Cómo depurar tu app

Android Studio proporciona un depurador que te permite realizar las siguientes acciones y más:

* Seleccionar un dispositivo en el cual depurarás tu app
* Establecer interrupciones en tu código Java, Kotlin y C/C++
* Examinar variables y evaluar expresiones en el tiempo de ejecución

En esta página, se incluyen instrucciones para las operaciones básicas de depuración. Para obtener más documentación, también puedes consultar los [documentos de depuración de IDEA de IntelliJ](https://www.jetbrains.com/help/idea/2019.1/debugging.html).

Cómo habilitar la depuración

Antes de comenzar a depurar, debes prepararte de la siguiente manera:

* **Instala el LLDB:**

Si tu proyecto incluye código C o C++, necesitarás instalar el **LLBD** desde [SDK Manager](https://developer.android.com/studio/intro/update.html#sdk-manager).

* **Habilita la depuración en tu dispositivo:**

Si estás usando el emulador, esta opción estará activada de forma predeterminada. Sin embargo, en el caso de un dispositivo conectado, deberás [habilitar la depuración en las opciones para desarrolladores del dispositivo](https://developer.android.com/studio/debug/dev-options.html).

* **Ejecuta una variante de compilación depurable:**

Debes usar una [variante de compilación](https://developer.android.com/studio/build/build-variants.html) que incluya [debuggable true](http://google.github.io/android-gradle-dsl/current/com.android.build.gradle.internal.dsl.BuildType.html#com.android.build.gradle.internal.dsl.BuildType:debuggable) en la configuración de compilación. Por lo general, simplemente puedes seleccionar la variante de "depuración" predeterminada que se incluye en cada proyecto de Android Studio (aunque no esté visible en el archivo build.gradle). Sin embargo, si defines nuevos tipos de compilación que deberían ser depurables, debes agregar "debuggable true" al tipo de compilación:

    android {  
        buildTypes {  
            customDebugType {  
                debuggable true  
                ...  
            }  
        }  
    }

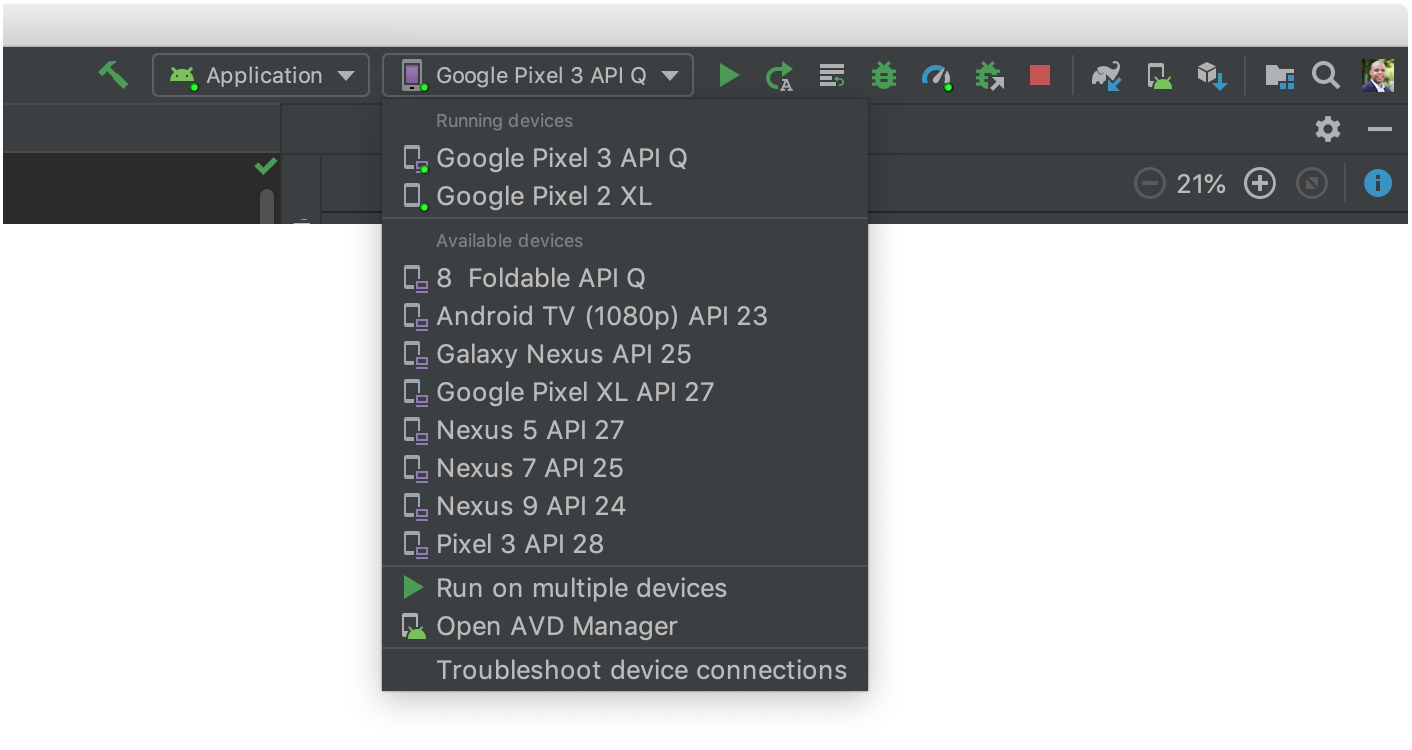
Esta propiedad también se aplica a los [módulos con código C o C++](https://developer.android.com/studio/projects/add-native-code.html) (ya no se usa la propiedad jniDebuggable).

**Nota**: Si tu app depende de un módulo de biblioteca que también quieres depurar, esta biblioteca también debe empaquetarse con **debuggable true** de manera que conserve sus símbolos de depuración. Para garantizar que las variantes depurables del proyecto de tu app reciban la variante depurable de un módulo de biblioteca, asegúrate de [publicar versiones no predeterminadas de tu biblioteca](https://developer.android.com/studio/projects/android-library.html#publish_multiple_variants).

Cómo iniciar la depuración

Puedes iniciar una sesión de depuración de la siguiente manera:

1. Establece algunas [interrupciones](https://developer.android.com/studio/debug#breakPoints) en el código de la app.
2. En la barra de herramientas, selecciona el dispositivo en el que depurarás la app en el menú desplegable de dispositivos.

