|  |  |  |
| --- | --- | --- |
|  | **2, LEBUH ACHEH, GEORGE TOWN**  **10300 GEORGE TOWN**  **PULAU PINANG**  **INFORMATION SHEET** | |
| **PROGRAM’S CODE & NAME** | J620-002-4:2020 FRONT-END SOFTWARE DEVELOPMENT | |
| **LEVEL** | FOUR (4) | |
| **COMPETENCY UNIT NO. AND TITLE** | J620-002-4:2020-C04 MOBILE APPLICATION WITH THIRD PARTY API DEVELOPMENT | |
| **WORK ACTIVITIES NO. AND STATEMENT** | 1. CREATE MOBILE APP DESIGN MOCK-UP ELEMENTS. 2. PLAN MOBILE APP DESIGN STRUCTURE. 3. TRANSFORM MOCK-UP TO MOBILE APP. 4. INTEGRATE MOBILE APP WITH DATA SOURCE. 5. **VERIFY SUCCESSFUL API INTEGRATION** 6. VERIFY DEVELOPED MOBILE APP. 7. VERIFY MOBILE APP ACCESSIBLE GLOBALLY. | |
| **CODE NO.** | J620-002-4:2020-C04/IS(9/15) | Page: 1 of |

**TITLE**:

**DEBUGGING AND LOGGING TOOLS ON INTEGRATED DEVELOPMENT ENVIRONMENT**

**PURPOSE**:

This information sheet is intended to provide insight and knowledge to trainees with regards to the fundamentals of debugging and logging tools in the integrated development environment.

**INFORMATION:**

This information sheet provides useful notes and explanations on fundamental concepts of debugging and logging tools in the integrated development environment.

# **DEBUGGING YOUR APPLICATION**

Android Studio provides a debugger that allows you to do the following and more:

* Select a device to debug your app on.
* Set breakpoints in your Java, Kotlin, and C/C++ code.
* Examine variables and evaluate expressions at runtime.

This page includes instructions for basic debugger operations. For more documentation, also see the [IntelliJ IDEA debugging docs](https://www.jetbrains.com/help/idea/2020.2/debugging.html).

## Enabling debugging

Before you can begin debugging, you need to prepare as follows:

1. Enable debugging on your device:

If you're using the emulator, this is enabled by default. But for a connected device, you need to enable debugging in the device developer options.

1. Run a debuggable build variant:

You must use a build variant that includes debuggable true in the build configuration. Usually, you can just select the default "debug" variant that's included in every Android Studio project (even though it's not visible in the build.gradle file). But if you define new build types that should be debuggable, you must add `debuggable true` to the build type:

android {

buildTypes {

create("customDebugType") {

debuggable = true

...

}

}

}

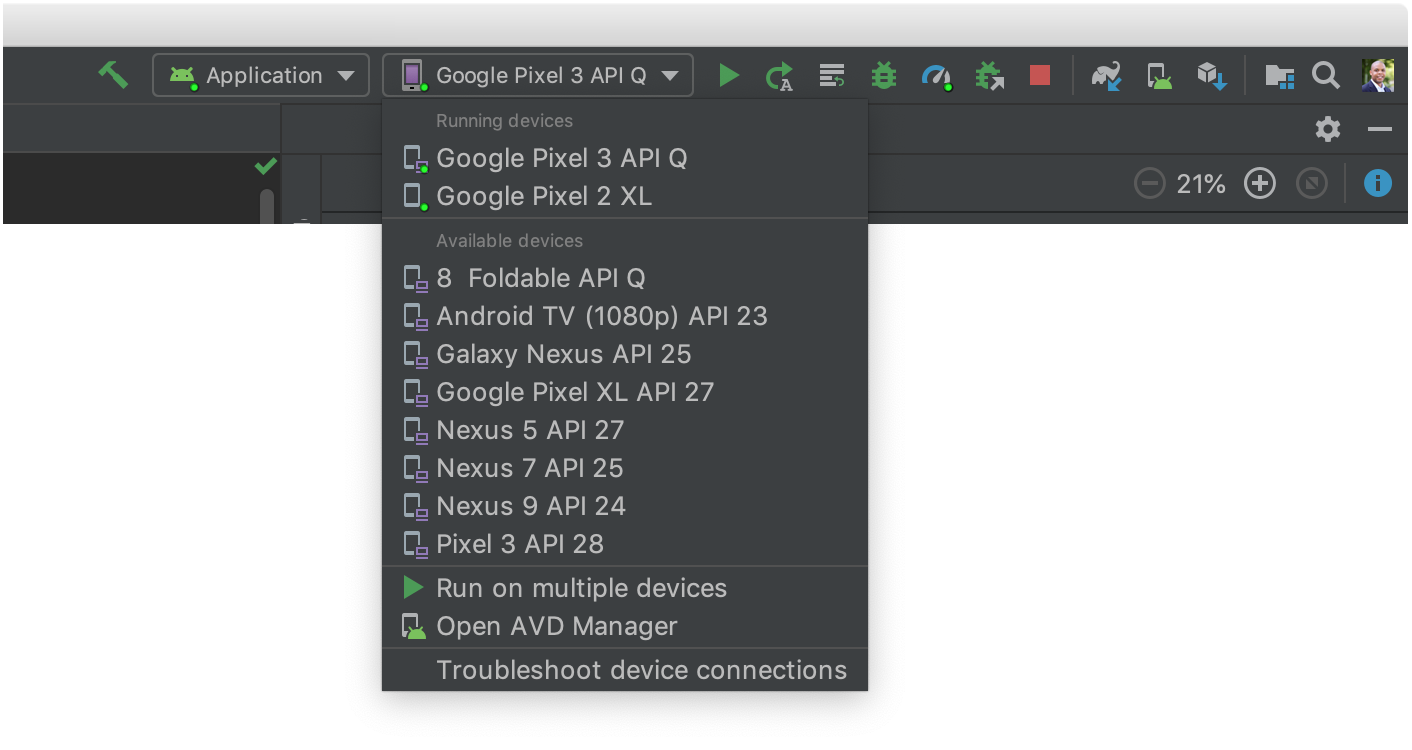
This property also applies to modules with C/C++ code. (The jniDebuggable property is no longer used.)

