AVD is active, the APK file is uploaded to the AVD and installed.

5. **The AVD starts the main activity associated with the app.**
Your app gets displayed on the AVD screen, and it's all ready for you to test out.

You can watch progress in the console

Set up environment
Build app
Run app
Change app

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

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

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

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

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

Install the app.

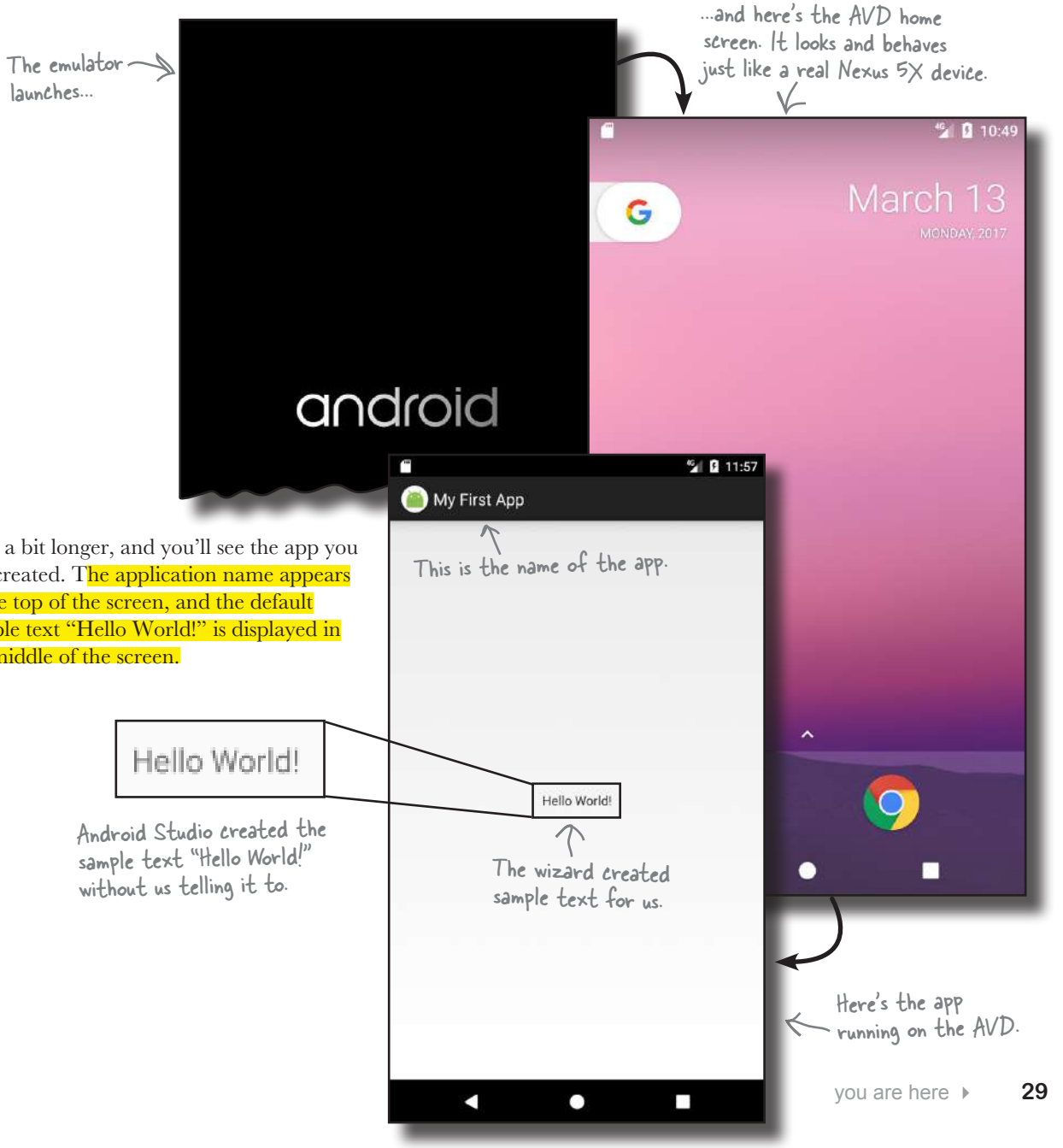
Android Studio has finished launching the AVD we just set up.

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

Test drive

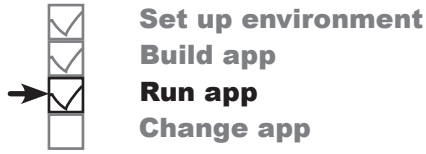
So let's look at what actually happens onscreen when you run your app.

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



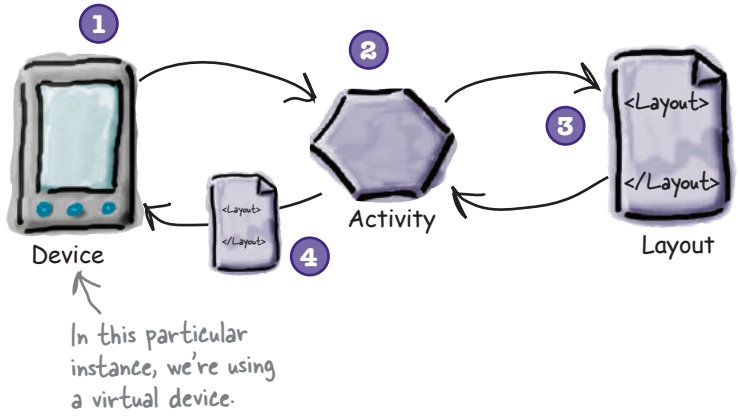
Wait a bit longer, and you'll see the app you just created. The application name appears at the top of the screen, and the default sample text "Hello World!" is displayed in the middle of the screen.

What just happened?



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

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



there are no Dumb Questions

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

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

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

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

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

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

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

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

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

Set up environment
Build app
Run app
Change app

Refine the app

Over the past several pages, you've built a basic Android app and seen it running in the emulator. Next, we're going to refine the app.

At the moment, the app displays the sample text "Hello World!" that the wizard put in as a placeholder. You're going to change that text to say something else instead. So what do we need to change in order to achieve that? To answer that, let's take a step back and look at how the app is currently built.

The app has one activity and one layout

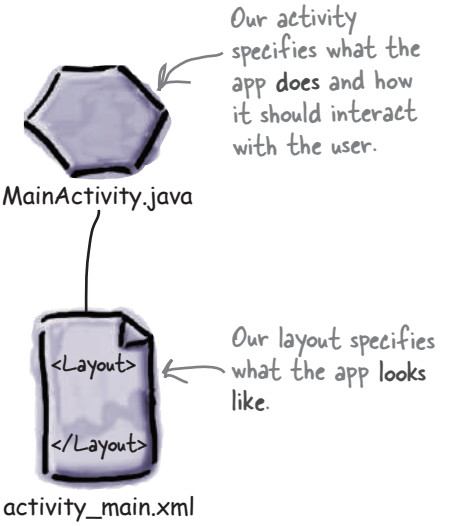
When we built the app, we told Android Studio how to configure it, and the wizard did the rest. The wizard created an activity for us, and also a default layout.

The activity controls what the app does

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

The layout controls the app's appearance

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



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

What's in the layout?

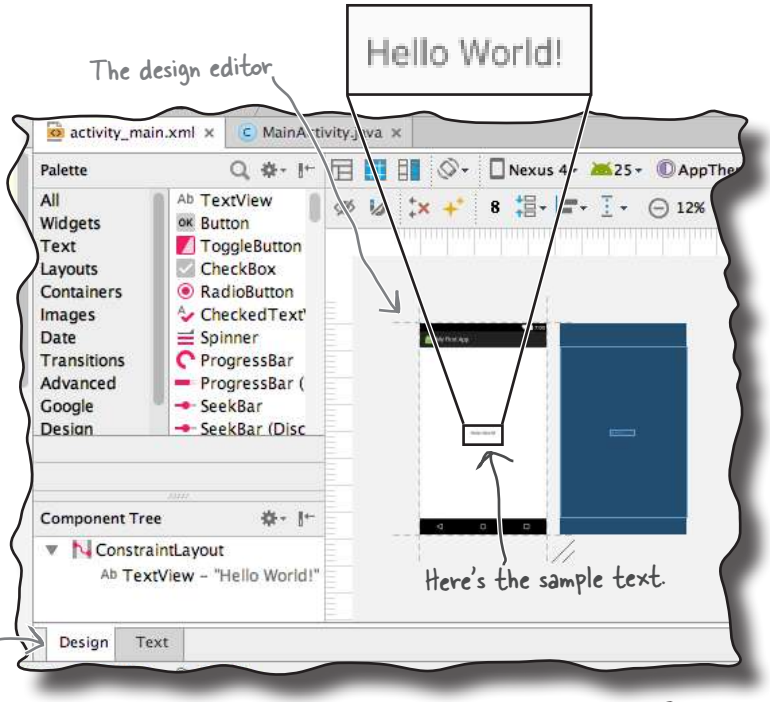
We want to change the sample "Hello World!" text that Android Studio created for us, so let's start with the layout file *activity_main.xml*. If it isn't already open in an editor, open it now by finding the file in the *app/src/main/res/layout* folder in the explorer and double-clicking on it.

