3
The Activity Lifecycle

What good is an app that resets itself when the user rotates the device? At the end of Chapter 2 you discovered that the geography question displayed is reset to the first question every time the device is rotated, regardless of what question is displayed prior to rotation. In this chapter you will address the dreaded – and very common – "rotation problem." To fix it, you will learn the basics of the *activity lifecycle*.

Every instance of **`Activity`** has a lifecycle. During this lifecycle, an activity transitions between four states: resumed, paused, stopped, and nonexistent. 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

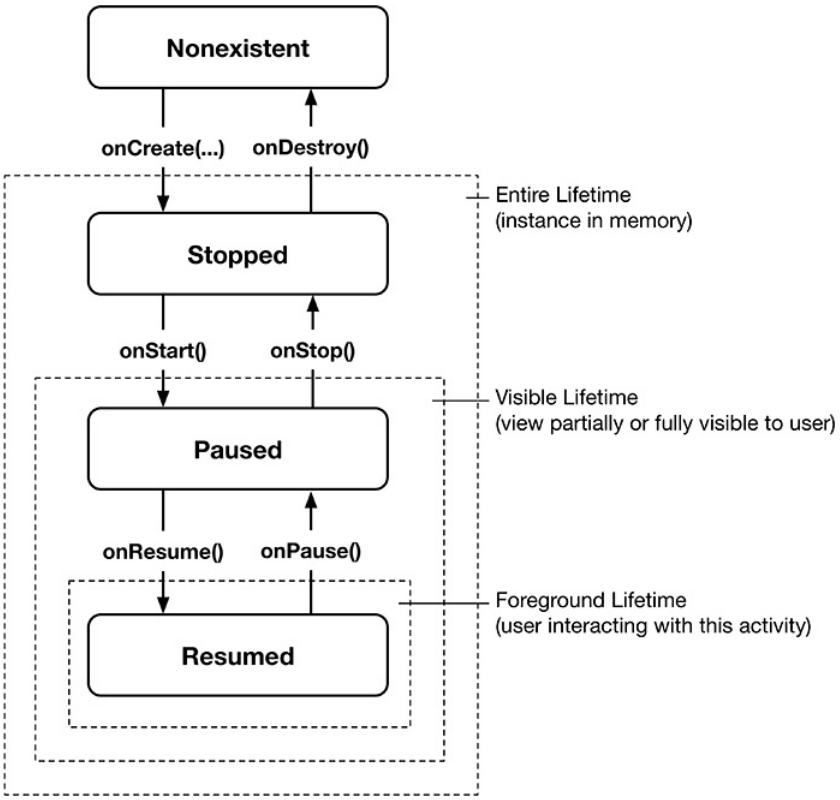


Figure 3.1 indicates for each state whether the activity has an instance in memory, is visible to the user, or is active in the foreground (accepting user input). Table 3.1 summarizes this information.

Table 3.1 Activity States

State	In memory?	Visible to user?	In foreground?
nonexistent	no	no	no
stopped	yes	no	no
paused	yes	yes/partially*	no
resumed	yes	yes	yes

(*Depending on the circumstances, a paused activity may be fully or partially visible. This is discussed further in the section called *Exploring the activity lifecycle by example*.)

The resumed state represents the activity the user is currently interacting with. Only one activity across all the apps on the device can be in the resumed state at any given time.

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. These methods are often called *lifecycle callbacks*.

You are already acquainted with one of these lifecycle callback 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(Bundle)`** to prepare the specifics of its UI:

- 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(Bundle)`** or any of the other **`Activity`** lifecycle methods yourself. You simply override the callbacks in your activity subclass. Then Android calls the lifecycle callbacks at the appropriate time (in relation to what the user is doing and what is happening across the rest of the system) to notify the activity that its state is changing.

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. This will help you see how **`QuizActivity`**'s state changes at runtime in relation to what the user is doing.

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.

