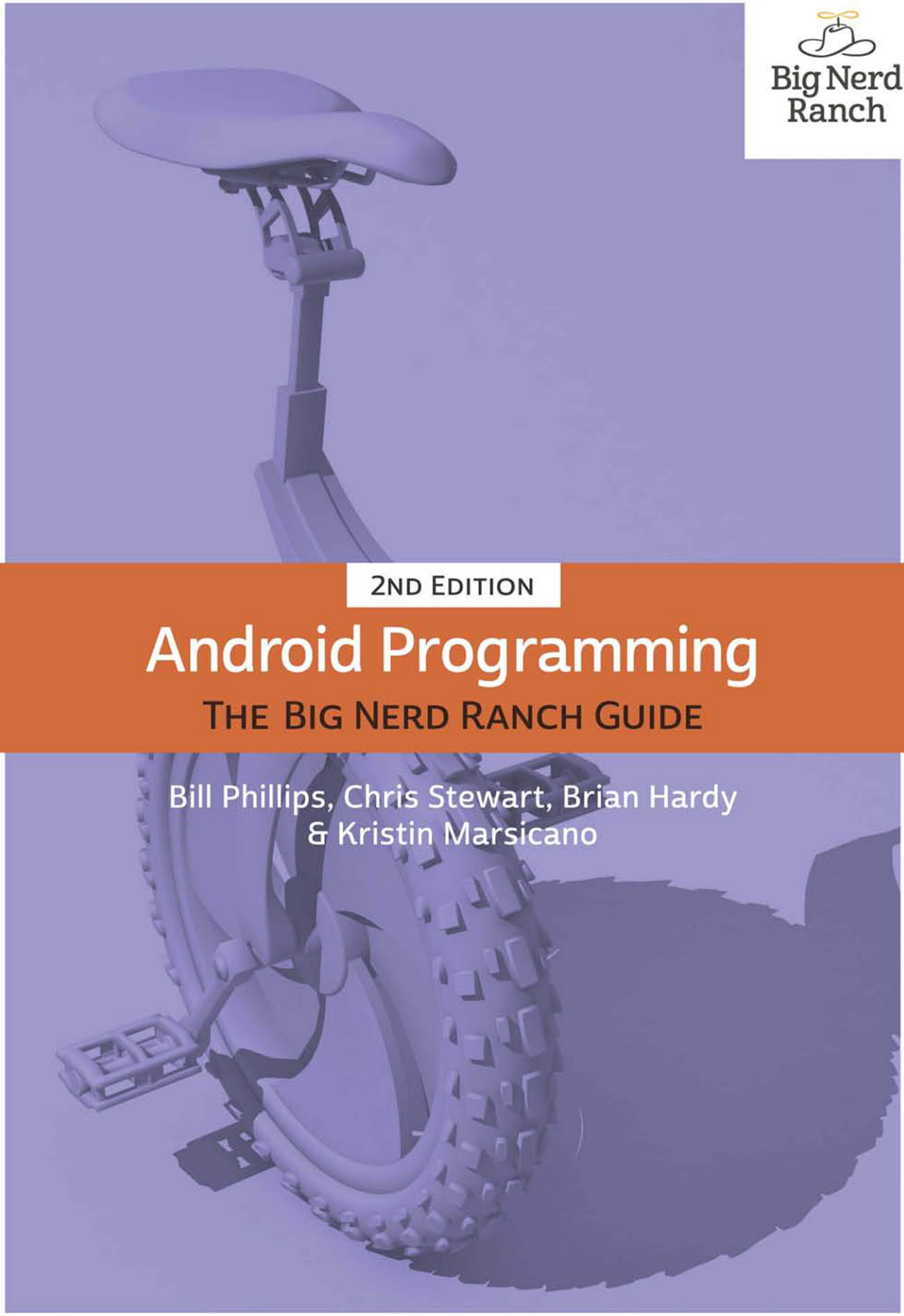


Big Nerd Ranch

2ND EDITION

Android Programming

THE BIG NERD RANCH GUIDE

Bill Phillips, Chris Stewart, Brian Hardy
& Kristin Marsicano

Android Programming: The Big Nerd Ranch Guide

by Bill Phillips, Chris Stewart, Brian Hardy and Kristin Marsicano

Big Nerd Ranch, LLC.
200 Arizona Ave NE
Atlanta, GA 30307
(770) 817-6373
http://www.bignerdranch.com/
book-comments@bignerdranch.com

The 10-gallon hat with propeller logo is a trademark of Big Nerd Ranch, Inc.

Exclusive worldwide distribution of the English edition of this book by

Pearson Technology Group
800 East 96th Street
Indianapolis, IN 46240 USA
http://www.informit.com

ISBN-10 0134171497
ISBN-13 978-0134171494

Second edition, first printing, August 2015
Release D.2.1.1

Dedication

To God, or to whatever it is that you personally have faith in. Reader, I hope that you find the many explanations in this book useful. Please don't ask me how they got here, though. I once thought that I was responsible. Fortunately for you, I was wrong.

— B.P.

To my dad, David, for teaching me the value of hard work. To my mom, Lisa, for pushing me to always do the right thing.

— C.S.

For Donovan. May he live a life filled with activities and know when to use fragments.

— B.H.

To my dad, Dave Vadas, for inspiring and encouraging me to pursue a career in computing. And to my mom, Joan Vadas, for cheering me on through all the ups and downs (and for reminding me that watching an episode of The Golden Girls always makes things better).

— K.M.

This page intentionally left blank

Acknowledgments

We feel a bit sheepish having our names on the cover of this book. The truth is that without an army of collaborators, this book could never have happened. We owe them all a debt of gratitude.

- Our co-instructors and members of our Android development team, Andrew Lunsford, Bolot Kerimbaev, Brian Gardner, David Greenhalgh, Jason Atwood, Josh Skeen, Kurt Nelson, Matt Compton, Paul Turner, and Sean Farrell. We thank them for their patience in teaching work-in-progress material, as well as their suggestions and corrections. If we could give ourselves additional brains to do with as we pleased, we would not. We would just put the new brains in a big pile, and share them with our colleagues. We trust them at least as much as we trust our own selves.

- Special thanks to Sean Farrell for graciously updating many screen shots as Android Studio evolved, and to Matt Compton for publishing all of our sample apps to the Google Play Store.

- Kar Loong Wong and Zack Simon, members of Big Nerd Ranch's amazing design team. Kar made BeatBox look intimidating and polished, and provided advice and imagery for the material design chapter. Zack took time out of his schedule to design MockWalker for us. Kar and Zack's design abilities seem like unknowable superpowers to us. We thank them, and bid them fond returns to their home planet.

- Our technical reviewers, Frank Robles and Roy Kravitz, who helped us find and fix flaws.

- Thanks to Aaron Hillegass. Aaron’s faith in people is one of the great and terrifying forces of nature. Without it, we would never have had the opportunity to write this book, nor would we ever have completed it. (He also gave us money, which was very friendly of him.)

- Our editor, Elizabeth Holaday, who many times saved us from going down rabbit holes. She kept our writing focused on what our readers actually care about and spared you all from confusing, boring, and irrelevant detours. Thank you, Liz, for being organized and patient, and for being a constant supportive presence, even though you live many miles away.

- Ellie Volckhausen, who designed our cover.

- Simone Payment, our copy-editor, who found and smoothed rough spots.