The design editor

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

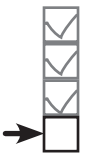
When you choose the design option, you can see that the sample text "Hello World!" appears in the layout as you might expect. But what's in the underlying XML?

Let's see by switching to the code editor.



The code editor

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



Set up environment
Build app
Run app
Change app

activity_main.xml has two elements

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

Here's our code:

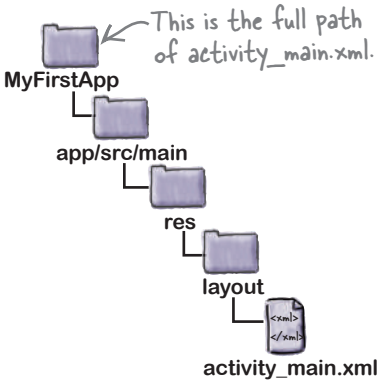
```
<?xml version="1.0" encoding="utf-8"?>
<android.support.constraint.ConstraintLayout
    ... >
    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hello World!"
        ... />
</android.support.constraint.ConstraintLayout>
```

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

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

This is the <TextView> element.

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



As you can see, the code contains two elements.

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

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

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

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

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

This is the text that's being displayed.

Let's change the text to something else.

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

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

Set up environment
Build app
Run app
Change app

Update the text displayed in the layout

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

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

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

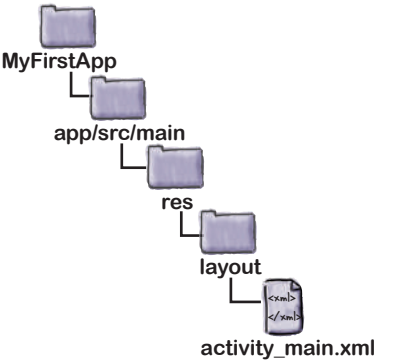
Display the text... `android:text="Hello World!" />` ...."Hello World!"

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

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

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

Change the text here from "Hello World!" to "Sup doge".



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

there are no Dumb Questions

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

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

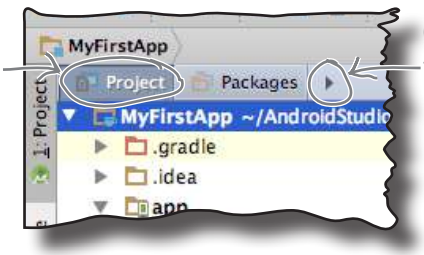
Q: Am I right in thinking we're hardcoding the text that's displayed?

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

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

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

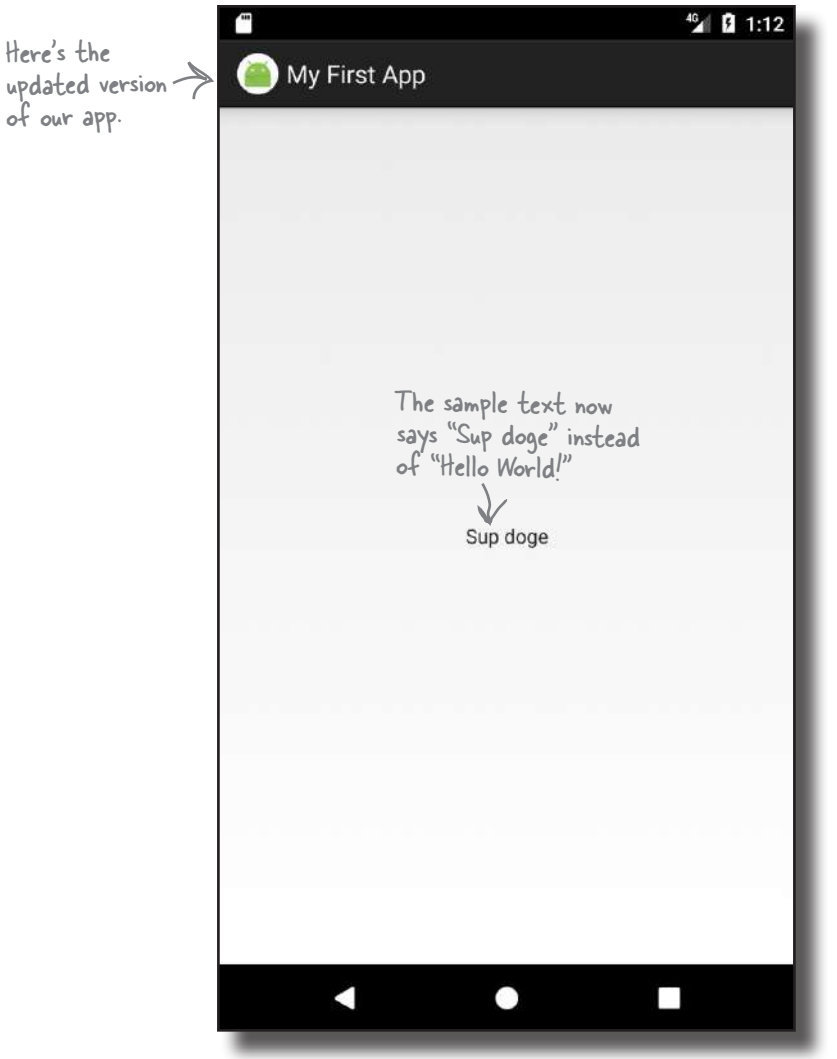
We're using the Project view.



Click on this arrow to change the explorer view.

Take the app for a test drive

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



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

Your Android Toolbox

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

You can download the full code for the chapter from https://tinyurl.com/HeadFirstAndroid.

BULLET POINTS

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

Apps That Do Something

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

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

Let's build a Beer Adviser app

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

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

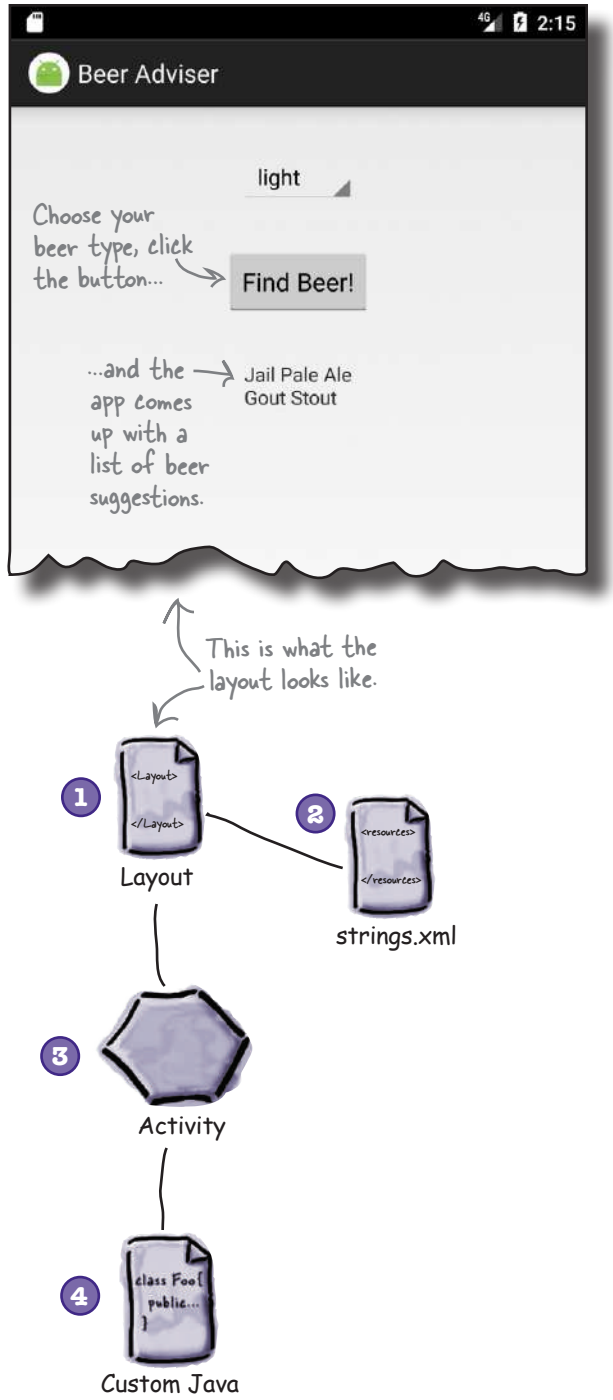
Here's how the app will be structured:

1. **The layout specifies what the app will look like.**
   It includes three GUI components:
   - A drop-down list of values called a spinner, which allows the user to choose which type of beer they want.
   - A button that when pressed will return a selection of beer types.
   - A text field that displays the types of beer.

