## **Introduce a bug**

Remember the division by zero example from the debugging lesson in Unit 1? On the final iteration of the loop, when the app attempts to perform division by zero, the app crashes with a java.langArithmeticException, because division by zero is impossible. That bug was found and fixed by examining the stack trace, and this assumption was verified using log statements.

As you're already familiar with that example, it will be used to demonstrate how breakpoints can be used. Breakpoints step through code one line at a time without the need to add log statements and re-running your app first.

1. Open up MainActivity.kt and replace the code with the following:

import androidx.appcompat.app.AppCompatActivity

import android.os.Bundle

public val TAG = "MainActivity"

class MainActivity : AppCompatActivity() {

override fun onCreate(savedInstanceState: Bundle?) {

super.onCreate(savedInstanceState)

setContentView(R.layout.activity\_main)

division()

}

fun division() {

val numerator = 60

var denominator = 4

repeat(5) {

Log.v(TAG, "${numerator / denominator}")

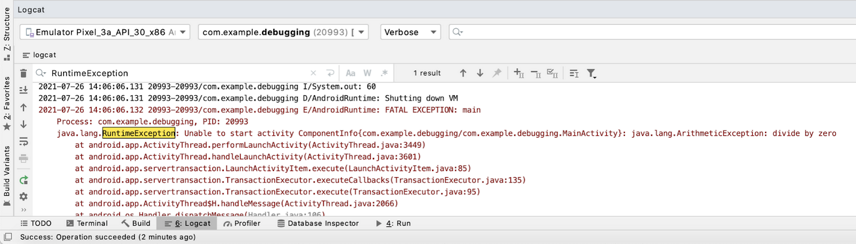
denominator--

}

}

}

Execute the app. Observe that the app crashes as expected.



3. Debug with breakpoints

When you learned about logging, you learned to strategically place logs to help identify bugs, and verify that they've been fixed. However, when faced with bugs that you didn't introduce, it's not always clear where to place log statements, or which variables to print out. Often, you can only find this information at runtime.

fun division() {

val numerator = 60

var denominator = 4

repeat(5) {

Log.v(TAG, "${numerator / denominator}")

denominator--

}

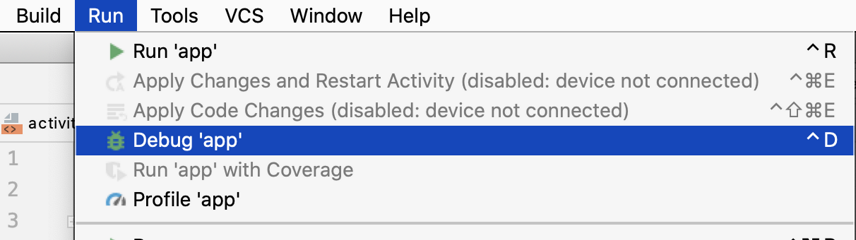
}

This is where breakpoints come in! Even if you have a vague idea of what's causing the bug based on information in the stack trace, you can add a breakpoint serving as a stop sign for a specific line of code. Once a breakpoint is reached, it will pause execution, allowing you to use other debugging tools at runtime to get an up-close look at what's happening, and what's gone wrong.

## **Attach the debugger**

Behind the scenes, Android Studio uses a tool called **Android Debug Bridge**, also known abbreviated as ADB. This is a command-line tool that's integrated into Android Studio, and provides debugging capabilities, such as breakpoints, to your running apps. A tool for debugging is often called a debugger.

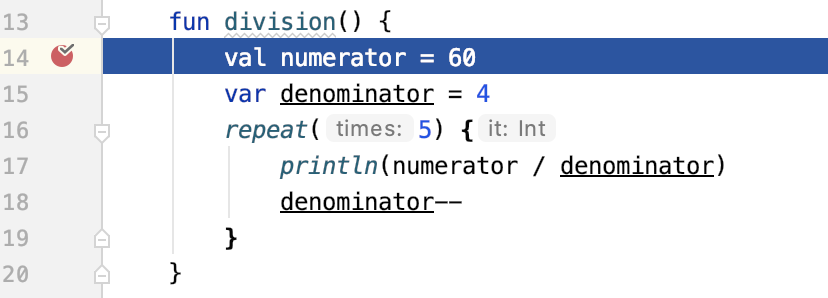
To use or attach the debugger to an app, you can't simply run the app with **Run > Run** like before. Instead, you run the app with **Run > Debug ‘app'**.