- Chris Loper at IntelligentEnglish.com, who designed and produced the print book and the EPUB and Kindle versions. His DocBook toolchain made life much easier, too.

Finally, thanks to our students. We wish that we had room to thank every single student who gave us a correction or opinion on the book as it was shaping up. It is your curiosity we have worked to satisfy, your confusions we have worked to clarify. Thank you.

This page intentionally left blank

Table of Contents

Learning Android

As a beginning Android programmer, you face a steep learning curve. Learning Android is like moving to a foreign city. Even if you speak the language, it will not feel like home at first. Everyone around you seems to understand things that you are missing. Things you already knew turn out to be dead wrong in this new context.

Android has a culture. That culture speaks Java, but knowing Java is not enough. Getting your head around Android requires learning many new ideas and techniques. It helps to have a guide through unfamiliar territory.

That's where we come in. At Big Nerd Ranch, we believe that to be an Android programmer, you must:

- *write* Android applications
- *understand* what you are writing

This guide will help you do both. We have trained hundreds of professional Android programmers using it. We lead you through writing several Android applications, introducing concepts and techniques as needed. When there are rough spots, when some things are tricky or obscure, you will face them head on, and we will do our best to explain why things are the way they are.

This approach allows you to put what you have learned into practice in a working app right away rather than learning a lot of theory and then having to figure out how to apply it all later. You will come away with the experience and understanding you need to get going as an Android developer.

Prerequisites

To use this book, you need to be familiar with Java, including classes and objects, interfaces, listeners, packages, inner classes, anonymous inner classes, and generic classes.

If these ideas do not ring a bell, you will be in the weeds by page 2. Start instead with an introductory Java book and return to this book afterward. There are many excellent introductory books available, so you can choose one based on your programming experience and learning style.

If you are comfortable with object-oriented programming concepts, but your Java is a little rusty, you will probably be OK. We will provide some brief reminders about Java specifics (like interfaces and anonymous inner classes). Keep a Java reference handy in case you need more support as you go through the book.

What's New in the Second Edition?

This second edition shows how to use the Android Studio integrated development environment to write practical applications for Android 5.1 (Lollipop) that are backwards-compatible through Android 4.1 (Jelly Bean). It includes updated coverage of the fundamentals of Android programming as well as new Lollipop tools like the toolbar and material design. It also covers new tools from the support libraries, like **`RecyclerView`** and Google Play Services, plus some key standard library tools, like **`SoundPool`**, animations, and assets.

How to Use This Book

This book is not a reference book. Its goal is to get you over the initial hump to where you can get the most out of the reference and recipe books available. It is based on our five-day class at Big Nerd Ranch. As such, it is meant to be worked through from the beginning. Chapters build on each other and skipping around is unproductive.

In our classes, students work through these materials, but they also benefit from the right environment – a dedicated classroom, good food and comfortable board, a group of motivated peers, and an instructor to answer questions.

As a reader, you want your environment to be similar. That means getting a good night's rest and finding a quiet place to work. These things can help, too:

- Start a reading group with your friends or coworkers.
- Arrange to have blocks of focused time to work on chapters.
- Participate in the forum for this book at `http://forums.bignerdranch.com`.
- Find someone who knows Android to help you out.

How This Book is Organized

As you work through this book, you will write eight Android apps. A couple are very simple and take only a chapter to create. Others are more complex. The longest app spans 11 chapters. All are designed to teach you important concepts and techniques and give you direct experience using them.

GeoQuiz	In your first app, you will explore the fundamentals of Android projects, activities, layouts, and explicit intents.
CriminalIntent	The largest app in the book, CriminalIntent lets you keep a record of your colleagues' lapses around the office. You will learn to use fragments, master-detail interfaces, list-backed interfaces, menus, the camera, implicit intents, and more.
BeatBox	Intimidate your foes with this app while you learn more about fragments, media playback, themes, and drawables.
NerdLauncher	Building this custom launcher will give you insight into the intent system and tasks.
PhotoGallery	A Flickr client that downloads and displays photos from Flickr's public feed, this app will take you through services, multithreading, accessing web services, and more.

DragAndDraw	In this simple drawing app, you will learn about handling touch events and creating custom views.
Sunset	In this toy app, you will create a beautiful representation of a sunset over open water while learning about animations.
Locatr	This app lets you query Flickr for pictures around your current location and display them on a map. In it, you will learn how to use location services and maps.

Challenges

Most chapters have a section at the end with exercises for you to work through. This is your opportunity to use what you have learned, explore the documentation, and do some problem solving on your own.

We strongly recommend that you do the challenges. Going off the beaten path and finding your way will solidify your learning and give you confidence with your own projects.

If you get lost, you can always visit `http://forums.bignerdranch.com` for some assistance.

Are you more curious?

There are also sections at the ends of chapters labeled "For the More Curious." These sections offer deeper explanations or additional information about topics presented in the chapter. The information in these sections is not absolutely essential, but we hope you will find it interesting and useful.

Code Style

There are two areas where our choices differ from what you might see elsewhere in the Android community:

We use anonymous inner classes for listeners.

> This is mostly a matter of opinion. We find it makes for cleaner code in the applications in this book because it puts the listener's method implementations right where you want to see them. In high-performance contexts or large applications, anonymous inner classes may cause problems, but for most circumstances they work fine.

After we introduce fragments in Chapter 7, we use them for all user interfaces.

> Fragments are not an absolutely necessary tool but we find that, when used correctly, they are a valuable tool in any Android developer's toolkit. Once you get comfortable with fragments, they are not that difficult to work with. Fragments have clear advantages over activities that make them worth the effort, including flexibility in building and presenting your user interfaces.

Typographical Conventions

To make this book easier to read, certain items appear in certain fonts. Variables, constants, and types appear in a fixed-width font. Class names, interface names, and method names appear in a bold, fixed-width font.

All code and XML listings are in a fixed-width font. Code or XML that you need to type in is always bold. Code or XML that should be deleted is struck through. For example, in the following method implementation, you are deleting the call to **`makeText(…)`** and adding the call to **`checkAnswer(true)`**.

```
@Override
public void onClick(View v) {
    ~~Toast.makeText(QuizActivity.this, R.string.incorrect_toast,~~
                    ~~Toast.LENGTH_SHORT).show();~~
    checkAnswer(true);
}
```

Android Versions

This book teaches Android development for all widely used versions of Android. As of this writing, that is Android 4.1 (Jelly Bean) - Android 5.1 (Lollipop). While there is a small amount of market-share on older versions of Android, we find that for most developers the amount of effort required to support those versions is not worth the reward. For more info on the support of versions of Android earlier than 4.1 (in particular, Android 2.2 and Android 2.3), see the first edition of this book.

As Android releases new versions, the techniques you learn in this book will continue to work thanks to Android's backwards compatibility support (see Chapter 6 for details). We will keep track of changes at `http://forums.bignerdranch.com` and offer notes on using this book with the latest version.

The Necessary Tools

To get started with this book, you will need Android Studio. Android Studio is an integrated development environment used for Android development that is based off of the popular IntelliJ IDEA.

An install of Android Studio includes:

Android SDK

: the latest version of the Android SDK

Android SDK tools and platform-tools

: tools for debugging and testing your apps

A system image for the Android emulator

: lets you create and test your apps on different virtual devices

As of this writing, Android Studio is under active development and is frequently updated. Be aware that you may find differences between your version of Android Studio and what you see in this book. Visit `http://forums.bignerdranch.com` for help with these differences.

