

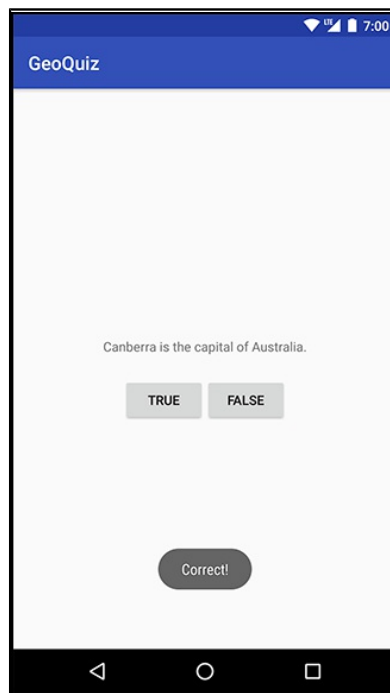
Your First Android Application

This first chapter is full of new concepts and moving parts required to build an Android application. It is OK if you do not understand everything by the end of this chapter. You will be revisiting these ideas in greater detail as you proceed through the book.

The application you are going to create is called GeoQuiz. GeoQuiz tests the user's knowledge of geography. The user presses TRUE or FALSE to answer the question on screen, and GeoQuiz provides instant feedback.

Figure 1.1 shows the result of a user pressing the TRUE button.

Figure 1.1 Do you come from a land down under?



App Basics

Your GeoQuiz application will consist of an *activity* and a *layout*:

- An *activity* is an instance of **Activity**, a class in the Android SDK. An activity is responsible for managing user interaction with a screen of information.

You write subclasses of **Activity** to implement the functionality that your app requires. A simple application may need only one subclass; a complex application can have many.

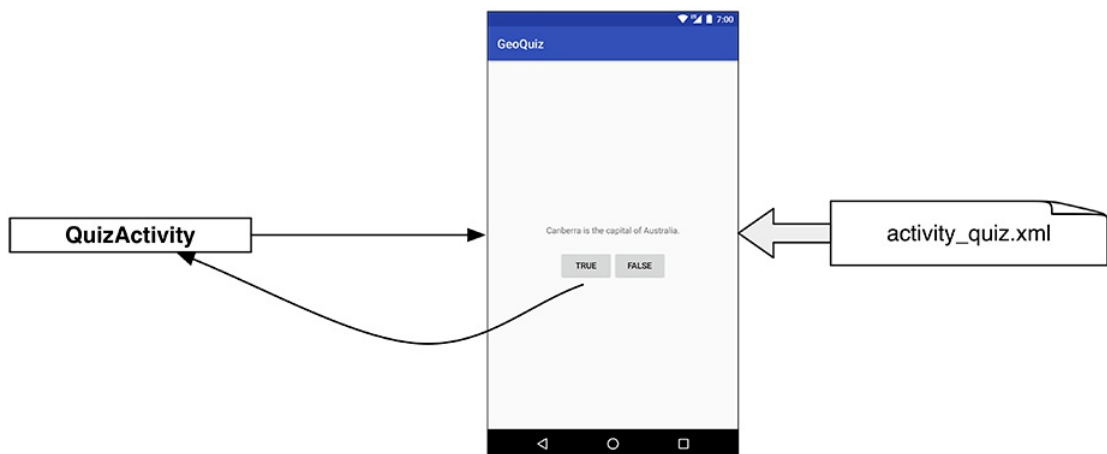
GeoQuiz is a simple app, so it will have a single **Activity** subclass named **QuizActivity**. **QuizActivity** will manage the user interface, or UI, shown in Figure 1.1.

- A *layout* defines a set of UI objects and their positions on the screen. A layout is made up of definitions written in XML. Each definition is used to create an object that appears on screen, like a button or some text.

GeoQuiz will include a layout file named `activity_quiz.xml`. The XML in this file will define the UI shown in Figure 1.1.

The relationship between **QuizActivity** and `activity_quiz.xml` is diagrammed in Figure 1.2.

Figure 1.2 **QuizActivity** manages what `activity_quiz.xml` defines



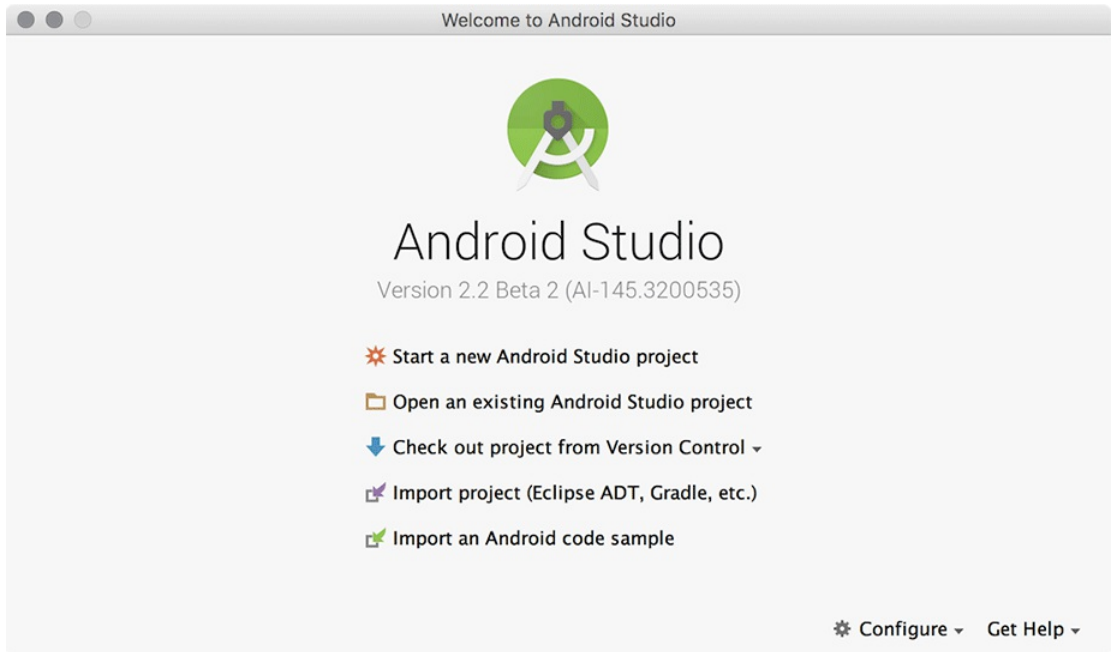
With those ideas in mind, let's build an app.

Creating an Android Project

The first step is to create an Android *project*. An Android project contains the files that make up an application. To create a new project, first open Android Studio.

If this is your first time running Android Studio, you will see the Welcome dialog, as in Figure 1.3.

Figure 1.3 Welcome to Android Studio



From the dialog, choose Start a new Android Studio project. If you do not see the dialog, you may have created projects before. In this case, choose File → New → New Project....

You should see the New Project wizard (Figure 1.4). In the first screen of the wizard, enter GeoQuiz as the application name. For the company domain, enter `android.bignerdranch.com`. As you do this, you will see the generated package name change to `com.bignerdranch.android.geoquiz`. For the project location, you can use any location on your filesystem that you want.