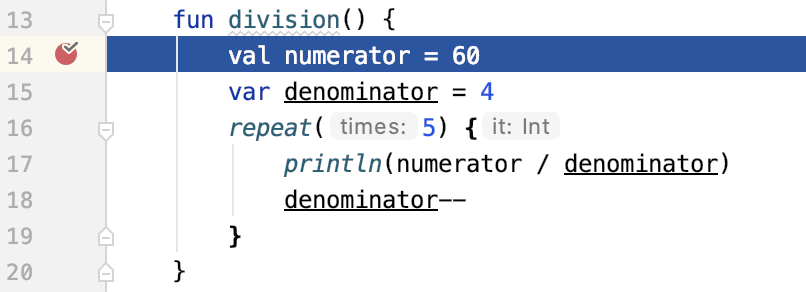
## **Add breakpoints to your project**

Perform the following steps to see breakpoints in action:

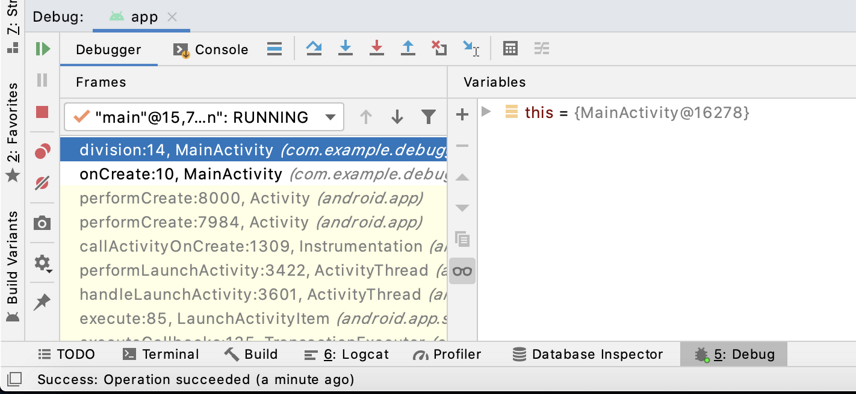
1. Add a breakpoint by clicking the gutter next to the line number you want to pause at. A dot will appear next to the line number, and the line will be highlighted.



Run your app with the debugger attached using **Run > Debug ‘app'** or the  icon in the toolbar. When the app launches, you should see a screen like this



At the bottom of the screen where you previously viewed the Logcat window, a new **Debug** tab has opened up.

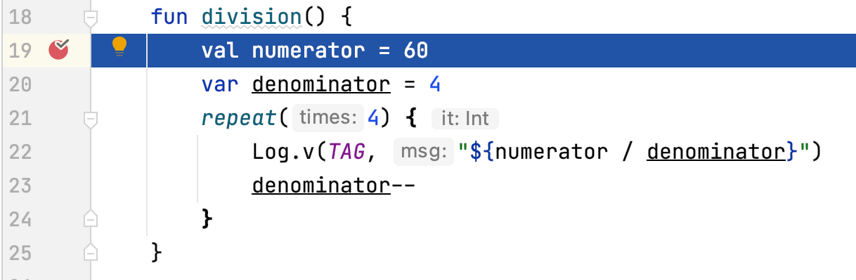


On the left side is a list of functions, which are the same as the list that appeared in the stack trace. On the right side is a pane where you can check the values of individual variables in the current function (i.e. division()). At the top, there are also buttons that help you navigate your program while it's paused. The button you'll use most often is **Step Over,** which executes the single highlighted line of code.