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

3. **The activity specifies how the app should interact with the user.**
   It takes the type of beer the user chooses, and uses this to display a list of beers the user might be interested in. It achieves this with the help of a custom Java class.

4. **The custom Java class contains the application logic for the app.**
   It includes a method that takes a type of beer as a parameter, and returns a list of beers of this type. The activity calls the method, passes it the type of beer, and uses the response.



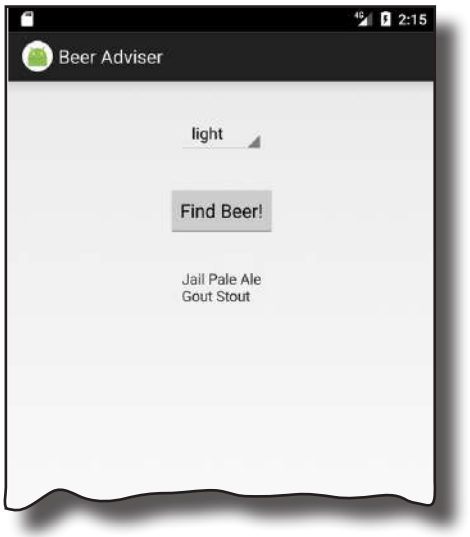
Here's what we're going to do

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

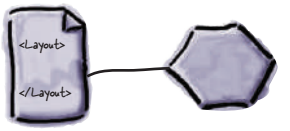
1. **Create a project.**
You're creating a brand-new app, so you'll need to create a new project. Just like before, you'll need to create an empty activity with a layout.

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

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



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



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

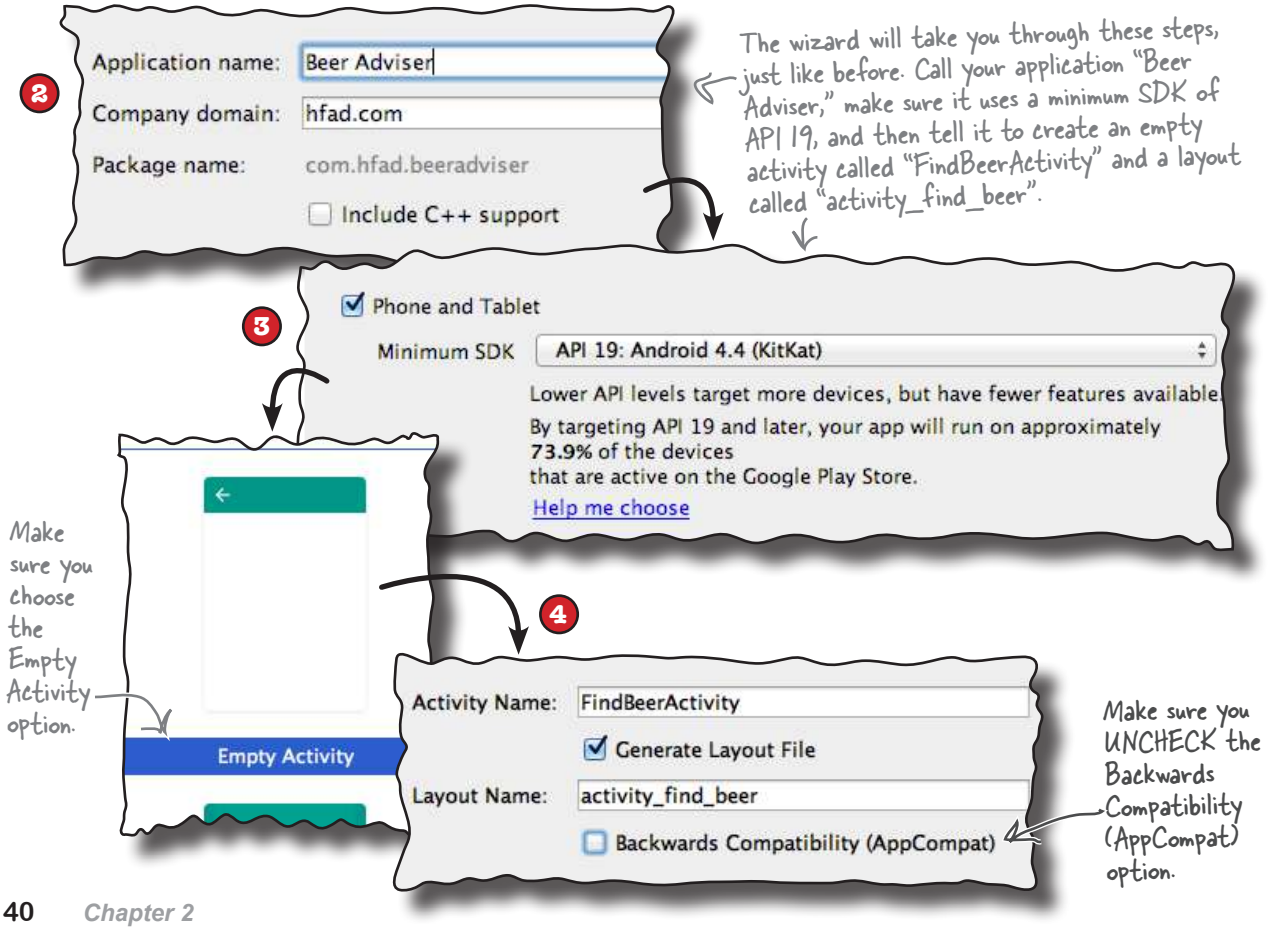
Create project
Update layout
Connect activity
Write logic

Create the project

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

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

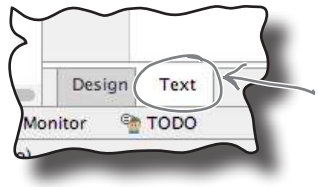
If your version of Android Studio has an option to include Kotlin support, uncheck this option too.



We've created a default activity and layout

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

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



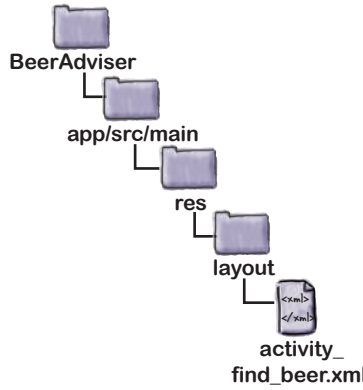
Click on the Text tab to open the code editor.

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

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

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

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

This is used to display text.

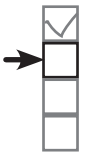


We've just changed the code Android Studio gave us so that it uses a `<LinearLayout>`. This is used to display GUI components next to each other, either vertically or horizontally. If it's vertically, they're displayed in a single column, and if it's horizontally, they're displayed in a single row. You'll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

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

Click on the Design tab to open the design editor.

A closer look at the design editor

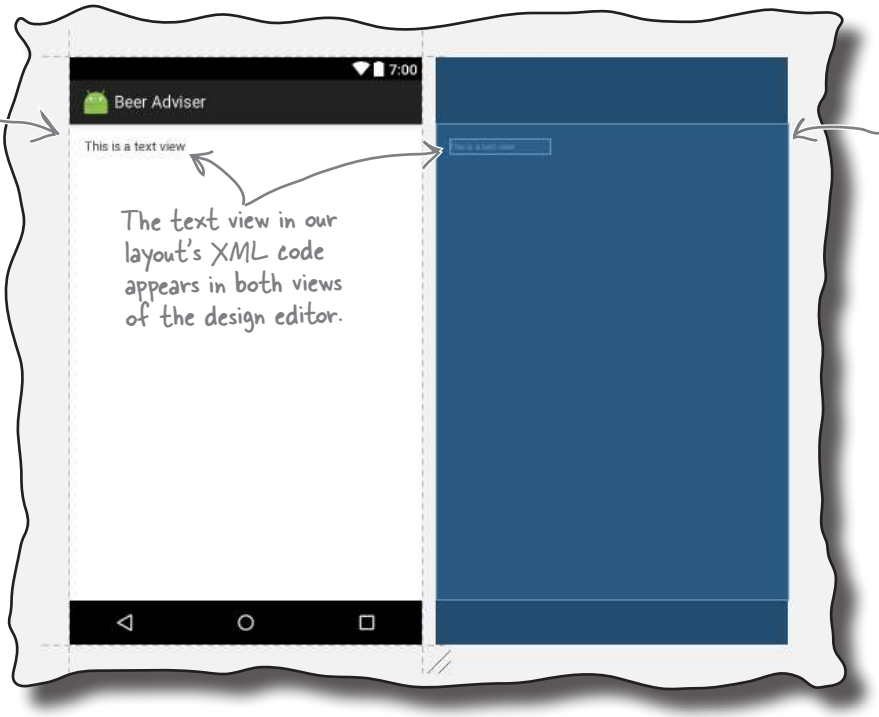


Create project
Update layout
Connect activity
Write logic

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

If Android Studio doesn't show you both views of the layout, click on the "Show Design + Blueprint" icon in the design editor's toolbar.