If your app depends on a library module that you also want to debug, that library must also be packaged with debuggable true so it retains its debug symbols. To ensure that the debuggable variants of your app project receive the debuggable variant of a library module, be sure that you publish non-default versions of your library.

## Starting debugging

You can start a debugging session as follows:

1. Set some breakpoints in the app code.
2. In the toolbar, select a device to debug your app on from the target device drop-down menu.



If you don't have any devices configured, then you need to either connect a device via USB or create an AVD to use the Android Emulator.

1. In the toolbar, click Debug .

If you see a dialog asking if you want to "switch from Run to Debug," that means your app is already running on the device and it will restart in order to begin debugging. If you'd rather keep the same instance of the app running, click Cancel Debug and instead attach the debugger to a running app.

Otherwise, Android Studio builds an APK, signs it with a debug key, installs it on your selected device, and runs it. If you add C and C++ code to your project, Android Studio also runs the LLDB debugger in the Debug window to debug your native code.

1. If the Debug window is not open, select View > Tool Windows > Debug (or click Debug in the tool window bar), and then click the Debugger tab, as shown in figure 1.

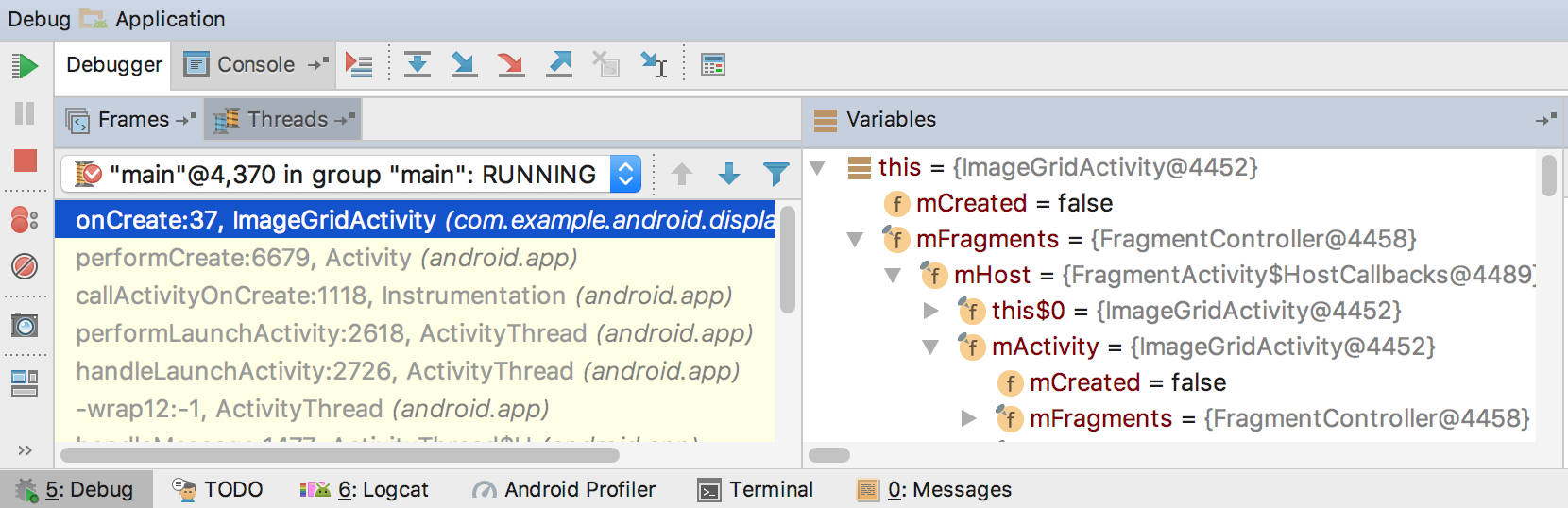


Figure 1. The Debugger window, showing the current thread and the object tree for a variable

## Attach the debugger to a running app

If your app is already running on your device, you can start debugging without restarting your app as follows:

1. Click Attach debugger to Android process .
2. In the Choose Process dialog, select the process you want to attach the debugger to.

If you're using an emulator or a rooted device, you can check Show all processes to see all processes.

From the Use Android Debugger Settings from drop-down menu, you can select an existing run/debug configuration. (For C and C++ code, this lets you reuse the LLDB startup commands, LLDB post-attach commands, and symbol directories in an existing configuration.) If you don't have an existing run/debug configuration, select Create New. This selection enables the Debug Type drop-down menu, where you can select a different debug type. By default, Android Studio uses the Auto debug type to select the best debugger option for you, based on whether your project includes Java or C/C++ code.

1. Click OK.
2. The Debug window appears.

## Change the debugger type

Because different debugger tools are required to debug Java/Kotlin code and C/C++ code, the Android Studio debugger allows you to select which debugger type to use. By default, Android Studio decides which debugger to use based on which languages it detects in your project (with the Auto debugger type). However, you can manually select the debugger in the debug configuration (click Run > Edit Configurations) or in the dialog that appears when you click Run > Attach debugger to Android process.

The debug types available include the following:

1. Auto

Select this debug type if you want Android Studio to automatically choose the best option for the code you are debugging. For example, if you have any C or C++ code in your project, Android Studio automatically uses the Dual debug type. Otherwise, Android Studio uses the Java debug type.

1. Java

Select this debug type if you want to debug only code written in Java or Kotlin—the Java debugger ignores any breakpoints or watches you set in your native code.