Downloading and Installing Android Studio

Android Studio is available from Android's developer site at `https://developer.android.com/sdk/`.

If you do not already have it installed, you will need to install the Java Development Kit (JDK7), which you can download from `http://www.oracle.com`.

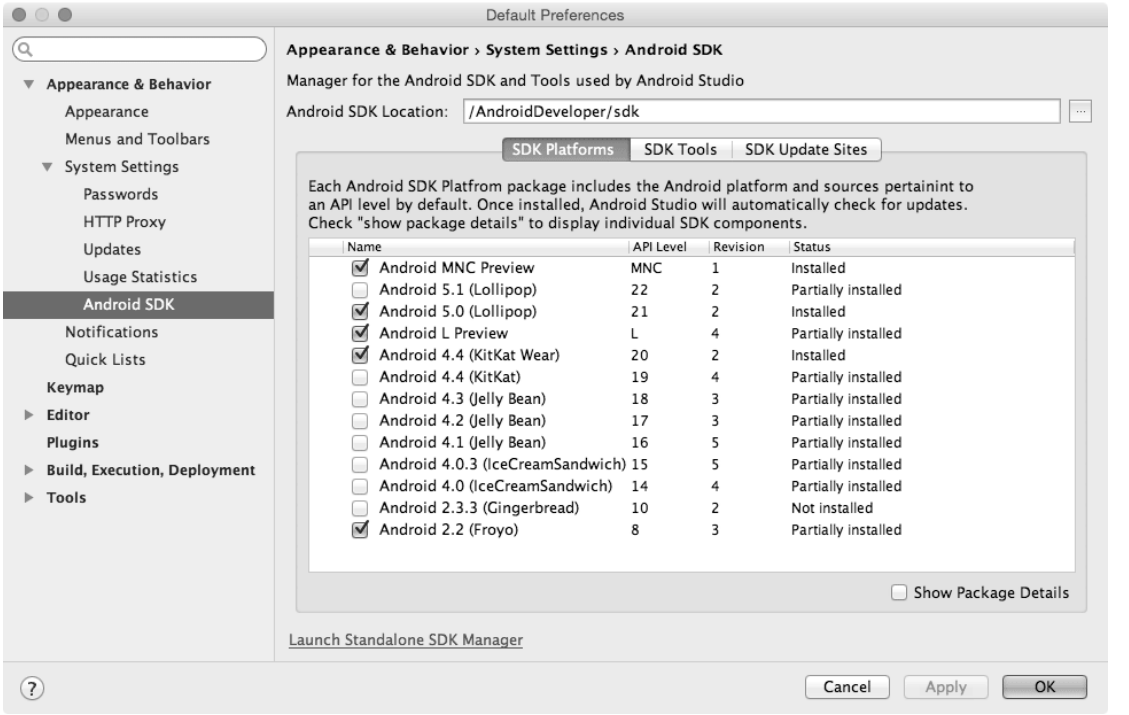
If you are still having problems, return to `https://developer.android.com/sdk/` for more information.

Downloading Earlier SDK Versions

Android Studio provides the SDK and the emulator system image from the latest platform. However, you may want to test your apps on earlier versions of Android.

You can get components for each platform using the Android SDK Manager. In Android Studio, select Tools → Android → SDK Manager. (You will only see the Tools menu if you have a project open. If you have not created a project yet, you can instead access the SDK Manager from the Android Setup Wizard screen. Under the Quick Start section, select Configure → SDK Manager, as shown in Figure 1.)

Figure 1 Android SDK Manager



Select and install each version of Android that you want to test with. Note that downloading these components may take a while.

The Android SDK Manager is also how to get Android's latest releases, like a new platform or an update of the tools.

An Alternative Emulator

The speed of the Android emulator has improved significantly over time and it is a reasonable way to run the code that you write in this book.

As an alternative, Genymotion is a popular, third-party Android emulator. You will occasionally see references to the Genymotion emulator in this book. For more information on Genymotion, visit `http://genymotion.com/`.

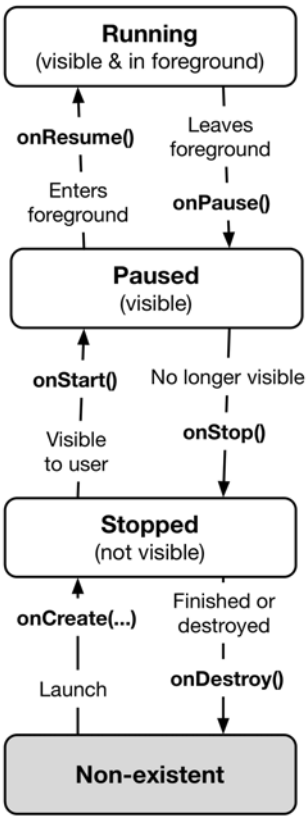
A Hardware Device

The emulator and Genymotion are useful for testing apps. However, they are no substitute for an actual Android device when measuring performance. If you have a hardware device, we recommend using that device at times when working through this book.

3
The Activity Lifecycle

Every instance of **`Activity`** has a lifecycle. During this lifecycle, an activity transitions between three states: running, paused, and stopped. For each transition, there is an **`Activity`** method that notifies the activity of the change in its state. Figure 3.1 shows the activity lifecycle, states, and methods.

Figure 3.1 Activity state diagram



Subclasses of **`Activity`** can take advantage of the methods named in Figure 3.1 to get work done at critical transitions in the activity's lifecycle.

You are already acquainted with one of these methods – **`onCreate(Bundle)`**. The OS calls this method after the activity instance is created but before it is put on screen.

Typically, an activity overrides **`onCreate(…)`** to prepare the specifics of its user interface:

- inflating widgets and putting them on screen (in the call to (**`setContentView(int)`**)
- getting references to inflated widgets
- setting listeners on widgets to handle user interaction
- connecting to external model data

It is important to understand that you never call **`onCreate(…)`** or any of the other **`Activity`** lifecycle methods yourself. You override them in your activity subclasses, and Android calls them at the appropriate time.

Logging the Activity Lifecycle

In this section, you are going to override lifecycle methods to eavesdrop on **`QuizActivity`**'s lifecycle. Each implementation will simply log a message informing you that the method has been called.

Making log messages

In Android, the **`android.util.Log`** class sends log messages to a shared system-level log. **`Log`** has several methods for logging messages. Here is the one that you will use most often in this book:

```
public static int d(String tag, String msg)
```

The **`d`** stands for "debug" and refers to the level of the log message. (There is more about the **`Log`** levels in the final section of this chapter.) The first parameter identifies the source of the message, and the second is the contents of the message.

The first string is typically a `TAG` constant with the class name as its value. This makes it easy to determine the source of a particular message.

In `QuizActivity.java`, add a `TAG` constant to **`QuizActivity`**:

Listing 3.1 Adding TAG constant (`QuizActivity.java`)

```
public class QuizActivity extends AppCompatActivity {

    private static final String TAG = "QuizActivity";

    ...

}
```

Next, in **`onCreate(…)`**, call **`Log.d(…)`** to log a message.