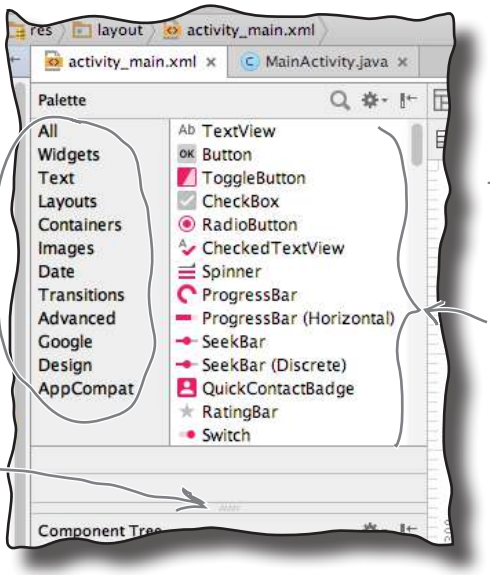
This view of the design gives you an idea of how your layout will look on an actual device.



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

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

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



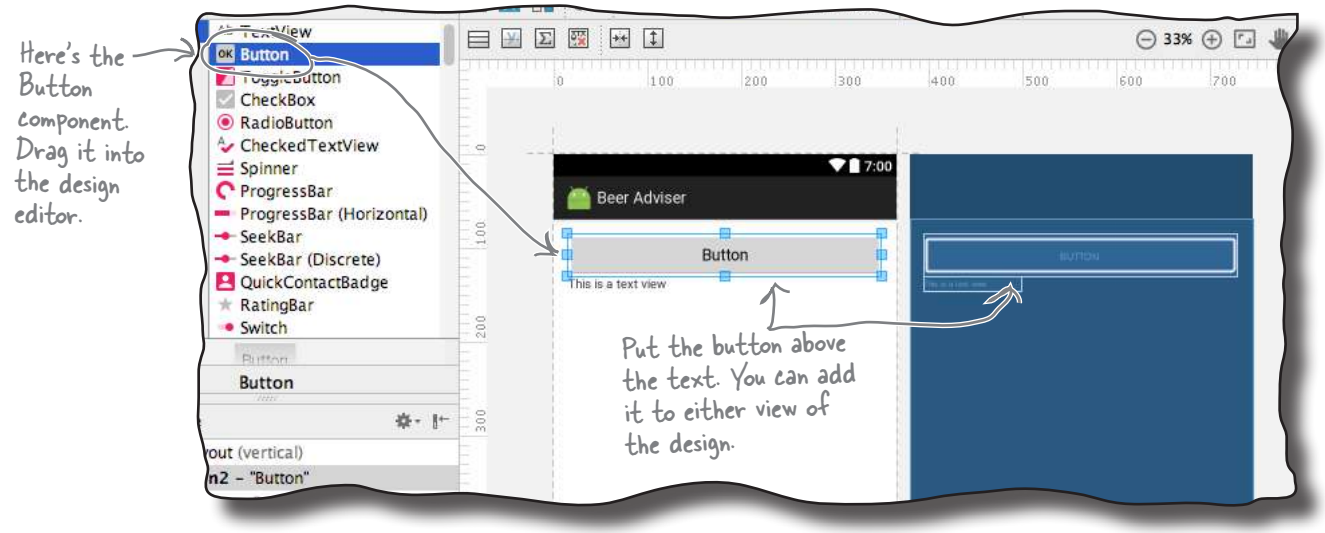
This list shows you the different categories of component you can add to your layout. You can click on them to filter the components displayed in the palette.

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

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

Add a button using the design editor

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



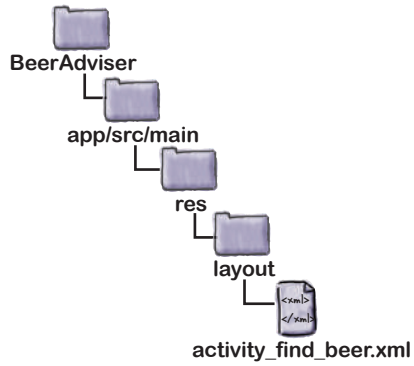
Changes in the design editor are reflected in the XML

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

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

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

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

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



activity_find_beer.xml has a new button

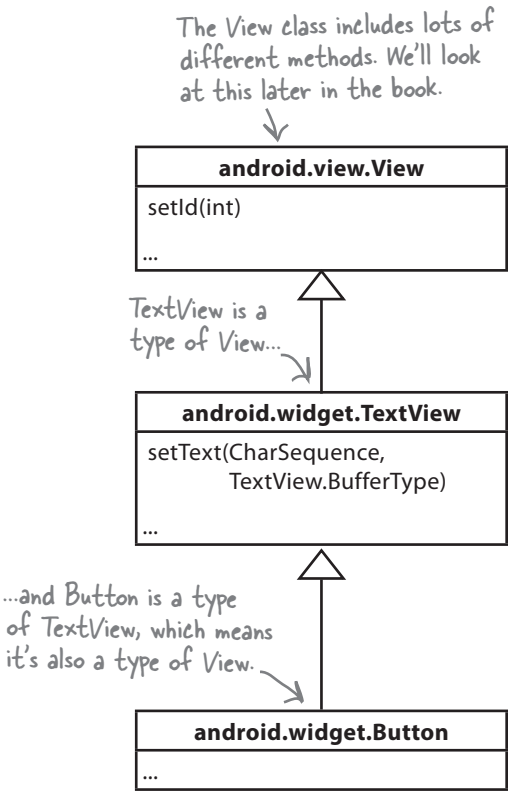
The editor added a new <Button> element to *activity_find_beer.xml*:

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

A button in Androidville is a pushbutton that the user can press to trigger an action. The <Button> element includes properties controlling its size and appearance. These properties aren't unique to buttons—other GUI components including text views have them too.

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

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



android:id

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

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

android:layout_width, android:layout_height

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

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

android:text

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

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

A closer look at the layout code

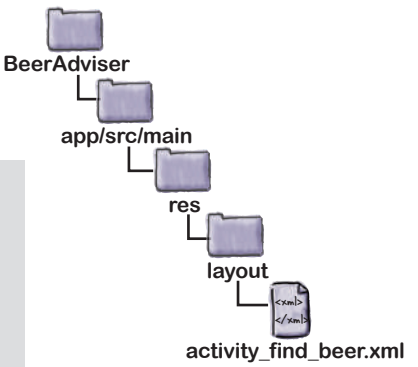
Let's take a closer look at the layout code, and break it down so that you can see what it's actually doing (don't worry if your code looks a little different, just follow along with us):

The <LinearLayout> element

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

This is the button.

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



This is the text view.

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

```
</LinearLayout>
```

This closes the <LinearLayout> element.

The LinearLayout element

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

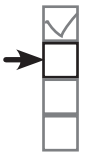
There are other ways of laying out your GUI components too. You'll find out more about these later in the book.

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

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

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

A closer look at the layout code (continued)



Create project
Update layout
Connect activity
Write logic

The <LinearLayout> contains two elements: a <Button> and a <TextView>.

The Button element

The first element is the <Button>:

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

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

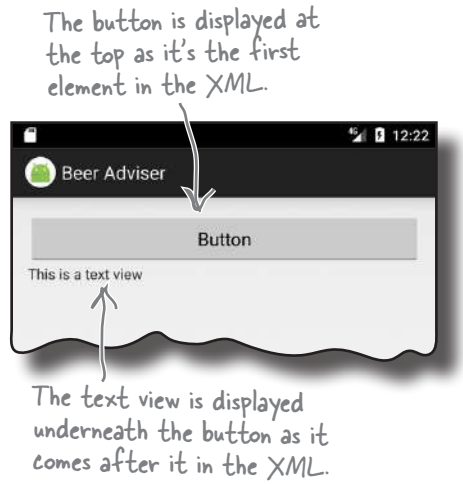
Using a linear layout means that GUI components are displayed in a single row or column.