Native (available only with C/C++ code)

Select this debug type if you want to use only LLDB to debug your code. When using this debug type, the Java debugger session view is not available. By default, LLDB inspects only your native code and ignores breakpoints in your Java code. If you want to also debug your Java code, you should switch to either the **Auto** or **Dual** debug type.

Native debugging only works on devices that meet the following requirements:

* The device supports run-as.

To check whether the device supports run-as, run the following command on the ADB shell that is connected to your device:

run-as your-package-name pwd

* The device has ptrace enabled.

To check whether ptrace is enabled, run the following command on the ADB shell that is connected to your device:

sysctl kernel.yama.ptrace\_scope

## Dual (available only with C/C++ code)

Select this debug type if you want to switch between debugging both Java and native code. Android Studio attaches both the Java debugger and LLDB to your app process, one for the Java debugger and one for LLDB, so you can inspect breakpoints in both your Java and native code without restarting your app or changing your debug configuration.

In figure 2, notice the two tabs to the right of the Debug window title. Because the app has both Java and C++ code, one tab is for debugging the native code, and the other for debugging Java code, as indicated by -java.

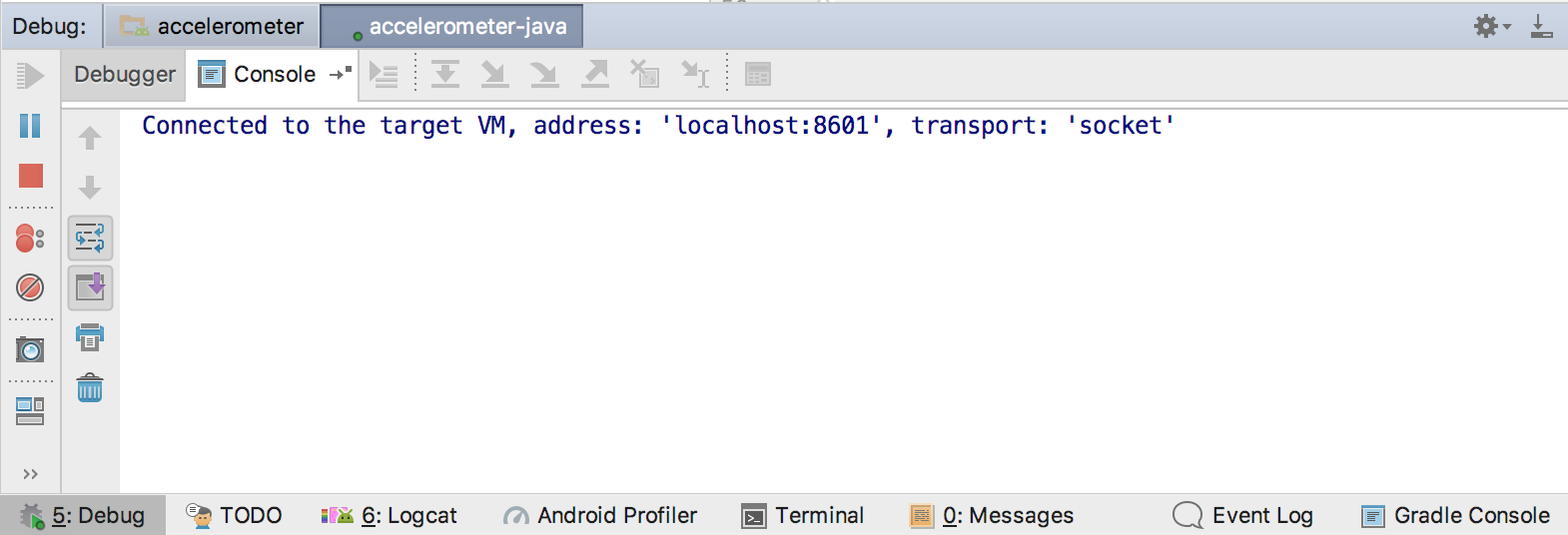


Figure 2. Tab for debugging native code and tab for debugging Java code

1. Use the system log

The system log shows system messages while you debug your app. These messages include information from apps running on the device. If you want to use the system log to debug your app, make sure your code writes log messages and prints the stack trace for exceptions while your app is in the development phase.

# **WRITING LOG MESSAGES IN YOUR CODE**

To write log messages in your code, use the Log class. Log messages help you understand the execution flow by collecting the system debug output while you interact with your app. Log messages can tell you what part of your application failed. For more information about logging, see Write and view logs.

The following example shows how you might add log messages to determine if previous state information is available when your activity starts:

import android.util.Log

...

private val TAG: String = MyActivity::class.java.simpleName

...

class MyActivity : Activity() {

...

override fun onCreate(savedInstanceState: Bundle?) {

...

if (savedInstanceState != null) {

Log.d(TAG, "onCreate() Restoring previous state")

/\* restore state \*/

} else {

Log.d(TAG, "onCreate() No saved state available")

/\* initialize app \*/

}

}

}

During development, your code can also catch exceptions and write the stack trace to the system log:

fun someOtherMethod() {

try {

...

} catch (e : SomeException) {

Log.d(TAG, "someOtherMethod()", e)

}

}

## Viewing the system log

You can view and filter debug and other system messages in the Logcat window. For example, you can see messages when garbage collection occurs, or messages that you add to your app with the Log class.

To use logcat, start debugging and select the Logcat tab in the bottom toolbar as shown in figure 3.