Figure 1.4 Creating a new application

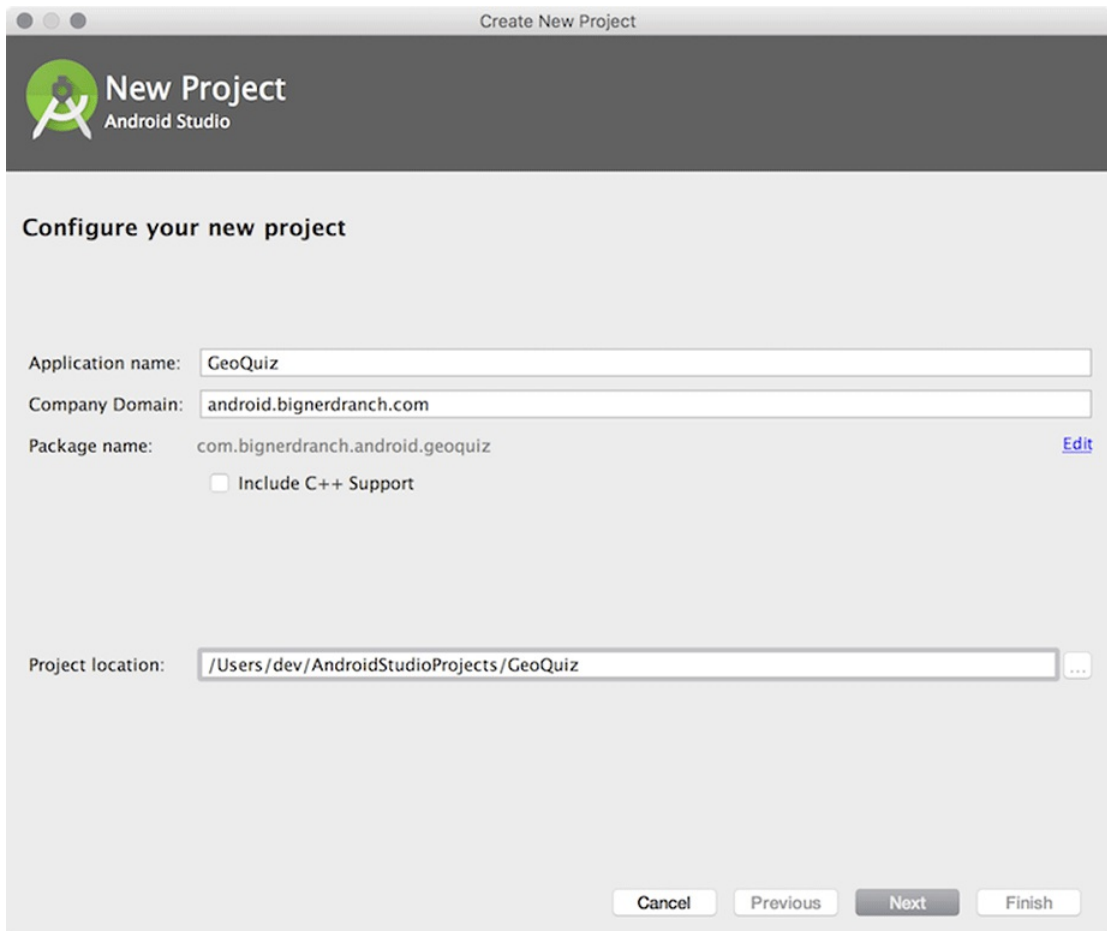


Figure 1.4 shows the "Create New Project" wizard in Android Studio. The window title is "Create New Project". The header area displays the Android Studio logo and the text "New Project" and "Android Studio". The main section is titled "Configure your new project". It contains the following fields and options:

- Application name:** GeoQuiz
- Company Domain:** android.bignerdranch.com
- Package name:** com.bignerdranch.android.geoquiz (with an [Edit](#) link)
- ☐ Include C++ Support
- Project location:** /Users/dev/AndroidStudioProjects/GeoQuiz

At the bottom of the wizard, there are four buttons: "Cancel", "Previous", "Next", and "Finish".

Notice that the package name uses a “reverse DNS” convention: The domain name of your organization is reversed and suffixed with further identifiers. This convention keeps package names unique and distinguishes applications from each other on a device and on Google Play.

Click Next. The next screen allows you to specify details about which devices you want to support. GeoQuiz will only support phones, so just check Phone and Tablet. Select a minimum SDK version of API 19: Android 4.4 (KitKat) (Figure 1.5). You will learn about the different versions of Android in Chapter 6.

Figure 1.5 Specifying device support

The screenshot shows the 'Create New Project' dialog in Android Studio, specifically the 'Target Android Devices' screen. The title bar says 'Create New Project'. The main heading is 'Target Android Devices' with the Android logo. Below this, the instruction is 'Select the form factors your app will run on' and a note says 'Different platforms may require separate SDKs'.

There are five form factor options, each with a checkbox and a 'Minimum SDK' dropdown menu:

- ☒ Phone and Tablet: Minimum SDK is 'API 19: Android 4.4 (KitKat)'. Below this, text reads: 'Lower API levels target more devices, but have fewer features available. By targeting API 19 and later, your app will run on approximately 73.9% of the devices that are active on the Google Play Store.' A link '[Help me choose](#)' is provided.
- ☐ Wear: Minimum SDK is 'API 21: Android 5.0 (Lollipop)'.
- ☐ TV: Minimum SDK is 'API 21: Android 5.0 (Lollipop)'.
- ☐ Android Auto
- ☐ Glass: Minimum SDK is 'Glass Development Kit Preview (API 19)'.

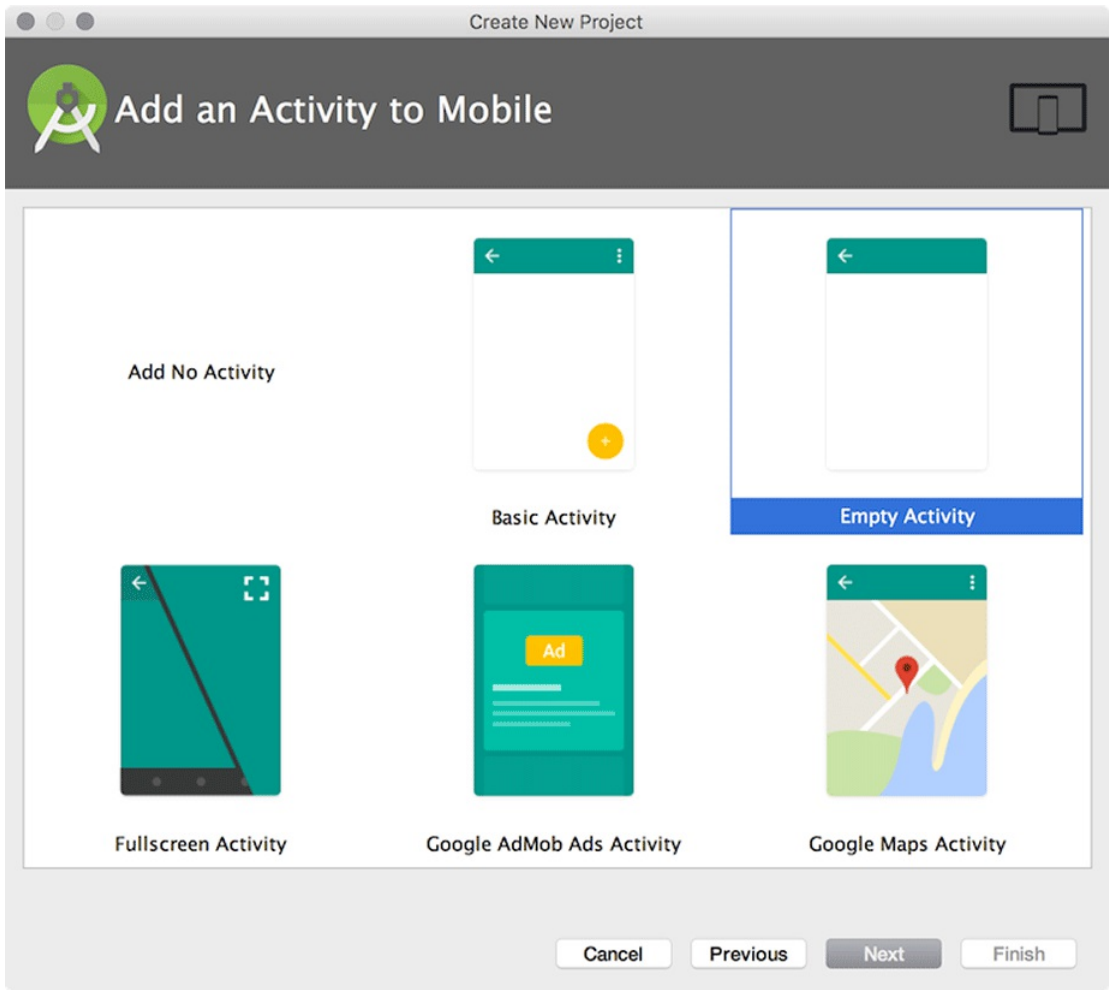
At the bottom, there are four buttons: 'Cancel', 'Previous', 'Next' (which is highlighted), and 'Finish'.

Click Next.

In the next screen, you are prompted to choose a template for the first screen of GeoQuiz (Figure 1.6). You want the most basic template available. Choose Empty Activity and click Next.