The TextView element

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

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

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

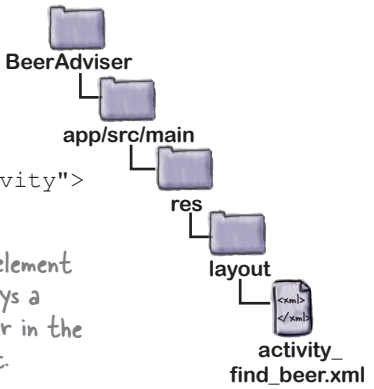


Changes to the XML...

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

Try this now. Update your *activity_find_beer.xml* code with the following changes (highlighted in bold):

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

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

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

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



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

This element displays a spinner in the layout.

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

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

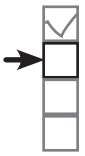
Center the button horizontally and give it a margin.

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

Center the text view and apply a margin.

Do this!

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

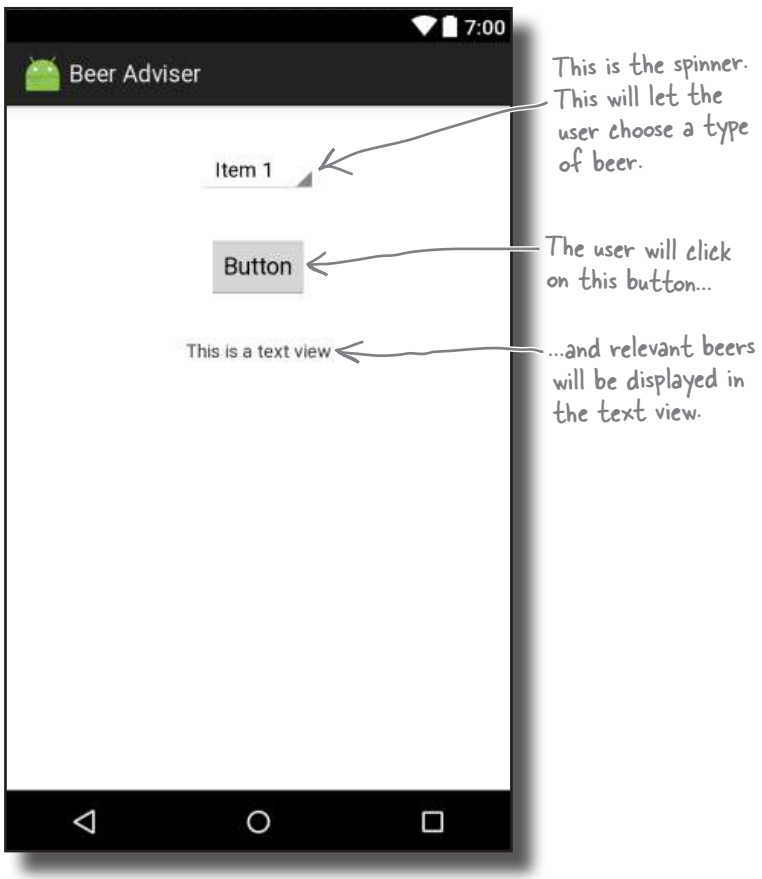


Create project
Update layout
Connect activity
Write logic

...are reflected in the design editor

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

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

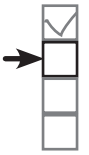


A spinner provides a drop-down list of values. It allows you to choose a single value from a set of values.

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

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

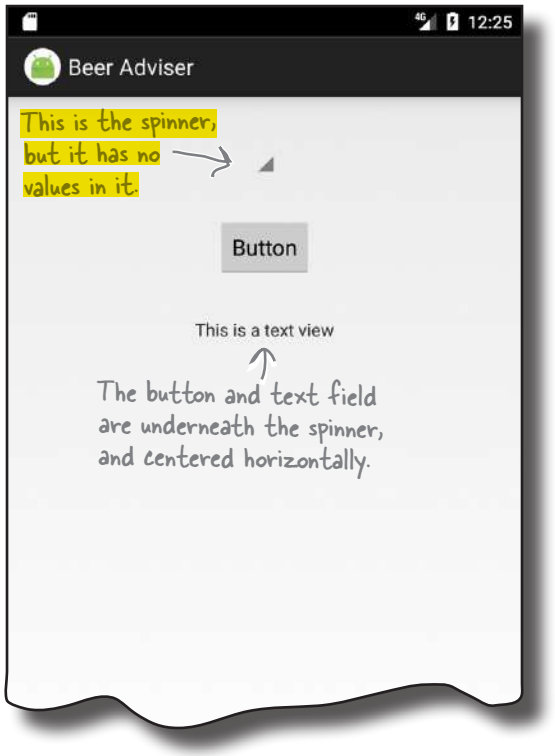
Let's take the app for a test drive



Create project
Update layout
Connect activity
Write logic

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

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



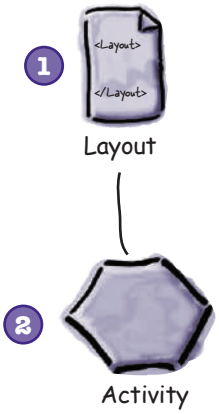
Try pressing the spinner. It's not immediately obvious, but when you press it, the spinner presents you with a drop-down list of values—it's just at this point we haven't added any values to it.

Here's what we've done so far

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

1. **We've created a layout that specifies what the app looks like.**
   It includes a spinner, a button, and a text view.

2. **The activity specifies how the app should interact with the user.**
   Android Studio has created an activity for us, but we haven't done anything with it yet.

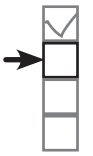


The next thing we'll do is look at replacing the hardcoded String values for the text view and button text.

there are no Dumb Questions

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

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



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

Hardcoding text makes localization hard

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

Display the text... ...“Hello World!”

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

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

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

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

Put the text in a String resource file

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

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

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

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

How do you use String resources?

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

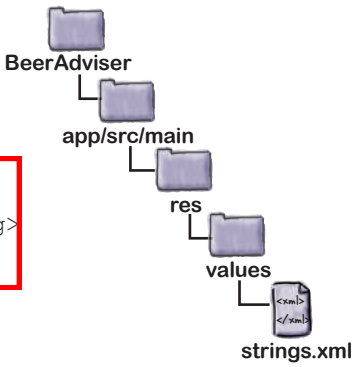
1. **Create the String resource by adding it to strings.xml.**
2. **Use the String resource in your layout.**

Let's see how this is done.

Create the String resource

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

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



The file should look something like this:

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

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

This indicates that this is a String resource.

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

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

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

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

This adds a new String resource called "find_beer".

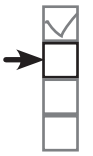
Then add a new resource named `"brands"` with a value of `No beers selected`:

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

This will be the default text in the text view.

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

Use the String resource in your layout



Create project
Update layout
Connect activity
Write logic

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

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

You've seen the `android:text` part of the code before; it specifies what text should be displayed. But what does `"@string/find_beer"` mean?

Let's start with the first part, `@string`. This is just a way of telling Android to look up a text value from a String resource file. In our case, this is the file *strings.xml* that you just edited.

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

Display the text... `android:text="@string/find_beer" />` ...for the String resource find_beer.

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

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

- Change the line:

  `android:text="Button"`

  to:

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

- Change the line:

  `android:text="TextView"`

  to:

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

You can see the code on the next page.

Watch it!

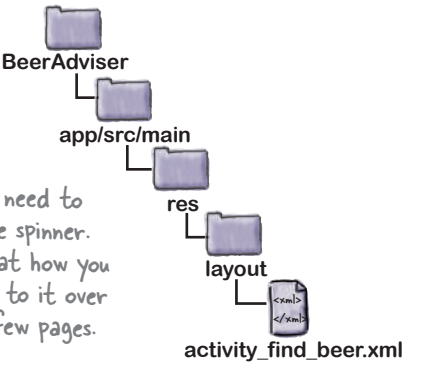
Android Studio sometimes displays the values of references in the code editor in place of actual code.

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

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

The code for activity_find_beer.xml

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




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

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

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

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

Delete the hardcoded text.

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

Delete this hardcoded text too.

When you're done, save your changes.

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

String Resource Files Up Close

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

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

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

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

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

- **The file is held in the folder *app/src/main/res/values*.**
  XML files held in this folder contain simple values, such as Strings and colors.
- **The file has a <resources> element, which contains one or more <string> elements.**
  The format of the file itself indicates that it's a resource file containing Strings. The `<resources>` element tells Android that the file contains resources, and the `<string>` element identifies each String resource.