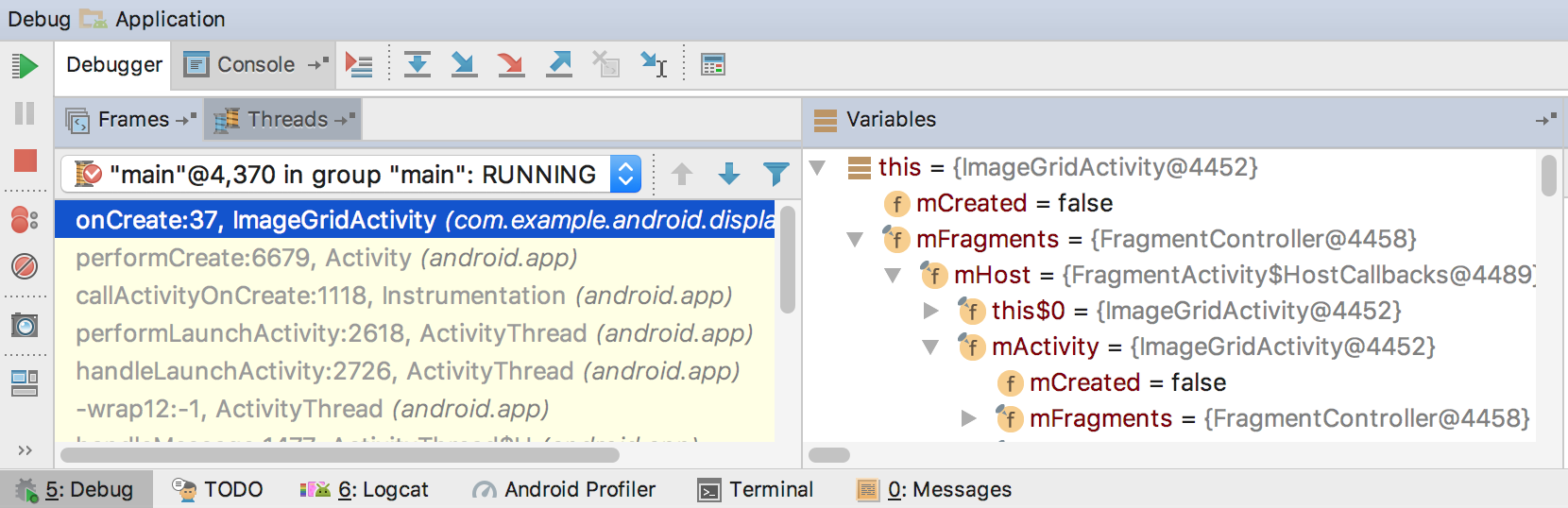


Si no tienes ningún dispositivo configurado, debes [conectar un dispositivo mediante USB](https://developer.android.com/studio/run/device#connect) o [crear un AVD](https://developer.android.com/studio/run/managing-avds#createavd) para usar el [emulador de Android](https://developer.android.com/studio/run/emulator.html).

1. En la barra de herramientas, haz clic en **Debug**.

Si se muestra el diálogo "switch from Run to Debug" (cambiar del modo de ejecución al de depuración), significa que tu app ya se está ejecutando en el dispositivo y se reiniciará para comenzar con la depuración. Si prefieres conservar la misma instancia de la app en ejecución, haz clic en **Cancel Debug** y, en su lugar, [adjunta el depurador a la app en ejecución](https://developer.android.com/studio/debug#attach-debugger).

Android Studio compila un APK, lo firma con una clave de depuración, lo instala en el dispositivo seleccionado y lo ejecuta. Si [agregas código C y C++ a tu proyecto](https://developer.android.com/studio/projects/add-native-code.html), Android Studio también ejecutará el [depurador LLDB](http://lldb.llvm.org/) en la ventana **Debug** para depurar tu código nativo.

1. Si la ventana **Debug** no está abierta, selecciona **View > Tool Windows > Debug** (o haz clic en **Debug**  en la barra de la ventana de herramientas) y, luego, haz clic en la pestaña **Debugger**, como se muestra en la figura 1. 

**Figura 1:** La ventana "Debugger", en la que se muestra el subproceso actual y el árbol de objetos para una variable

Cómo adjuntar el depurador en una app en ejecución

Si tu app ya se está ejecutando en tu dispositivo, puedes comenzar con la depuración sin reiniciarla, de la siguiente manera:

1. Haz clic en **Attach debugger to Android process** .
2. En el diálogo **Choose Process**, selecciona el proceso al que desees adjuntar el depurador.

Si estás usando un emulador o un dispositivo con derechos de administrador, puedes marcar la opción **Show all processes** para ver todos los procesos.

En el menú desplegable **Debugger**, puedes [seleccionar un tipo de depuración diferente](https://developer.android.com/studio/debug#debug-types). De forma predeterminada, Android Studio usa el tipo de depuración **Auto** para seleccionar la mejor opción de depuración en función de si tus proyectos incluyen código Java o C/C++.

1. Haz clic en **OK**.

Aparecerá la ventana **Debug**.

**Nota:** El depurador y el recolector de elementos no utilizados de Android Studio se vinculan de forma flexible. La máquina virtual de Android garantiza que cualquier objeto que el depurador identifique no se recolecte como elemento no utilizado hasta que se desconecte el depurador, lo cual puede generar una acumulación de objetos con el tiempo mientras el depurador esté conectado. Por ejemplo, si el depurador ve un subproceso en ejecución, no se recolecta el objeto [**Thread**](https://developer.android.com/reference/java/lang/Thread.html) asociado como elemento no utilizado hasta que se desconecta el depurador, aunque haya finalizado el subproceso.

Cómo cambiar el tipo de depurador

Debido a que se requieren diferentes herramientas de depuración para depurar el código Java o Kotlin y el código C/C ++, el depurador de Android Studio te permite seleccionar qué tipo de depurador usar. De forma predeterminada, Android Studio decide qué depurador usar según los idiomas que detecta en tu proyecto (con el tipo de depurador **Auto**). Sin embargo, puedes seleccionar manualmente el depurador en la [configuración de depuración](https://developer.android.com/studio/run/rundebugconfig.html) (haz clic en **Run > Edit > Configurations**) o en el diálogo que aparece cuando haces clic en **Run > Attach debugger to Android process**.

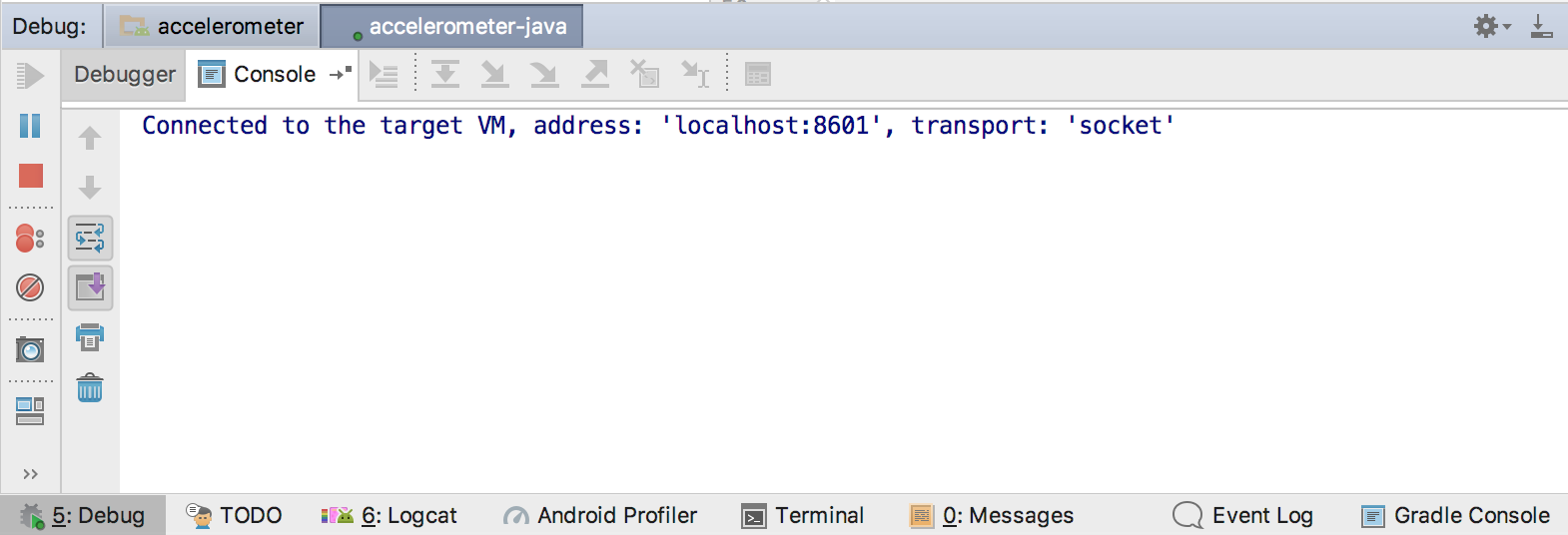
Entre los tipos de depuración disponibles, se incluyen los siguientes:

* **Auto**: Selecciónalo si deseas que Android Studio elija de forma automática la mejor opción para el código que depures. Por ejemplo, si tienes código C o C++ en tu proyecto, Android Studio usa automáticamente el tipo de depuración **Dual**. De lo contrario, usa el tipo de depuración **Java**.
* **Java**: Selecciónalo si solo quieres depurar código escrito en Java o Kotlin (el depurador de Java ignora las interrupciones o los controles que estableces en tu código nativo).
* **Native**: (Disponible solo con el código C/C++). Selecciónalo si solo deseas usar LLDB para depurar tu código. Cuando usas este tipo de depuración, no está disponible la vista de la sesión del depurador de Java. De manera predeterminada, LLDB inspecciona solo tu código nativo y omite las interrupciones en tu código Java. Si también deseas depurar tu código Java, debes realizar un cambio al tipo de depuración **Auto** o **Dual**.

**Nota:** La depuración nativa no funciona en Windows de 32 bits en Android Studio 3.0 y versiones posteriores. Si estás usando Windows de 32 bits y necesitas depurar código nativo, debes usar [Android Studio 2.3](https://developer.android.com/studio/archive.html#android-studio-2-3-3).

* **Dual**: (Disponible solo con código C/C ++). Selecciónalo si deseas realizar un cambio entre la depuración de Java y código nativo. Android Studio adjunta el depurador de Java y LLDB al proceso de tu app, uno para el depurador de Java y uno para LLDB, a fin de que puedas inspeccionar las interrupciones tanto en tu código Java como en tu código nativo sin reiniciar tu app ni cambiar la configuración de depuración.

En la figura 2, observa las dos pestañas a la derecha del título de la ventana **Debug**. Como la app tiene códigos Java y C++, una pestaña es para depurar el código nativo y la otra para el código Java, como lo indica **-java**.



**Figura 2:** La pestaña para depurar código nativo y la pestaña para depurar código Java

**Nota**: Si estás depurando código nativo optimizado por el compilador, es posible que recibas el siguiente mensaje de advertencia: **This function was compiled with optimizations enabled. Some debugger features may not be available**. Cuando se usan indicadores de optimización, como los [indicadores **-O**](https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html), el compilador realiza cambios en el código compilado para que se ejecute de manera más eficiente. Esta acción puede provocar que el depurador registre información incorrecta o inesperada porque le resulta difícil volver a asignar el código compilado y optimizado al código fuente original. Por este motivo, debes inhabilitar las optimizaciones del compilador mientras depuras tu código nativo.

Cómo usar el registro del sistema

El registro del sistema muestra mensajes del sistema mientras depuras tu app. En estos mensajes, se incluye información sobre las apps que se están ejecutando en el dispositivo. Si deseas usar el registro del sistema para depurar tu app, asegúrate de que tu código escriba mensajes de registro y también imprima el seguimiento de pila para las excepciones mientras tu app se encuentre en etapa de desarrollo.

Cómo escribir mensajes de registro en tu código

Para escribir mensajes de registro en tu código, usa la clase [Log](https://developer.android.com/reference/android/util/Log.html). Estos mensajes te ayudan a comprender el flujo de ejecución mediante la recopilación de los resultados de depuración del sistema mientras interactúas con tu app. Además, pueden indicarte cuál es la parte de tu app que falló. Para obtener más información sobre el registro, consulta [Cómo escribir y ver registros](https://developer.android.com/studio/debug/am-logcat.html).

En el siguiente ejemplo, se muestra cómo puedes agregar mensajes de registro para determinar si hay información disponible sobre el estado anterior cuando inicias tu actividad:

[KOTLIN](https://developer.android.com/studio/debug#kotlin)[JAVA](https://developer.android.com/studio/debug#java)

    import android.util.Log;  
    ...  
    public class MyActivity extends Activity {  
        private static final String TAG = MyActivity.class.getSimpleName();  
        ...  
        @Override  
        public void onCreate(Bundle savedInstanceState) {  
           ...  
           if (savedInstanceState != null) {  
                Log.d(TAG, "onCreate() Restoring previous state");  
                /\* restore state \*/  
            } else {  
                Log.d(TAG, "onCreate() No saved state available");  
                /\* initialize app \*/  
            }  
        }  
    }

Durante el desarrollo, tu código también puede detectar excepciones y escribir el seguimiento de pila en el registro del sistema:

[KOTLIN](https://developer.android.com/studio/debug#kotlin)[JAVA](https://developer.android.com/studio/debug#java)