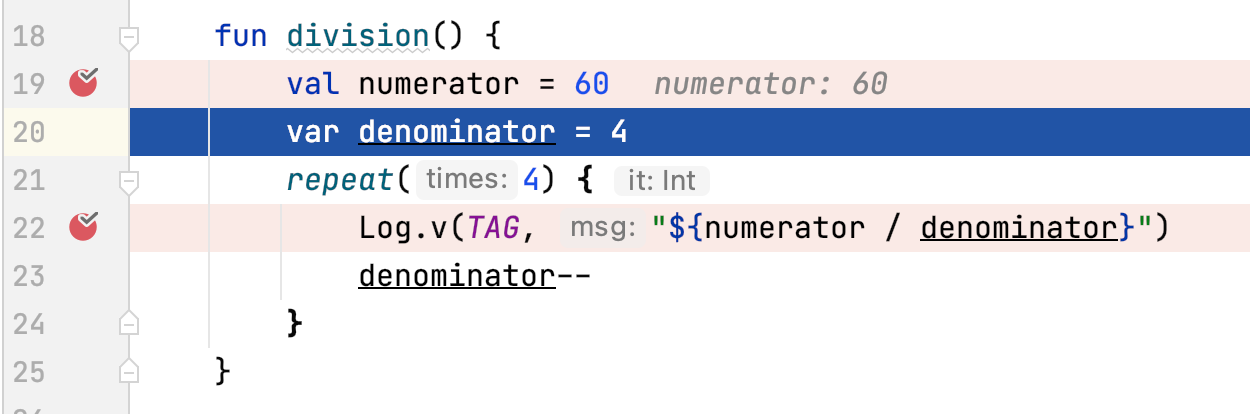


Perform the following steps to debug the code:

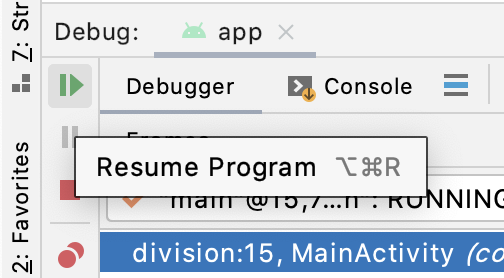
1. After the breakpoint was reached, line 19 (declaring the numerator variable) is now highlighted but hasn't run yet. Use the **Step Over**  button to execute line 19. Now line 20 will be highlighted.



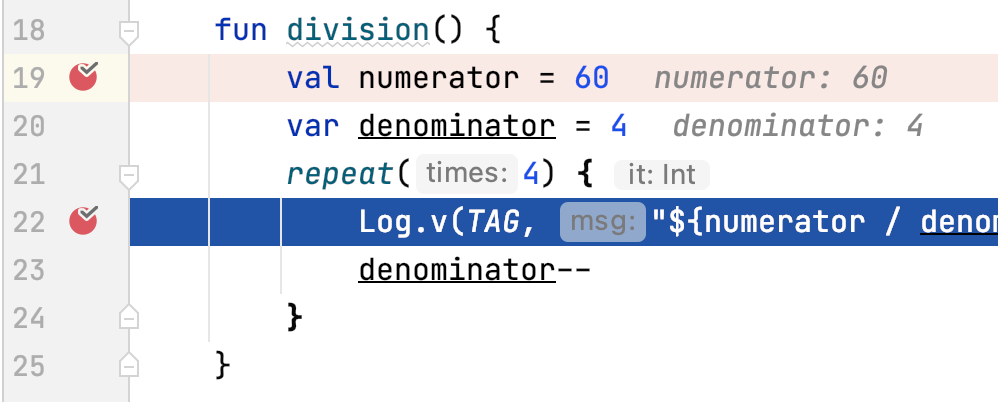
Set a breakpoint at line 22. This is where the division occurred and is the line that the stack trace reported the exception.



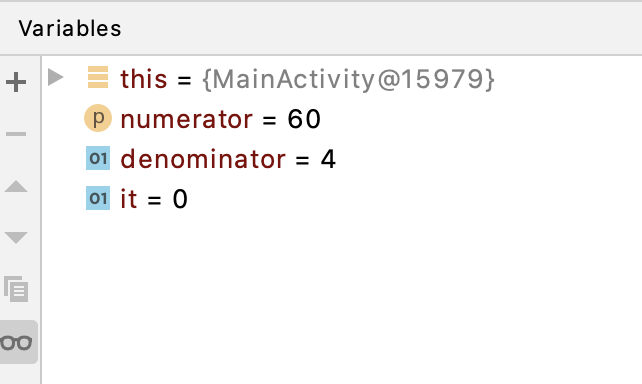
1. Use the **Resume Program**  button at the left of the **Debug** window to go to the next breakpoint. Run the rest of the division() function.