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

Each name/value pair takes the form:

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

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

A layout can retrieve the value of the String using:

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

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

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

Time for a test drive

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

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

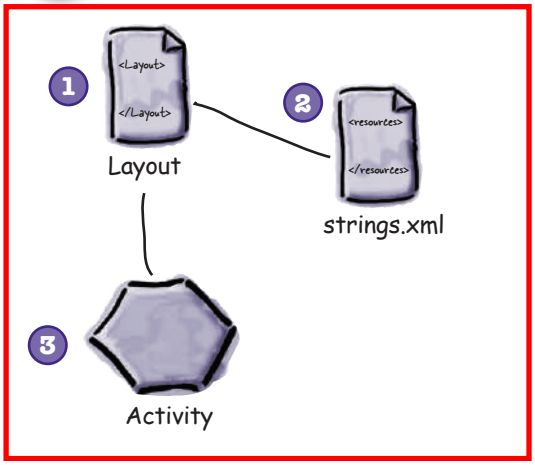
Here's what we've done so far

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

1. **We've created a layout that specifies what the app looks like.**
   It includes a spinner, a button, and a text view.

2. **The file strings.xml includes the String resources we need.**
   We've added a label for the button, and default text for the list of suggested beer brands to try.

3. **The activity specifies how the app should interact with the user.**
   Android Studio has created an activity for us, but we haven't done anything with it yet.



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

there are no Dumb Questions

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

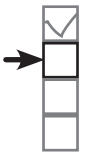
A: It's not mandatory, but Android gives you warning messages if you hardcode text values. Using a String resource file might seem like a lot of effort at first, but it makes things like localization much easier. It's also easier to use String resources to start off with, rather than patching them in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

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

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

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

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



Create project
Update layout
Connect activity
Write logic

Add values to the spinner

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

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

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

Adding an array resource is similar to adding a String

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

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

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

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

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

This is the name of the array.

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

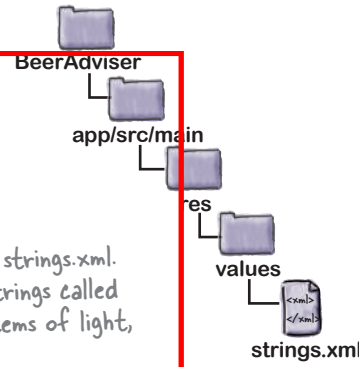
where `string_array_name` is the name of the array, and `string_value1`, `string_value2`, `string_value3` are the individual String values that make up the array.

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

Add the string-array to strings.xml

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

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



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

Get the spinner to reference a string-array

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

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

you use the syntax:

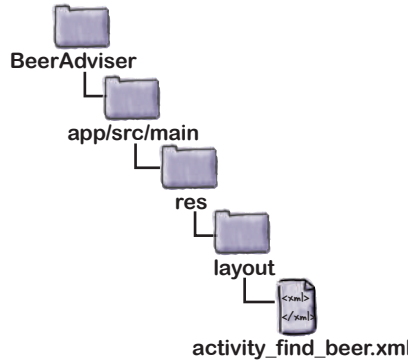
```
"@array/array_name"
```

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

where `array_name` is the name of the array.

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

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

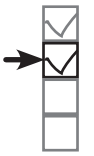


This means "the entries for the spinner come from array beer_colors".

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

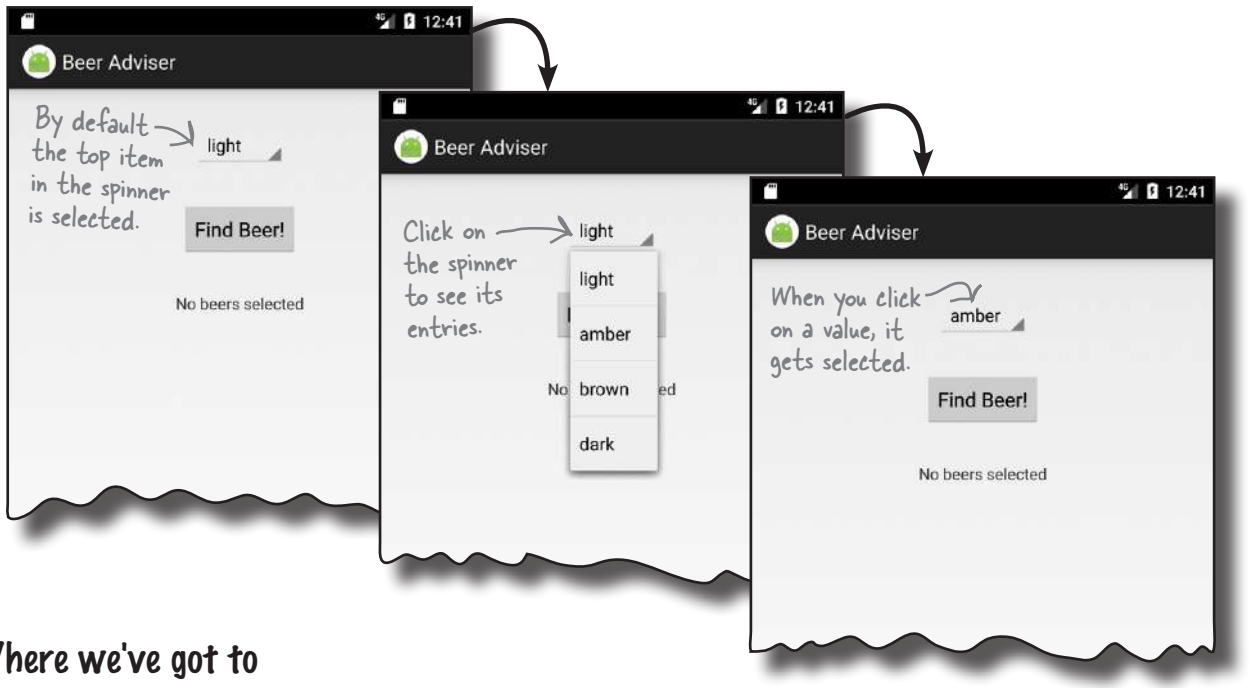
test drive

Test drive the spinner



Create project
Update layout
Connect activity
Write logic

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



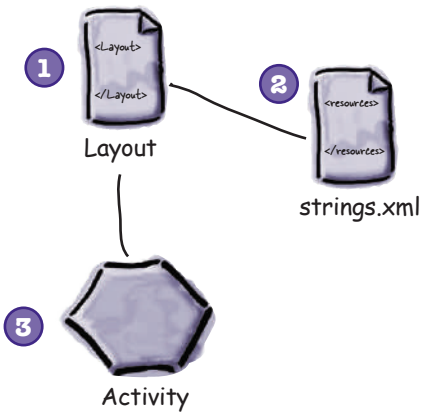
Where we've got to

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

1. **We've created a layout that specifies what the app looks like.**
   It includes a spinner, a button, and a text view.

2. **The file strings.xml includes the String resources we need.**
   We've added a label for the button, default text for the suggested beer brands, and an array of values for the spinner.

3. **The activity specifies how the app should interact with the user.**
   Android Studio has created an activity for us, but we haven't done anything with it yet.



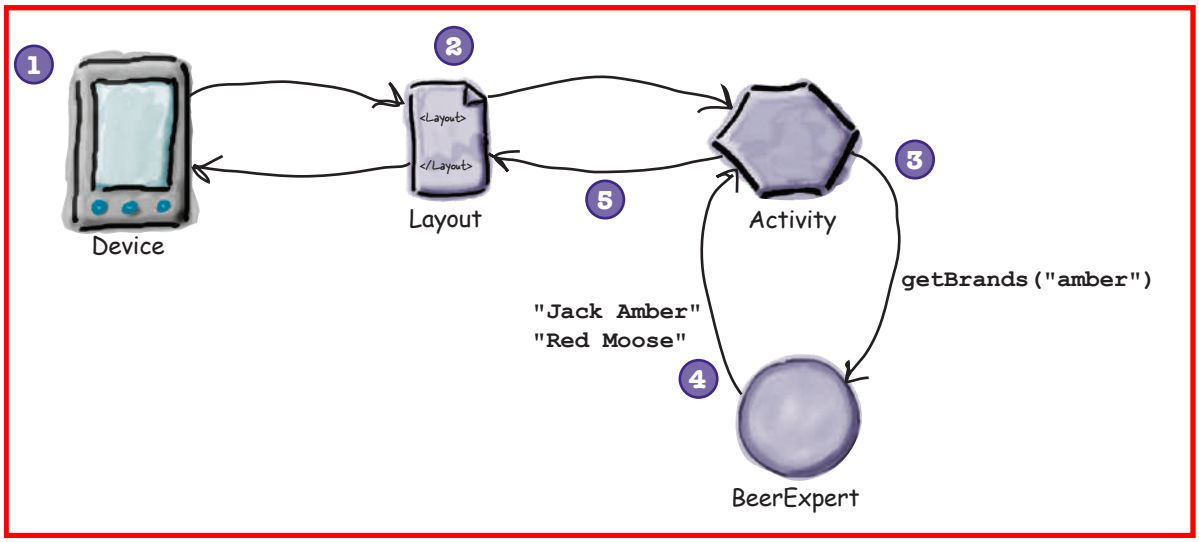
So what's next?