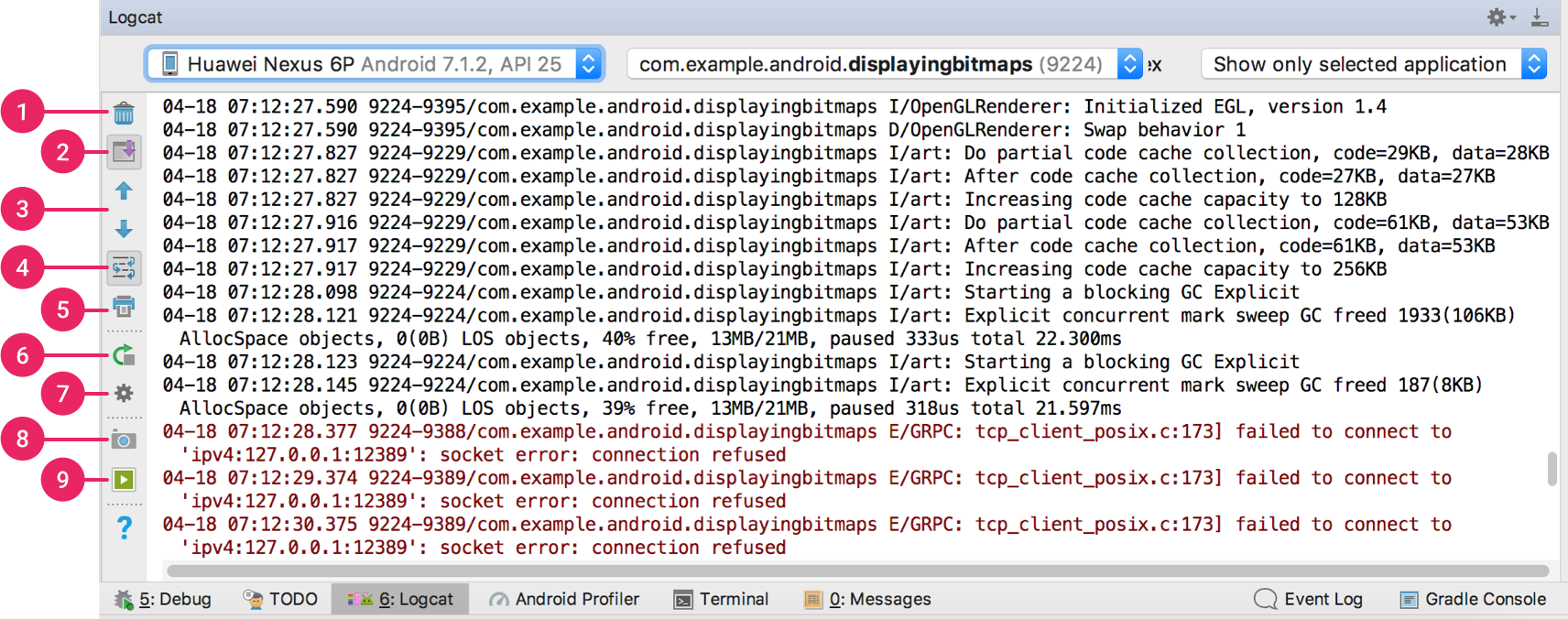


Figure 3. Logcat window with filter settings

## Work with breakpoints

Android Studio supports several types of breakpoints that trigger different debugging actions. The most common type is a line breakpoint that pauses the execution of your app at a specified line of code. While paused, you can examine variables, evaluate expressions, then continue execution line by line to determine the causes of runtime errors.

To add a line breakpoint, proceed as follows:

1. Locate the line of code where you want to pause execution, then either click the left gutter along that line of code or place the caret on the line and press Control+F8 (on Mac, Command+F8).
2. If your app is already running, you don't need to update it to add the breakpoint—just click Attach debugger to Android proccess . Otherwise, start debugging by clicking Debug .



Figure 3. A red dot appears next to the line when you set a breakpoint

When your code execution reaches the breakpoint, Android Studio pauses execution of your app. You can then use the tools in the Debugger tab to identify the state of the app:

* To examine the object tree for a variable, expand it in the Variables view. If the Variables view is not visible, click Restore Variables View .
* To evaluate an expression at the current execution point, click Evaluate Expression .
* To advance to the next line in the code (without entering a method), click Step Over .
* To advance to the first line inside a method call, click Step Into .
* To advance to the next line outside the current method, click Step Out .
* To continue running the app normally, click Resume Program .

If your project uses any native code, by default the Auto debug type attaches both the Java debugger and LLDB to your app as two separate processes, so you can switch between inspecting Java and C/C++ breakpoints without restarting your app or changing settings.

When Android Studio deploys your app to your target device, the Debug window opens with a tab or debug session view for each debugger process, as shown in figure 4.