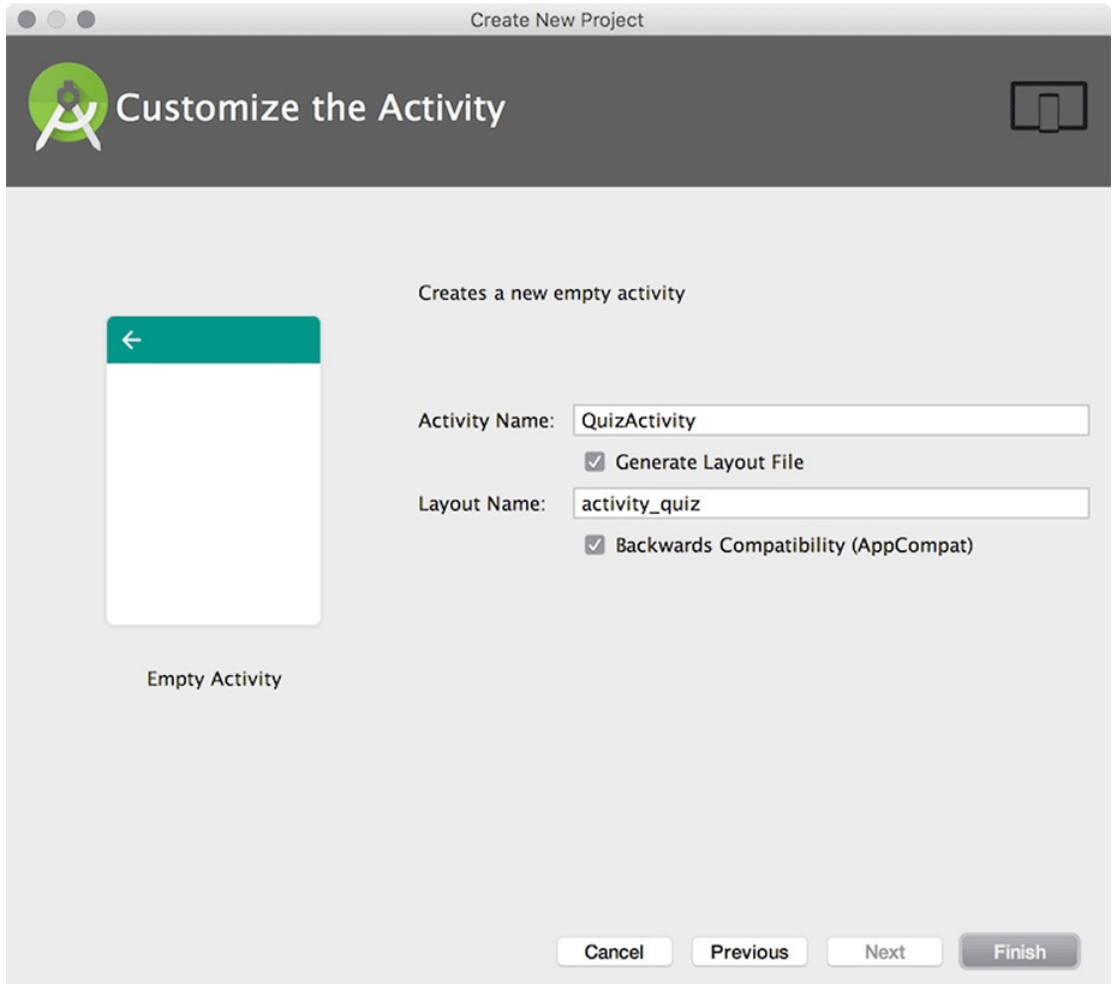
(Android Studio updates regularly, so your wizard may look slightly different from what we are showing you. This is usually not a problem; the choices should be similar. If your wizard looks very different, then the tools have changed more drastically. Do not panic. Head to this book’s forum at forums.bignerdranch.com and we will help you navigate the latest version.)

Figure 1.6 Choosing a type of activity



In the final dialog of this wizard, name the activity subclass **QuizActivity** (Figure 1.7). Notice the **Activity** suffix on the class name. This is not required, but it is an excellent convention to follow.

Figure 1.7 Configuring the new activity



Leave **Generate Layout File** checked. The layout name will automatically update to `activity_quiz` to reflect the activity's new name. The layout name reverses the order of the activity name, is all lowercase, and has underscores between words. This naming style is recommended for layouts as well as other resources that you will learn about later.

If your version of Android Studio has other options on this screen, leave them as is. Click **Finish**. Android Studio will create and open your new project.

Navigating in Android Studio

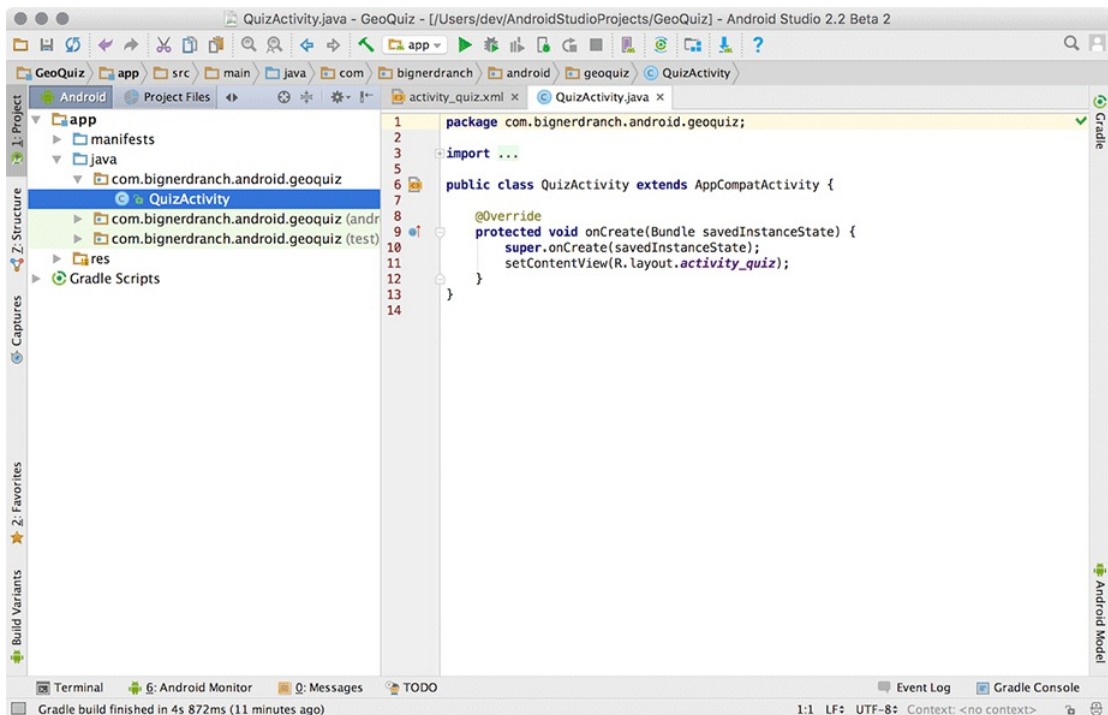
Android Studio opens your project in a window, as shown in Figure 1.8.

The different panes of the project window are called *tool windows*.

The lefthand view is the *project tool window*. From here, you can view and manage the files associated with your project.

The main view is the *editor*. To get you started, Android Studio has opened `QuizActivity.java` in the editor.

Figure 1.8 A fresh project window



You can toggle the visibility of the various tool windows by clicking on their names in the strips of tool buttons on the left, right, and bottom of the screen. There are keyboard shortcuts for many of these as well. If you do not see the tool button strips, click the gray square button in the lower-left corner of the main window or choose `View → Tool Buttons`.

Laying Out the UI

Open `app/res/layout/activity_quiz.xml`. If you see a graphical preview of the file, select the Text tab at the bottom to see the backing XML.

Currently, `activity_quiz.xml` defines the default activity layout. The defaults change frequently, but the XML will look something like Listing 1.1.