We need to make the button do something

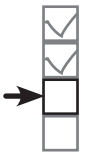
What we need to do next is make the app react to the value we select in the spinner when the Find Beer button is clicked. We want our app to behave something like this:

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

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



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



Create project
Update layout
Connect activity
Write logic

Make the button call a method

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

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

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

Let's start with the layout.

Use onClick to say which method the button calls

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

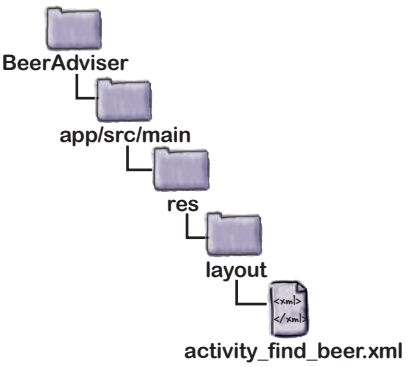
```
android:onClick="method_name"
```

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

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

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

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

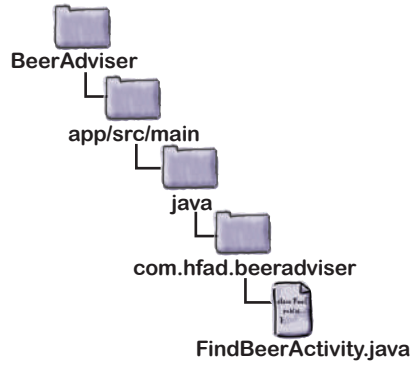


Once you've made these changes, save the file.

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

What activity code looks like

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



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

```
package com.hfad.beeradviser;

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

public class FindBeerActivity extends Activity {

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

Make sure class extends the Android Activity class.

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

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

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

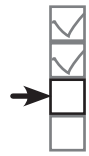
All activities (not just this one) have to extend the `Activity` class or one of its subclasses. The `Activity` class contains a bunch of methods that transform your Java class from a plain old Java class into a full-fledged, card-carrying Android activity.

All activities also need to implement the `onCreate()` method. This method gets called when the activity object gets created, and it's used to perform basic setup such as what layout the activity is associated with. This is done via the `setContentView()` method. In the example above, `setContentView(R.layout.activity_find_beer)` tells Android that this activity uses `activity_find_beer` as its layout.

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

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

Add an onClickFindBeer() method to the activity



Create project
Update layout
Connect activity
Write logic

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

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

The method must be public.

The method must have a void return value.

The method must have a single parameter of type View.

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

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

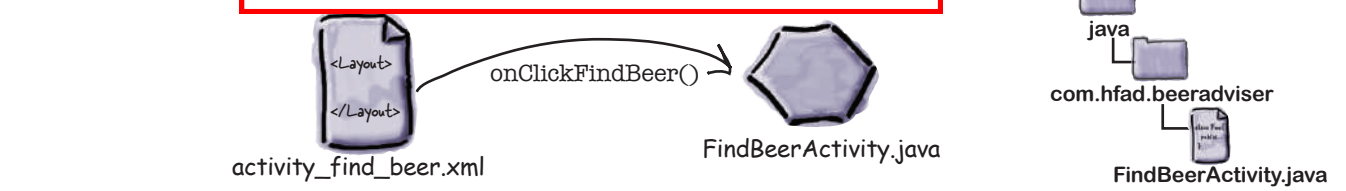
So let's update our activity code. Add the `onClickFindBeer()` method below to your activity code (*FindBeerActivity.java*):

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

We're using this class, so we need to import it.

Add the onClickFindBeer() method to FindBeerActivity.java.

```
...
import android.view.View;

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



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

onClickFindBeer() needs to do something

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

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

Use findViewById() to get a reference to a view

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

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

We want the view with an ID of brands.

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

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

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

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

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

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

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

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

***R.java* gets generated for you.**

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

Create project
Update layout
Connect activity
Write logic

Once you have a view, you can access its methods

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

Setting the text in a text view

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

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

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

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

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

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

Retrieving the selected value in a spinner

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

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

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

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

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

The code:

```
color.getSelectedItem()
```

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

Update the activity code

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

Activity Magnets

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

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

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

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

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

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

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

TextView

color

findViewById

setText

R.id.color

R.view.brands

R.id.brands

(TextView)

findView

Button

findView

R.view.color

View

findViewById

(Spinner)

getSelectedItem()

beerType

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

Activity Magnets Solution

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

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

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

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

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

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

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

R.view.brands

findView

Button

R.view.color

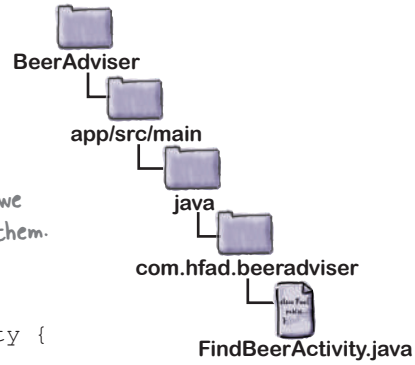
findView

You didn't need to use these magnets.

The first version of the activity

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

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




```
package com.hfad.beeradviser;

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

public class FindBeerActivity extends Activity {

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

    //Called when the button gets clicked
    public void onClickFindBeer(View view) {
        //Get a reference to the TextView
        TextView brands = (TextView) findViewById(R.id.brands);
        //Get a reference to the Spinner
        Spinner color = (Spinner) findViewById(R.id.color);
        //Get the selected item in the Spinner
        String beerType = String.valueOf(color.getSelectedItem());
        //Display the selected item
        brands.setText(beerType);
    }
}
```

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

We've not changed this method.

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

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

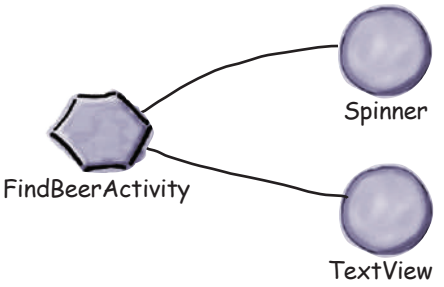
What the code does

Create project
Update layout
Connect activity
Write logic

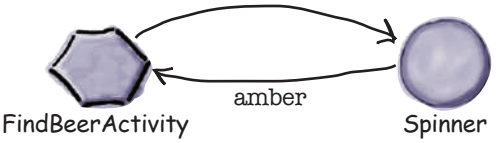
Before we take the app for a test drive, let's look at what the code actually does.

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

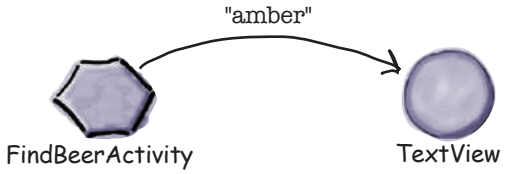
2. **The activity gets references to the Spinner and TextView GUI components using calls to the findViewById() method.**



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

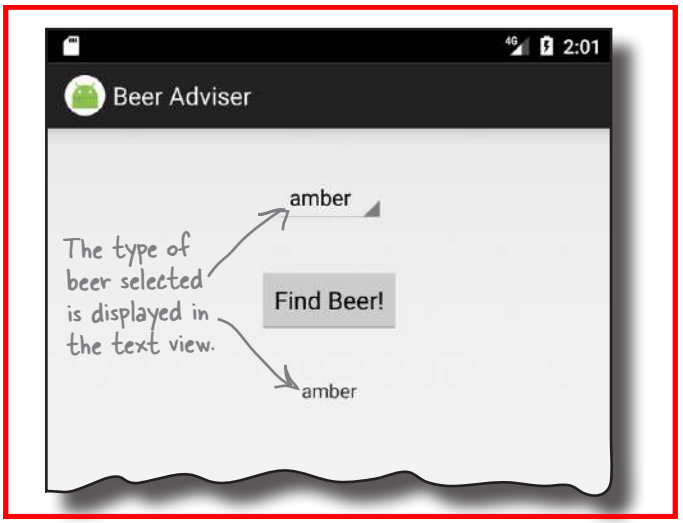


4. **The activity then sets the text property of the TextView to reflect the currently selected item in the spinner.**



Test drive the changes

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



there are no Dumb Questions

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

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

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

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

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

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

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

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

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

Q: Does the name of `onClickFindBeer` matter?