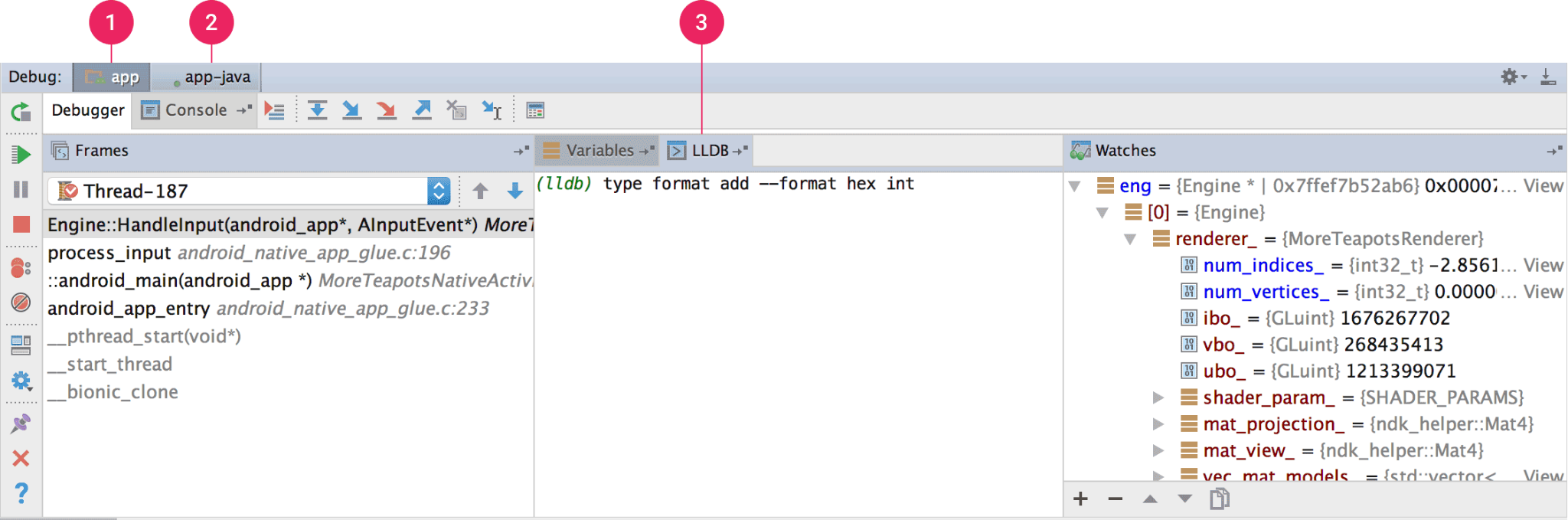


Figure 4. Debugging native code using LLDB

1. Android Studio switches to the <your-module> tab when LLDB debugger encounters a breakpoint in your C/C++ code. The Frames, Variables, and Watches panes are also available and work exactly as they would if you were debugging Java code. Although the Threads pane is not available in the LLDB session view, you can access your app processes using the drop-down list in the Frames pane. You can learn more about these panes in the sections about how to debug window frames and inspect variables.
2. Android Studio switches to the <your-module>-java tab when the Java debugger encounters a breakpoint in your Java code.
3. While debugging with LLDB, you can use the LLDB terminal in the LLDB session view to pass command line options to LLDB. If you have certain commands that you would like LLDB to execute each time you start debugging your app, either just before or just after the debugger attaches to your app process, you can add those commands to your debug configuration.

While debugging C/C++ code, you can also set special types of breakpoints, called watchpoints, that can suspend your app process when your app interacts with a particular block of memory. To learn more, read the section about how to add watchpoints.

1. View and configure breakpoints

To view all the breakpoints and configure breakpoint settings, click View Breakpoints on the left side of the Debug window. The Breakpoints window appears, as shown in figure 5.

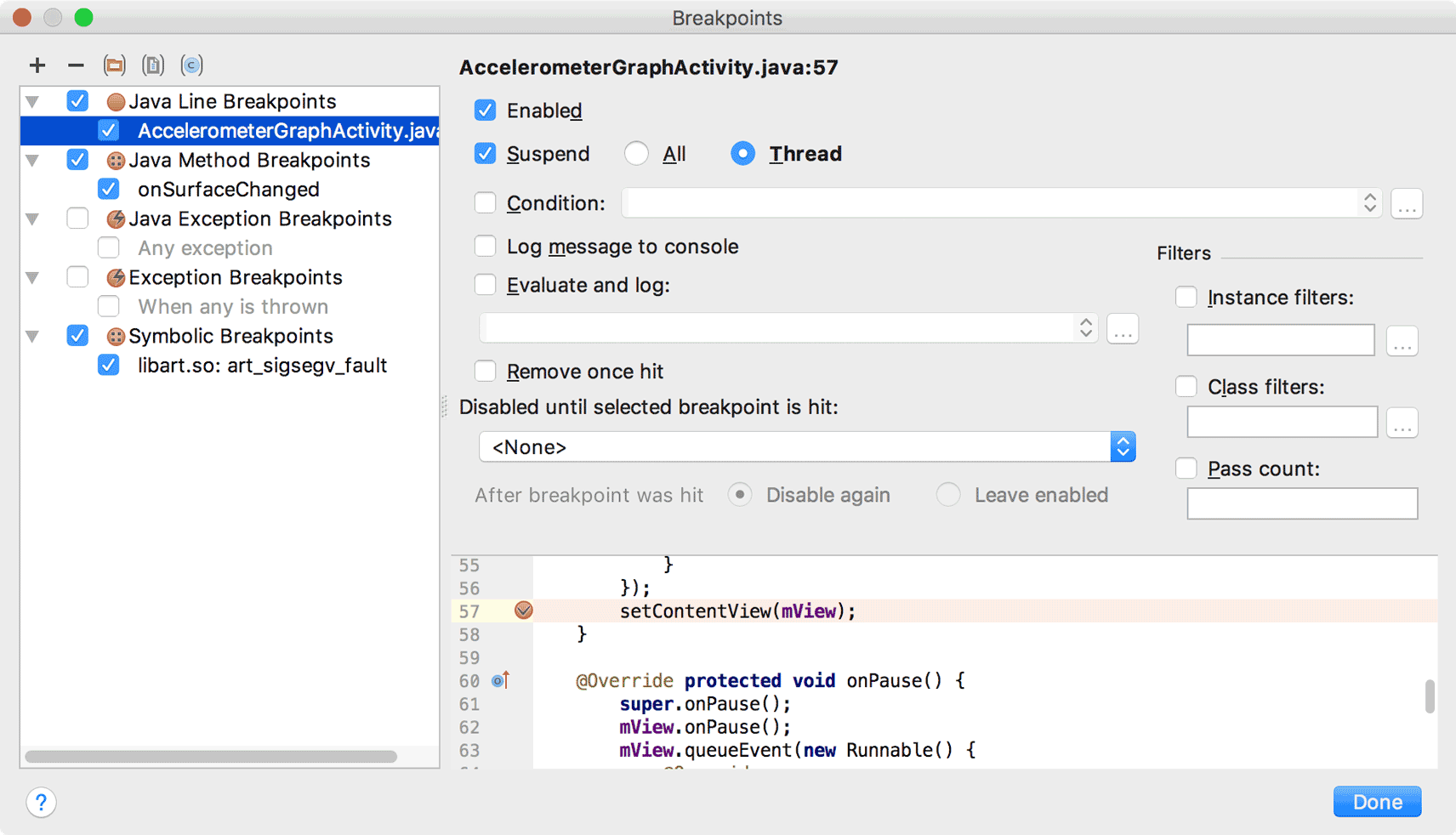


Figure 5. The Breakpoints window lists all the current breakpoints and includes behavior settings for each

The Breakpoints window lets you enable or disable each breakpoint from the list on the left. If a breakpoint is disabled, Android Studio does not pause your app when it hits that breakpoint. Select a breakpoint from the list to configure its settings. You can configure a breakpoint to be disabled at first and have the system enable it after a different breakpoint is hit. You can also configure whether a breakpoint should be disabled after it is hit. To set a breakpoint for any exception, select Exception Breakpoints in the list of breakpoints.

