4 Debugging Android Apps

In this chapter, you will find out what to do when apps get buggy. You will learn how to use Logcat, Android Lint, and the debugger that comes with Android Studio.

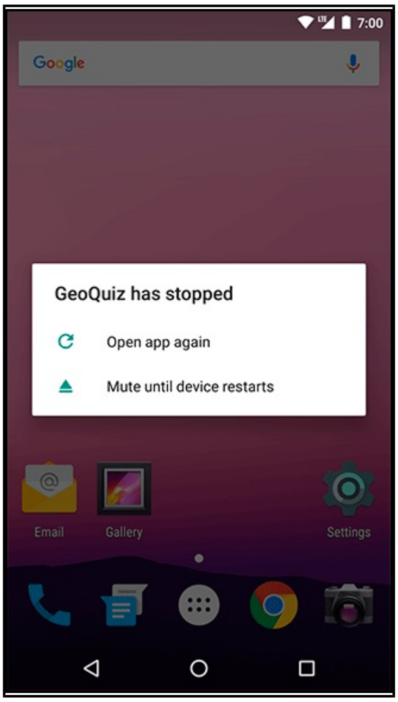
To practice debugging, the first step is to break something. In `QuizActivity.java`, comment out the code in **`onCreate(Bundle)`** where you pull out `mQuestionTextView`.

Listing 4.1 Commenting out a crucial line (`QuizActivity.java`)

```
@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);
    }

    // mQuestionTextView = (TextView)findViewById(R.id.question_text_view);

    mTrueButton = (Button)findViewById(R.id.true_button);
    mTrueButton.setOnClickListener(new View.OnClickListener() {
        ...
    });
    ...
}
```

Run GeoQuiz and see what happens. Figure 4.1 shows the message that appears when your app crashes and burns. Different versions of Android will have slightly different messages, but they all mean the same thing.

Figure 4.1 GeoQuiz is about to E.X.P.L.O.D.E.


GeoQuiz has stopped

Open app again

Mute until device restarts

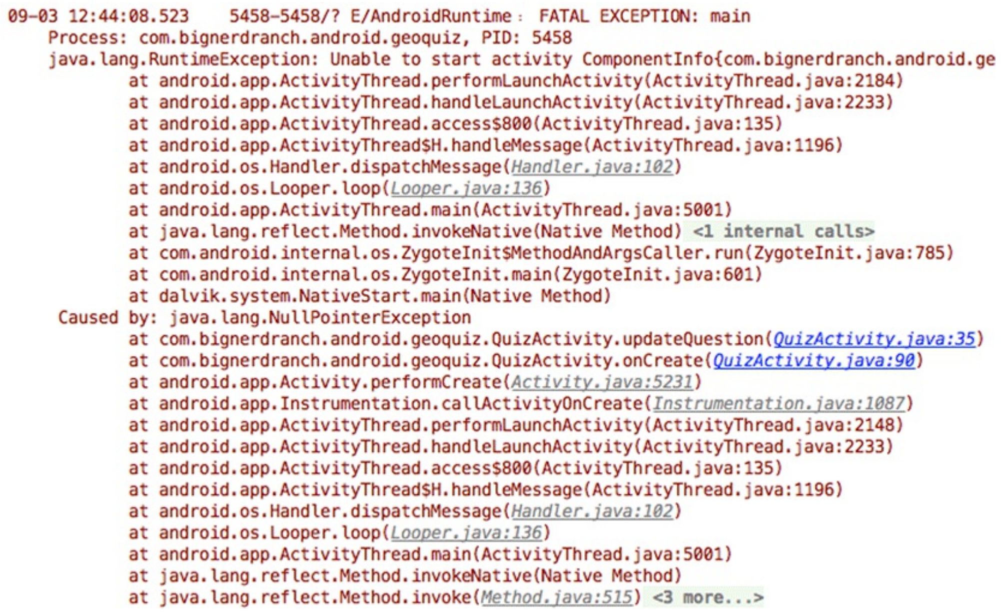
Of course, you know what is wrong with your app, but if you did not, it might help to look at your app from a new perspective.

Exceptions and Stack Traces

Expand the Android Monitor tool window so that you can see what has happened. If you scroll up and down in Logcat, you should eventually find an expanse of red, as shown in Figure 4.2. This is a standard `AndroidRuntime` exception report.

If you do not see much in Logcat and cannot find the exception, you may need to select the No Filters option in the filter dropdown. On the other hand, if you see too much in Logcat, you can adjust the Log Level to Error, which will show only the most severe log messages. You can also search for the text “FATAL EXCEPTION,” which will bring you straight to the exception that caused the app to crash.

Figure 4.2 Exception and stack trace in Logcat



```
09-03 12:44:08.523    5458-5458/? E/AndroidRuntime﹕ FATAL EXCEPTION: main
    Process: com.bignerdranch.android.geoquiz, PID: 5458
    java.lang.RuntimeException: Unable to start activity ComponentInfo{com.bignerdranch.android.ge
            at android.app.ActivityThread.performLaunchActivity(ActivityThread.java:2184)
            at android.app.ActivityThread.handleLaunchActivity(ActivityThread.java:2233)
            at android.app.ActivityThread.access$800(ActivityThread.java:135)
            at android.app.ActivityThread$H.handleMessage(ActivityThread.java:1196)
            at android.os.Handler.dispatchMessage(Handler.java:102)
            at android.os.Looper.loop(Looper.java:136)
            at android.app.ActivityThread.main(ActivityThread.java:5001)
            at java.lang.reflect.Method.invokeNative(Native Method) <1 internal calls>
            at com.android.internal.os.ZygoteInit$MethodAndArgsCaller.run(ZygoteInit.java:785)
            at com.android.internal.os.ZygoteInit.main(ZygoteInit.java:601)
            at dalvik.system.NativeStart.main(Native Method)
     Caused by: java.lang.NullPointerException
            at com.bignerdranch.android.geoquiz.QuizActivity.updateQuestion(QuizActivity.java:35)
            at com.bignerdranch.android.geoquiz.QuizActivity.onCreate(QuizActivity.java:90)
            at android.app.Activity.performCreate(Activity.java:5231)
            at android.app.Instrumentation.callActivityOnCreate(Instrumentation.java:1087)
            at android.app.ActivityThread.performLaunchActivity(ActivityThread.java:2148)
            at android.app.ActivityThread.handleLaunchActivity(ActivityThread.java:2233)
            at android.app.ActivityThread.access$800(ActivityThread.java:135)
            at android.app.ActivityThread$H.handleMessage(ActivityThread.java:1196)
            at android.os.Handler.dispatchMessage(Handler.java:102)
            at android.os.Looper.loop(Looper.java:136)
            at android.app.ActivityThread.main(ActivityThread.java:5001)
            at java.lang.reflect.Method.invokeNative(Native Method)
            at java.lang.reflect.Method.invoke(Method.java:515) <3 more...>
```