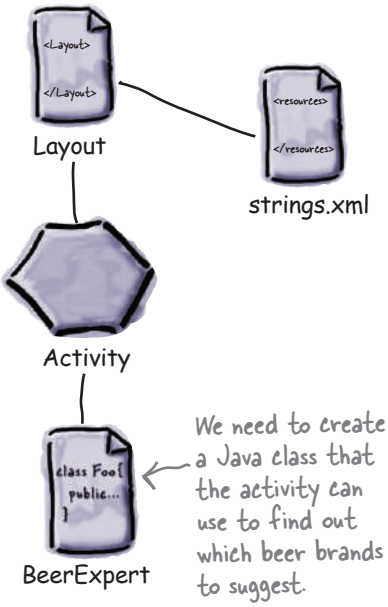
A: All that matters is that the name of the method in the activity code matches the name used in the button's `onClick` attribute in the layout.

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

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

Build the custom Java class

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



Custom Java class spec

The custom Java class should meet the following requirements:

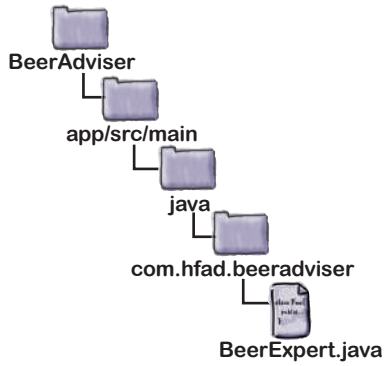
- The package name should be `com.hfad.beeradviser`.
- The class should be called `BeerExpert`.
- It should expose one method, `getBrands()`, that takes a preferred beer color (as a String), and return a `List<String>` of recommended beers.

Build and test the Java class

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

```
package com.hfad.beeradviser;

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

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

This is pure Java code; nothing Androidy about it.



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

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

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

Sharpen your pencil

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

```
package com.hfad.beeradviser;

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

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




    }
}
```

You need to update the onClickFindBeer() method.

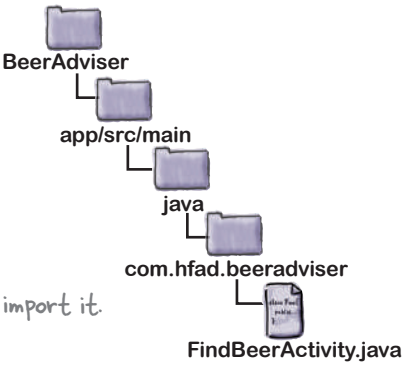
Enhance the activity so that it calls the `BeerExpert getBrands()` method and displays the results in the text view.

```
package com.hfad.beeradviser;

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

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

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

        StringBuilder brandsFormatted = new StringBuilder(); <- Build a String using
                                                                the values in the List.
        for (String brand : brandsList) {

            brandsFormatted.append(brand).append('\n'); <- Display each brand
                                                           on a new line.
        }

        //Display the beers

        brands.setText(brandsFormatted); <- Display the results in
                                            the text view.

    }
}
```

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

Activity code version 2

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




```
package com.hfad.beeradviser;

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

public class FindBeerActivity extends Activity {
    private BeerExpert expert = new BeerExpert();

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

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

We're using this extra class so we need to import it.

Add an instance of BeerExpert as a private variable.

Use the BeerExpert class to get a List of brands.

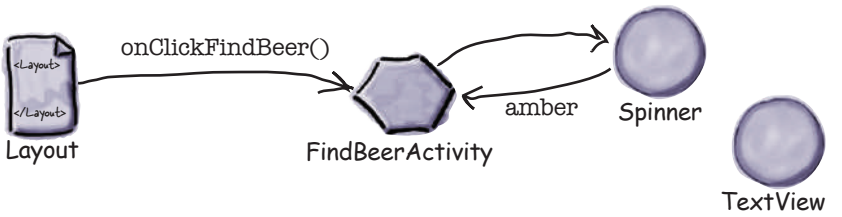
Build a String, displaying each brand on a new line.

Display the String in the TextView.

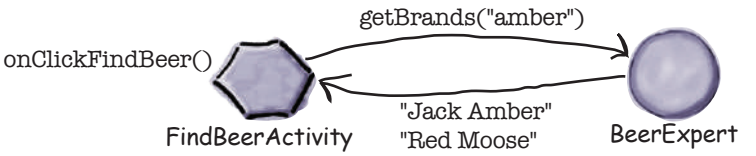
Delete this line.

What happens when you run the code

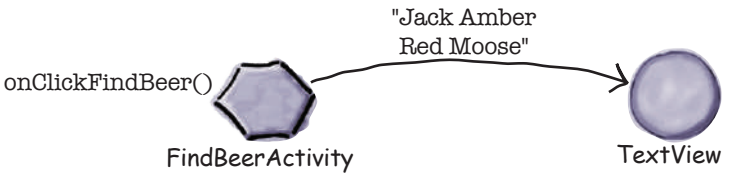
1 **When the user clicks on the Find Beer button, the onClickFindBeer() method in the activity gets called.**
The method creates a reference to the spinner and text view, and gets the currently selected value from the spinner.



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

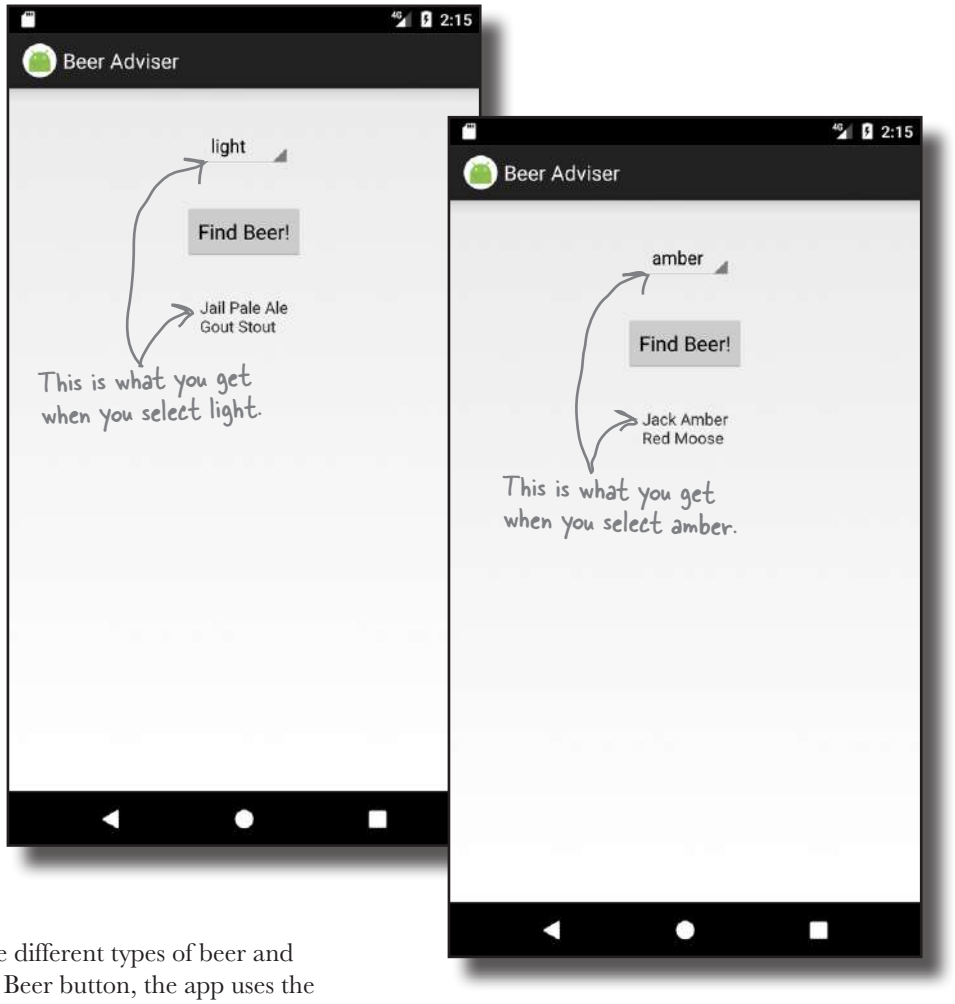


3 **The onClickFindBeer() method formats the list of brands and uses it to set the text property in the text view.**



Test drive your app

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



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

Your Android Toolbox

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

You can download the full code for the chapter from https://tinyurl.com/HeadFirstAndroid.

BULLET POINTS

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

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

- Reference a String in the layout using:

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

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

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

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

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

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

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

There needs to be a corresponding method in the activity:

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

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