Open `QuizActivity.java` and add a `TAG` constant to **`QuizActivity`**:

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

```
public class QuizActivity extends AppCompatActivity {

    private static final String TAG = "QuizActivity";
    ...
}
```

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

Listing 3.2 Adding a log statement to **`onCreate(Bundle)`** (`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)`**:

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

```
public class QuizActivity extends AppCompatActivity {
    ...
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        ...
    }

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

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

    @Override
    public void onPause() {
        super.onPause();
        Log.d(TAG, "onPause() 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 should be the first line of each callback method override implementation.

You may have been wondering about the `@Override` annotation. This asks the compiler to ensure that the class actually has the method that you want 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);
        ...
    }
    ...
}
```

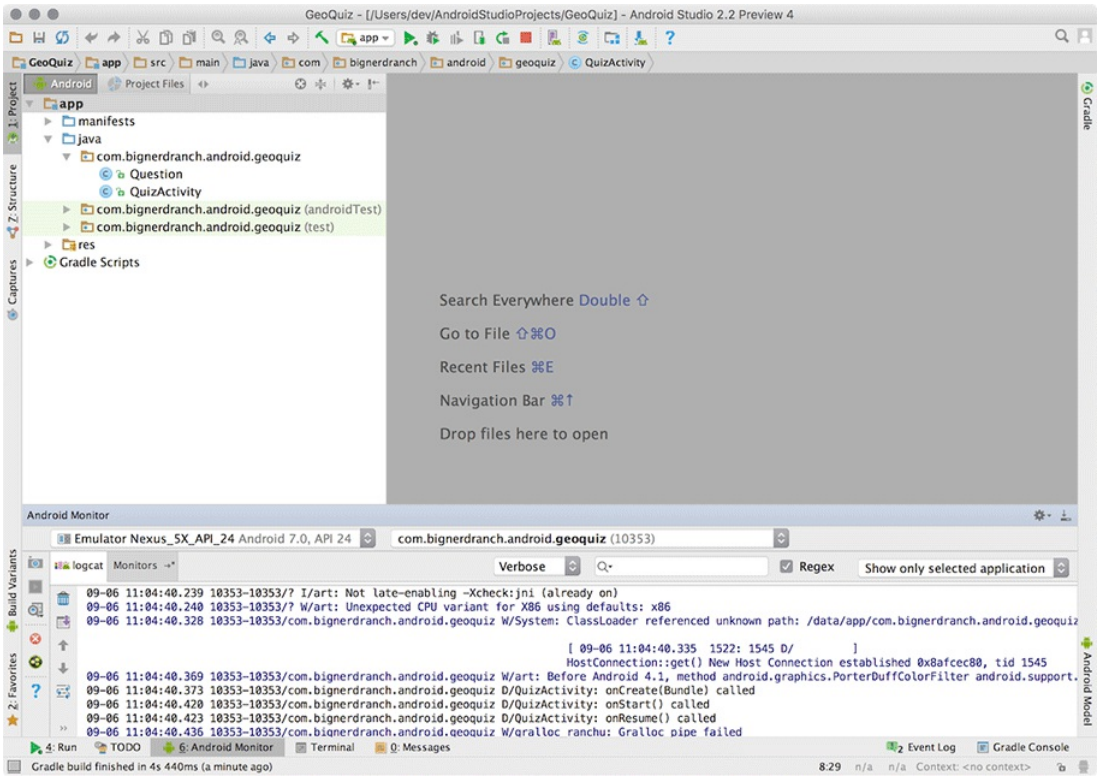
The parent **`AppCompatActivity`** class does not have an **`onCreat(Bundle)`** method, so the compiler will complain. This way you can fix the typo now rather than waiting until you run the app and see strange behavior to discover the error.

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 Monitor tool window near the bottom of the screen and ensure that the logcat tab is selected.

Figure 3.2 Android Studio with Logcat

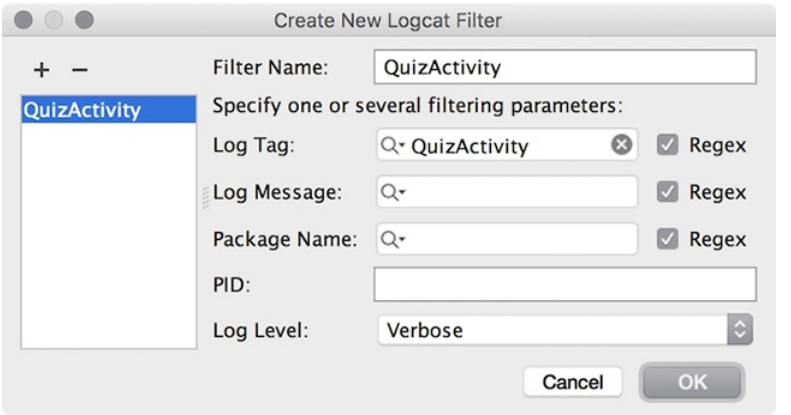


Run GeoQuiz and messages will start materializing in Logcat. By default, log statements that are generated with your app's package name are shown. You will see your own messages along with some system output.