Listing 3.2 Adding log statement to **`onCreate(…)`** (`QuizActivity.java`)

```
public class QuizActivity extends AppCompatActivity {

    ...

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        Log.d(TAG, "onCreate(Bundle) called");
        setContentView(R.layout.activity_quiz);

        ...
    }
}
```

Now override five more methods in **`QuizActivity`** by adding the following after **`onCreate(Bundle)`** and before **`onCreateOptionsMenu(Menu)`**:

Listing 3.3 Overriding more lifecycle methods (`QuizActivity.java`)

```
    @Override
    public void onStart() {
        super.onStart();
        Log.d(TAG, "onStart() called");
    }

    @Override
    public void onPause() {
        super.onPause();
        Log.d(TAG, "onPause() called");
    }

    @Override
    public void onResume() {
        super.onResume();
        Log.d(TAG, "onResume() called");
    }

    @Override
    public void onStop() {
        super.onStop();
        Log.d(TAG, "onStop() called");
    }

    @Override
    public void onDestroy() {
        super.onDestroy();
        Log.d(TAG, "onDestroy() called");
    }

    ...
}
```

Notice that you call the superclass implementations before you log your messages. These superclass calls are required. Calling the superclass implementation before you do anything else is critical in **`onCreate(…)`**; the order is less important in the other methods.

You may have been wondering about the `@Override` annotation. This asks the compiler to ensure that the class actually has the method that you are attempting to override. For example, the compiler would be able to alert you to the following misspelled method name:

```
public class QuizActivity extends AppCompatActivity {

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

    ...
```

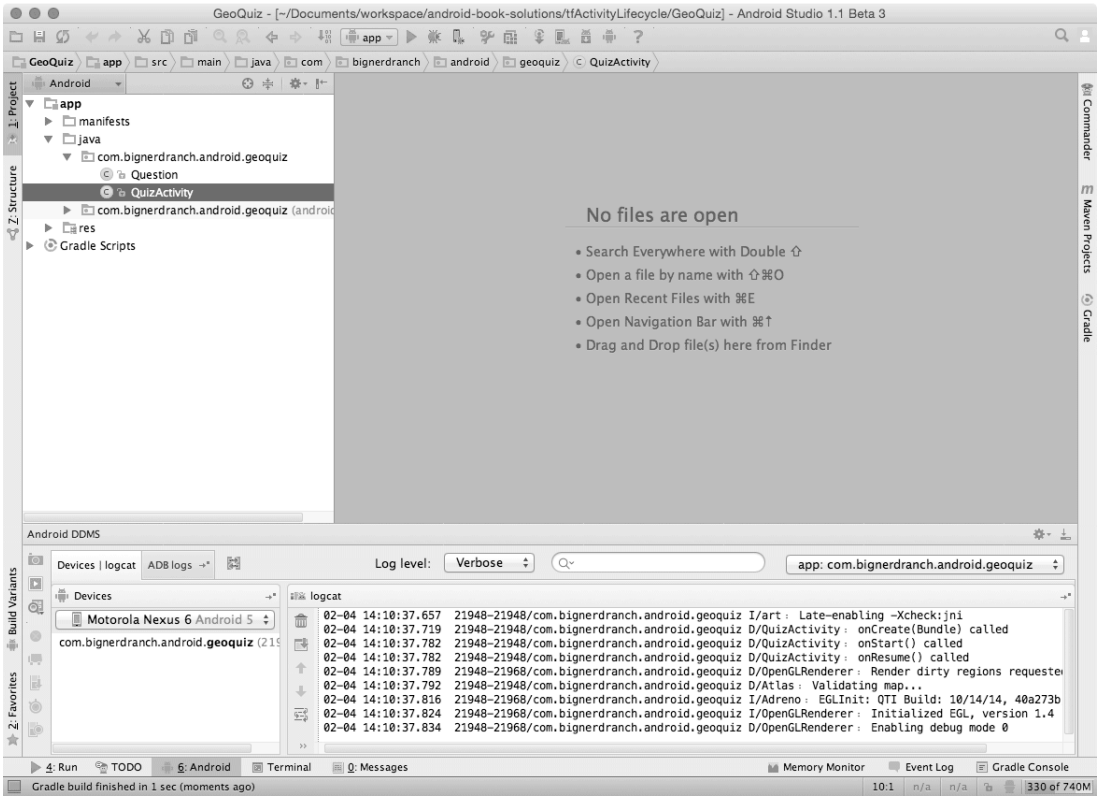
The **`Activity`** class does not have an **`onCreat(Bundle)`** method, so the compiler will complain. Then you can fix the typo rather than accidentally implementing **`QuizActivity.onCreat(Bundle)`**.

Using LogCat

To access the log while the application is running, you can use LogCat, a log viewer included in the Android SDK tools.

When you run GeoQuiz, you should see LogCat appear at the bottom of Android Studio, as shown in Figure 3.2. If LogCat is not visible, select the Android tool window near the bottom of the screen and ensure that the Devices | logcat tab is selected.

Figure 3.2 Android Studio with LogCat

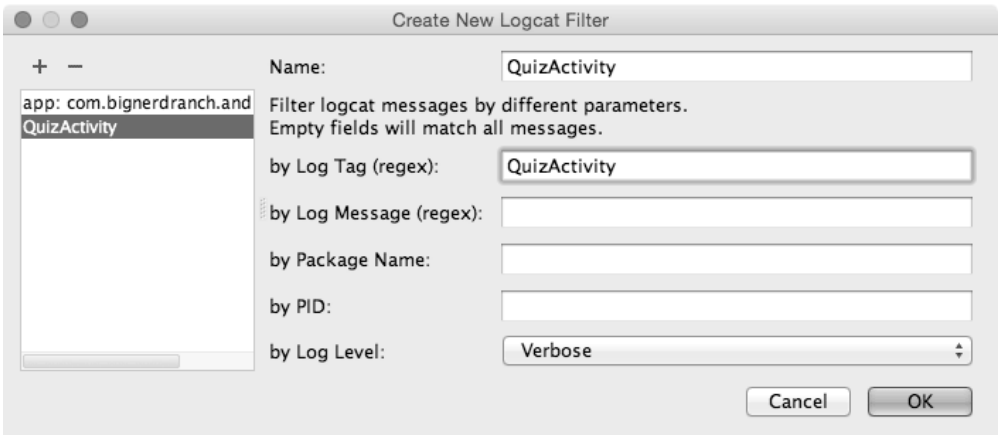


Run GeoQuiz and messages will start materializing in LogCat. By default, log statements that are generated with your app's package name are shown. You will see your own messages along with some system output.

To make your messages easier to find, you can filter the output using the `TAG` constant. In LogCat, click the filter drop-down box in the top right of the LogCat pane. Notice the existing filter, which is set up to show messages from only your app. Selecting No Filters will show log messages generated from all over the system.

In the filter dropdown, select Edit Filter Configuration. Use the + button to create a brand-new filter. Name the filter **`QuizActivity`** and enter **`QuizActivity`** in the by Log Tag: field (Figure 3.3).

Figure 3.3 Creating a filter in LogCat



Click OK, and only messages tagged **`QuizActivity`** will be visible (Figure 3.4).

Three lifecycle methods were called after GeoQuiz was launched and the initial instance of **`QuizActivity`** was created.