The report tells you the top-level exception and its stack trace, then the exception that caused that exception and *its* stack trace, and so on until it finds an exception with no cause.

In most of the code you will write, that last exception with no cause is the interesting one. Here the exception without a cause is a `java.lang.NullPointerException`. The line just below this exception is the first line in its stack trace. This line tells you the class and method where the exception occurred as well as what file and line number the exception occurred on. Click the blue link, and Android Studio will take you to that line in your source code.

The line to which you are taken is the first use of the `mQuestionTextView` variable, inside **`updateQuestion()`**. The name `NullPointerException` gives you a hint to the problem: This variable was not initialized.

Uncomment the line initializing `mQuestionTextView` to fix the bug.

When you encounter runtime exceptions, remember to look for the last exception in Logcat and the first line in its stack trace that refers to code that you have written. That is where the problem occurred, and it is the best place to start looking for answers.

If a crash occurs while a device is not plugged in, all is not lost. The device will store the latest lines written to the log. The length and expiration of the stored log depends on the device, but you can usually count on retrieving log results within 10 minutes. Just plug in the device and select it in the Devices view. Logcat will fill itself with the stored log.

Diagnosing misbehaviors

Problems with your apps will not always be crashes. In some cases, they will be misbehaviors. For example, suppose that every time you pressed the NEXT button, nothing happened. That would be a noncrashing, misbehaving bug.

In `QuizActivity.java`, make a change to the `mNextButton` listener to comment out the code that increments `mCurrentIndex`.

Listing 4.2 Forgetting a critical line of code (`QuizActivity.java`)

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    ...
    mNextButton = (Button)findViewById(R.id.next_button);
    mNextButton.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View v) {
            // mCurrentIndex = (mCurrentIndex + 1) % mQuestionBank.length;
            updateQuestion();
        }
    });
    ...
}
```

Run GeoQuiz and press the NEXT button. You should see no effect.

This bug is trickier than the last bug. It is not throwing an exception, so fixing the bug is not a simple matter of making the exception go away. On top of that, this misbehavior could be caused in two different ways: The index might not be changed, or **`updateQuestion()`** might not be called.

If you had no idea what was causing the problem, you would need to track down the culprit. In the next few sections, you will see two ways to do this: diagnostic logging of a stack trace and using the debugger to set a breakpoint.

Logging stack traces

In **`QuizActivity`**, add a log statement to **`updateQuestion()`**.

Listing 4.3 **`Exception`** for fun and profit (`QuizActivity.java`)

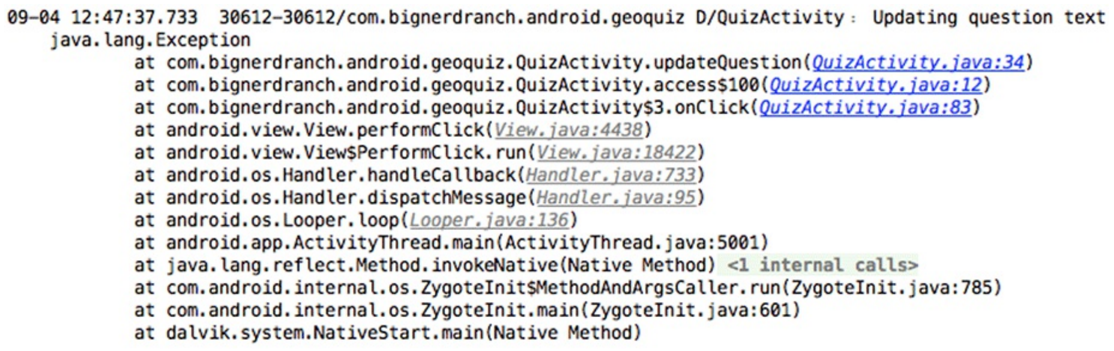
```
public class QuizActivity extends AppCompatActivity {
    ...
    private void updateQuestion() {
        Log.d(TAG, "Updating question text", new Exception());
        int question = mQuestionBank[mCurrentIndex].getTextResId();
        mQuestionTextView.setText(question);
    }
```

The **`Log.d(String, String, Throwable)`** version of **`Log.d`** logs the entire stack trace just like the `AndroidRuntime` exception you saw earlier. The stack trace will tell you where the call to **`updateQuestion()`** was made.

The exception that you pass to **Log.d(String, String, Throwable)** does not have to be a thrown exception that you caught. You can create a brand new **Exception** and pass it to the method without ever throwing it, and you will get a report of where the exception was created.

Run GeoQuiz, press the NEXT button, and then check the output in Logcat (Figure 4.3).

Figure 4.3 The results



