Debug Your App

In this document

1. [Use the System Log](https://developer.android.com/studio/debug/index.html#systemLog)
   1. [Write log messages in your code](https://developer.android.com/studio/debug/index.html#systemLogWrite)
   2. [View the system log](https://developer.android.com/studio/debug/index.html#systemLogView)
2. [Work with Breakpoints](https://developer.android.com/studio/debug/index.html#breakPoints)
   1. [View and configure breakpoints](https://developer.android.com/studio/debug/index.html#breakPointsView)
3. [Track Object Allocation](https://developer.android.com/studio/debug/index.html#allocTracker)
4. [Analyze Runtime Metrics to Optimize your App](https://developer.android.com/studio/debug/index.html#deviceMonitor)
5. [Capture Screenshots and Videos](https://developer.android.com/studio/debug/index.html#screenCap)

See also

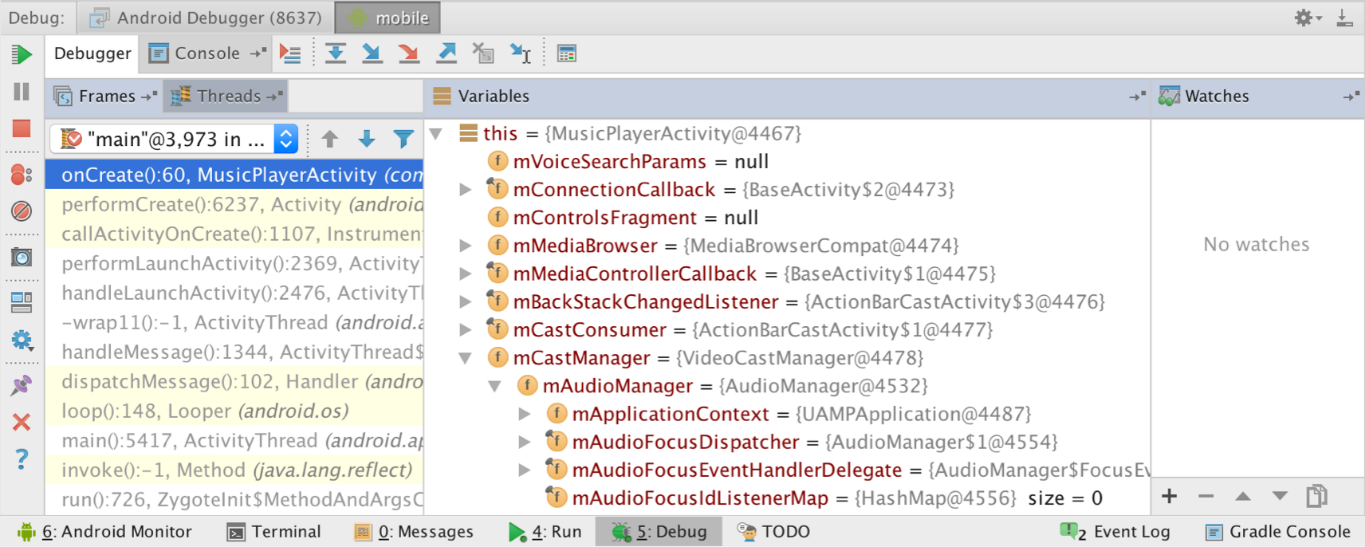
* [Android Monitor](https://developer.android.com/tools/help/android-monitor.html)
* [Device Monitor](https://developer.android.com/tools/help/monitor.html)
* [Using DDMS](https://developer.android.com/tools/debugging/ddms.html)
* [Create and Edit Run/Debug Configurations](https://developer.android.com/studio/run/rundebugconfig.html)

Android Studio includes a debugger that allows you to debug apps running on the Android Emulator or a connected Android device. With the Android Studio debugger, you can:

* Select a device to debug your app on.
* Set breakpoints in your code.
* Examine variables and evaluate expressions at runtime.
* Capture screenshots and videos of your app.