Listing 1.1 Default activity layout (`activity_quiz.xml`)

```
<RelativeLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/activity_quiz"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingBottom="16dp"
    android:paddingLeft="16dp"
    android:paddingRight="16dp"
    android:paddingTop="16dp"
    tools:context="com.bignerdranch.android.geoquiz.QuizActivity">

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

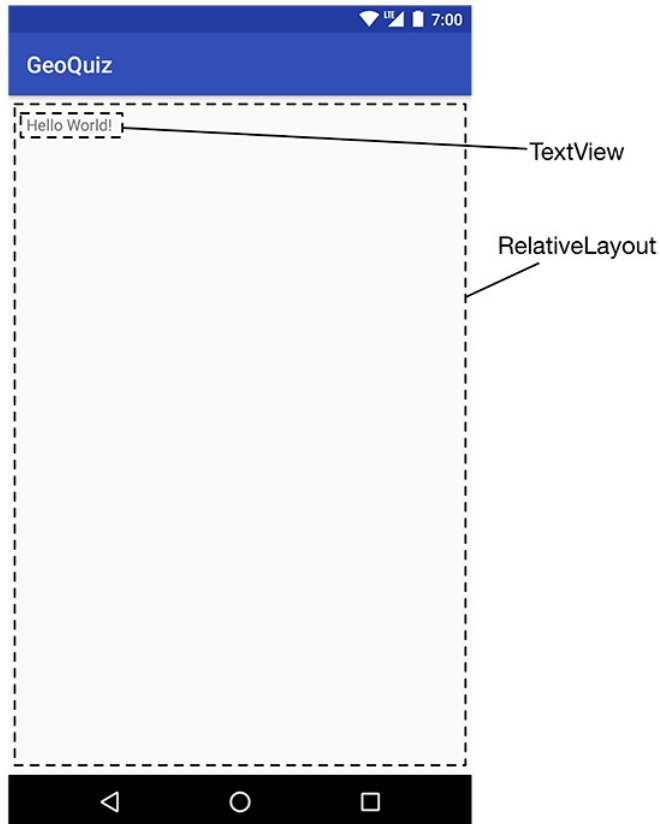
The default activity layout defines two *widgets*: a **RelativeLayout** and a **TextView**.

Widgets are the building blocks you use to compose a UI. A widget can show text or graphics, interact with the user, or arrange other widgets on the screen. Buttons, text input controls, and checkboxes are all types of widgets.

The Android SDK includes many widgets that you can configure to get the appearance and behavior you want. Every widget is an instance of the **View** class or one of its subclasses (such as **TextView** or **Button**).

Figure 1.9 shows how the **RelativeLayout** and **TextView** defined in Listing 1.1 would appear on screen.

Figure 1.9 Default widgets as seen on screen

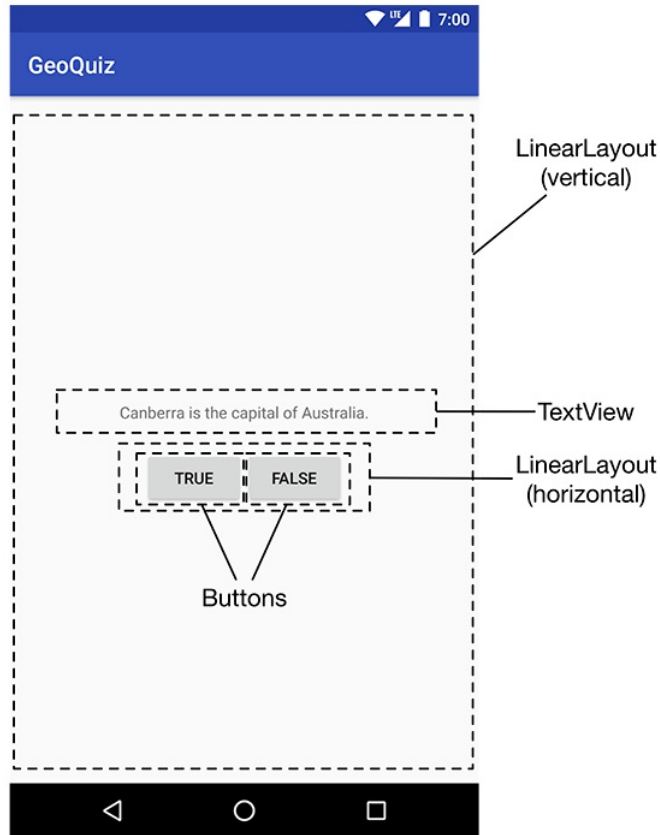


But these are not the widgets you are looking for. The interface for **QuizActivity** requires five widgets:

- a vertical **LinearLayout**
- a **TextView**
- a horizontal **LinearLayout**
- two **Buttons**

Figure 1.10 shows how these widgets compose **QuizActivity**'s interface.

Figure 1.10 Planned widgets as seen on screen



Now you need to define these widgets in `activity_quiz.xml`.

In the project tool window, find the `app/res/layout` directory, reveal its contents, and open `activity_quiz.xml`. Make the changes shown in Listing 1.2. The XML that you need to delete is struck through, and the XML that you need to add is in bold font. This is the pattern we will use throughout this book.

Do not worry about understanding what you are typing; you will learn how it works next. However, do be careful. Layout XML is not validated, and typos will cause problems sooner or later.

You will see errors on the three lines that start with `android:text`. Ignore these errors for now; you will fix them soon.

Listing 1.2 Defining widgets in XML (`activity_quiz.xml`)

```
<RelativeLayout
    android:id="@id/activity_quiz"
    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:paddingBottom="@dimen/activity_vertical_margin"
    android:paddingLeft="@dimen/activity_horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin"
    tools:context="com.bignerdranch.android.geoquiz.QuizActivity">

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

<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:gravity="center"
    android:orientation="vertical" >

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:padding="24dp"
        android:text="@string/question_text" />

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

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

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

    </LinearLayout>

</LinearLayout>
```

Compare your XML with the UI shown in Figure 1.10. Every widget has a corresponding XML element, and the name of the element is the type of the widget.

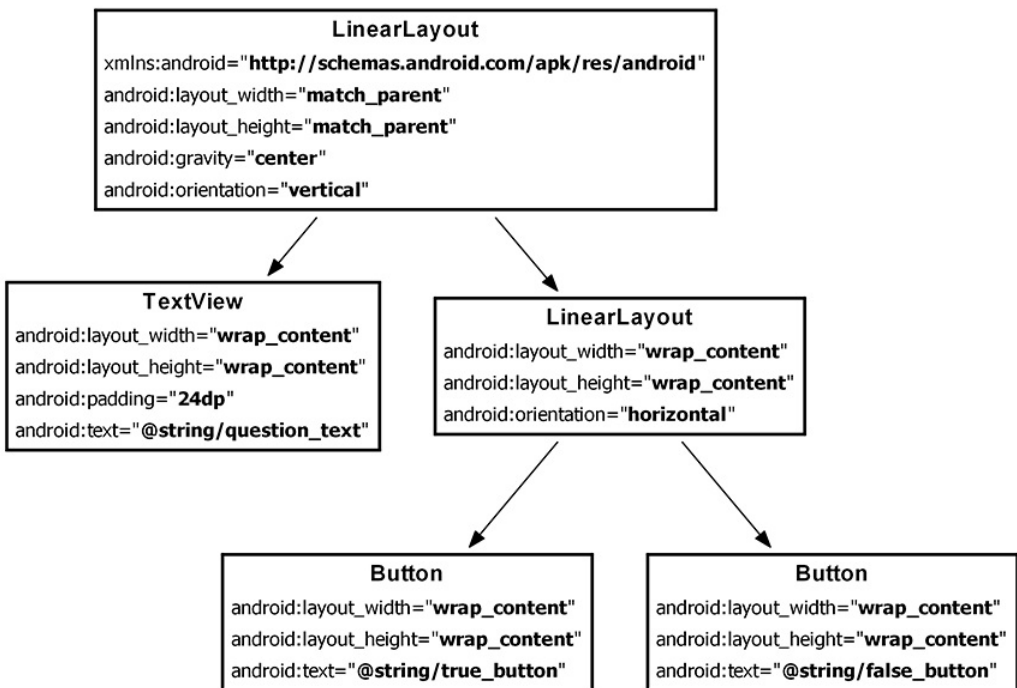
Each element has a set of XML *attributes*. Each *attribute* is an instruction about how the widget should be configured.

To understand how the elements and attributes work, it helps to look at the layout from a hierarchical perspective.

The view hierarchy

Your widgets exist in a hierarchy of **View** objects called the *view hierarchy*. Figure 1.11 shows the view hierarchy that corresponds to the XML in Listing 1.2.

Figure 1.11 Hierarchical layout of widgets and attributes



The root element of this layout's view hierarchy is a **LinearLayout**. As the root element, the **LinearLayout** must specify the Android resource XML namespace at <http://schemas.android.com/apk/res/android>.

LinearLayout inherits from a subclass of **View** named **ViewGroup**. A **ViewGroup** is a widget that contains and arranges other widgets. You use a **LinearLayout** when you want widgets arranged in a single column or row. Other **ViewGroup** subclasses are **FrameLayout**, **TableLayout**, and **RelativeLayout**.

When a widget is contained by a **ViewGroup**, that widget is said to be a *child* of the **ViewGroup**. The root **LinearLayout** has two children: a **TextView** and another **LinearLayout**. The child **LinearLayout** has two **Button** children of its own.

Widget attributes

Let's go over some of the attributes that you have used to configure your widgets.

android:layout_width and **android:layout_height**

The `android:layout_width` and `android:layout_height` attributes are required for almost every type of widget. They are typically set to either `match_parent` or `wrap_content`:

`match_parent` view will be as big as its parent

`wrap_content` view will be as big as its contents require

(You may see `fill_parent` in some places. This deprecated value is equivalent to `match_parent`.)

For the root **LinearLayout**, the value of both the height and width attributes is `match_parent`. The **LinearLayout** is the root element, but it still has a parent – the view that Android provides for your app's view hierarchy to live in.

The other widgets in your layout have their widths and heights set to `wrap_content`. You can see in Figure 1.10 how this determines their sizes.

The **TextView** is slightly larger than the text it contains due to its `android:padding="24dp"` attribute. This attribute tells the widget to add the specified amount of space to its contents when determining its size. You are using it to get a little breathing room between the question and the buttons. (Wondering about the `dp` units? These are density-independent pixels, which you will learn about in Chapter 9.)

android:orientation

The `android:orientation` attribute on the two **LinearLayout** widgets determines whether their children will appear vertically or horizontally. The root **LinearLayout** is vertical; its child **LinearLayout** is horizontal.

The order in which children are defined determines the order in which they appear on screen. In a vertical **LinearLayout**, the first child defined will appear topmost. In a horizontal **LinearLayout**, the first child defined will be leftmost. (Unless the device is set to a language that runs right to left, such as Arabic or Hebrew. In that case, the first child will be rightmost.)

android:text

The **TextView** and **Button** widgets have `android:text` attributes. This attribute tells the widget what text to display.

Notice that the values of these attributes are not literal strings. They are references to *string resources*.

A *string resource* is a string that lives in a separate XML file called a *strings file*. You can give a widget a hardcoded string, like `android:text="True"`, but it is usually not a good idea. Placing strings into a separate file and then referencing them is better because it makes localization easy.

The string resources you are referencing in `activity_quiz.xml` do not exist yet. Let's fix that.

Creating string resources

Every project includes a default strings file named `strings.xml`.

Open `res/values/strings.xml`. The template has already added one string resource for you. Add the three new strings that your layout requires.

Listing 1.3 Adding string resources (`strings.xml`)