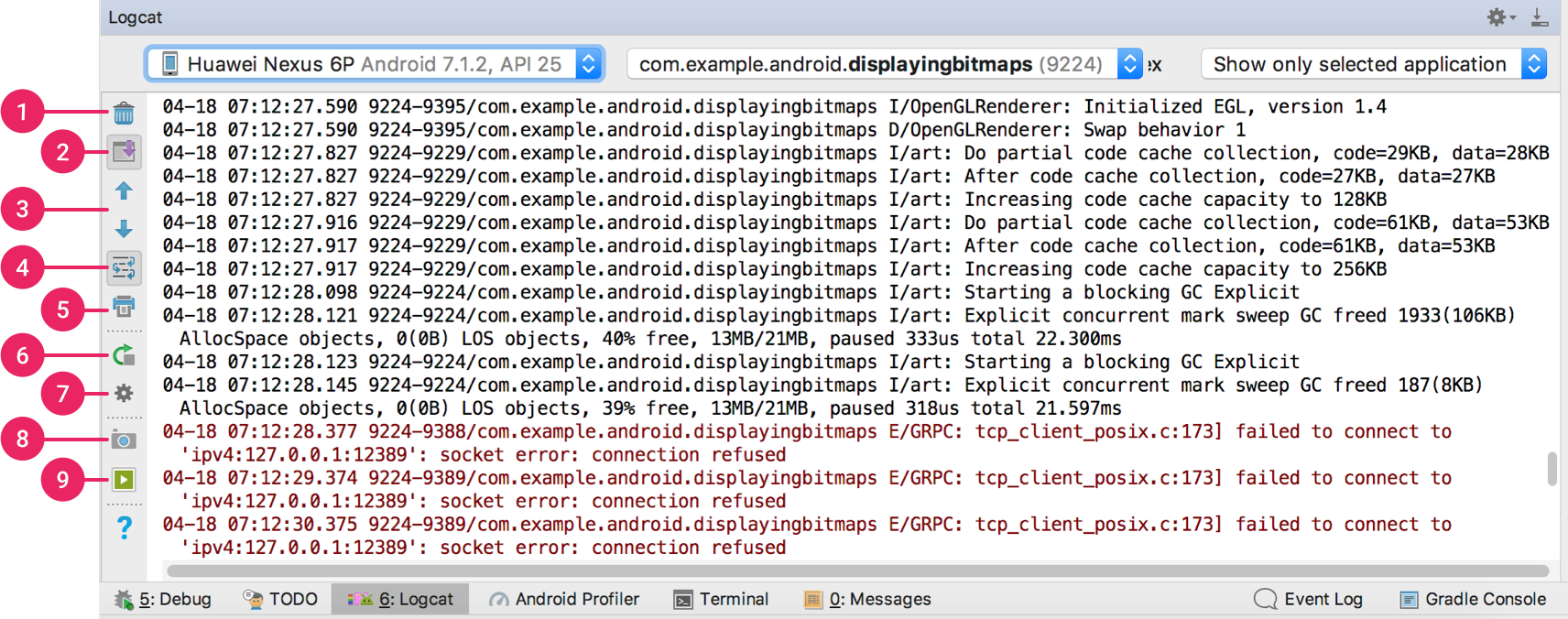
    void someOtherMethod() {  
        try {  
            ...  
        } catch (SomeException e) {  
            Log.d(TAG, "someOtherMethod()", e);  
        }  
    }

**Nota:** Cuando estés listo para publicar tu app, quita los mensajes del registro de depuración y las llamadas de impresión de seguimiento de pila de tu código. También puedes hacerlo si configuras un indicador **DEBUG** y colocas mensajes del registro de depuración en las declaraciones condicionales.

Cómo ver el registro del sistema

Puedes ver y filtrar la depuración y otros mensajes del sistema en la ventana **Logcat**. Por ejemplo, puedes ver mensajes cuando se produce la recolección de elementos no utilizados, o bien mensajes que agregas a tu app con la clase [Log](https://developer.android.com/reference/android/util/Log.html).

Para usar Logcat, debes [iniciar la depuración](https://developer.android.com/studio/debug#startdebug) y seleccionar la pestaña **Logcat** en la barra de herramientas inferior, como se muestra en la figura 3.



**Figura 3:** La ventana "Logcat" con configuración de filtros

Para obtener una descripción de Logcat y sus opciones de filtrado, consulta [Cómo escribir y ver registros con Logcat](https://developer.android.com/studio/debug/am-logcat.html).

Cómo trabajar con interrupciones

Android Studio admite varios tipos de interrupciones que activan diferentes acciones de depuración. El tipo más común es un punto de interrupción de línea que detiene la ejecución de tu app en una línea de código especificada. Mientras la app está detenida, puedes examinar variables, evaluar expresiones y, luego, continuar la ejecución línea por línea para determinar las causas de los errores en el tiempo de ejecución.

Para agregar un punto de interrupción de línea, haz lo siguiente:

1. Localiza la línea de código en la que desees detener la ejecución, luego haz clic en el margen izquierdo de esta o coloca el símbolo de intercalación en la línea y presiona Control + F8 (en Mac, Command + F8).
2. Si tu app ya está en ejecución, no necesitas actualizarla para agregar el punto de interrupción. Simplemente, haz clic en **Attach debugger to Android proccess** . De lo contrario, inicia la depuración haciendo clic en **Debug** .



**Figura 3:** Cuando se establece un punto de interrupción, aparece un punto rojo junto a la línea.

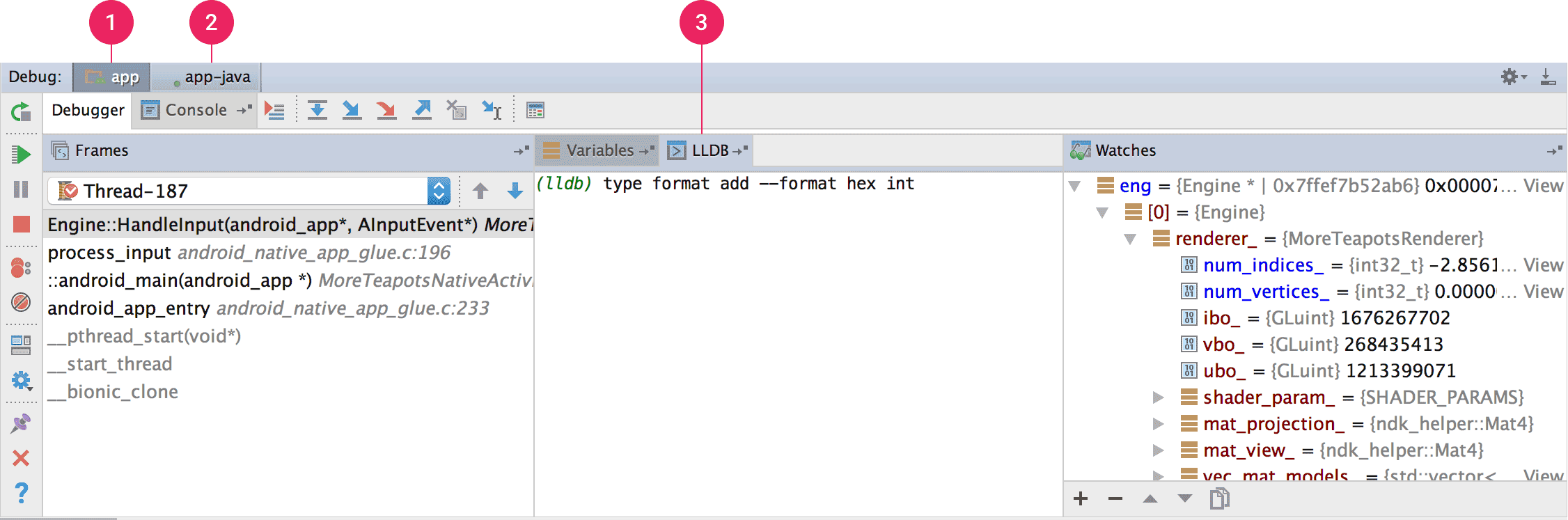
Cuando la ejecución del código alcanza la interrupción, Android Studio detiene la ejecución de tu app. Luego, puedes usar las herramientas de la pestaña **Debugger** para identificar el estado de la app:

* Para examinar el árbol de objetos de una variable, expándelo en la vista **Variables**. Si la vista **Variables** no está visible, haz clic en **Restore Variables View** .
* Para evaluar una expresión en el punto de ejecución actual, haz clic en **Evaluate Expression** .
* Para avanzar a la siguiente línea del código (sin ingresar a un método), haz clic en **Step Over** .
* Para avanzar a la primera línea dentro de una llamada a un método, haz clic en **Step Into** .
* Para avanzar a la siguiente línea fuera del método actual, haz clic en **Step Out** .
* Para continuar ejecutando la app normalmente, haz clic en **Resume Program** .