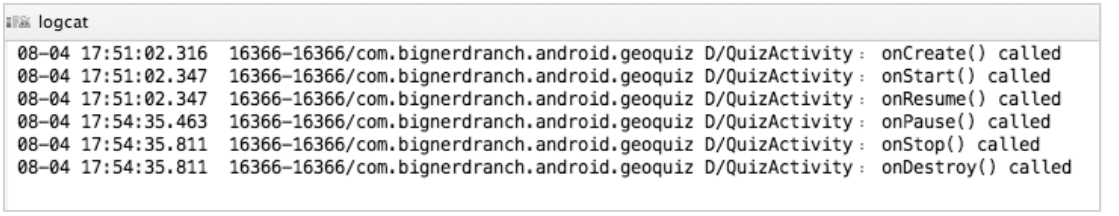
Figure 3.4 Launching GeoQuiz creates, starts, and resumes an activity

```
logcat
08-04 17:51:02.316  16366-16366/com.bignerdranch.android.geoquiz D/QuizActivity: onCreate() called
08-04 17:51:02.347  16366-16366/com.bignerdranch.android.geoquiz D/QuizActivity: onStart() called
08-04 17:51:02.347  16366-16366/com.bignerdranch.android.geoquiz D/QuizActivity: onResume() called
```

(If you are not seeing the filtered list, select the QuizActivity filter from LogCat's filter dropdown.)

Now let's have some fun. Press the Back button on the device and then check LogCat. Your activity received calls to **`onPause()`**, **`onStop()`**, and **`onDestroy()`** (Figure 3.5).

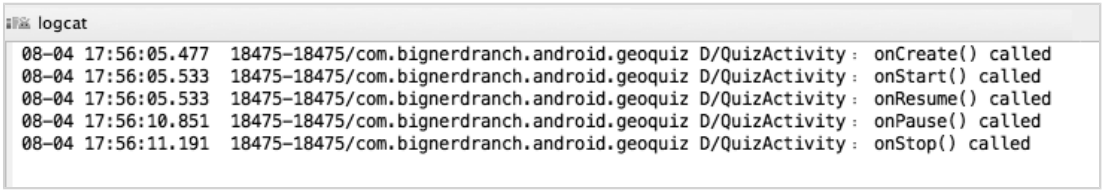
Figure 3.5 Pressing the Back button destroys the activity



When you pressed the Back button, you told Android, "I'm done with this activity, and I won't need it anymore." Android then destroyed your activity. This is Android's way of being frugal with your device's limited resources.

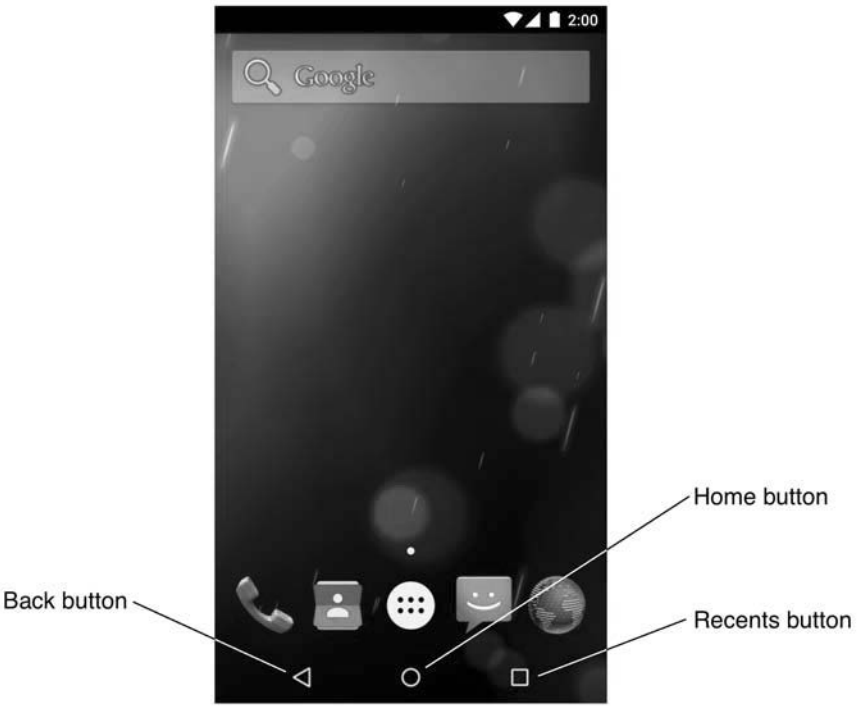
Relaunch GeoQuiz. Press the Home button and then check LogCat. Your activity received calls to **`onPause()`** and **`onStop()`**, but not **`onDestroy()`** (Figure 3.6).

Figure 3.6 Pressing the Home button stops the activity



On the device, pull up the task manager: On newer devices, press the Recents button next to the Home button (Figure 3.7). On devices without a Recents button, long-press the Home button.

Figure 3.7 Home, Back, and Recents buttons



In the task manager, press GeoQuiz and then check LogCat. The activity was started and resumed, but it did not need to be created.

Pressing the Home button tells Android, "I'm going to go look at something else, but I might come back." Android pauses and stops your activity but tries not to destroy it in case you come back.

However, a stopped activity's survival is not guaranteed. When the system needs to reclaim memory, it will destroy stopped activities.

Another situation that pauses an activity is when it is obscured from the user, such as by a pop-up window. Even if the window only partially covers the activity, the activity is paused and cannot be interacted with. The activity resumes when the pop-up window is dismissed.

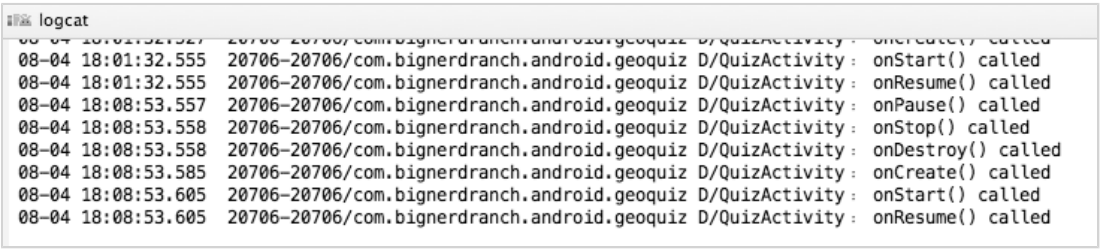
As you continue through the book, you will override the different activity lifecycle methods to do real things for your application. When you do, you will learn more about the uses of each method.

Rotation and the Activity Lifecycle

Let's get back to the bug you found at the end of Chapter 2. Run GeoQuiz, press the Next button to reveal the second question, and then rotate the device. (On the emulator, press Fn+Control+F12/Ctrl +F12 to rotate.)

After rotating, GeoQuiz will display the first question again. Check LogCat to see what has happened. Your output should look like Figure 3.8.

Figure 3.8 **QuizActivity** is dead. Long live **QuizActivity**!



When you rotated the device, the instance of **QuizActivity** that you were looking at was destroyed, and a new one was created. Rotate the device again to witness another round of destruction and rebirth.

This is the source of your bug. Each time a new **QuizActivity** is created, `mCurrentIndex` is initialized to `0`, and the user starts over at the first question. You will fix this bug in a moment. First, let's take a closer look at why this happens.

Device configurations and alternative resources

Rotating the device changes the *device configuration*. The *device configuration* is a set of characteristics that describe the current state of an individual device. The characteristics that make up the configuration include screen orientation, screen density, screen size, keyboard type, dock mode, language, and more.

Typically, applications provide alternative resources to match different device configurations. You saw an example of this when you added multiple arrow icons to your project for different screen densities.