To make your messages easier to find, you can filter the output using the `TAG` constant. In Logcat, click the dropdown in the top right of the Logcat pane that reads Show only selected application. This is the filter dropdown, which is currently set to show messages from only your app. Selecting No Filters will show log messages generated from all over the system.

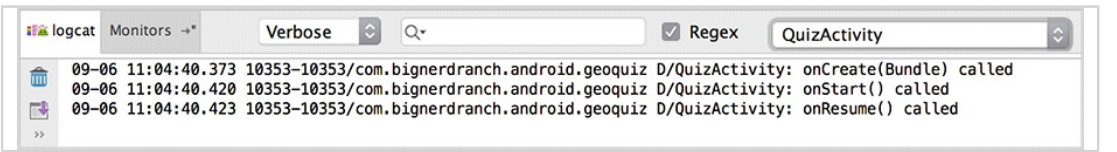
In the filter dropdown, select Edit Filter Configuration to create a new filter. Name the filter **`QuizActivity`** and enter **`QuizActivity`** in the Log Tag field (Figure 3.3).

Figure 3.3 Creating a filter in Logcat



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

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



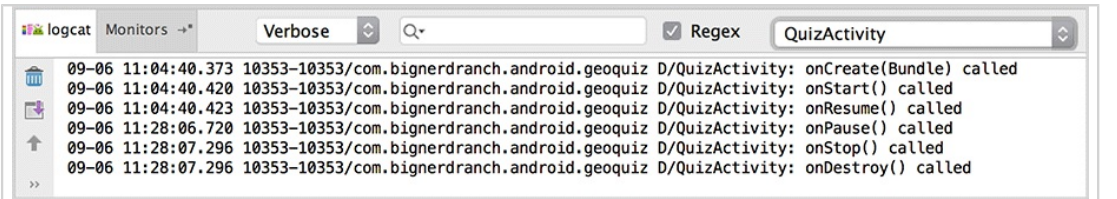
Exploring the activity lifecycle by example

Three lifecycle methods were called after GeoQuiz was launched and the initial instance of **`QuizActivity`** was created: **`onCreate(Bundle)`**, **`onStart()`**, and **`onResume()`** (Figure 3.4). Your **`QuizActivity`** instance is now in the resumed state (in memory, visible, and active in the foreground).

(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). Your **`QuizActivity`** instance is now in the nonexistent state (not in memory and thus not visible – and certainly not active in the foreground).

Figure 3.5 Pressing the Back button destroys the activity

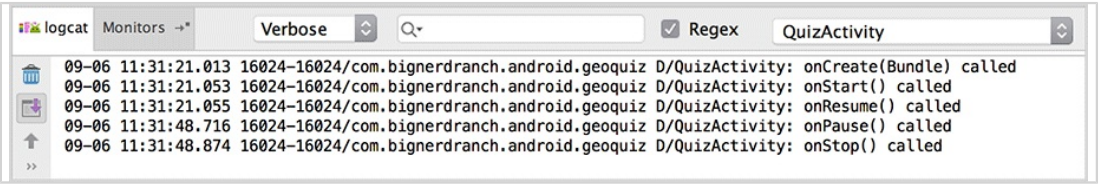


When you pressed the Back button, you told Android, "I'm done with this activity, and I won't need it anymore." Android then destroyed your activity's view and removed all traces of the activity from memory. This is Android's way of being frugal with your device's limited resources.

Launch GeoQuiz again by clicking the GeoQuiz app icon. Android creates a new instance of **`QuizActivity`** from scratch and calls **`onCreate()`**, **`onStart()`**, and **`onResume()`** to move **`QuizActivity`** from nonexistent to resumed.

Now press the Home button. The home screen displays and **`QuizActivity`** moves completely out of view. What state is **`QuizActivity`** in now? Check Logcat for a hint. Your activity received calls to **`onPause()`** and **`onStop()`**, but not **`onDestroy()`** (Figure 3.6).