Si tu proyecto usa código nativo, de forma predeterminada, el tipo de depuración **Auto** adjunta el depurador Java y LLDB a tu app como dos procesos independientes, de manera que puedas realizar el cambio entre la inspección de interrupciones de Java y C/C++ sin tener que reiniciar tu app ni cambiar la configuración.

**Nota**: Para que Android Studio detecte las interrupciones en su código C o C++, debes usar un tipo de depuración que sea compatible con LLDB, como Auto, Native o Dual. Para cambiar el tipo de depuración que usa Android Studio, [edita tu configuración de depuración](https://developer.android.com/studio/run/rundebugconfig.html). Para obtener más información sobre los diferentes tipos de depuración, lee la sección sobre cómo usar otros [tipos de depuración](https://developer.android.com/studio/debug#debug-types).

Cuando Android Studio implementa tu app en tu dispositivo de destino, se abre la ventana "Debug" con una pestaña o la vista de la sesión de depuración para cada proceso del depurador, como se muestra en la figura 4.



**Figura 4:** Depuración de código nativo mediante LLDB

1. Android Studio cambia a la pestaña ***<your-module>*** cuando el depurador LLDB encuentra una interrupción en tu código C o C++. También se encuentran disponibles los paneles **Frames**, **Variables** y **Watches**, que funcionan exactamente como lo harían si depuraras código Java. Si bien el panel **Threads** no está disponible en la vista de la sesión de LLDB, puedes acceder a los procesos de tu app por medio de la lista desplegable del panel **Frames**. Para obtener más información sobre estos paneles, consulta las secciones sobre cómo [inspeccionar variables](https://developer.android.com/studio/debug#variablesAndWatches) y [depurar marcos de ventanas](https://developer.android.com/studio/debug#stackFrames).

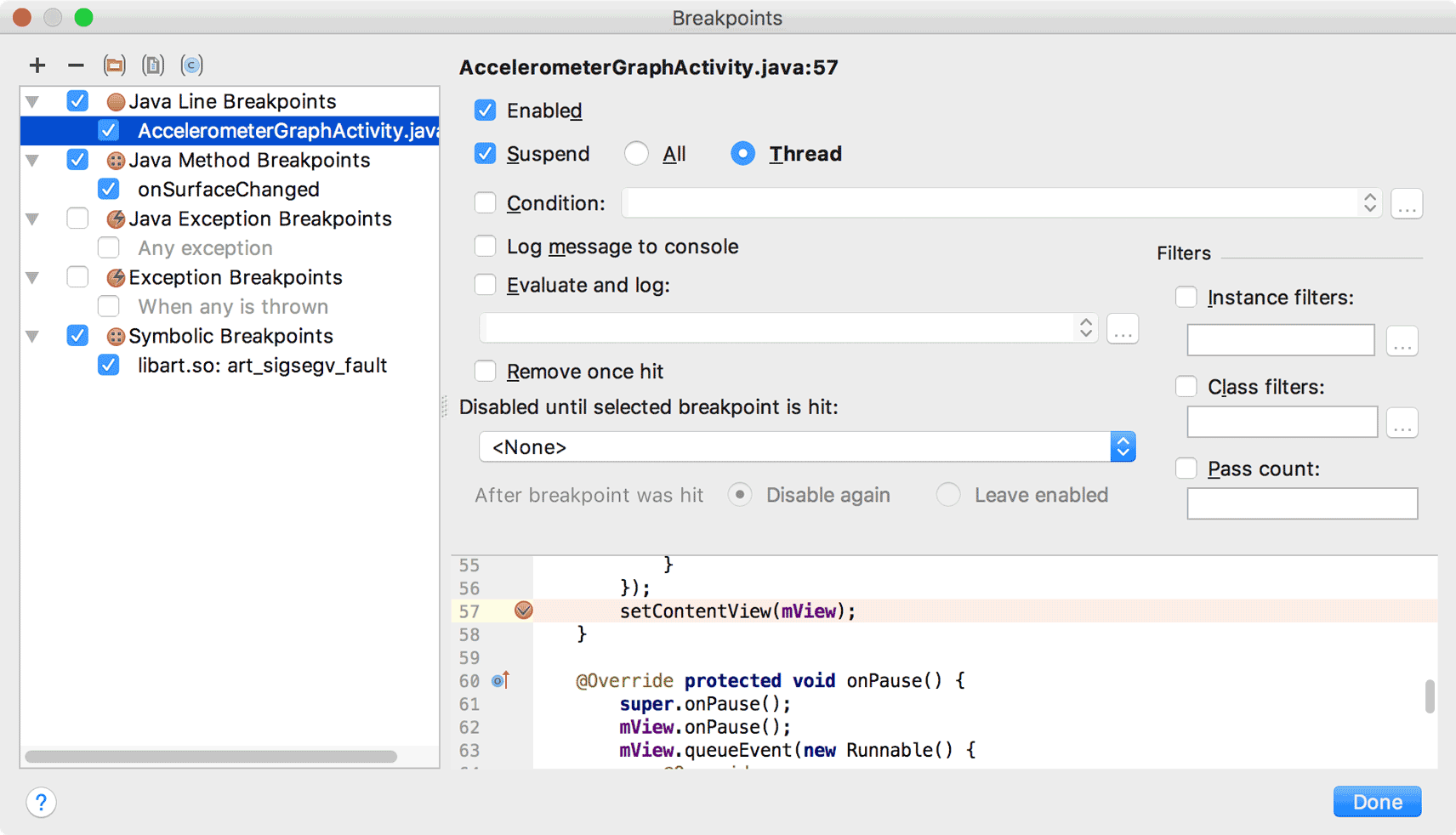
**Nota:** Mientras inspeccionas un punto de interrupción en tu código nativo, el sistema de Android suspende la máquina virtual que ejecuta el código de byte Java de tu app, lo que implica que no podrás interactuar con el depurador de Java ni recuperar información sobre el estado de tu sesión de depuración de Java mientras inspeccionas un punto de interrupción en tu código nativo.

1. Android Studio cambia a la pestaña ***<your-module>-java*** cuando el depurador de Java encuentra una interrupción en tu código Java.
2. Mientras realizas la depuración con LLDB, puedes usar el terminal **LLDB** en la vista de sesión LLDB para pasar las [opciones de la línea de comandos a LLDB](http://lldb.llvm.org/tutorial.html). Si tienes comandos específicos que quisieras que LLDB ejecute cada vez que comience a depurar tu app (ya sea justo antes o después de que el depurador se adjunte al proceso de tu app), puedes [agregar esos comandos a tu configuración de depuración](https://developer.android.com/studio/run/rundebugconfig.html#debugger-tab).