To start debugging, click **Debug** https://developer.android.com/images/tools/as-debugbutton.png in the toolbar. Android Studio builds an APK, signs it with a debug key, installs it on your selected device, then runs it and opens the **Debug** window.

If no devices appear in the **Select Deployment Target** window after you click **Debug**, then you need to either [connect a device](https://developer.android.com/tools/device.html) or click **Create New Emulator** to [setup the Android Emulator](https://developer.android.com/tools/devices/emulator.html).



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

If your app is already running on a connected device or emulator, you can start debugging as follows:

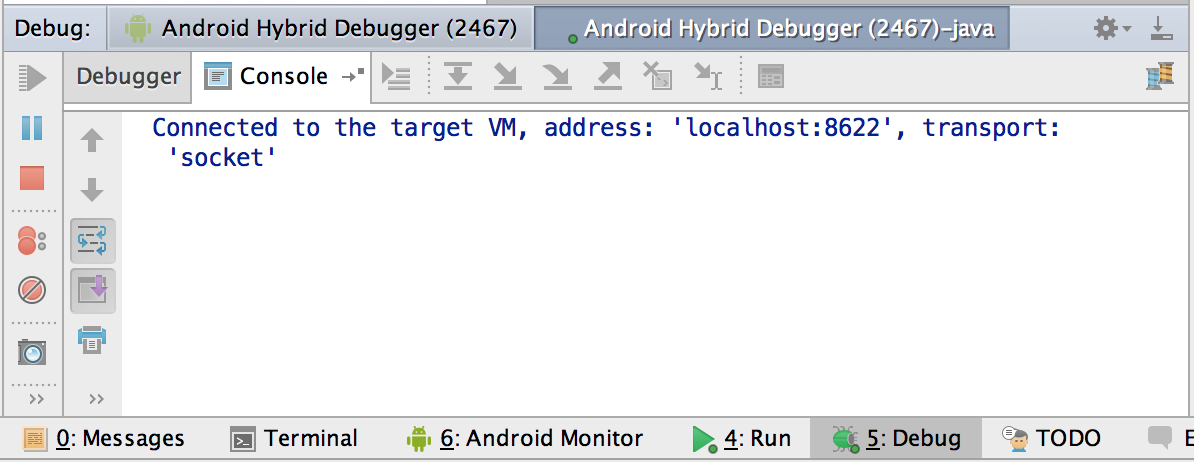
1. Click **Attach debugger to Android process** https://developer.android.com/images/tools/as-attach.png.
2. In the *Choose Process* dialog, select the process you want to attach the debugger to.

By default, the debugger shows the device and app process for the current project, as well as any connected hardware devices or virtual devices on your computer. Select **Show all processes** to show all processes on all devices; the display includes any services that your app created as well as system processes, for example.

From the **Debugger** menu, you can select **Java**, **Native**, or **Hybrid**. The latter two options are available only if your project contains some native C or C++ source code.

1. Click **OK**.

The *Debug* window appears. In this case, notice the two tabs to the right of the Debug window title: one tab is for debugging native code and the other for Java code, as indicated by **-java**.



Separate debugging sessions have separate tabs and different port numbers, which are displayed in parentheses in the tab.

1. To end a debugging session, click the tab for the session, and then click **Terminate** https://developer.android.com/images/tools/am-iterminate.png.

**Note:** The Android Studio debugger and garbage collector are loosely integrated. The Andriod virtual machine guarantees that any object the debugger is aware of is not garbage collected until after the debugger disconnects. This can result in a buildup of objects over time while the debugger is connected. For example, if the debugger sees a running thread, the associated [Thread](https://developer.android.com/reference/java/lang/Thread.html) object is not garbage collected even after the thread terminates.

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.

Write log messages in your code

To write log messages in your code, use the [Log](https://developer.android.com/reference/android/util/Log.html) 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 [Reading and Writing Logs](https://developer.android.com/tools/debugging/debugging-log.html).