Figure 3.6 Pressing the Home button stops the activity

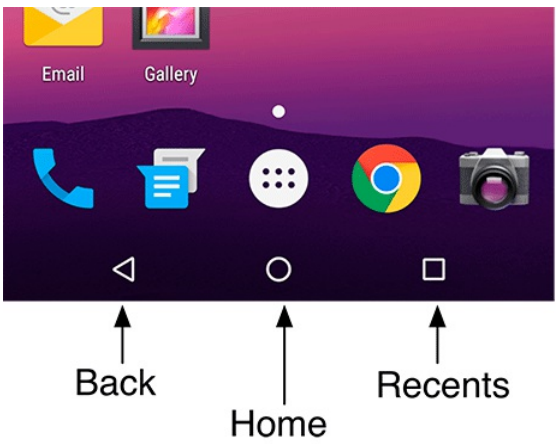


Pressing the Home button means the user is telling Android, "I'm going to go look at something else, but I might come back. I'm not really done with this screen yet." Android pauses and ultimately stops your activity. This means, after pressing Home, your instance of **`QuizActivity`** hangs out in the stopped state (in memory, not visible, and not active in the foreground). Android does this so it can quickly and easily restart **`QuizActivity`** where you left off when you come back to GeoQuiz later.

(This is not the whole story about going Home. Stopped activities can be destroyed at the discretion of the OS. See the section called *The Activity Lifecycle, Revisited* for the rest of the story.)

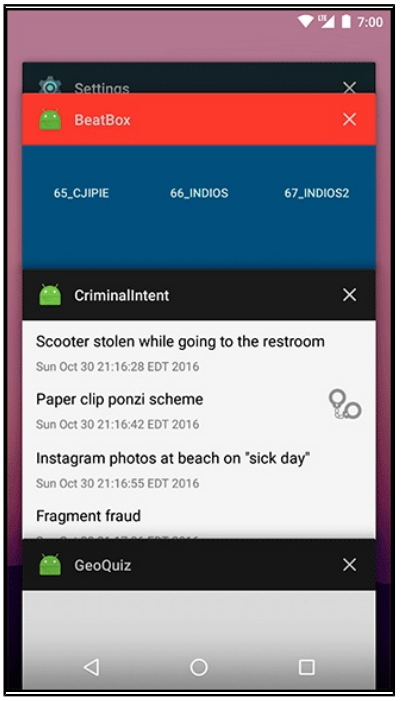
Go back to GeoQuiz by selecting the GeoQuiz task card from the overview screen. To do this, press the Recents button next to the Home button (Figure 3.7). (On devices without a Recents button, long-press the Home button.)

Figure 3.7 Back, Home, and Recents buttons



Each card in the overview screen represents an app the user has interacted with in the past (Figure 3.8). (The overview screen is often called the “Recents screen” or “task manager” by users. We defer to the developer documentation, which calls it the “overview screen.”)

Figure 3.8 Overview screen



Click on the GeoQuiz task card in the overview screen. **`QuizActivity`** will fill the screen.

A quick look at Logcat shows that your activity got calls to **`onStart()`** and **`onResume()`**. Note that **`onCreate()`** was not called. This is because **`QuizActivity`** was in the stopped state after the user pressed the Home button. Because the activity instance was still in memory, it did not need to be created. Instead, the activity only had to be started (moved to the paused/visible state) and then resumed (moved to the resumed/foreground state).

It is also possible for an activity to hang out in the paused state (fully or partially visible, but not in the foreground). The partially visible paused scenario can occur when a new activity with either a transparent background or a smaller-than-screen size is launched on top of your activity. The fully visible scenario occurs in multi-window mode (only available on Android 6.0 Nougat and higher) when the user interacts with a window that does not contain your activity, and yet your activity remains fully visible in the other window.

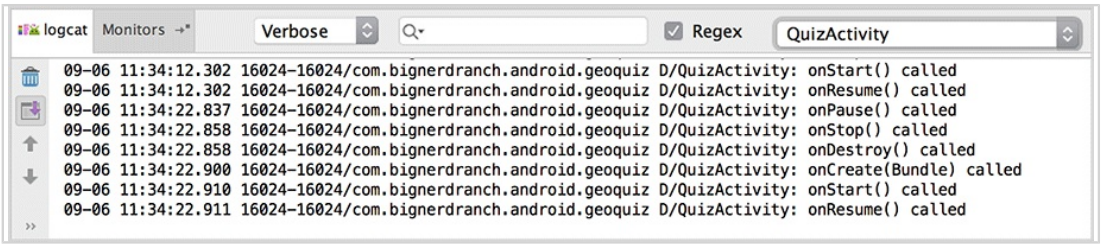
As you continue through the book, you will override the different activity lifecycle methods to do real things for your application. When you do, you will learn more about the uses of each method.