```
09-04 12:47:37.733  30612-30612/com.bignerdranch.android.geoquiz D/QuizActivity: Updating question text
    java.lang.Exception
            at com.bignerdranch.android.geoquiz.QuizActivity.updateQuestion(QuizActivity.java:34)
            at com.bignerdranch.android.geoquiz.QuizActivity.access$100(QuizActivity.java:12)
            at com.bignerdranch.android.geoquiz.QuizActivity$3.onClick(QuizActivity.java:83)
            at android.view.View.performClick(View.java:4438)
            at android.view.View$PerformClick.run(View.java:18422)
            at android.os.Handler.handleCallback(Handler.java:733)
            at android.os.Handler.dispatchMessage(Handler.java:95)
            at android.os.Looper.loop(Looper.java:136)
            at android.app.ActivityThread.main(ActivityThread.java:5001)
            at java.lang.reflect.Method.invokeNative(Native Method) <1 internal calls>
            at com.android.internal.os.ZygoteInit$MethodAndArgsCaller.run(ZygoteInit.java:785)
            at com.android.internal.os.ZygoteInit.main(ZygoteInit.java:601)
            at dalvik.system.NativeStart.main(Native Method)
```

The top line in the stack trace is the line where you logged out the **Exception**. A few lines after that you can see where **updateQuestion()** was called from within your **onClick(View)** implementation. Click the link on this line, and you will be taken to where you commented out the line to increment your question index. But do not get rid of the bug; you are going to use the debugger to find it again in a moment.

Logging out stack traces is a powerful tool, but it is also a verbose one. Leave a bunch of these hanging around, and soon Logcat will be an unmanageable mess. Also, a competitor might steal your ideas by reading your stack traces to understand what your code is doing.

On the other hand, sometimes a stack trace showing what your code does is exactly what you need. If you are seeking help with a problem at `stackoverflow.com` or `forums.bignerdranch.com`, it often helps to include a stack trace. You can copy and paste lines directly from Logcat.

Before continuing, delete the log statement in `QuizActivity.java`.

Listing 4.4 Farewell, old friend (`QuizActivity.java`)

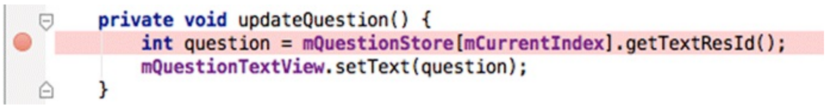
```
public class QuizActivity extends AppCompatActivity {
    ...
    private void updateQuestion() {
        ~~Log.d(TAG, "Updating question text", new Exception());~~
        int question = mQuestionBank[mCurrentIndex].getTextResId();
        mQuestionTextView.setText(question);
    }
```

Setting breakpoints

Now you will use the debugger that comes with Android Studio to track down the same bug. You will set a *breakpoint* on **`updateQuestion()`** to see whether it was called. A breakpoint pauses execution before the line executes and allows you to examine line by line what happens next.

In `QuizActivity.java`, return to the **`updateQuestion()`** method. Next to the first line of this method, click the gray bar in the lefthand margin. You should now see a red circle in the gray bar like the one shown in Figure 4.4. This is a breakpoint.

Figure 4.4 A breakpoint



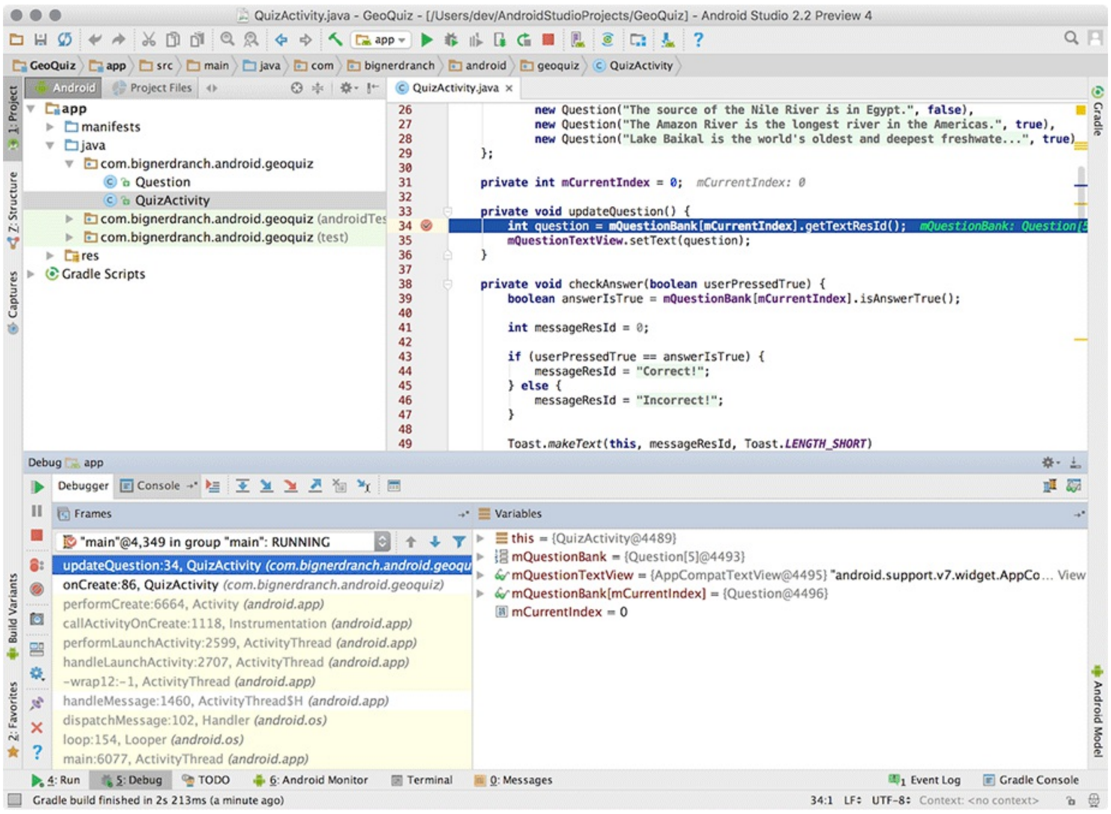
```
private void updateQuestion() {
    int question = mQuestionStore[mCurrentIndex].getTextResId();
    mQuestionTextView.setText(question);
}
```

To engage the debugger and trigger your breakpoint, you need to debug your app instead of running it. To debug your app, click the debug button (represented by a bug), which is next to the run button. You can also navigate to Run → Debug 'app' in the menu bar. Your device will report that it is waiting for the debugger to attach, and then it will proceed normally.

Once your app is up and running with the debugger attached, it will pause. Firing up GeoQuiz called **`QuizActivity.onCreate(Bundle)`**, which called **`updateQuestion()`**, which hit your breakpoint.

In Figure 4.5, you can see that this editor has opened `QuizActivity.java` and highlighted the line with the breakpoint where execution has paused.

Figure 4.5 Stop right there!


QuizActivity.java - GeoQuiz - [/Users/dev/AndroidStudioProjects/GeoQuiz] - Android Studio 2.2 Preview 4
GeoQuiz › app › src › main › java › com › bignerdranch › android › geoquiz › QuizActivity
Android
Project Files
QuizActivity.java
app
manifests
java
com.bignerdranch.android.geoquiz
Question
QuizActivity
com.bignerdranch.android.geoquiz (androidTes
com.bignerdranch.android.geoquiz (test)
res
Gradle Scripts
Debug app
Debugger
Console
Frames
Variables
4: Run
5: Debug
TODO
6: Android Monitor
Terminal
0: Messages
Event Log
Gradle Console