1. Debug window frames

In the Debugger window, the Frames pane lets you inspect the stack frame that caused the current breakpoint to be hit. This enables you to navigate and examine the stack frame and also inspect the list of threads in your Android app. To select a thread, use the thread selector drop-down and view its stack frame. Clicking the elements in the frame opens the source in the editor. You can also customize the thread presentation and export the stack frame as discussed in the Window Frames guide.

1. Inspect variables

In the Debugger window, the Variables pane lets you inspect variables when the system stops your app on a breakpoint and you select a frame from the Frames pane. The Variables pane also lets you evaluate ad-hoc expressions using static methods and/or variables available within the selected frame.

The Watches pane provides similar functionality except that expressions added to the Watches pane persist between debugging sessions. You should add watches for variables and fields that you access frequently or that provide state that is helpful for the current debugging session. The Variables and Watches panes appear as shown in figure 5.

To add a variable or expression to the Watches list, follow these steps:

1. Begin debugging.
2. In the Watches pane, click Add .
3. In the text box that appears, type the name of the variable or expression you want to watch and then press Enter.
4. To remove an item from the Watches list, select the item and then click Remove .

You can reorder the elements in the Watches list by selecting an item and then clicking Up or Down .

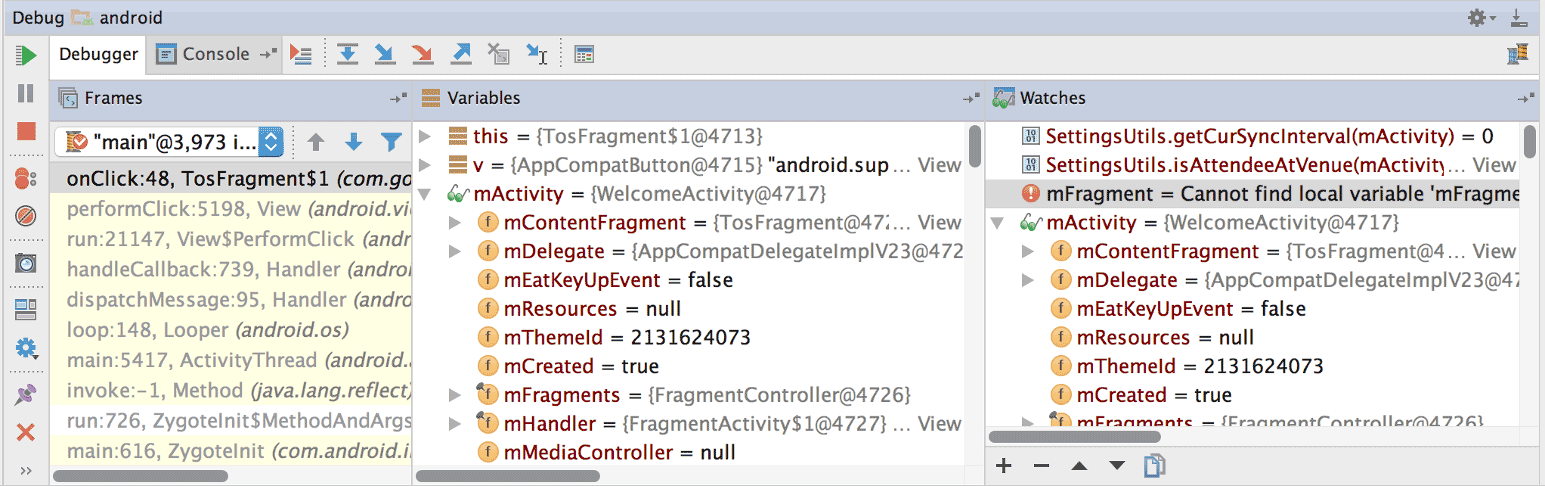


Figure 6. The Variables and Watches panes in the Debugger window

1. Add watchpoints

While debugging C/C++ code, you can set special types of breakpoints, called watchpoints, that can suspend your app process when your app interacts with a particular block of memory. For example, if you set two pointers to a block of memory and assign a watchpoint to it, using either pointer to access that block of memory triggers the watchpoint.

In Android Studio, you can create a watchpoint during runtime by selecting a specific variable, but LLDB assigns the watchpoint to only the block of memory the system allocates to that variable, not the variable itself. This is different from adding a variable to the Watches pane, which enables you to observe the value of a variable but doesn’t allow you to suspend your app process when the system reads or changes its value in memory.

To set a watchpoint, you must meet the following requirements:

* Your target physical device or emulator uses an x86 or x86\_64 CPU. If your device uses an ARM CPU, then you must align the boundary of your variable’s address in memory to either 4 bytes for 32-bit processors, or 8 bytes for 64-bit processors. You can align a variable in your native code by specifying \_\_attribute\_\_((aligned(num\_bytes))) in the variable deceleration, as shown below:

// For a 64-bit ARM processor

int my\_counter \_\_attribute\_\_((aligned(8)));

* You have assigned three or fewer watchpoints already. Android Studio only supports up to four watchpoints on x86 or x86\_64 target devices. Other devices may support fewer watchpoints.