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 Command+Right Arrow/Ctrl+Right Arrow or click the rotation icon in the toolbar to rotate.)

After rotating, GeoQuiz will display the first question again. Check Logcat to see what has happened. Your output should look like Figure 3.9.

Figure 3.9 **`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 you rotate the device, the current **`QuizActivity`** instance is completely destroyed. The value that was stored in `mCurrentIndex` in that instance is wiped from memory. This means that when you rotate, GeoQuiz forgets which question you were looking at. As rotation finishes, Android creates a new instance of **`QuizActivity`** from scratch. `mCurrentIndex` is initialized to `0` in **`onCreate(Bundle)`**, 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 device configurations. You saw an example of this when you added multiple arrow icons to your project for different screen densities.

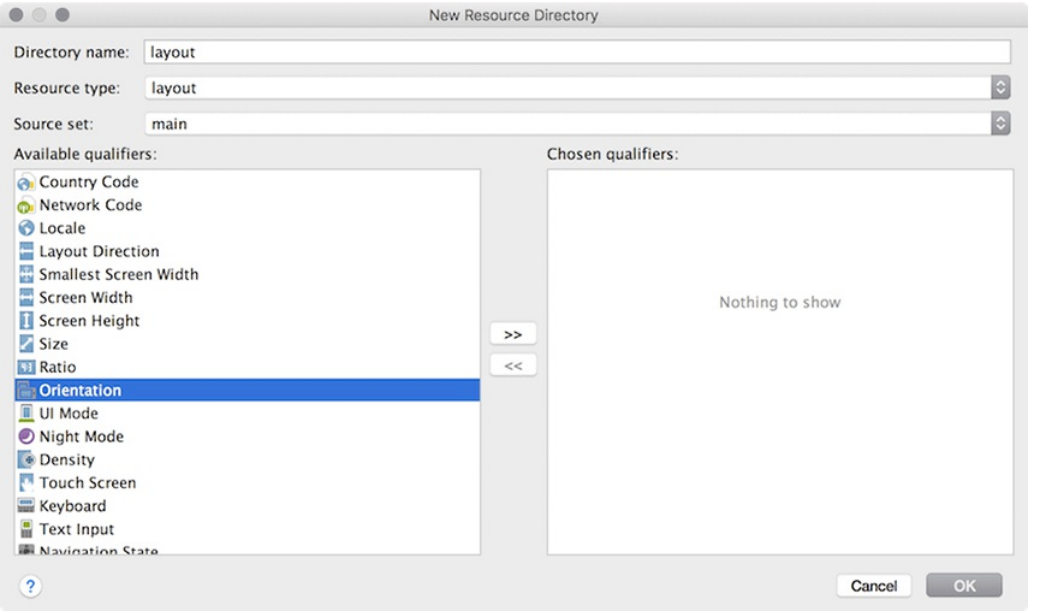
Screen density is a fixed component of the device configuration; it cannot change at runtime. On the other hand, some components, like screen orientation, *can* change at runtime. (There are other configuration changes that can occur at runtime, such as keyboard availability, language, and multi-window mode.)

When a *runtime configuration change* occurs, there may be resources that are a better match for the new configuration. So Android destroys the activity, looks for resources that are the best fit for the new configuration, and then rebuilds a new instance of the activity with those resources. 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 3.10 that lists the resource types and qualifiers for those types. Select layout in the Resource type dropdown. Leave the Source set option set to main.