```
            new Question("The source of the Nile River is in Egypt.", false),
            new Question("The Amazon River is the longest river in the Americas.", true),
            new Question("Lake Baikal is the world's oldest and deepest freshwate...", true)
    };

    private int mCurrentIndex = 0;  mCurrentIndex: 0

    private void updateQuestion() {
        int question = mQuestionBank[mCurrentIndex].getTextResId();  mQuestionBank: Question[5
        mQuestionTextView.setText(question);
    }

    private void checkAnswer(boolean userPressedTrue) {
        boolean answerIsTrue = mQuestionBank[mCurrentIndex].isAnswerTrue();

        int messageResId = 0;

        if (userPressedTrue == answerIsTrue) {
            messageResId = "Correct!";
        } else {
            messageResId = "Incorrect!";
        }

        Toast.makeText(this, messageResId, Toast.LENGTH_SHORT)
```

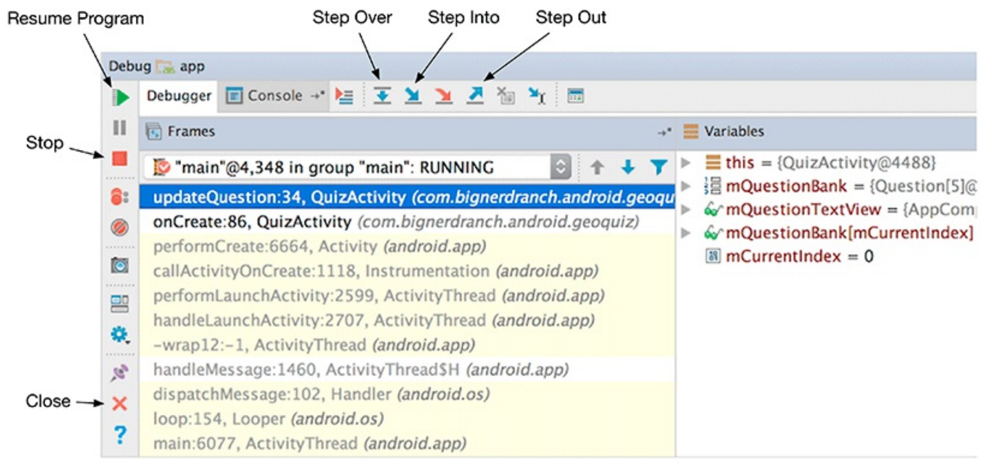
```
"main"@4,349 in group "main": RUNNING
updateQuestion:34, QuizActivity (com.bignerdranch.android.geoqu
onCreate:86, QuizActivity (com.bignerdranch.android.geoquiz)
performCreate:6664, Activity (android.app)
callActivityOnCreate:1118, Instrumentation (android.app)
performLaunchActivity:2599, ActivityThread (android.app)
handleLaunchActivity:2707, ActivityThread (android.app)
-wrap12:-1, ActivityThread (android.app)
handleMessage:1460, ActivityThread$H (android.app)
dispatchMessage:102, Handler (android.os)
loop:154, Looper (android.os)
main:6077, ActivityThread (android.app)
```

```
this = {QuizActivity@4489}
mQuestionBank = {Question[5]@4493}
mQuestionTextView = {AppCompatTextView@4495} "android.support.v7.widget.AppCo... View
mQuestionBank[mCurrentIndex] = {Question@4496}
mCurrentIndex = 0
```

```
Gradle build finished in 2s 213ms (a minute ago)          34:1  LF÷  UTF-8÷  Context: <no context>
```

The debug tool window at the bottom of the screen is now visible. It contains the Frames and Variables views (Figure 4.6).

Figure 4.6 The debug tool window



```
"main"@4,348 in group "main": RUNNING
updateQuestion:34, QuizActivity (com.bignerdranch.android.geoqu
onCreate:86, QuizActivity (com.bignerdranch.android.geoquiz)
performCreate:6664, Activity (android.app)
callActivityOnCreate:1118, Instrumentation (android.app)
performLaunchActivity:2599, ActivityThread (android.app)
handleLaunchActivity:2707, ActivityThread (android.app)
-wrap12:-1, ActivityThread (android.app)
handleMessage:1460, ActivityThread$H (android.app)
dispatchMessage:102, Handler (android.os)
loop:154, Looper (android.os)
main:6077, ActivityThread (android.app)
```