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;  
...  
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 \*/  
        }  
    }  
}

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

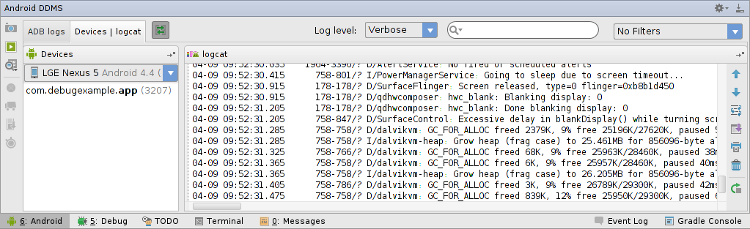
void someOtherMethod() {  
    try {  
        ...  
    } catch (SomeException e) {  
        Log.d(TAG, "someOtherMethod()", e);  
    }  
}

**Note:** Remove debug log messages and stack trace print calls from your code when you are ready to publish your app. You could do this by setting a DEBUG flag and placing debug log messages inside conditional statements.

View the system log

Both the Android DDMS (Dalvik Debug Monitor Server) and the **Android Monitor** windows show logs from the system and any particular app process. To view the system log on the **Android DDMS** tool window:

1. Start your app as described in [Run your App in Debug Mode](https://developer.android.com/studio/debug/index.html#runDebug).
2. Click **Android Monitor** https://developer.android.com/images/tools/as-android.png.
3. If the system log is empty in the *Logcat view*, click **Restart** https://developer.android.com/images/tools/as-restart.png.



**Figure 2.** The system log in the Android DDMS tool window.

The **Android DDMS** tool window gives you access to some DDMS features from Android Studio. For more information about DDMS, see [Using DDMS](https://developer.android.com/tools/debugging/ddms.html).

The system log shows messages from Android services and other Android apps. To filter the log messages to view only the ones you are interested in, use the tools in the **Android DDMS** window:

* To show only log messages for a particular process, select the process in the **Devices** view and then click **Only Show Logcat from Selected Process** https://developer.android.com/images/tools/as-currentproc.png. If the **Devices** view is not available, click **Restore Devices View** https://developer.android.com/images/tools/as-showdevview.png on the right of the **Android DDMS** tool window. This button is only visible when you hide the **Devices** window.
* To filter log messages by log level, select a level from the Log Level drop-down on the top of the **Android DDMS** window.
* To show only log messages that contain a particular string, enter the string in the search box and press **Enter**.

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** https://developer.android.com/images/tools/as-attach.png. Otherwise, start debugging by clicking **Debug** https://developer.android.com/images/tools/as-debugbutton.png.