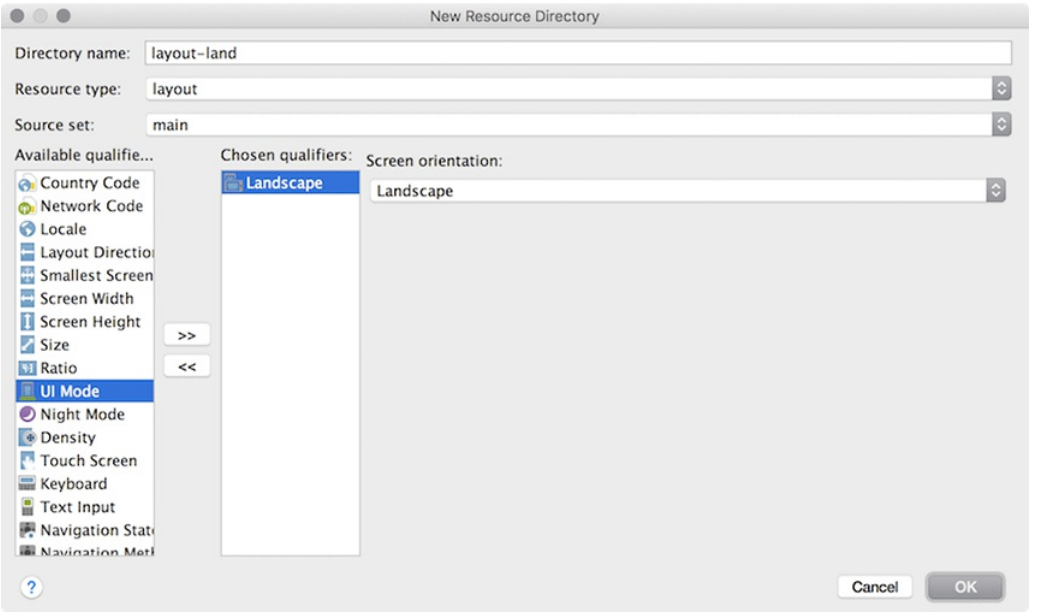
Figure 3.10 Creating a new resource directory



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.

Finally, ensure that Landscape is selected in the Screen orientation dropdown, as shown in Figure 3.11. 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 3.11 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 `developer.android.com/guide/topics/resources/providing-resources.html`.

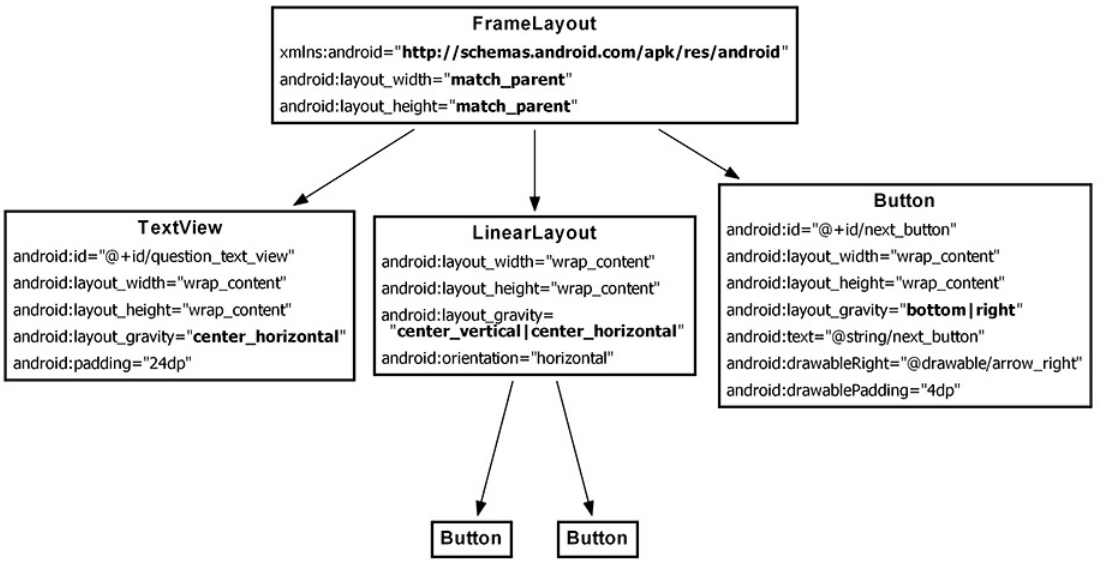
When the device is in landscape orientation, Android will find and use resources in the `res/layout-land` directory. Otherwise, it will stick with the default in `res/layout/`. However, at the moment there are no resources in the `res/layout-land` directory. Let's fix that.