```
this = {QuizActivity@4488}
mQuestionBank = {Question[5]@
mQuestionTextView = {AppCom
mQuestionBank[mCurrentIndex]
mCurrentIndex = 0
```

You can use the arrow buttons at the top of the view to step through your program. You can see from the stack trace that **`updateQuestion()`** has been called from inside **`onCreate(Bundle)`**. But you are interested in investigating the NEXT button's behavior, so click the resume program button to continue execution. Then press the NEXT button in GeoQuiz to see if your breakpoint is hit and execution is stopped. (It should be.)

Now that you are stopped at an interesting point of execution, you can take a look around. The Variables view allows you to examine the values of the objects in your program. You should see the variables that you have created in **`QuizActivity`** as well as an additional value: `this` (the **`QuizActivity`** instance itself).

You could expand the `this` variable to see all the variables declared in **`QuizActivity`**'s superclass, **`Activity`**, in **`Activity`**'s superclass, in its super-superclass, and so on. But for now, focus on the variables that you created.

You are only interested in one value: `mCurrentIndex`. Scroll down in the variables view until you see `mCurrentIndex`. Sure enough, it still has a value of `0`.

This code looks perfectly fine. To continue your investigation, you need to step out of this method. Click the step out button.

Check the editor view. It has now jumped you over to your `mNextButton`'s **`OnClickListener`**, right after **`updateQuestion()`** was called. Pretty nifty.

You will want to fix this implementation, but before you make any changes to code, you should stop debugging your app. You can do this in two ways: You can stop the program, or you can simply disconnect the debugger. To stop the program, click the stop button shown in Figure 4.6. Usually it is easier to simply disconnect the debugger. To do that, click the close button also labeled in Figure 4.6.

Now return your **OnClickListener** to its former glory.

Listing 4.5 Returning to normalcy (`QuizActivity.java`)

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    ...
    mNextButton = (Button)findViewById(R.id.next_button);
    mNextButton.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View v) {
            ~~//~~ mCurrentIndex = (mCurrentIndex + 1) % mQuestionBank.length;
            updateQuestion();
        }
    });
    ...
}
```

You have tried out two ways of tracking down a misbehaving line of code: stack trace logging and setting a breakpoint in the debugger. Which is better? Each has its uses, and one or the other will probably end up being your favorite.

Logging out stack traces has the advantage that you can see stack traces from multiple places in one log. The downside is that to learn something new you have to add new log statements, rebuild, deploy, and navigate through your app to see what happened. The debugger is more convenient. If you run your app with the debugger attached, then you can set a breakpoint while the application is still running and poke around to get information about multiple issues.

Using exception breakpoints

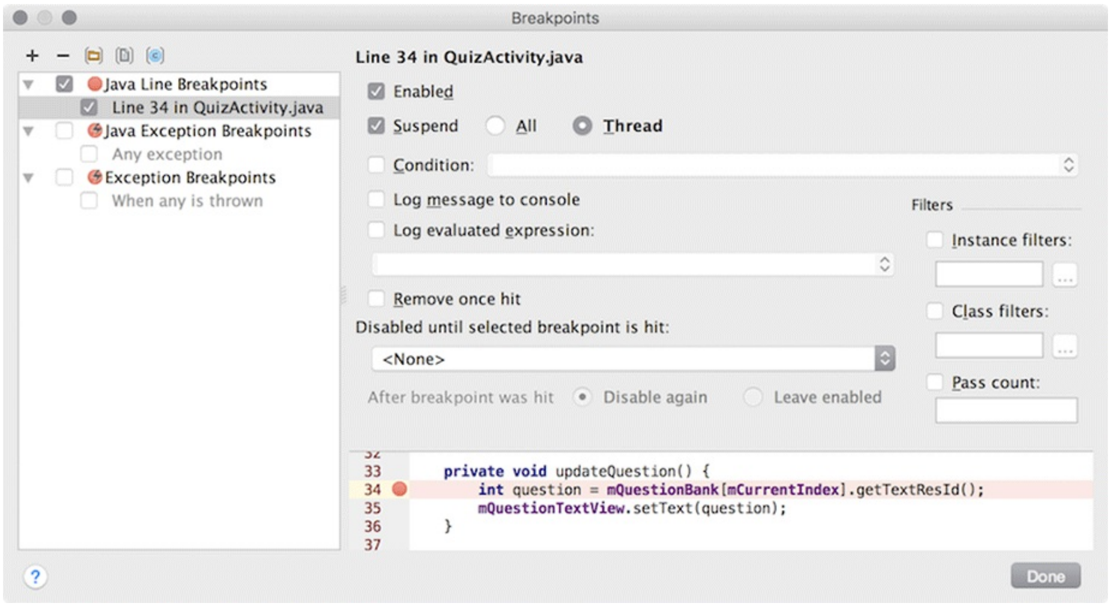
As if that were not enough choices, you can also use the debugger to catch exceptions. Return to **QuizActivity**'s **onCreate** method and comment out a line of code that will cause the app to crash.

Listing 4.6 Making GeoQuiz crash again (`QuizActivity.java`)

```
@Override
protected void onCreate(Bundle savedInstanceState) {
super.onCreate(savedInstanceState);
    ...
    // mNextButton = (Button) findViewById(R.id.next_button);
    mNextButton.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View v) {
            mCurrentIndex = (mCurrentIndex + 1) % mQuestionBank.length;
            updateQuestion();
        }
    });
    ...
}
```

Now select Run → View Breakpoints... to pull up the breakpoints dialog, as shown in Figure 4.7.

Figure 4.7 Setting an exception breakpoint



[✓ Java Line Breakpoints]
[✓ Line 34 in QuizActivity.java]
Java Exception Breakpoints
Any exception
Exception Breakpoints
When any is thrown

Line 34 in QuizActivity.java

[✓ Enabled]
[✓ Suspend] All [✓ Thread]
Condition:
Log message to console
Log evaluated expression:
Remove once hit
Disabled until selected breakpoint is hit:
<None>
After breakpoint was hit [✓ Disable again] Leave enabled

Filters
Instance filters:
Class filters:
Pass count:

```
    private void updateQuestion() {
        int question = mQuestionBank[mCurrentIndex].getTextResId();
        mQuestionTextView.setText(question);
    }
```

This dialog shows all of your currently set breakpoints. Remove the breakpoint you added earlier by highlighting it and clicking the - button.

The breakpoints dialog also allows you to set a breakpoint that is triggered when an exception is thrown, wherever it might happen. You can limit it to only uncaught exceptions or apply it to both caught and uncaught exceptions.

Click the + button to add a new breakpoint. Choose Java Exception Breakpoints in the drop-down list. You can now select the type of exception that you want to catch. Type in **`RuntimeException`** and choose RuntimeException (java.lang) from the suggestions. **`RuntimeException`** is the superclass of **`NullPointerException`**, **`ClassCastException`**, and other runtime problems, so it makes a nice catch-all.

Click Done and launch GeoQuiz with the debugger attached. This time, your debugger will jump right to the line where the exception was thrown as soon as it happens. Exquisite.