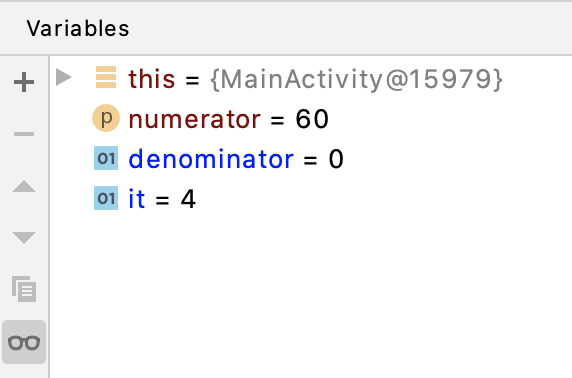
Notice that execution stops at line 17 before executing it.



The values of each variable numerator and denominator are shown next to their declarations. The values of the variables can be seen in the debug window in the **Variables** tab.



Press the **Resume Program** button on the left of the debug window four more times. Each time the loop pauses and observes the values of numerator and denominator. On the last iteration, numerator should be 60, and denominator should be 0. And you can't divide 60 by 0!



Now you know the exact line of code that causes the bug, and know the exact reason. Like before, you can fix the bug by changing the number of times to repeat the code from 5 to 4.

fun division() {

val numerator = 60

var denominator = 4

repeat(4) {

Log.v(TAG, "${numerator / denominator}")

denominator--

}

}

To remove a breakpoint, click on the dot that appears next to the line number.

## [**4. Set conditions for breakpoints**](https://developer.android.com/codelabs/basic-android-kotlin-training-debugging-with-breakpoints#3)

In the previous section, you had to step through each iteration of the loop until the denominator was zero. In more complicated apps, this can be tedious when you have less information about the bug. However, if you have an assumption, such as the app only crashing when the denominator is zero, you can modify your breakpoint so that it will only be reached when this assumption is met, rather than having to step through each iteration of the loop.

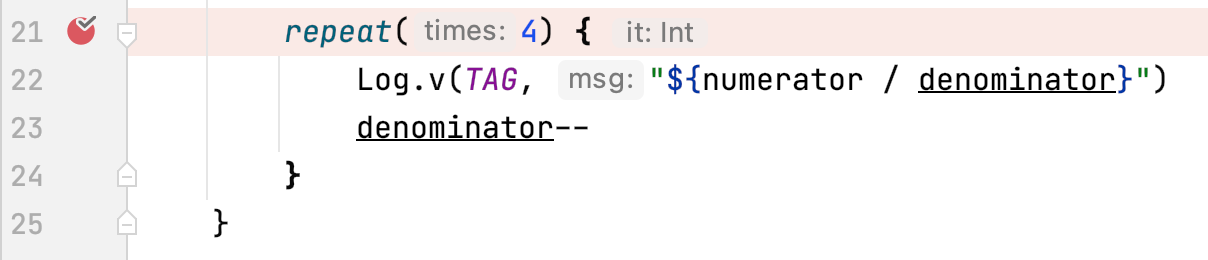
1. If necessary, re-introduce the bug by changing 4 to 5 in the repeat loop.

repeat(4) {

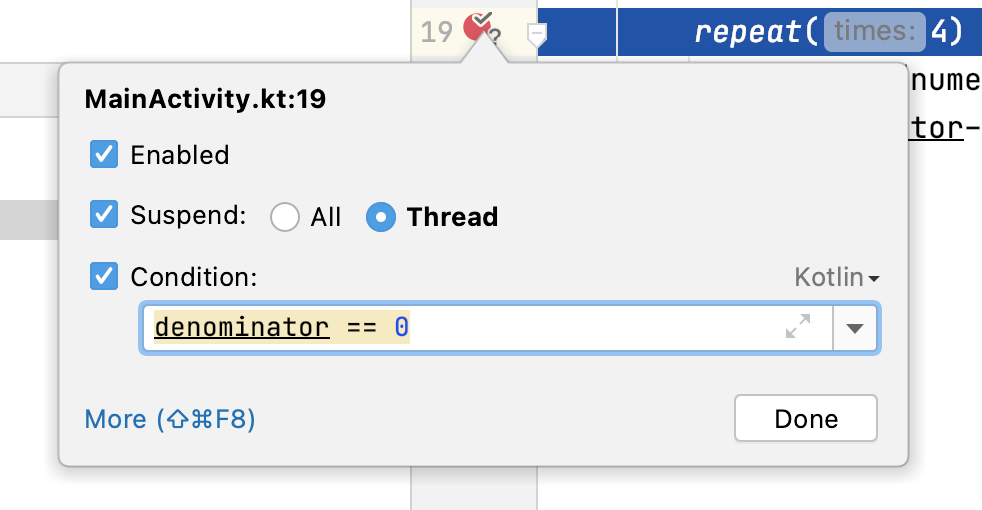
    ...