If you meet the requirements above, you can add a watchpoint as follows:

1. While your app is suspended on a breakpoint, navigate to the Variables pane in your LLDB session view.
2. Right-click on a variable that occupies the block of memory you want to track and select Add Watchpoint. A dialog to configure your watchpoint appears, as shown in figure 7.

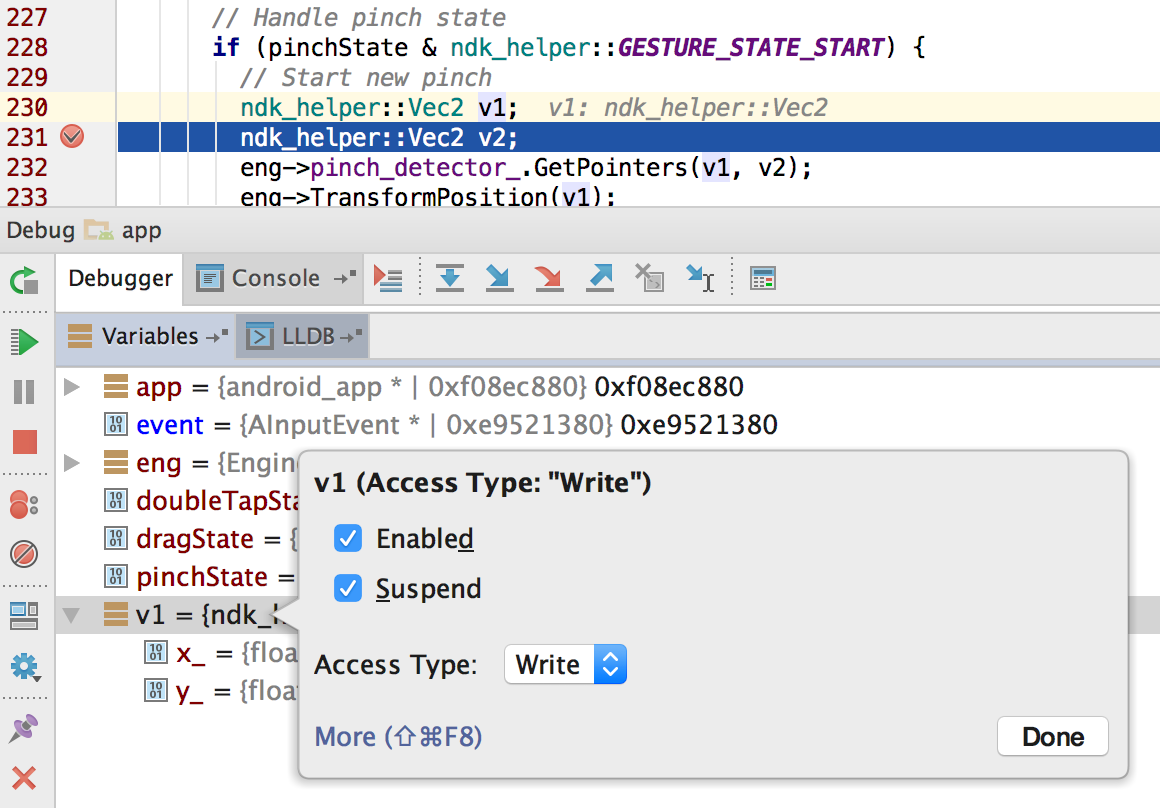


Figure 7 Adding a watchpoint to a variable in memory

1. Configure your watchpoint with the following options:

* Enabled: You can deselect this option if you want to tell Android Studio to ignore the watchpoint for the time being. Android Studio still saves your watchpoint so you can access it later in your debug session.
* Suspend: By default, the Android system suspends your app process when it accesses a block of memory you assign to a watchpoint. You can deselect this option if you don’t want this behavior—this reveals additional options you can use to customize behavior when the system interacts with your watchpoint: Log message to console and Remove [the watchpoint] when hit.
* Access Type: Select whether your app should trigger your watchpoint when it tries to Read or Write to the block of memory the system allocates to the variable. To trigger your watchpoint on either a read or write, select Any.

1. Click Done.

To view all your watchpoints and configure watchpoint settings, click View Breakpoints on the left side of the Debug window. The Breakpoints dialog appears, as shown in figure 8.