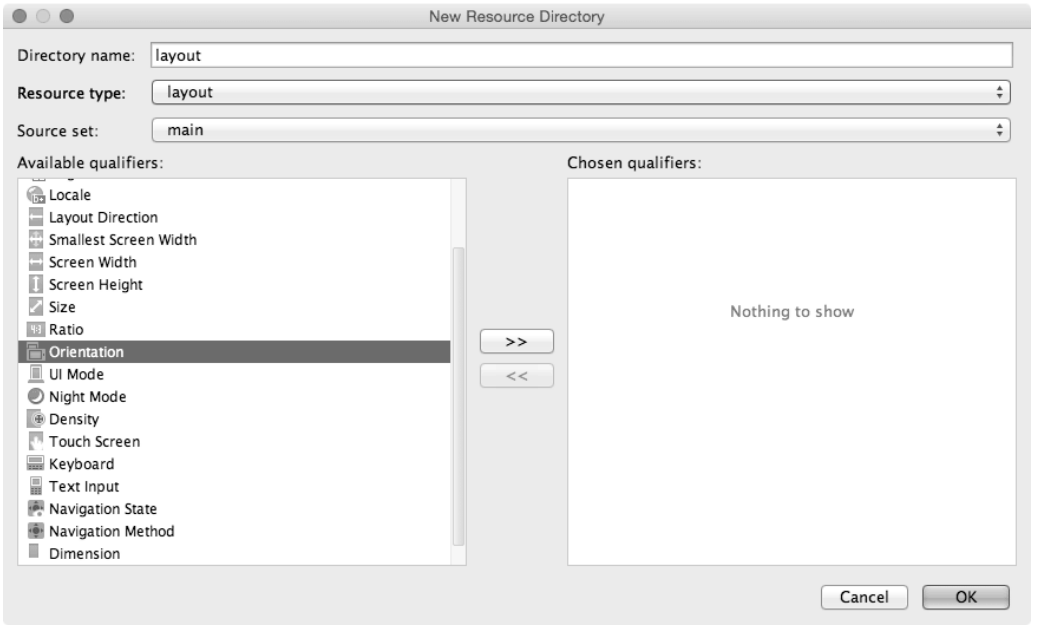
Screen density is a fixed component of the device configuration; it cannot change at runtime. On the other hand, some components, like screen orientation, *can* change at runtime.

When a *runtime configuration change* occurs, there may be resources that are a better match for the new configuration. To see this in action, let's create an alternative resource for Android to find and use when the device's screen orientation changes to landscape.

Creating a landscape layout

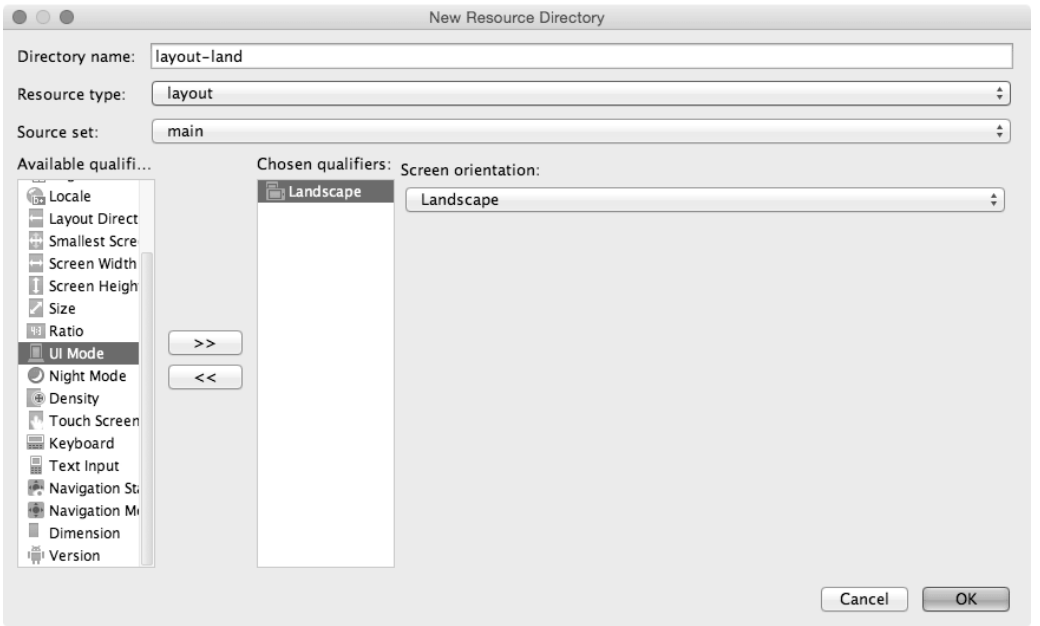
In the Project tool window, right-click the `res` directory and select New → Android resource directory. You should see a window similar to Figure 3.9 that lists the resource types and qualifiers for those types. Select layout in the Resource type drop-down box. Leave the Source set option set to main. Next, you will choose how the layout resources will be qualified. Select Orientation in the Available qualifiers list and click the >> button to move Orientation to the Chosen qualifiers section.

Figure 3.9 Creating a new resource directory



Finally, ensure that Landscape is selected in the Screen Orientation dropdown, as shown in Figure 3.10. Verify that the Directory name now indicates that your directory is called `layout-land`. While this window looks fancy, its purpose is just to set the name of your directory. Click OK and Android Studio will create the `res/layout-land/` folder.

Figure 3.10 Creating `res/layout-land`



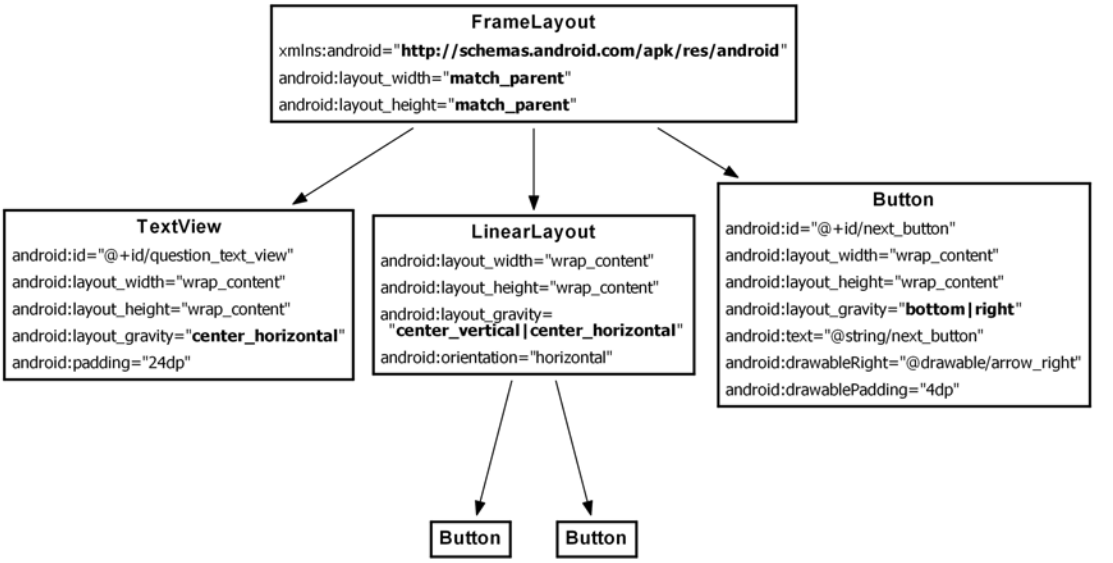
The `-land` suffix is another example of a configuration qualifier. Configuration qualifiers on `res` subdirectories are how Android identifies which resources best match the current device configuration. You can find the list of configuration qualifiers that Android recognizes and the pieces of the device configuration that they refer to at `http://developer.android.com/guide/topics/resources/providing-resources.html`.