}

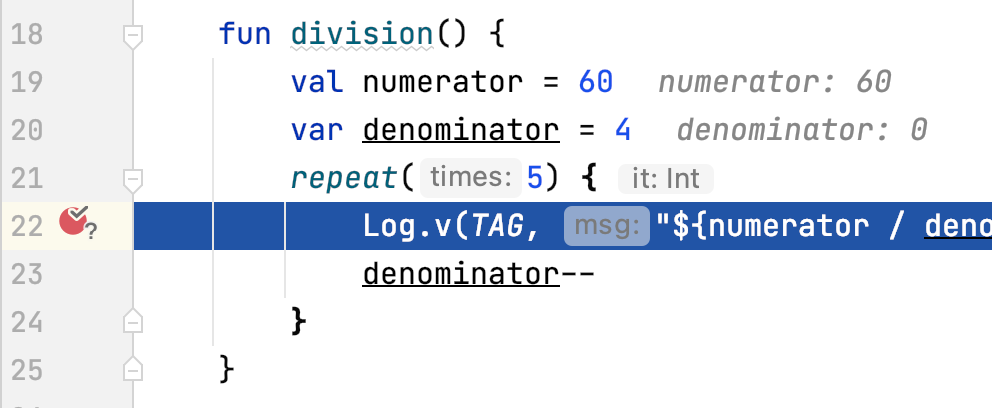
1. Place a new breakpoint on the line with the repeat statement.



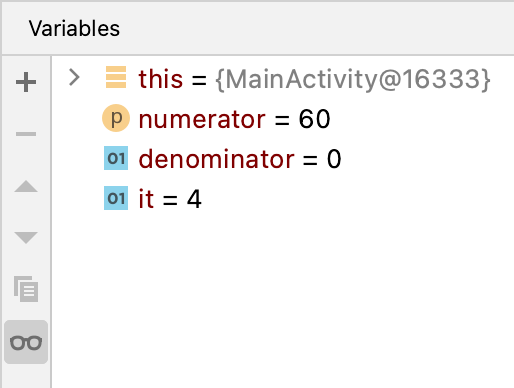
1. Right click on the red breakpoint icon. A menu will appear with a few options, such as whether or not the breakpoint is enabled. A disabled breakpoint still exists, but won't be triggered at runtime. You also have an option to add a Kotlin expression that if it evaluates to true, the breakpoint will be triggered. For example, if you used the expression denominator > 3, the breakpoint would only be triggered on the first iteration of the loop. To only trigger the breakpoint when your app is potentially going to divide by zero, set the expression to denominator == 0. The options for the breakpoint should look like the following:



1. Run your app using **Run > Debug ‘app'** and observe that the breakpoint is reached.



You can see that the denominator is already 0. The breakpoint was only triggered when the condition was met, saving you time and effort to step through the code.

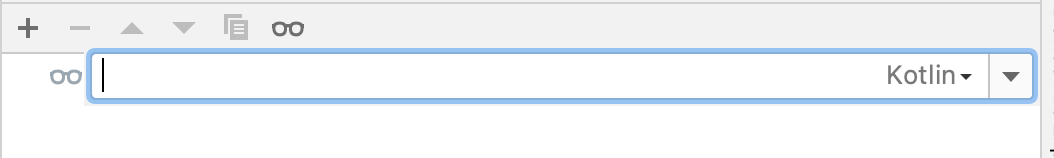


1. Like before, you see that the bug is caused by the loop executing one too many times, where the denominator was set to 0.

## **Add watches**

If you want to monitor a specific value while debugging, you don't need to search in the **Variables** tab to find it. You can add something called Watches to monitor specific variables. These variables will be visible in the debug pane. When execution is paused and that variable is in scope, it will be visible in the Watches pane. This makes your debugging more efficient when working with larger projects. You'll be able to keep track of all the relevant variables in one place.

1. In the debug view, to the right of the variables pane there should be another empty pane called **Watches**. Click the plus  button in the top left corner. You may see a menu option that says **New Watch**.



1. Type the name of the variable, denominator, in the provided field and click enter.
2. Re-run your app with **Run > debug ‘app'**, and observe that when the breakpoint is hit, you'll see the value of the denominator in the Watches pane.