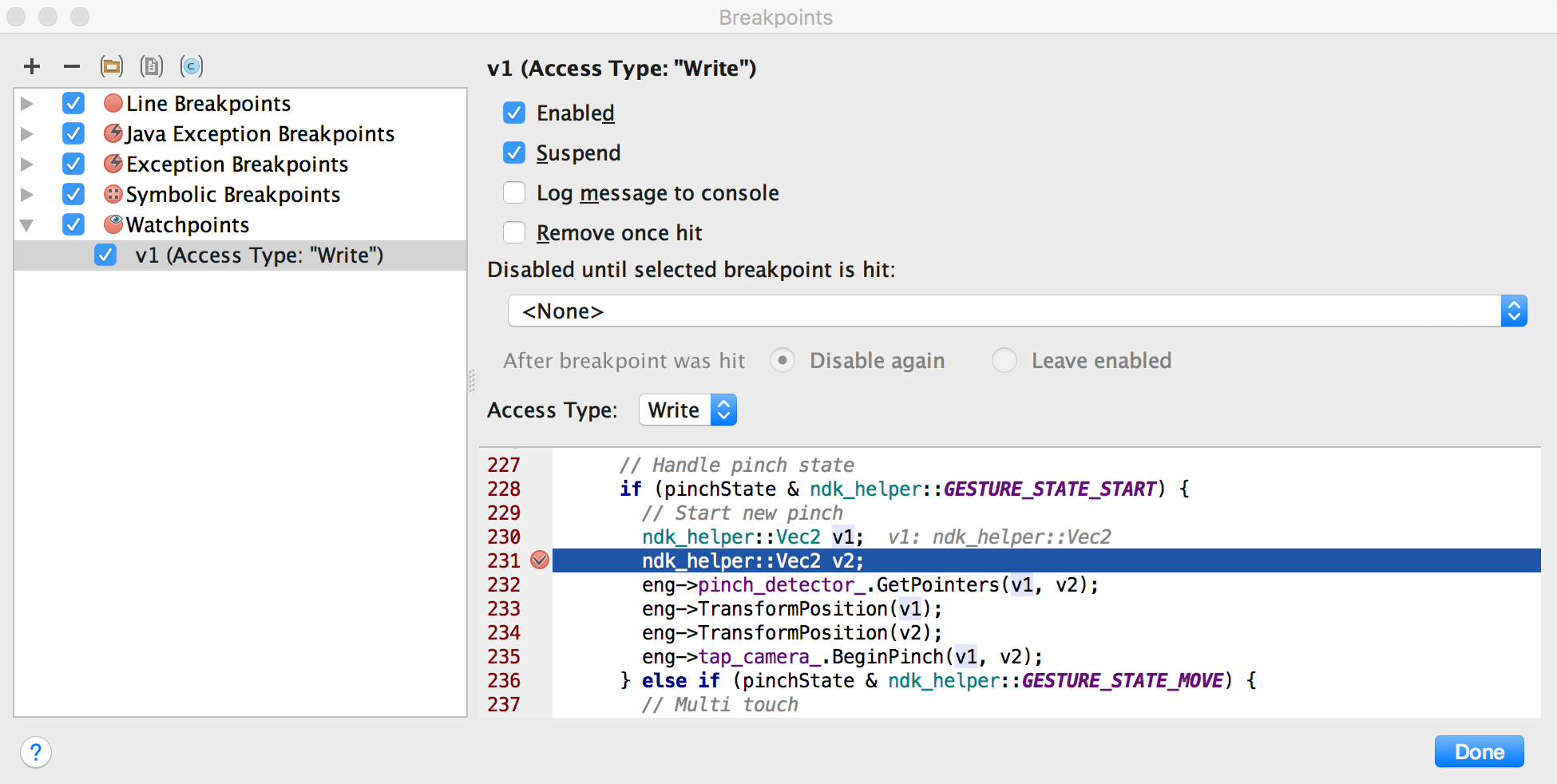


Figure 8. The Breakpoints dialogue lists your current watchpoints and includes behavior settings for each

After you add your watchpoint, click Resume Program on the left side of the Debug window to resume your app process. By default, if your app tries to access a block of memory that you have set a watchpoint to, the Android system suspends your app process and a watchpoint icon appears next to the line of code that your app executed last, as shown in figure 9.

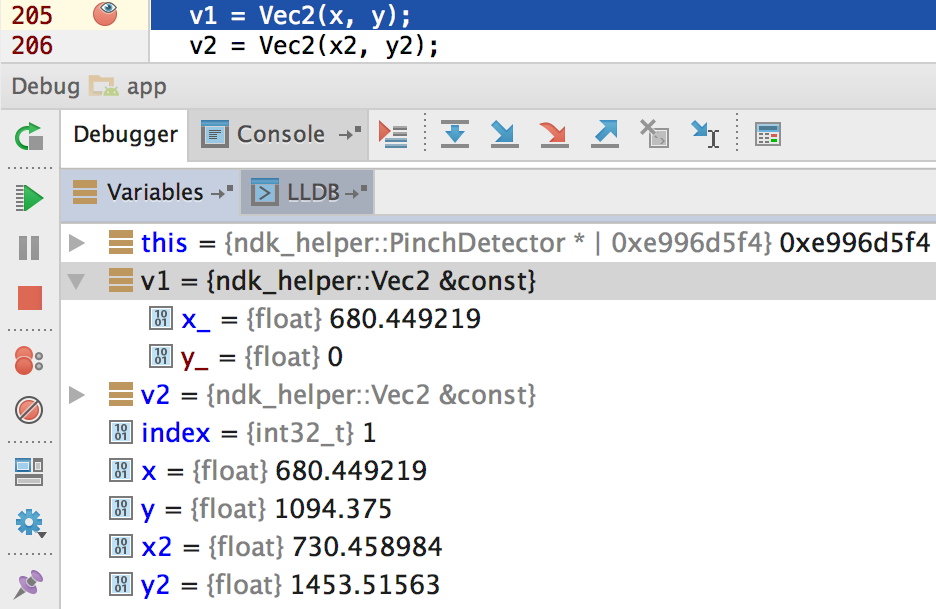


Figure 9. Android Studio indicates the line of code that your app executes just before triggering a watchpoint

1. View and change resource value display format

In debug mode, you can view resource values and select a different display format for variables in your Java code. With the Variables tab displayed and a frame selected, do the following:

1. In the Variables list, right-click anywhere on a resource line to display the drop-down list.
2. In the drop-down list, select View as and select the format you want to use. The available formats depend on the data type of the resource you selected. You might see any one or more of the following options:

* Class: Display the class definition.
* toString: Display string format.
* Object: Display the object (an instance of a class) definition.
* Array: Display in an array format.
* Timestamp: Display date and time as follows: yyyy-mm-dd hh:mm:ss.
* Auto: Android Studio chooses the best format based on the data type.
* Binary: Display a binary value using zeroes and ones.
* MeasureSpec: The value passed from the parent to the selected child. See MeasureSpec.
* Hex: Display as a hexadecimal value.
* Primitive: Display as a numeric value using a primitive data type.
* Integer: Display a numeric value of type Integer.

You can create a custom format (data type renderer), as follows:

1. Right-click the resource value.
2. Select View as.
3. Select Create. The Java Data Type Renderers dialog displays.
4. Follow the instructions at Java Data type renderers

**QUESTIONS:**

1. Different debugger tools are required to debug Java/Kotlin code and C/C++ code. Explain the types of debugger available in Android studio.

**Answer:**

* **Auto**
* **Java**
* **Native (available only with C/C++ code)**

**REFERENCE:**

<https://developer.android.com/studio/debug>