When the device is in landscape orientation, Android will find and use resources in the `res/layout-land` directory. Otherwise, it will stick with the default in `res/layout/`. However, at the moment there are no resources in the `res/layout-land` directory. Let's fix that.

Copy the `activity_quiz.xml` file from `res/layout/` to `res/layout-land/`. You now have a landscape layout and a default layout. Keep the filename the same. The two layout files must have the same filename so that they can be referenced with the same resource ID.

Now make some changes to the landscape layout so that it is different from the default. Figure 3.11 shows the changes that you are going to make.

Figure 3.11 An alternative landscape layout



The **`FrameLayout`** will replace the **`LinearLayout`**. **`FrameLayout`** is the simplest **`ViewGroup`** and does not arrange its children in any particular manner. In this layout, child views will be arranged according to their `android:layout_gravity` attributes.

The **`TextView`**, **`LinearLayout`**, and **`Button`** children of the **`FrameLayout`** need `android:layout_gravity` attributes. The **`Button`** children of the **`LinearLayout`** will stay exactly the same.

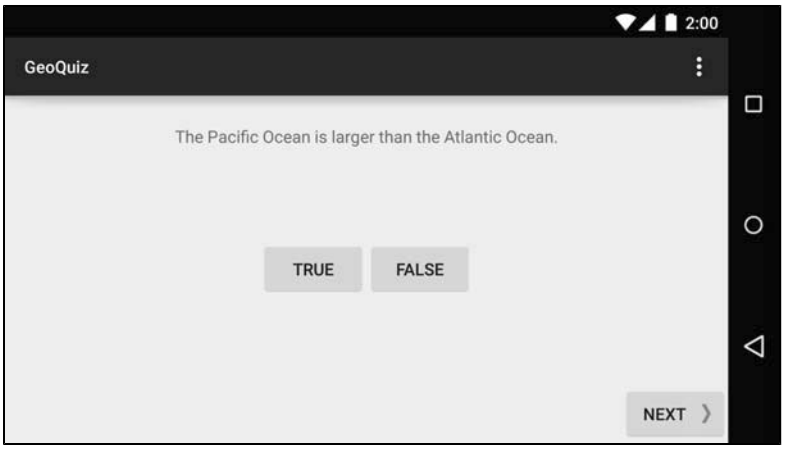
Open `layout-land/activity_quiz.xml` and make the necessary changes using Figure 3.11. You can use Listing 3.4 to check your work.

Listing 3.4 Tweaking the landscape layout (`layout-land/activity_quiz.xml`)

```
<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" >

<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="match_parent" >

  <TextView
    android:id="@+id/question_text_view"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_gravity="center_horizontal"
    android:padding="24dp" />

  <LinearLayout
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_gravity="center_vertical|center_horizontal"
    android:orientation="horizontal" >

    ...

  </LinearLayout>

  <Button
    android:id="@+id/next_button"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_gravity="bottom|right"
    android:text="@string/next_button"
    android:drawableRight="@drawable/arrow_right"
    android:drawablePadding="4dp"
    />

</LinearLayout>
</FrameLayout>
```

Run GeoQuiz again. Rotate the device to landscape to see the new layout (Figure 3.12). Of course, this is not just a new layout – it is a new **`QuizActivity`** as well.

Figure 3.12 QuizActivity in landscape orientation



Rotate back to portrait to see the default layout and yet another new **`QuizActivity`**.

Android does the work of determining the best resource for you, but it has to create a new activity from scratch to do it. For a **`QuizActivity`** to display a different layout, **`setContentView(R.layout.activity_quiz)`** must be called again. And this will not happen unless **`QuizActivity.onCreate(…)`** is called again. Thus, Android destroys the current **`QuizActivity`** on rotation and starts fresh to ensure that it has the resources that best match the new configuration.

Note that Android destroys the current activity and creates a new one whenever any runtime configuration change occurs. A change in keyboard availability or language could also occur at runtime, but a change in screen orientation is the runtime change that occurs most frequently.

Saving Data Across Rotation

Android does a great job of providing alternative resources at the right time. However, destroying and re-creating activities on rotation can cause headaches, too, like GeoQuiz's bug of reverting back to the first question when the device is rotated.

To fix this bug, the post-rotation **`QuizActivity`** needs to know the old value of `mCurrentIndex`. You need a way to save this data across a runtime configuration change, like rotation. One way to do this is to override the **`Activity`** method:

```
protected void onSaveInstanceState(Bundle outState)
```

This method is normally called by the system before **`onPause()`**, **`onStop()`**, and **`onDestroy()`**.

The default implementation of **`onSaveInstanceState(…)`** directs all of the activity's views to save their state as data in the **`Bundle`** object. A **`Bundle`** is a structure that maps string keys to values of certain limited types.

You have seen this **`Bundle`** before. It is passed into **`onCreate(Bundle)`**:

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

When you override **`onCreate(…)`**, you call **`onCreate(…)`** on the activity's superclass and pass in the bundle you just received. In the superclass implementation, the saved state of the views is retrieved and used to re-create the activity's view hierarchy.

Overriding onSaveInstanceState(Bundle)

You can override **`onSaveInstanceState(…)`** to save additional data to the bundle and then read that data back in **`onCreate(…)`**. This is how you are going to save the value of `mCurrentIndex` across rotation.

First, in `QuizActivity.java`, add a constant that will be the key for the key-value pair that will be stored in the bundle.

Listing 3.5 Adding a key for the value (`QuizActivity.java`)

```
public class QuizActivity extends AppCompatActivity {

    private static final String TAG = "QuizActivity";
    private static final String KEY_INDEX = "index";

    private Button mTrueButton;
    ...
```

Next, override **`onSaveInstanceState(…)`** to write the value of `mCurrentIndex` to the bundle with the constant as its key.

Listing 3.6 Overriding **`onSaveInstanceState(…)`** (`QuizActivity.java`)

```
        mNextButton.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                mCurrentIndex = (mCurrentIndex + 1) % mQuestionBank.length;
                updateQuestion();
            }
        });

        updateQuestion();
    }

    @Override
    public void onSaveInstanceState(Bundle savedInstanceState) {
        super.onSaveInstanceState(savedInstanceState);
        Log.i(TAG, "onSaveInstanceState");
        savedInstanceState.putInt(KEY_INDEX, mCurrentIndex);
    }
```

Finally, in **`onCreate(…)`**, check for this value. If it exists, assign it to `mCurrentIndex`.

Listing 3.7 Checking bundle in **`onCreate(…)`** (`QuizActivity.java`)

```
        ...

        if (savedInstanceState != null) {
            mCurrentIndex = savedInstanceState.getInt(KEY_INDEX, 0);
        }

        updateQuestion();
    }
```

Run GeoQuiz and press Next. No matter how many device rotations you perform, the newly minted **`QuizActivity`** will "remember" what question you were on.

Note that the types that you can save to and restore from a **`Bundle`** are primitive types and classes that implement the **`Serializable`** or **`Parcelable`** interfaces. It is usually a bad practice to put objects of custom types into a **`Bundle`**, however, because the data might be stale when you get it back out. It is a better choice to use some other kind of storage for the data and put a primitive identifier into the **`Bundle`** instead.