```
<resources>
    <string name="app_name">GeoQuiz</string>
    <string name="question_text">Canberra is the capital of Australia.</string>
    <string name="true_button">True</string>
    <string name="false_button">False</string>
</resources>
```

(Depending on your version of Android Studio, you may have additional strings. Do not delete them. Deleting them could cause cascading errors in other files.)

Now, whenever you refer to `@string/false_button` in any XML file in the GeoQuiz project, you will get the literal string “False” at runtime.

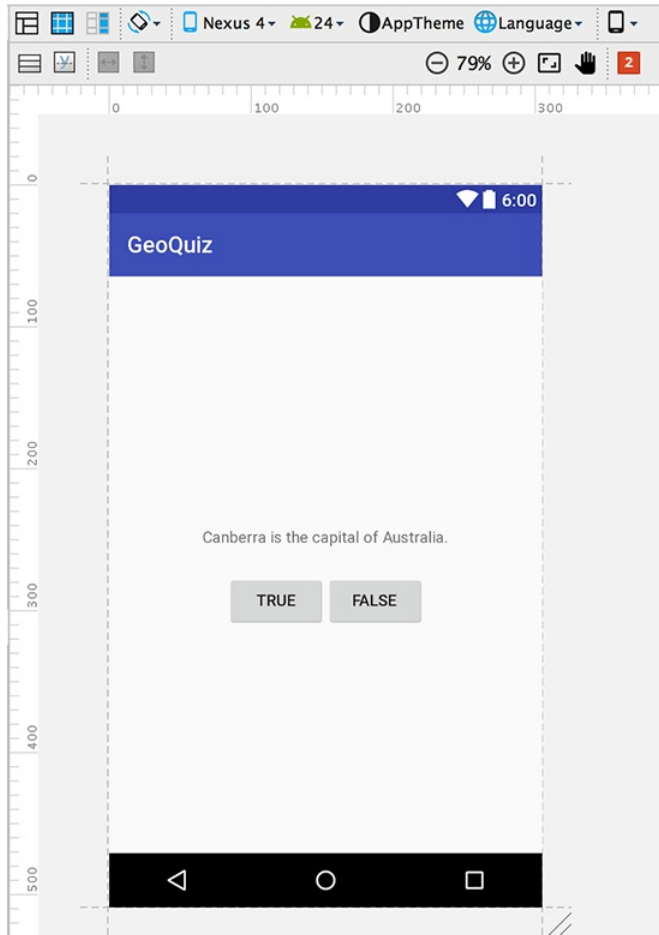
If you had errors in `activity_quiz.xml` about the missing string resources, they should now be gone. (If you still have errors, check both files for typos.)

Although the default strings file is named `strings.xml`, you can name a strings file anything you want. You can also have multiple strings files in a project. As long as the file is located in `res/values/`, has a `resources` root element, and contains child `string` elements, your strings will be found and used.

Previewing the layout

Your layout is now complete, and you can preview the layout in the graphical layout tool (Figure 1.12). First, make sure that your files are error free. Then return to `activity_quiz.xml` and open the preview tool window (if it is not already open) using the tab to the right of the editor.

Figure 1.12 Previewing `activity_quiz.xml` in graphical layout tool



From Layout XML to View Objects

How do XML elements in `activity_quiz.xml` become **View** objects? The answer starts in the **QuizActivity** class.

When you created the GeoQuiz project, a subclass of **Activity** named **QuizActivity** was created for you. The class file for **QuizActivity** is in the `app/java` directory of your project. The `java` directory is where the Java code for your project lives.

In the project tool window, reveal the contents of the `app/java` directory and then the contents of the `com.bignerdranch.android.geoquiz` package. Open the `QuizActivity.java` file and take a look at its contents.

Listing 1.4 Default class file for **QuizActivity** (`QuizActivity.java`)

```
package com.bignerdranch.android.geoquiz;

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

public class QuizActivity extends AppCompatActivity {

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

(Wondering what **AppCompatActivity** is? It is a subclass of Android's **Activity** class that provides compatibility support for older versions of Android. You will learn much more about **AppCompatActivity** in Chapter 13.)

If you are not seeing all of the import statements, click the symbol to the left of the first import statement to reveal the others.

This file has one **Activity** method: **onCreate(Bundle)**.

(If your file has **onOptionsItemSelected(Menu)** and **onOptionsItemSelected(MenuItem)** methods, ignore them for now. You will return to menus in detail in Chapter 13.)

The **onCreate(Bundle)** method is called when an instance of the activity subclass is created. When an activity is created, it needs a UI to manage. To get the activity its UI, you call the following **Activity** method:

```
public void setContentView(int layoutResID)
```

This method *inflates* a layout and puts it on screen. When a layout is inflated, each widget in the layout file is instantiated as defined by its attributes. You specify which layout to inflate by passing in the layout's resource ID.

Resources and resource IDs

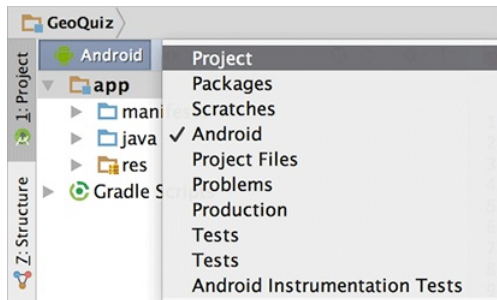
A layout is a *resource*. A *resource* is a piece of your application that is not code – things like image files, audio files, and XML files.

Resources for your project live in a subdirectory of the `app/res` directory. In the project tool window, you can see that `activity_quiz.xml` lives in `res/layout/`. Your strings file, which contains string resources, lives in `res/values/`.

To access a resource in code, you use its *resource ID*. The resource ID for your layout is `R.layout.activity_quiz`.

To see the current resource IDs for GeoQuiz, you must first change your project view. By default, Android Studio uses the Android project view (Figure 1.13). This view hides the true directory structure of your Android project so that you can focus on the files and folders that you need most often.

Figure 1.13 Changing the project view



Locate the dropdown at the top of the project tool window and change from the Android view to the Project view. The Project view will show you the files and folders in your project as they actually are.

To see the resources for GeoQuiz, reveal the contents of the `app/build/generated/source/r/debug` directory. In this directory, find your project’s package name and open `R.java` within that package. Because this file is generated by the Android build process, you should not change it, as you are subtly warned at the top of the file.

After making a change to your resources, you may not see this file instantly update. Android Studio maintains a hidden `R.java` that your code builds against. The `R.java` file in Listing 1.5 is the one that is generated for your app just before it is installed on a device or emulator. You will see this file update when you run your app.

Listing 1.5 Current GeoQuiz resource IDs (R.java)

```

/* AUTO-GENERATED FILE. DO NOT MODIFY.
 *
 * This class was automatically generated by the
 * aapt tool from the resource data it found. It
 * should not be modified by hand.
 */

package com.bignerdranch.android.geoquiz;

public final class R {
    public static final class anim {
        ...
    }

    ...

    public static final class id {
        ...
    }
    public static final class layout {
        ...
        public static final int activity_quiz=0x7f030017;
    }
    public static final class mipmap {
        public static final int ic_launcher=0x7f030000;
    }
    public static final class string {
        ...
        public static final int app_name=0x7f0a0010;
        public static final int false_button=0x7f0a0012;
        public static final int question_text=0x7f0a0014;
        public static final int true_button=0x7f0a0015;
    }
}

```

The R.java file can be large, and much of this file is omitted from Listing 1.5.

This is where the R.layout.activity_quiz comes from – it is an integer constant named activity_quiz within the **layout** inner class of **R**.

Your strings also have resource IDs. You have not yet referred to a string in code, but if you did, it would look like this:

```
setTitle(R.string.app_name);
```

Android generated a resource ID for the entire layout and for each string, but it did not generate IDs for the individual widgets in `activity_quiz.xml`. Not every widget needs a resource ID. In this chapter, you will only interact with the two buttons in code, so only they need resource IDs.

Before generating the resource IDs, switch back to the Android project view. Throughout this book, the Android project view will be used – but feel free to use the Project version if you prefer.

To generate a resource ID for a widget, you include an `android:id` attribute in the widget's definition. In `activity_quiz.xml`, add an `android:id` attribute to each button.