https://developer.android.com/images/tools/as-breakpointline.png

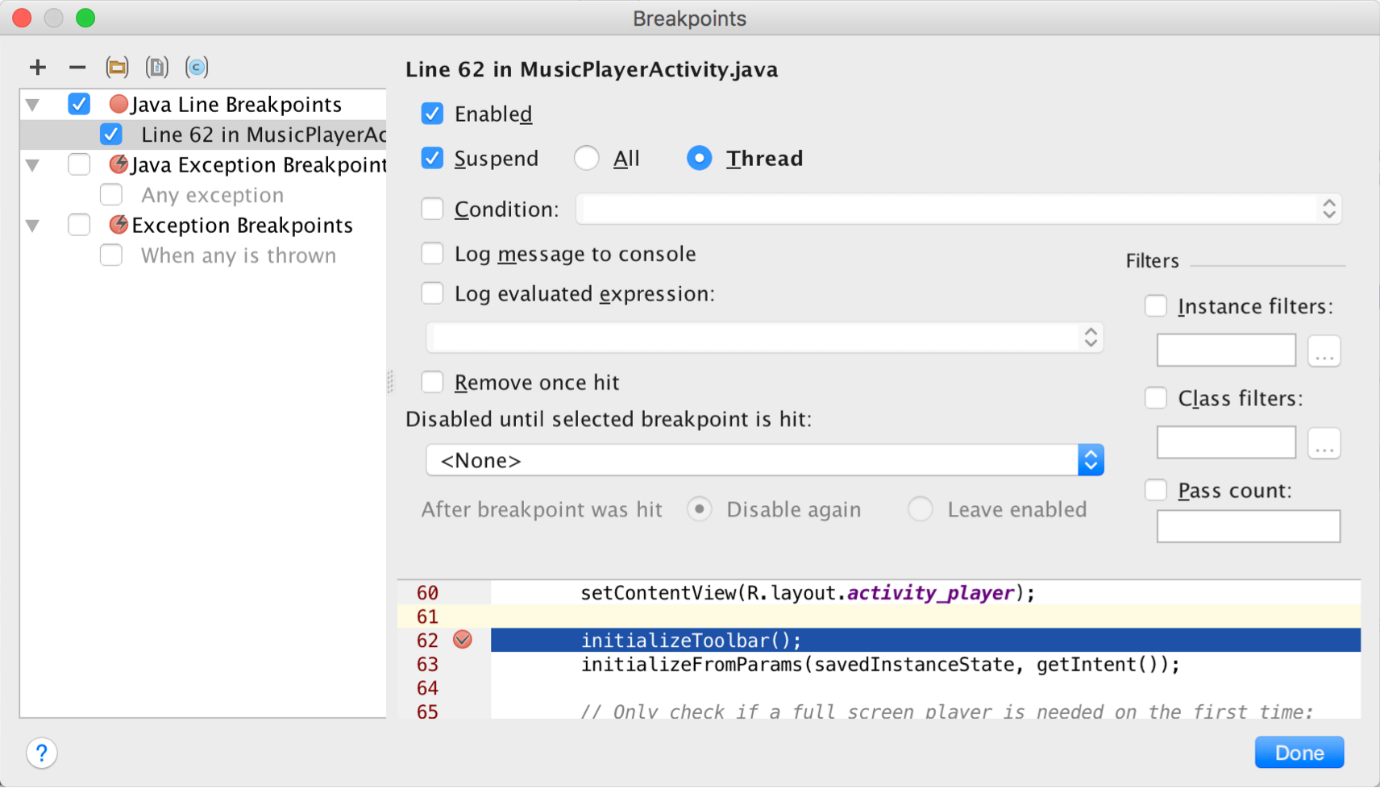
**Figure 3.** A red dot appears next to the line when you set a breakpoint.

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 cause of the error:

* 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** https://developer.android.com/images/tools/as-varviewbutton.png.
* To evaluate an expression at the current execution point, click **Evaluate Expression** https://developer.android.com/studio/images/debug/debugger_button_evaluate_2-0.png.
* To advance to the next line in the code (without entering a method), click **Step Over** https://developer.android.com/studio/images/debug/debugger_button_stepover_2-0.png.
* To advance to the first line inside a method call, click **Step Into** https://developer.android.com/studio/images/debug/debugger_button_stepinto_2-0.png.
* To advance to the next line outside the current method, click **Step Out** https://developer.android.com/studio/images/debug/debugger_button_stepout_2-0.png.
* To continue running the app normally, click **Resume Program** https://developer.android.com/studio/images/debug/debugger_button_resume_2-0.png.

View and configure breakpoints

To view all the breakpoints and configure breakpoint settings, click **View Breakpoints** https://developer.android.com/images/tools/as-viewbreakbutton.png on the left side of the **Debug** window. The **Breakpoints**window appears, as shown in figure 4.