Copy the `activity_quiz.xml` file from `res/layout/` to `res/layout-land/`. (If you do not see `res/layout-land/` in the project tool window, select Project from the dropdown to switch from the Android view. Just be sure to switch back to the Android view when you are done. You can also copy and paste the file outside of Android Studio using your favorite file explorer or terminal app.)

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 3.12 shows the changes that you are going to make.

Figure 3.12 An alternative landscape layout



A **`FrameLayout`** will replace the top **`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.

This means that 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.

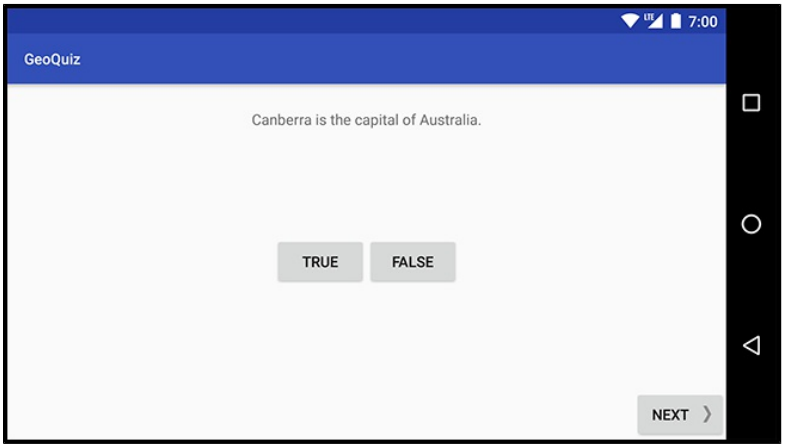
Open `layout-land/activity_quiz.xml` and make the necessary changes using Figure 3.12. You can use Listing 3.4 to check your work.

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 3.13). Of course, this is not just a new layout – it is a new **QuizActivity** as well.

Figure 3.13 QuizActivity in landscape orientation



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

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, such as GeoQuiz's bug of reverting back to the first question when the device is rotated.

To fix this bug, the post-rotation **QuizActivity** instance 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 called before **onStop()**, except when the user presses the Back button. (Remember, pressing Back tells Android the user is done with the activity, so Android wipes the activity from memory completely and does not make any attempt to save data to re-create it.)

The default implementation of **onSaveInstanceState(Bundle)** 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(Bundle)`**, you call **`onCreate(Bundle)`** 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(Bundle)`** to save additional data to the bundle and then read that data back in **`onCreate(Bundle)`**. 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(Bundle)`** to write the value of `mCurrentIndex` to the bundle with the constant as its key.

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

```
public class QuizActivity extends AppCompatActivity {
    ...
    @Override
    protected void onPause() {
        ...
    }

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

    @Override
    protected void onStop() {
        ...
    }
    ...
}
```

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

Listing 3.7 Checking bundle in **`onCreate(Bundle)`** (`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);

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

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.

The Activity Lifecycle, Revisited

Overriding **`onSaveInstanceState(Bundle)`** is not just for handling rotation or other runtime configuration changes. An activity can also be destroyed by the OS if the user navigates away for a while and Android needs to reclaim memory (e.g., if the user presses Home and then goes and watches a video or plays a game).

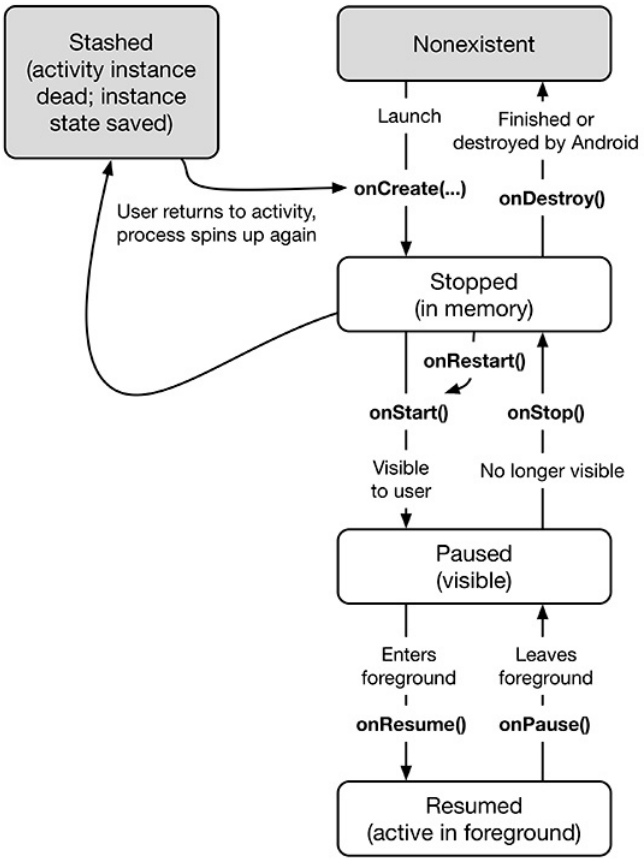
Practically speaking, the OS will not reclaim a visible (paused or resumed) activity. Activities are not marked as "killable" until **`onStop()`** is called and finishes executing.