Listing 1.6 Adding IDs to **Buttons** (`activity_quiz.xml`)

```
<LinearLayout ... >

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:padding="24dp"
        android:text="@string/question_text" />

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

        <Button
            android:id="@+id/true_button"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="@string/true_button" />

        <Button
            android:id="@+id/false_button"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="@string/false_button" />

    </LinearLayout>

</LinearLayout>
```

Notice that there is a + sign in the values for `android:id` but not in the values for `android:text`. This is because you are *creating* the IDs and only *referencing* the strings.

Wiring Up Widgets

Now that the buttons have resource IDs, you can access them in **QuizActivity**. The first step is to add two member variables.

Type the following code into `QuizActivity.java`. (Do not use code completion; type it in yourself.) After you save the file, it will report two errors.

Listing 1.7 Adding member variables (`QuizActivity.java`)

```
public class QuizActivity extends AppCompatActivity {  
  
    private Button mTrueButton;  
    private Button mFalseButton;  
  
    @Override  
    protected void onCreate(Bundle savedInstanceState) {  
        super.onCreate(savedInstanceState);  
        setContentView(R.layout.activity_quiz);  
    }  
}
```

You will fix the errors in just a second. First, notice the `m` prefix on the two member (instance) variable names. This prefix is an Android naming convention that we will follow throughout this book.

Now mouse over the red error indicators. They report the same problem: Cannot resolve symbol 'Button'.

These errors are telling you that you need to import the **android.widget.Button** class into `QuizActivity.java`. You could type the following import statement at the top of the file:

```
import android.widget.Button;
```

Or you can do it the easy way and let Android Studio do it for you. Just press Option+Return (or Alt+Enter) to let the IntelliJ magic under the hood amaze you. The new import statement now appears with the others at the top of the file. This shortcut is generally useful when something is not correct with your code. Try it often!

This should get rid of the errors. (If you still have errors, check for typos in your code and XML.)

Now you can wire up your button widgets. This is a two-step process:

- get references to the inflated **View** objects
- set listeners on those objects to respond to user actions

Getting references to widgets

In an activity, you can get a reference to an inflated widget by calling the following **Activity** method:

```
public View findViewById(int id)
```

This method accepts a resource ID of a widget and returns a **View** object.

In `QuizActivity.java`, use the resource IDs of your buttons to retrieve the inflated objects and assign them to your member variables. Note that you must cast the returned **View** to **Button** before assigning it.

Listing 1.8 Getting references to widgets (`QuizActivity.java`)

```
public class QuizActivity extends AppCompatActivity {  
  
    private Button mTrueButton;  
    private Button mFalseButton;  
  
    @Override  
    protected void onCreate(Bundle savedInstanceState) {  
        super.onCreate(savedInstanceState);  
        setContentView(R.layout.activity_quiz);  
  
        mTrueButton = (Button) findViewById(R.id.true_button);  
        mFalseButton = (Button) findViewById(R.id.false_button);  
    }  
}
```

Setting listeners

Android applications are typically *event driven*. Unlike command-line programs or scripts, event-driven applications start and then wait for an event, such as the user pressing a button. (Events can also be initiated by the OS or another application, but user-initiated events are the most obvious.)

When your application is waiting for a specific event, we say that it is “listening for” that event. The object that you create to respond to an event is called a *listener*, and the *listener* implements a *listener interface* for that event.

The Android SDK comes with listener interfaces for various events, so you do not have to write your own. In this case, the event you want to listen for is a button being pressed (or “clicked”), so your listener will implement the **View.OnClickListener** interface.

Start with the TRUE button. In `QuizActivity.java`, add the following code to **onCreate(Bundle)** just after the variable assignment.

Listing 1.9 Setting a listener for the TRUE button (`QuizActivity.java`)

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

    mTrueButton = (Button) findViewById(R.id.true_button);
    mTrueButton.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View v) {
            // Does nothing yet, but soon!
        }
    });

    mFalseButton = (Button) findViewById(R.id.false_button);
}
```

(If you have a `View` cannot be resolved to a type error, try using Option+Return (Alt+Enter) to import the **View** class.)

In Listing 1.9, you set a listener to inform you when the **Button** known as `mTrueButton` has been pressed. The **setOnClickListener(OnClickListener)** method takes a listener as its argument. In particular, it takes an object that implements **OnClickListener**.

Using anonymous inner classes

This listener is implemented as an *anonymous inner class*. The syntax is a little tricky, but it helps to remember that everything within the outermost set of parentheses is passed into **setOnClickListener(OnClickListener)**. Within these parentheses, you create a new, nameless class and pass its entire implementation.

```
mTrueButton.setOnClickListener(new View.OnClickListener() {  
    @Override  
    public void onClick(View v) {  
        // Does nothing yet, but soon!  
    }  
});
```

All of the listeners in this book will be implemented as anonymous inner classes. Doing so puts the implementations of the listeners' methods right where you want to see them. And there is no need for the overhead of a named class because the class will be used in one place only.

Because your anonymous class implements **OnClickListener**, it must implement that interface's sole method, **onClick(View)**. You have left the implementation of **onClick(View)** empty for now, and the compiler is OK with that. A listener interface requires you to implement **onClick(View)**, but it makes no rules about *how* you implement it.

(If your knowledge of anonymous inner classes, listeners, or interfaces is rusty, you may want to review some Java before continuing or at least keep a reference nearby.)

Set a similar listener for the FALSE button.

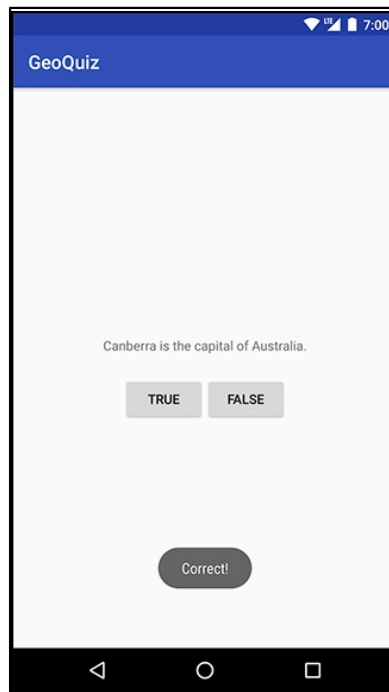
Listing 1.10 Setting a listener for the FALSE button (QuizActivity.java)

```
mTrueButton.setOnClickListener(new View.OnClickListener() {  
    @Override  
    public void onClick(View v) {  
        // Does nothing yet, but soon!  
    }  
});  
  
mFalseButton = (Button) findViewById(R.id.false_button);  
mFalseButton.setOnClickListener(new View.OnClickListener() {  
    @Override  
    public void onClick(View v) {  
        // Does nothing yet, but soon!  
    }  
});  
}
```


Making Toasts

Now to make the buttons fully armed and operational. You are going to have a press of each button trigger a pop-up message called a *toast*. A *toast* is a short message that informs the user of something but does not require any input or action. You are going to make toasts that announce whether the user answered correctly or incorrectly (Figure 1.14).

Figure 1.14 A toast providing feedback



First, return to `strings.xml` and add the string resources that your toasts will display.

Listing 1.11 Adding toast strings (`strings.xml`)

```
<resources>
    <string name="app_name">GeoQuiz</string>
    <string name="question_text">Canberra is the capital of Australia.</string>
    <string name="true_button">True</string>
    <string name="false_button">False</string>
    <string name="correct_toast">Correct!</string>
    <string name="incorrect_toast">Incorrect!</string>
</resources>
```

To create a toast, you call the following method from the **Toast** class:

```
public static Toast makeText(Context context, int resId, int duration)
```

The **Context** parameter is typically an instance of **Activity** (**Activity** is a subclass of **Context**). The second parameter is the resource ID of the string that the toast should display. The **Context** is needed by the **Toast** class to be able to find and use the string's resource ID. The third parameter is one of two **Toast** constants that specify how long the toast should be visible.

After you have created a toast, you call **Toast.show()** on it to get it on screen.

In **QuizActivity**, you are going to call **makeText(...)** in each button's listener. Instead of typing everything in, try using Android Studio's code completion feature to add these calls.

Using code completion

Code completion can save you a lot of time, so it is good to become familiar with it early.

Start typing the code addition shown in Listing 1.12. When you get to the period after the **Toast** class, a pop-up window will appear with a list of suggested methods and constants from the **Toast** class.

To choose one of the suggestions, use the up and down arrow keys to select it. (If you wanted to ignore code completion, you could just keep typing. It will not complete anything for you if you do not press the Tab key, press the Return key, or click on the pop-up window.)

From the list of suggestions, select **makeText(Context context, int resID, int duration)**. Code completion will add the complete method call for you.

Fill in the parameters for the **makeText** method until you have added the code shown in Listing 1.12.

Listing 1.12 Making toasts (QuizActivity.java)