Testing the implementation of **`onSaveInstanceState(…)`** is a good idea – especially if you are saving and restoring objects. Rotation is easy to test; testing low-memory situations is harder. There is information at the end of this chapter about how to simulate your activity being destroyed by Android to reclaim memory.

The Activity Lifecycle, Revisited

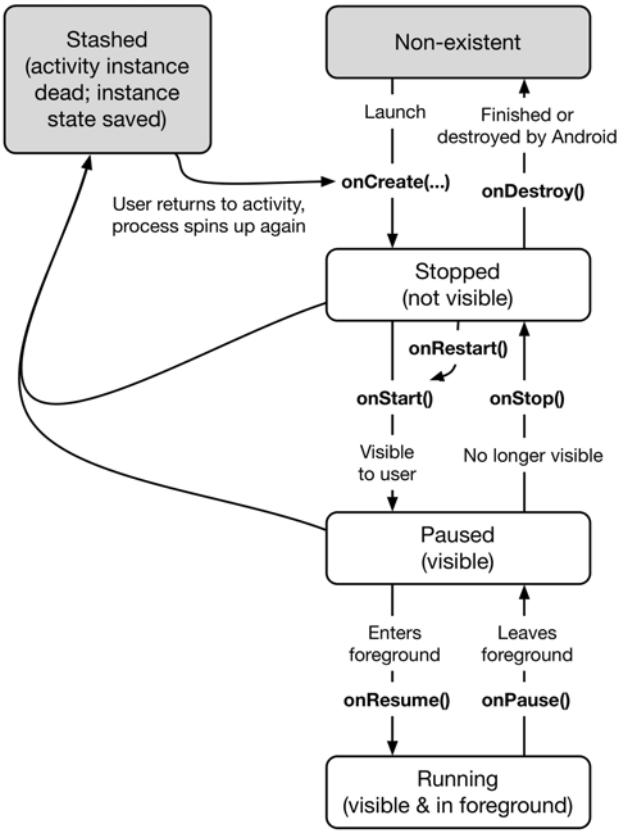
Overriding **`onSaveInstanceState(Bundle)`** is not just for handling rotation. An activity can also be destroyed if the user navigates away for a while and Android needs to reclaim memory.

Android will never destroy a running activity to reclaim memory – the activity must be in the paused or stopped state to be destroyed. If an activity is paused or stopped, then its **`onSaveInstanceState(…)`** method has been called.

When **`onSaveInstanceState(…)`** is called, the data is saved to the **`Bundle`** object. That **`Bundle`** object is then stuffed into your activity's *activity record* by the OS.

To understand the activity record, let's add a *stashed* state to the activity lifecycle (Figure 3.13).

Figure 3.13 The complete activity lifecycle



When your activity is stashed, an **`Activity`** object does not exist, but the activity record object lives on in the OS. The OS can reanimate the activity using the activity record when it needs to.

Note that your activity can pass into the stashed state without **`onDestroy()`** being called. However, you can always rely on **`onPause()`** and **`onSaveInstanceState(…)`** to be called. Typically, you override **`onSaveInstanceState(…)`** to stash small, transient states that belong to the current activity in your **`Bundle`** and **`onPause()`** for anything else that needs to be done.

Under some situations, Android will not only kill your activity but also completely shut down your application's process. This will only happen if the user is not currently looking at your application, but it can (and does) happen. Even in this case, the activity record will live on and enable a quick restart of your activity if the user returns.

So when does the activity record get snuffed? When the user presses the Back button, your activity really gets destroyed, once and for all. At that point, your activity record is discarded. Activity records are also typically discarded on reboot and may also be discarded if they are not used for a long time.

For the More Curious: Testing onSaveInstanceState(Bundle)

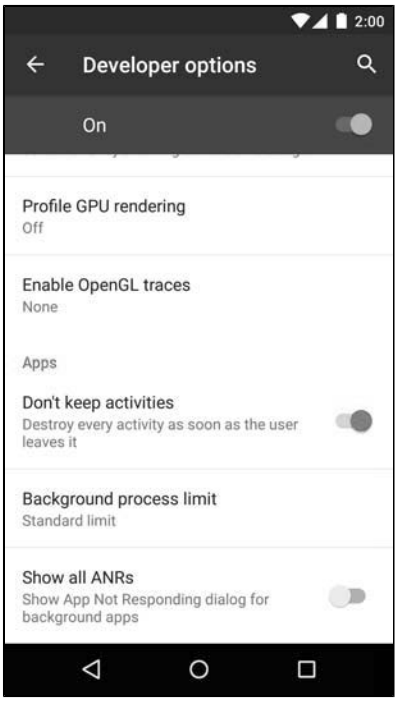
If you are overriding **`onSaveInstanceState(Bundle)`**, you should test that your state is being saved and restored as expected. This is easy to do on the emulator.

Start up a virtual device. Within the list of applications on the device, find the Settings app (Figure 3.14). This app is included with most system images used on the emulator.

Figure 3.14 Finding the Settings app

Launch Settings and select Developer options. Here you will see many possible settings. Turn on the setting labeled Don't keep activities, as shown in Figure 3.15.

Figure 3.15 Don't keep activities selected



Now run your app and press the Home button. Pressing Home causes the activity to be paused and stopped. Then the stopped activity will be destroyed just as if the Android OS had reclaimed it for its memory. Then you can restore the app to see if your state was saved as you expected. *Be sure to turn this setting off when you are done testing, as it will cause a performance decrease and some apps will perform poorly.*

Pressing the Back button instead of the Home button will always destroy the activity, regardless of whether you have this development setting on. Pressing the Back button tells the OS that the user is done with the activity.

To run the same test on a hardware device, you must install Dev Tools on the device. For more information, visit `http://developer.android.com/tools/debugging/debugging-devtools.html`.

For the More Curious: Logging Levels and Methods

When you use the **`android.util.Log`** class to send log messages, you control not only the content of a message, but also a *level* that specifies how important the message is. Android supports five log levels, shown in Figure 3.16. Each level has a corresponding method in the **`Log`** class. Sending output to the log is as simple as calling the corresponding **`Log`** method.

Figure 3.16 Log levels and methods

Log Level	Method	Notes
ERROR	`Log.e(...)`	Errors
WARNING	`Log.w(...)`	Warnings
INFO	`Log.i(...)`	Informational messages
DEBUG	`Log.d(...)`	Debug output; may be filtered out
VERBOSE	`Log.v(...)`	For development only!

In addition, each of the logging methods has two signatures: one which takes a *tag* string and a message string and a second that takes those two arguments plus an instance of **`Throwable`**, which makes it easy to log information about a particular exception that your application might throw. Listing 3.8 shows some sample log method signatures. Use regular Java string concatenation to assemble your message string, or **`String.format`** if you have fancier needs.

Listing 3.8 Different ways of logging in Android

```
// Log a message at "debug" log level
Log.d(TAG, "Current question index: " + mCurrentIndex);

Question question;
try {
    question = mQuestionBank[mCurrentIndex];
} catch (ArrayIndexOutOfBoundsException ex) {
    // Log a message at "error" log level, along with an exception stack trace
    Log.e(TAG, "Index was out of bounds", ex);
}
```

Index

Symbols

A

B

C

D

E

F

G

M

N

O

P

Q

R

S

T

U

V

W

X

Z