Stopped activities are fair game to be killed, though. Still, not to worry. If an activity is stopped, that means **`onSaveInstanceState(Bundle)`** was called. So resolving the data-loss-across-rotation bug also addresses the situation where the OS destroys your nonvisible activity to free up memory.

How does the data you stash in **`onSaveInstanceState(Bundle)`** survive the activity's death? When **`onSaveInstanceState(Bundle)`** 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 3.14).

Figure 3.14 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. You can rely on **`onStop()`** and **`onSaveInstanceState(Bundle)`** being called (unless something has gone horribly wrong on the device). Typically, you override **`onSaveInstanceState(Bundle)`** to stash small, transient-state data that belongs to the current activity in your **`Bundle`**. Override **`onStop()`** to save any permanent data, such as things the user is editing, because your activity may be killed at any time after this method 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 discarded on reboot.

For the More Curious: Current State of Activity Cleanup

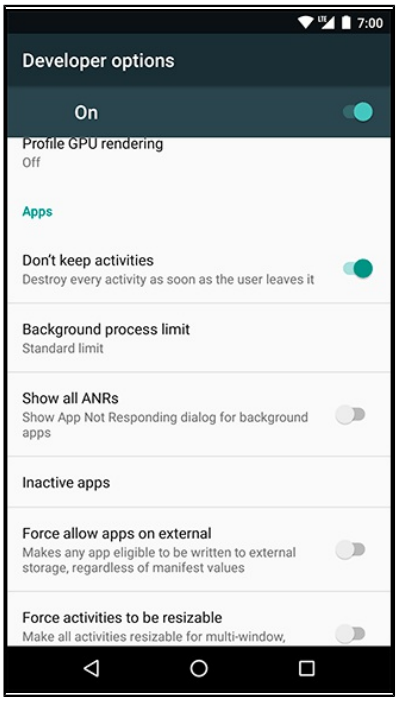
As of this writing, activities themselves are not individually destroyed in low-memory situations. Instead, Android clears an entire app process from memory, taking any of the app's in-memory activities with it. (Each application gets its own process. You will learn more about Android application processes in the section called *For the More Curious: Processes vs Tasks* in Chapter 24.)

Processes containing foreground (resumed) and/or visible (paused) activities get higher priority than other processes. When the OS needs to free up resources, it will select the lower priority processes first. Practically speaking, a process containing a visible activity will not reclaimed by the OS. If a foreground process does get reclaimed, that means something is horribly wrong with the device (and your app being killed is probably the least of the user's concerns).

If you are overriding **`onSaveInstanceState(Bundle)`**, you should test that your state is being saved and restored as expected. Rotation is easy to test. And, luckily, so is the low-memory situation. Try it out now to see for yourself.

Find and click on the Settings icon within the list of applications on the device. When the Settings screen appears, click Developer options (you will need to scroll down until you see the option you are looking for). On the Developer options screen 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



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. 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.

Remember that 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.

For the More Curious: Log 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 Table 3.2. Each level has a corresponding method in the **`Log`** class. Sending output to the log is as simple as calling the corresponding **`Log`** method.

Table 3.2 Log levels and methods

Log level	Method	Used for
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(…)`**	development only

In addition, each of the logging methods has two signatures: one that 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. You can 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);
}
```

Challenge: Preventing Repeat Answers

Once a user provides an answer for a particular question, disable the buttons for that question to prevent multiple answers being entered.

Challenge: Graded Quiz

After the user provides answers for all of the quiz questions, display a **`Toast`** with a percentage score for the quiz. Good luck!