Now, this is a fairly big hammer. If you leave this breakpoint on while debugging, you can expect it to stop on some framework code or in other places you do not expect. So you may want to turn it off when you are not using it. Go ahead and remove the breakpoint now by returning to Run → View Breakpoints....

Undo the change from Listing 4.6 to get GeoQuiz back to a working state.

Android-Specific Debugging

Most Android debugging is just like Java debugging. However, you will run into issues with Android-specific parts, such as resources, that the Java compiler knows nothing about. This is where Android Lint comes in.

Using Android Lint

Android Lint (or just "Lint") is a *static analyzer* for Android code. A static analyzer is a program that examines your code to find defects without running it. Lint uses its knowledge of the Android frameworks to look deeper into your code and find problems that the compiler cannot. In most cases, Lint's advice is worth taking.

In Chapter 6, you will see Lint warn you about compatibility problems. Lint can also perform type-checking for objects that are defined in XML. Make the following casting mistake in **QuizActivity**.

Listing 4.7 A simple mix-up (QuizActivity.java)

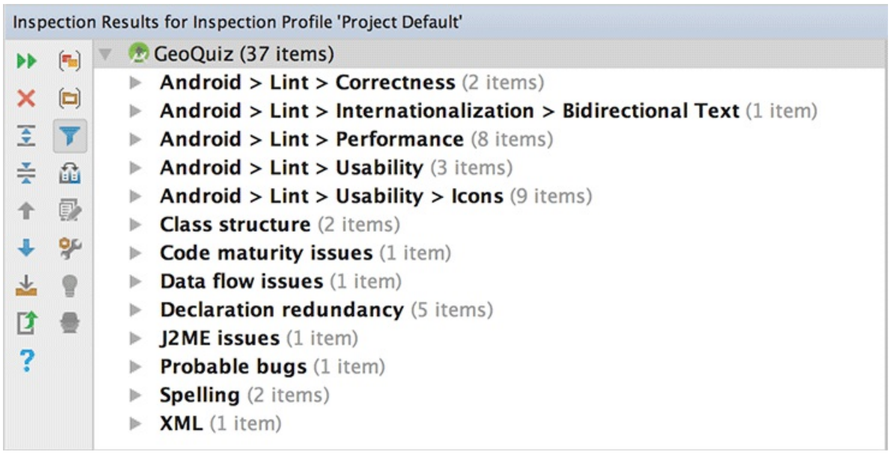
```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    Log.d(TAG, "onCreate(Bundle) called");
    setContentView(R.layout.activity_quiz);
    ...
    mQuestionTextView = (TextView)findViewById(R.id.question_text_view);

    ~~mTrueButton = (Button)findViewById(R.id.true_button);~~
    mTrueButton = (Button)findViewById(R.id.question_text_view);
    ...
}
```

Because you used the wrong resource ID, this code will attempt to cast a **TextView** as a **Button** at runtime. This will cause an improper cast exception. The Java compiler sees no problem with this code, but Lint will catch this error. You should see Lint immediately highlight this line of code to indicate that there is a problem.

You can manually run Lint to see all of the potential issues in your project, including those that are not as serious as the one above. Select Analyze → Inspect Code... from the menu bar. You will be asked which parts of your project you would like to inspect. Choose Whole project and click OK. Android Studio will now run Lint as well as a few other static analyzers on your code.

Once the scan is complete, you will see a few categories of potential issues in the inspection tool window. Expand the Android Lint categories to see Lint's information about your project (Figure 4.8).

Figure 4.8 Lint warnings



Inspection Results for Inspection Profile 'Project Default'

- GeoQuiz (37 items)
  - Android > Lint > Correctness (2 items)
  - Android > Lint > Internationalization > Bidirectional Text (1 item)
  - Android > Lint > Performance (8 items)
  - Android > Lint > Usability (3 items)
  - Android > Lint > Usability > Icons (9 items)
  - Class structure (2 items)
  - Code maturity issues (1 item)
  - Data flow issues (1 item)
  - Declaration redundancy (5 items)
  - J2ME issues (1 item)
  - Probable bugs (1 item)
  - Spelling (2 items)
  - XML (1 item)

You can select an issue in this list to see more detailed information and its location in your project.

The Mismatched view type warning in the Android > Lint > Correctness category is the one that you created above. Go ahead and correct the cast in **`onCreate(Bundle)`**.

Listing 4.8 Fixing that simple mix-up (`QuizActivity.java`)

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

    mQuestionTextView = (TextView)findViewById(R.id.question_text_view);