Durante la depuración de código C o C++, también puedes configurar tipos especiales de interrupciones, llamados *puntos de análisis*, que pueden suspender el proceso de tu app cuando esta interactúa con un bloque de memoria específico. Para obtener más información, consulta la sección sobre cómo [agregar puntos de análisis](https://developer.android.com/studio/debug#watchpoints).

Cómo ver y configurar interrupciones

Para ver todas las interrupciones y configurarlas, haz clic en **View Breakpoints**  a la izquierda de la ventana **Debug**. Aparecerá la ventana **Breakpoints**, como se muestra en la figura 5.



**Figura 5:** En la ventana "Breakpoints", se enumeran todas las interrupciones actuales y se incluye la configuración de comportamiento para cada una

La ventana **Breakpoints** te permite habilitar o inhabilitar cada punto de interrupción de la lista de la izquierda. Si se inhabilita un punto de interrupción, Android Studio no detendrá tu app cuando lo alcance. Selecciona un punto de interrupción de la lista para configurarlo. Puedes inhabilitar uno y hacer que el sistema lo habilite luego de que se alcance otro punto de interrupción. También puedes configurar un punto de interrupción para que se inhabilite una vez que se alcance. Para configurar un punto de interrupción para cualquier excepción, selecciona **Exception Breakpoints** en la lista de interrupciones.

Cómo depurar marcos de una ventana

En la ventana **Debugger**, el panel **Frames** te permite inspeccionar el marco de la pila que provocó que se alcance el punto de interrupción actual. De esta manera, podrás navegar y examinar el marco de la pila y, además, inspeccionar la lista de subprocesos en tu app de Android. Para seleccionar un subproceso, usa el selector de subprocesos desplegable y visualiza su marco de pila. Al hacer clic en los elementos del marco, se abrirá el origen en el editor. También puedes personalizar la presentación de subprocesos y exportar el marco de la pila como se explica en la [Guía de marcos de ventanas](https://www.jetbrains.com/help/idea/2019.1/debug-tool-window-frames.html).

Cómo inspeccionar variables

En la ventana **Debugger**, el panel **Variables** te permite inspeccionar variables cuando el sistema detiene tu app en un punto de interrupción y seleccionas un marco del panel **Frames**. El panel **Variables** también te permite evaluar expresiones ad hoc por medio de métodos estáticos o variables disponibles en el marco seleccionado.

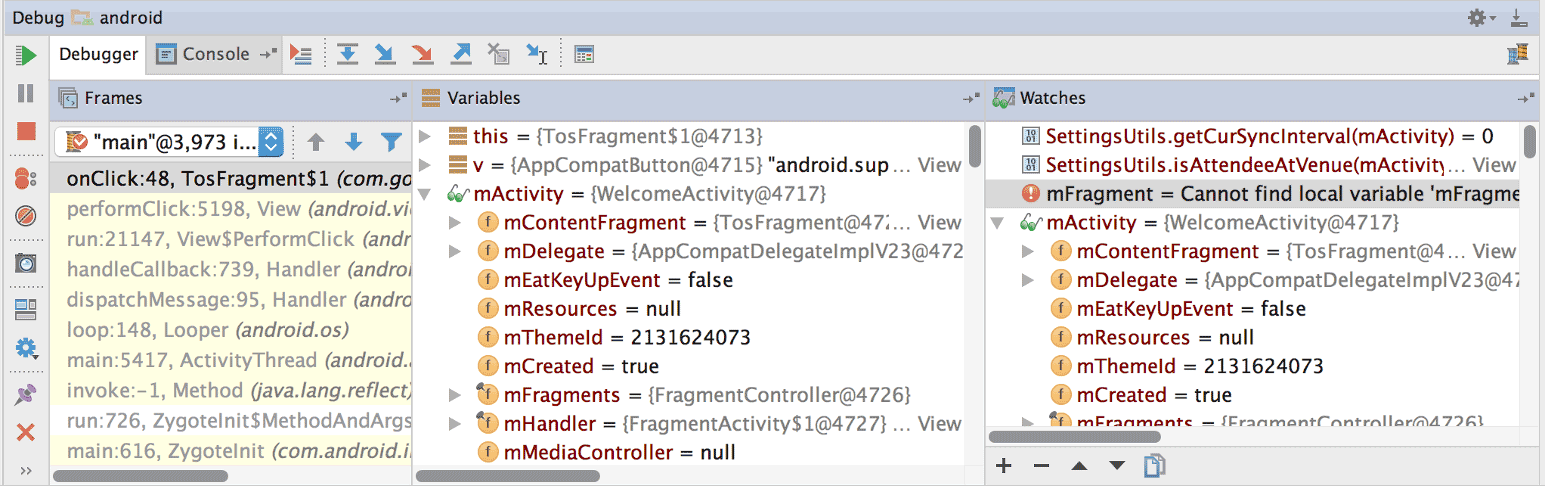
En el panel **Watches**, se proporciona una funcionalidad similar, con la excepción de que las expresiones agregadas al panel **Watches** se conservan entre sesiones de depuración. Debes agregar controles para variables y campos a los que accedas con frecuencia o que proporcionen un estado útil para la sesión de depuración actual. Los paneles **Variables** y **Watches** aparecen como se muestra en la figura 6.

Para agregar una variable o expresión a la lista de **Watches**, sigue estos pasos:

1. Inicia la depuración.
2. En el panel **Watches**, haz clic en **Add** .
3. En el cuadro de texto que aparece, escribe el nombre de la variable o la expresión que desees controlar y presiona "Intro".

Para quitar un elemento de la lista **Watches**, selecciónalo y, luego, haz clic en **Remove** .

Para volver a ordenar los elementos en la lista **Watches**, selecciona un elemento y haz clic en **Up**  o **Down** .



**Figura 6:** Los paneles "Variables" y "Watches" en la ventana "Debugger"

Cómo agregar puntos de control

Durante la depuración de código C y C++, puedes configurar tipos especiales de interrupciones, llamados *puntos de análisis*, que pueden suspender el proceso de tu app cuando esta interactúa con un bloque de memoria específico. Por ejemplo, si estableces dos punteros para un bloque de memoria y le asignas un punto de análisis, y usas cualquiera de los punteros para acceder al bloque de memoria que activa el punto de análisis.

En Android Studio, puedes crear un punto de análisis durante el tiempo de ejecución al seleccionar una variable específica, pero LLDB asigna el punto de análisis solo al bloque de memoria que el sistema asigna a esa variable, no a la variable en sí. Es distinto a agregar una variable al panel **Watches**, que te permite observar el valor de una variable, pero no suspender el proceso de tu app cuando el sistema lee o cambia su valor en la memoria.

**Nota:** Cuando el proceso de tu app sale de una función y el sistema desasigna sus variables locales de la memoria, tendrás que volver a asignar los puntos de análisis que creaste para esas variables.