```
mTrueButton.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) {
        Toast.makeText(QuizActivity.this,
                       R.string.correct_toast,
                       Toast.LENGTH_SHORT).show();
        // Does nothing yet, but soon!
    }
});
mFalseButton = (Button) findViewById(R.id.false_button);
mFalseButton.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) {
        Toast.makeText(QuizActivity.this,
                       R.string.incorrect_toast,
                       Toast.LENGTH_SHORT).show();
        // Does nothing yet, but soon!
    }
});
```

In **makeText(...)**, you pass the instance of **QuizActivity** as the **Context** argument. However, you cannot simply pass the variable **this** as you might expect. At this point in the code, you are defining the anonymous class where **this** refers to the **View.OnClickListener**.

Because you used code completion, you do not have to do anything to import the **Toast** class. When you accept a code completion suggestion, the necessary classes are imported automatically.

Now, let's see your new app in action.

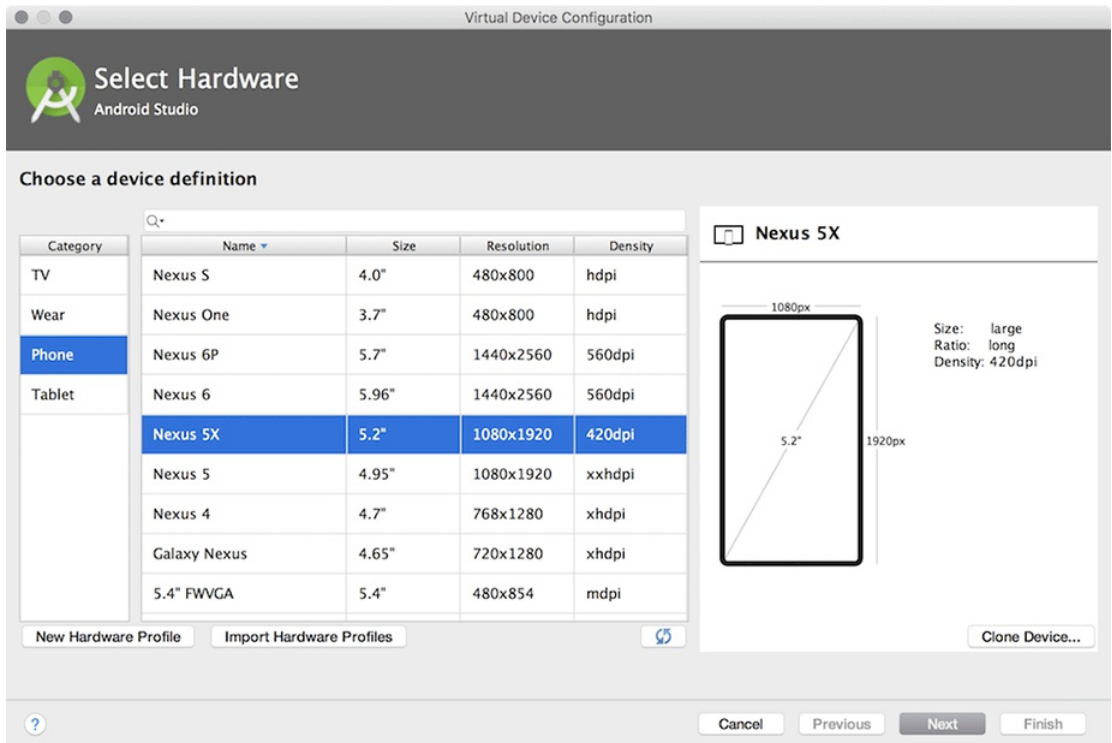
Running on the Emulator

To run an Android application, you need a device – either a hardware device or a *virtual device*. Virtual devices are powered by the Android emulator, which ships with the developer tools.

To create an Android virtual device (AVD), choose Tools → Android → AVD Manager. When the AVD Manager appears, click the +Create Virtual Device... button in the lower-left corner of the window.

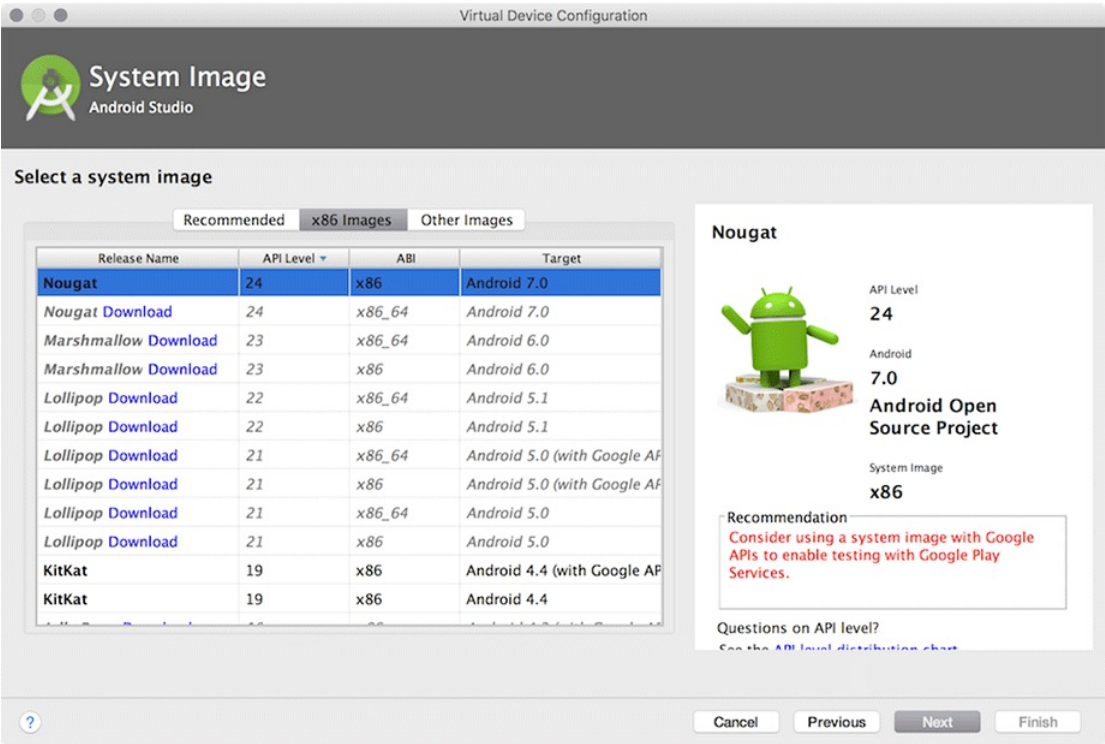
In the dialog that appears, you are offered many options for configuring a virtual device. For your first AVD, choose to emulate a Nexus 5X, as shown in Figure 1.15. Click Next.

Figure 1.15 Choosing a virtual device



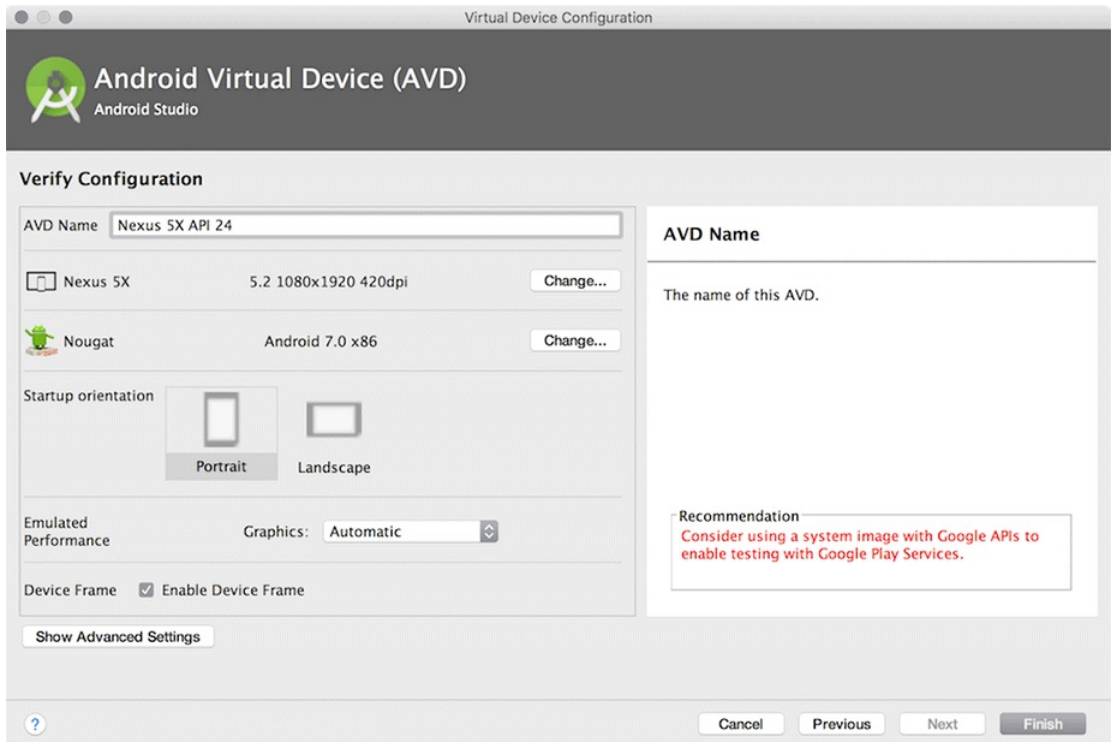
On the next screen, choose a system image that your emulator is based on. For this emulator, select an x86 Nougat emulator and select **Next** (Figure 1.16). (You may need to follow the steps to download the emulator’s components before you can click Next.)

Figure 1.16 Choosing a system image



Finally, you can review and tweak properties of the emulator. You can also edit the properties of an existing emulator later. For now, name your emulator something that will help you to identify it later and click **Finish** (Figure 1.17).

Figure 1.17 Updating emulator properties



Once you have an AVD, you can run GeoQuiz on it. From the Android Studio toolbar, click the run button (it looks like a green “play” symbol) or press Control+R. Android Studio will find the virtual device you created, start it, install the application package on it, and run the app.

Starting up the emulator can take a while, but eventually your GeoQuiz app will launch on the AVD that you created. Press buttons and admire your toasts.

If GeoQuiz crashes when launching or when you press a button, useful information will appear in the Logcat view in the Android DDMS tool window. (If Logcat did not open automatically when you ran GeoQuiz, you can open it by clicking the Android Monitor button at the bottom of the Android Studio window.) Look for exceptions in the log; they will be an eye-catching red color, as shown in Figure 1.18.

Figure 1.18 An example **NullPointerException** at line 21