    ~~mTrueButton = (Button)findViewById(R.id.question_text_view);~~
    mTrueButton = (Button)findViewById(R.id.true_button);
    ...
}
```

Run GeoQuiz once more and confirm that the app is back to normal.

Issues with the R class

You are familiar with build errors that occur when you reference resources before adding them or delete resources that other files refer to. Usually, resaving the files once the resource is added or the references are removed will cause Android Studio to rebuild without any fuss.

Sometimes, however, these build errors will persist or appear seemingly out of nowhere. If this happens to you, here are some things you can try:

Recheck the validity of the XML in your resource files

> If your `R.java` file was not generated for the last build, you will see errors in your project wherever you reference a resource. Often, this is caused by a typo in one of your XML files. Layout XML is not always validated, so typos in these files may not be pointedly brought to your attention. Finding the typo and resaving the file should cause `R.java` to regenerate.

Clean your project

> Select Build → Clean Project. Android Studio will rebuild the project from scratch, which often results in an error-free build. We can all use a deep clean every now and then.

Sync your project with Gradle

> If you make changes to your `build.gradle` file, you will need to sync those changes to update your project's build settings. Select Tools → Android → Sync Project with Gradle Files. Android Studio will rebuild the project from scratch with the correct project settings, which can help to resolve issues after changing your Gradle configuration.

Run Android Lint

> Pay close attention to the warnings from Lint. With this tool, you will often discover unexpected issues.

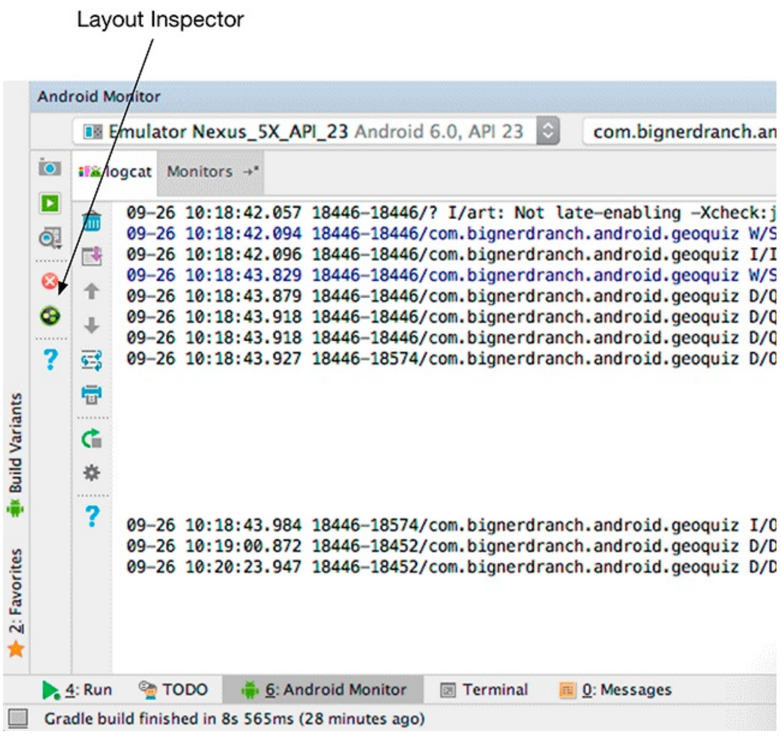
If you are still having problems with resources (or having different problems), give the error messages and your layout files a fresh look. It is easy to miss mistakes in the heat of the moment. Check out any Lint errors and warnings as well. A cool-headed reconsideration of the error messages may turn up a bug or typo.

Finally, if you are stuck or having other issues with Android Studio, check the archives at `stackoverflow.com` or visit the forum for this book at `forums.bignerdranch.com`.

Challenge: Exploring the Layout Inspector

For support debugging layout file issues, the layout inspector can be used to interactively inspect how a layout file is rendered to the screen. To use the layout inspector, make sure GeoQuiz is running in the emulator and click on the layout inspector icon in the left drawer within the Android Monitor tool window (Figure 4.9). Once the inspector is activated, you can explore the properties of your layout by clicking the elements within the layout inspector view.

Figure 4.9 The Layout Inspector button



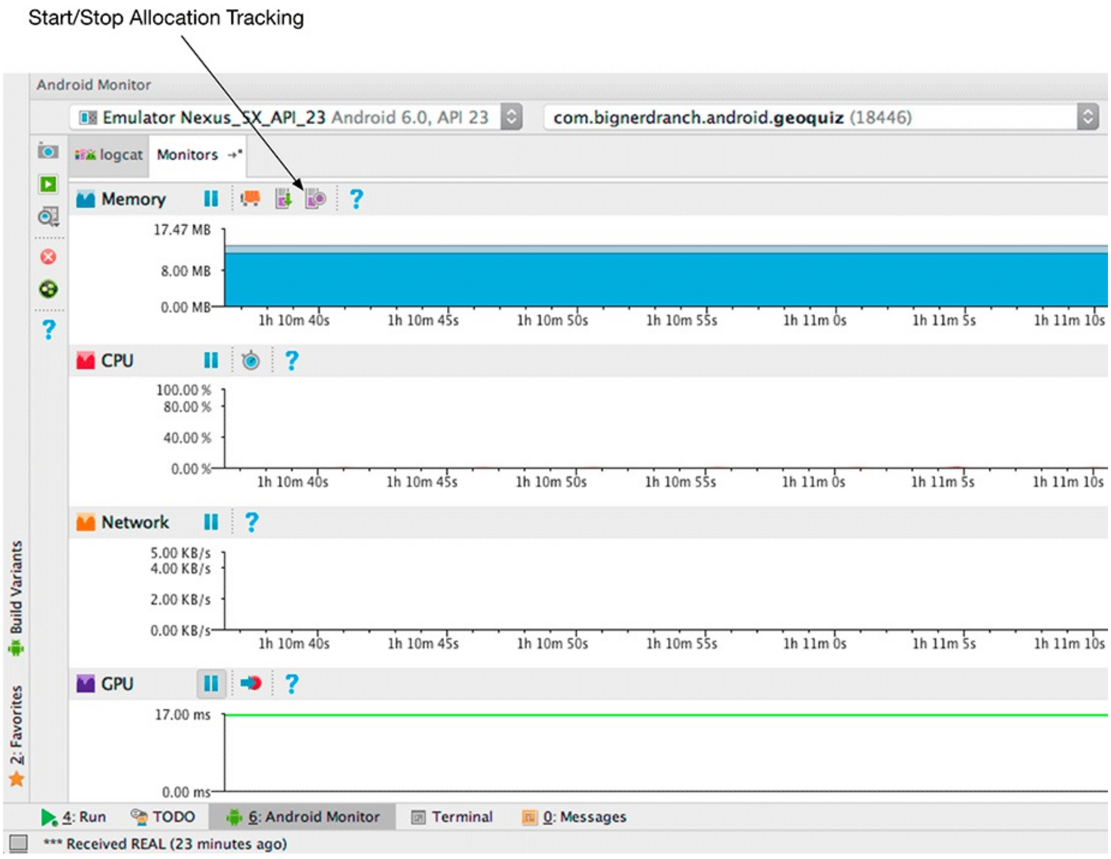
```
09-26 10:18:42.057 18446-18446/? I/art: Not late-enabling -Xcheck:j
09-26 10:18:42.094 18446-18446/com.bignerdranch.android.geoquiz W/S
09-26 10:18:42.096 18446-18446/com.bignerdranch.android.geoquiz I/I
09-26 10:18:43.829 18446-18446/com.bignerdranch.android.geoquiz W/S
09-26 10:18:43.879 18446-18446/com.bignerdranch.android.geoquiz D/O
09-26 10:18:43.918 18446-18446/com.bignerdranch.android.geoquiz D/O
09-26 10:18:43.918 18446-18446/com.bignerdranch.android.geoquiz D/O
09-26 10:18:43.927 18446-18574/com.bignerdranch.android.geoquiz D/O