Para establecer un punto de análisis, debes cumplir con los siguientes requisitos:

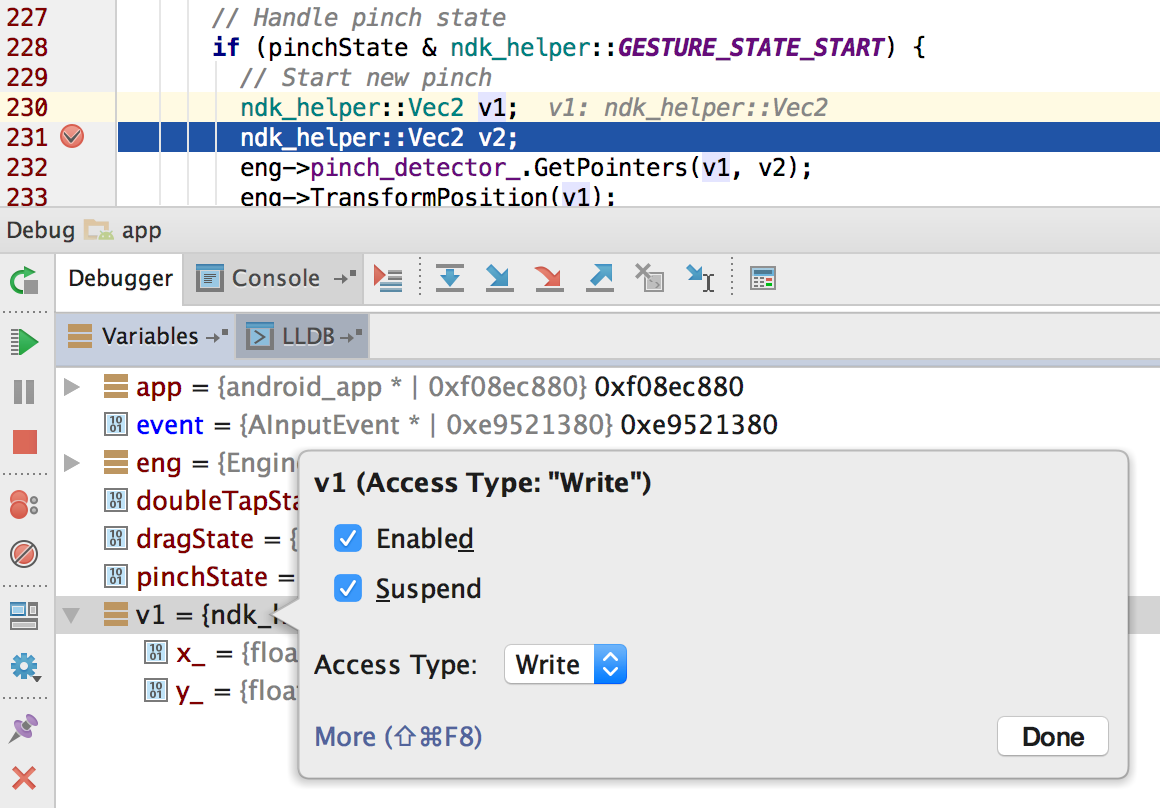
* Tu emulador o dispositivo físico usa una CPU x86 o x86\_64. Si usa una CPU ARM, entonces debes alinear el límite de la dirección de tu variable en la memoria a 4 bytes para procesadores de 32 bits o a 8 bytes para procesadores de 64 bits. Para alinear una variable en tu código nativo, especifica \_\_attribute\_\_((aligned(***num\_bytes***))) en la desaceleración de la variable, como se muestra a continuación:

    // For a 64-bit ARM processor  
    int my\_counter \_\_attribute\_\_((aligned(8)));

* Ya asignaste tres punto de análisis o menos. Android Studio solo admite hasta cuatro puntos de análisis en dispositivos de destino x86 o x86\_64. Es posible que otros dispositivos admitan menos puntos de análisis.

Si cumples los requisitos anteriores, puedes agregar un punto de control de la siguiente manera:

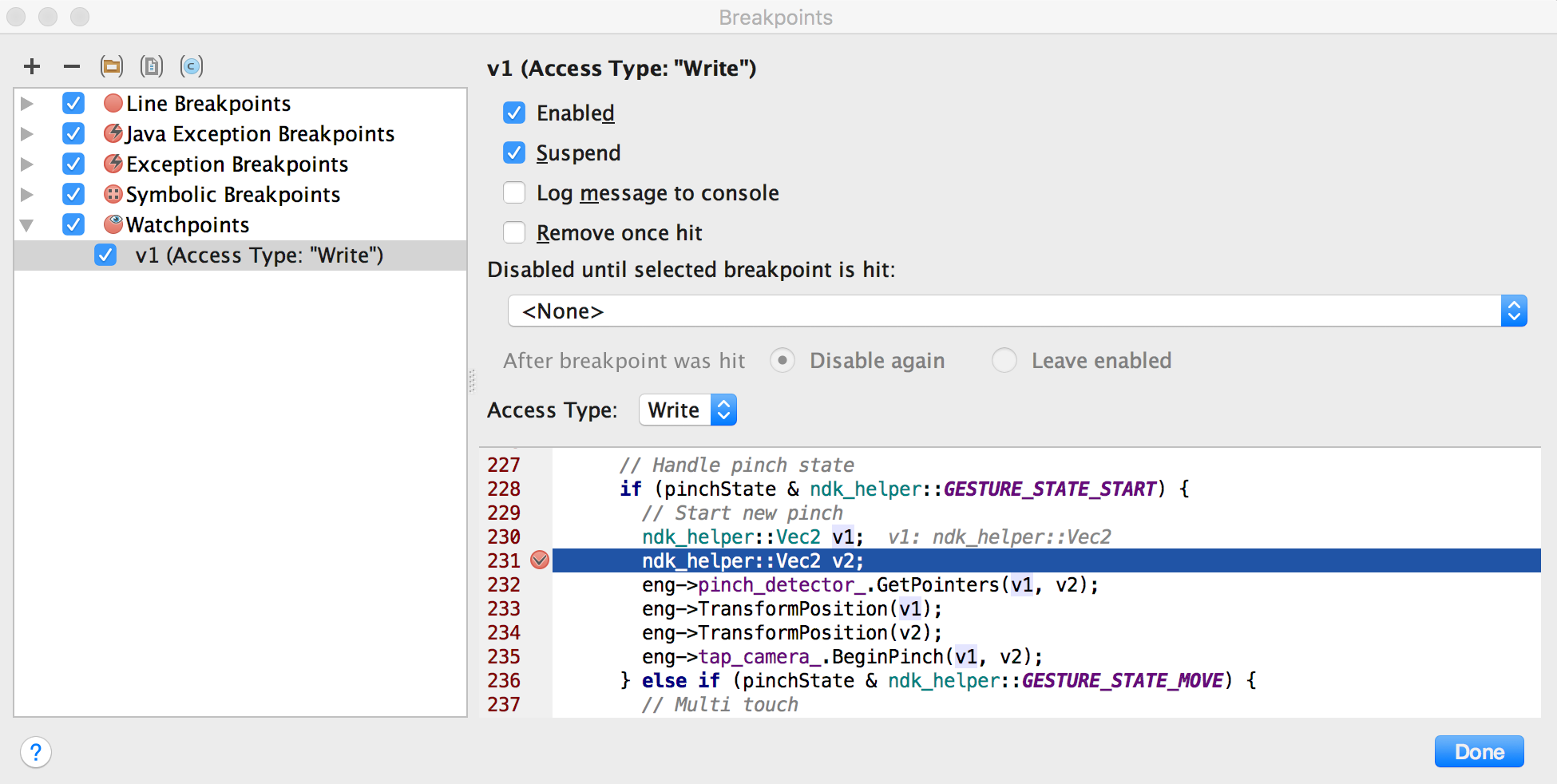
1. Mientras tu app esté suspendida en un punto de interrupción, navega al panel **Variables** en la vista de la sesión de LLDB.
2. Haz clic con el botón derecho en una variable que ocupe el bloque de memoria del cual desees realizar un seguimiento y selecciona **Add Watchpoint**. Aparecerá un diálogo para configurar tu punto de análisis, como se muestra en la figura 7.