```

Text
at dalvik.system.NativeStart.main(Native Method)
Caused by: java.lang.NullPointerException
at com.bignerdranch.android.geoquiz.QuizActivity.onCreate(QuizActivity.java:21)
at android.app.Activity.performCreate(Activity.java:5008)
at android.app.Instrumentation.callActivityOnCreate(Instrumentation.java:1079)

```

Compare your code with the code in the book to try to find the cause of the problem. Then try running again. (You will learn more about Logcat and debugging in the next two chapters.)

Keep the emulator running – you do not want to wait for it to launch on every run.

You can stop the app by pressing the Back button on the emulator. The Back button is shaped like a left-pointing triangle (on older versions of Android, it looks like an arrow that is making a U-turn). Then re-run the app from Android Studio to test changes.

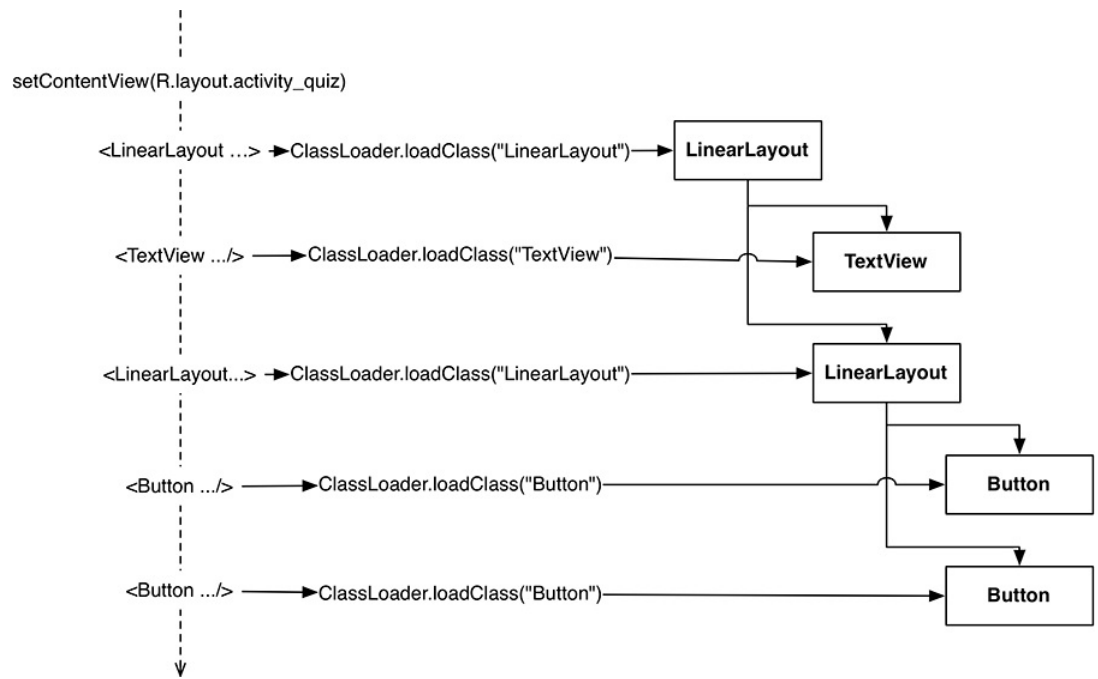
The emulator is useful, but testing on a real device gives more accurate results. In Chapter 2, you will run GeoQuiz on a hardware device. You will also give GeoQuiz more geography questions with which to test the user.

For the More Curious: Android Build Process

By now, you probably have some burning questions about how the Android build process works. You have already seen that Android Studio builds your project automatically as you modify it rather than on command. During the build process, the Android tools take your resources, code, and the `AndroidManifest.xml` file (which contains metadata about the application) and turn them into an `.apk` file. This file is then signed with a debug key, which allows it to run on the emulator. (To distribute your `.apk` to the masses, you have to sign it with a release key. There is more information about this process in the Android developer documentation at developer.android.com/tools/publishing/preparing.html.)

How do the contents of `activity_quiz.xml` turn into **View** objects in an application? As part of the build process, `aapt` (Android Asset Packaging Tool) compiles layout file resources into a more compact format. These compiled resources are packaged into the `.apk` file. Then, when `setContentView(...)` is called in the **QuizActivity**'s `onCreate(Bundle)` method, the **QuizActivity** uses the **LayoutInflater** class to instantiate each of the **View** objects as defined in the layout file (Figure 1.19).

Figure 1.19 Inflating `activity_quiz.xml`



(You can also create your view classes programmatically in the activity instead of defining them in XML. But there are benefits to separating your presentation from the logic of the application. The main one is taking advantage of configuration changes built into the SDK, which you will learn more about in Chapter 3.)

For more details on how the different XML attributes work and how views display themselves on the screen, see Chapter 9.

Android build tools

All of the builds you have seen so far have been executed from within Android Studio. This build is integrated into the IDE – it invokes standard Android build tools like `aapt`, but the build process itself is managed by Android Studio.

You may, for your own reasons, want to perform builds from outside of Android Studio. The easiest way to do this is to use a command-line build tool. The modern Android build system uses a tool called Gradle.

(You will know if this section applies to you. If it does not, feel free to read along but do not be concerned if you are not sure why you might want to do this or if the commands below do not seem to work. Coverage of the ins and outs of using the command line is beyond the scope of this book.)

To use Gradle from the command line, navigate to your project's directory and run the following command:

```
$ ./gradlew tasks
```

On Windows, your command will look a little different:

```
> gradlew.bat tasks
```

This will show you a list of available tasks you can execute. The one you want is called “`installDebug`”. Make it so with a command like this:

```
$ ./gradlew installDebug
```

Or, on Windows:

```
> gradlew.bat installDebug
```

This will install your app on whatever device is connected. However, it will not run the app. For that, you will need to pull up the launcher and launch the app by hand.

Challenges

Challenges are exercises at the end of the chapter for you to do on your own. Some are easy and provide practice doing the same thing you have done in the chapter. Other challenges are harder and require more problem solving.

We cannot encourage you enough to take on these challenges. Tackling them cements what you have learned, builds confidence in your skills, and bridges the gap between us teaching you Android programming and you being able to do Android programming on your own.

If you get stuck while working on a challenge, take a break and come back to try again fresh. If that does not help, check out the forum for this book at forums.bignerdranch.com. In the forum, you can review questions and solutions that other readers have posted as well as ask questions and post solutions of your own.

To protect the integrity of your current project, we recommend you make a copy and work on challenges in the new copy.

In your computer's file explorer, navigate to the root directory of your project. Copy the GeoQuiz folder and Paste a new copy next to the original (on macOS, use the Duplicate feature). Rename the new folder GeoQuiz Challenge. Back in Android Studio, select File → Import Project.... Inside the import window, navigate to GeoQuiz Challenge and select OK. The copied project will then appear in a new window ready for work.

Challenge: Customizing the Toast

In this challenge, you will customize the toast to show at the top instead of the bottom of the screen. To change how the toast is displayed, use the **Toast** class's **setGravity** method. Use **Gravity.TOP** for the gravity value. Refer to the developer documentation at [developer.android.com/reference/android/widget/Toast.html#setGravity\(int, int, int\)](http://developer.android.com/reference/android/widget/Toast.html#setGravity(int, int, int)) for more details.