09-26 10:18:43.984 18446-18574/com.bignerdranch.android.geoquiz I/O
09-26 10:19:00.872 18446-18452/com.bignerdranch.android.geoquiz D/D
09-26 10:20:23.947 18446-18452/com.bignerdranch.android.geoquiz D/D
```

Gradle build finished in 8s 565ms (28 minutes ago)

Challenge: Exploring Allocation Tracking

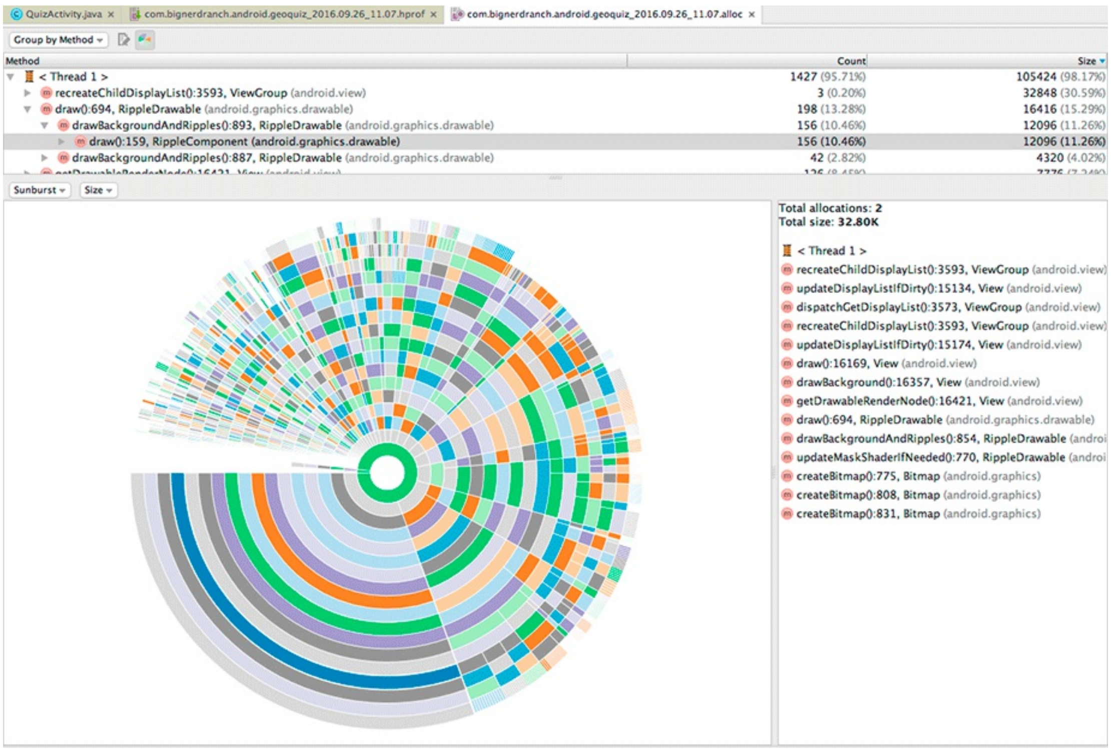
The Allocation Tracker tool creates detailed reports for the frequency and number of memory allocation calls in your program and is useful for performance-tuning your app. In the Android Monitor tool window, click the Allocation Tracker button (Figure 4.10).

Figure 4.10 Starting the Allocation Tracker


Start/Stop Allocation Tracking
Android Monitor
Emulator Nexus_5X_API_23 Android 6.0, API 23
com.bignerdranch.android.geoquiz (18446)
logcat
Monitors
Memory
CPU
Network
GPU
4: Run
TODO
6: Android Monitor
Terminal
0: Messages
*** Received REAL (23 minutes ago)
Build Variants
2: Favorites

This will begin recording memory allocations as you interact with your app. Once you have performed the interaction you are profiling, click the button again to stop allocation tracking. This will display the allocation report (Figure 4.11).

Figure 4.11 Allocation Tracker report


QuizActivity.java
com.bignerdranch.android.geoquiz_2016.09.26_11.07.hprof
com.bignerdranch.android.geoquiz_2016.09.26_11.07.alloc
Group by Method
Sunburst
Size

Method	Count	Size
< Thread 1 >	1427 (95.71%)	105424 (98.17%)
recreateChildDisplayList():3593, ViewGroup (android.view)	3 (0.20%)	32848 (30.59%)
draw():694, RippleDrawable (android.graphics.drawable)	198 (13.28%)	16416 (15.29%)
drawBackgroundAndRipples():893, RippleDrawable (android.graphics.drawable)	156 (10.46%)	12096 (11.26%)
draw():159, RippleComponent (android.graphics.drawable)	156 (10.46%)	12096 (11.26%)
drawBackgroundAndRipples():887, RippleDrawable (android.graphics.drawable)	42 (2.82%)	4320 (4.02%)

Total allocations: 2
Total size: 32.80K

< Thread 1 >
recreateChildDisplayList():3593, ViewGroup (android.view)
updateDisplayListIfDirty():15134, View (android.view)
dispatchGetDisplayList():3573, ViewGroup (android.view)
recreateChildDisplayList():3593, ViewGroup (android.view)
updateDisplayListIfDirty():15174, View (android.view)
draw():16169, View (android.view)
drawBackground():16357, View (android.view)
getDrawableRenderNode():16421, View (android.view)
draw():694, RippleDrawable (android.graphics.drawable)
drawBackgroundAndRipples():854, RippleDrawable (androi
updateMaskShaderIfNeeded():770, RippleDrawable (androi
createBitmap():775, Bitmap (android.graphics)
createBitmap():808, Bitmap (android.graphics)
createBitmap():831, Bitmap (android.graphics)

The allocation report shows the count of memory allocation events and the size of each in bytes in table form and as a visualization. You can select the report type at the top of the tool window.