**Figura 7:** Inclusión de un punto de análisis en una variable en la memoria

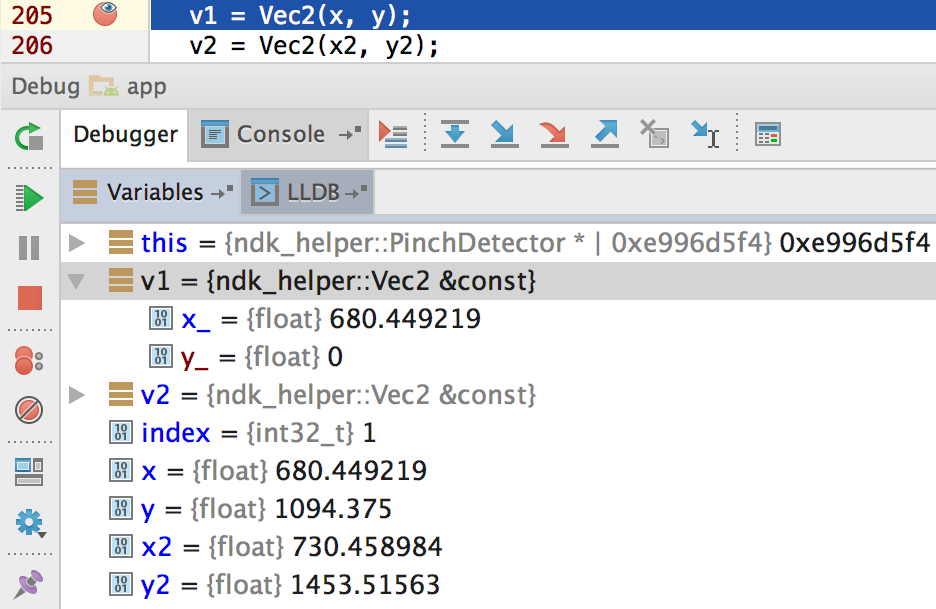
1. Configura tu punto de control con las siguientes opciones:
   * **Enabled:** Puedes anular la selección de esta opción si deseas indicarle a Android Studio que ignore el punto de análisis por el momento. Android Studio guarda tu punto de análisis para que puedas acceder a él más adelante en tu sesión de depuración.
   * **Suspend:** De forma predeterminada, el sistema Android suspenderá el proceso de tu app cuando esta acceda a un bloque de memoria que asignaste a un punto de análisis. Puedes anular la selección de esta opción si no deseas este comportamiento (esta acción revela opciones adicionales que puedes usar para personalizar el comportamiento cuando el sistema interactúa con su punto de análisis): **Log message to console** y **Remove [the watchpoint] when hit**.
   * **Access type:** Permite determinar si tu app debe activar el punto de análisis cuando intenta realizar operaciones de tipo **Read** (lectura) o **Write** (escritura) en el bloque de memoria que el sistema asigna a la variable. Para activar tu punto de análisis en lectura o escritura, selecciona **Any**.
2. Haz clic en **Done**.

Para ver todos los puntos de análisis o configurarlos, haz clic en **View Breakpoints**  a la izquierda de la ventana **Debug**. Aparecerá el diálogo **Breakpoints**, como se muestra en la figura 8.



**Figura 8:** En el diálogo "Breakpoints", se enumeran tus puntos de análisis actuales y se incluye la configuración de comportamiento para cada uno.

Una vez que agregues tu punto de análisis, haz clic en **Resume Program**  a la izquierda de la ventana **Debug** para reanudar el proceso de tu app. De forma predeterminada, si tu app intenta acceder a un bloque de memoria en el que configuraste un punto de análisis, el sistema Android suspende el proceso de tu app y aparece un ícono de punto de análisis () junto a la última línea de código que ejecutó tu app, como se muestra en la figura 9.



**Figura 9:** Android Studio indica la línea de código que tu app ejecuta justo antes de activar un punto de análisis.

Cómo ver y cambiar el formato de visualización de valores de los recursos

En el modo de depuración, puedes ver los valores de los recursos y seleccionar un formato de visualización diferente para las variables en tu código Java. Con la pestaña **Variables** visible en un marco seleccionado, haz lo siguiente:

1. En la lista **Variables**, haz clic con el botón derecho en cualquier parte de una línea de recursos para mostrar la lista desplegable.
2. En la lista desplegable, selecciona **View as** y elige el formato que quieras usar.

Los formatos disponibles dependen del tipo de datos del recurso que selecciones. Es posible que veas una o más de las siguientes opciones:

* + **Class:** Se muestra la definición de la clase.
  + **toString:** Se muestra el formato de la string.
  + **Object:** Se muestra la definición del objeto (una instancia de una clase).
  + **Array:** Se muestra un formato de arreglo.
  + **Timestamp:** Se muestra la fecha y la hora en el siguiente formato: aaaa-mm-dd hh:mm:ss.
  + **Auto:** Android Studio elige el mejor formato en función del tipo de datos.
  + **Binary:** Se muestra un valor binario con ceros y unos.
  + **MeasureSpec:** Se muestra el valor que se pasa del elemento principal al elemento secundario seleccionado. Consulta [MeasureSpec.](https://developer.android.com/reference/android/view/View.MeasureSpec.html)
  + **Hex:** Se muestra un valor hexadecimal.
  + **Primitive:** Se muestra un valor numérico con un tipo de datos primitivo.
  + **Integer**: Se muestra un valor numérico del tipo [Integer](https://developer.android.com/reference/java/lang/Integer.html).

Puedes crear un formato personalizado (representador del tipo de datos) de la siguiente manera:

1. Haz clic con el botón derecho en el valor del recurso.
2. Selecciona **View as**.
3. Selecciona **Create**. Aparecerá el diálogo **Java Data Type Renderers**.
4. Sigue las instrucciones de [Procesadores de tipos de datos de Java](https://www.jetbrains.com/help/idea/2019.1/java-type-renderers.html).