**Figure 4.** 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.

View variables and evaluate expressions

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

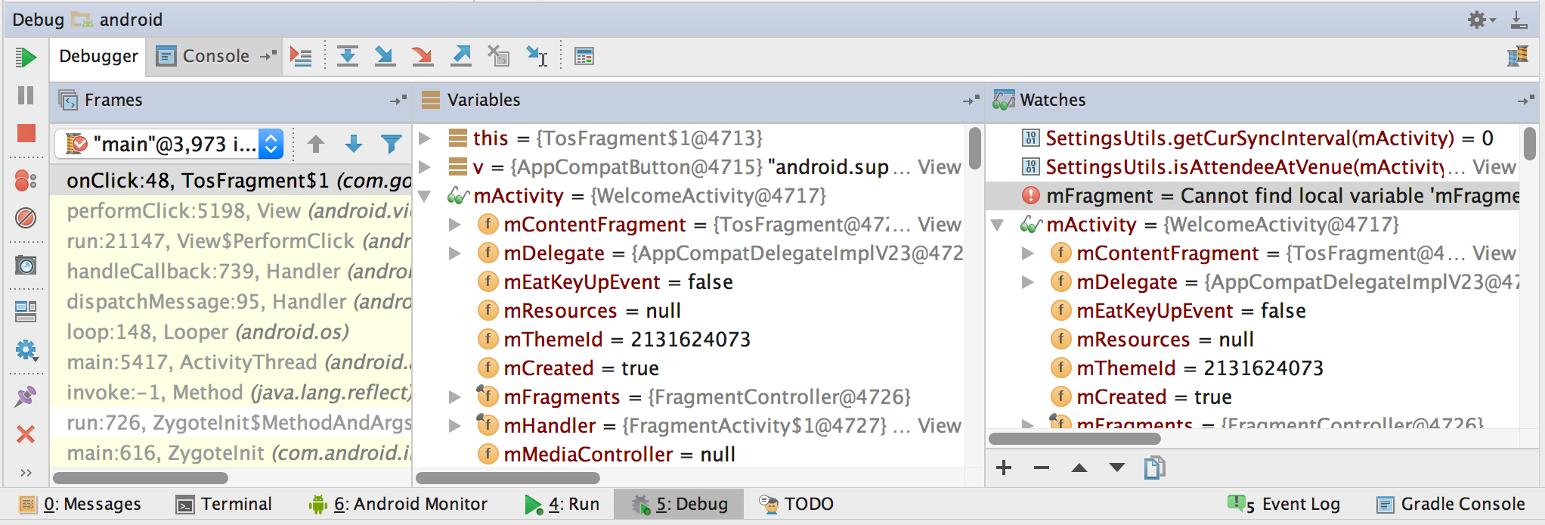
The **Watches** pane provides similar functionality except that expressions added to the **Watches** pane persist between debugging sessions. You should add watches for variables and fields that you access frequently or that provide state that is helpful for the current debugging session. The **Variables**and **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** https://developer.android.com/studio/images/buttons/ic_plus.png.
3. In the text box that appears, type the name of the variable or expression you want to watch and then press Enter.

To remove an item from the **Watches** list, select the item and then click **Remove** https://developer.android.com/studio/images/buttons/ic_minus.png.

You can reorder the elements in the **Watches** list by selecting an item and then clicking **Up** https://developer.android.com/studio/images/buttons/ic_arrow_up.png or **Down** https://developer.android.com/studio/images/buttons/ic_arrow_down.png.



**Figure 5.** The Variables and Watches panes in the Debugger window.

Track Object Allocation

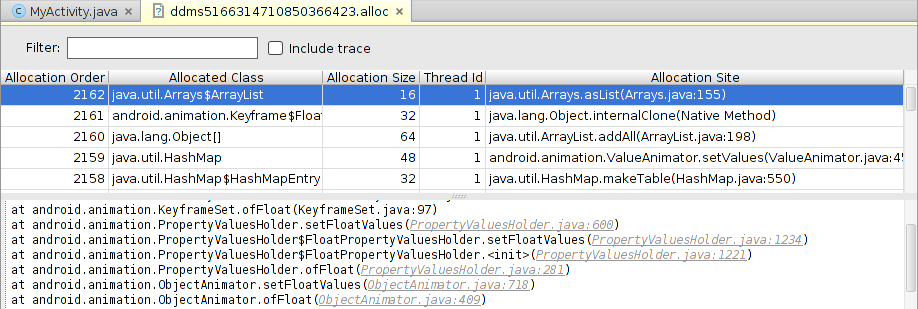
Android Studio lets you track objects that are being allocated on the Java heap and see which classes and threads are allocating these objects. This allows you to see the list of objects allocated during a period of interest. This information is valuable for assessing memory usage that can affect application performance.

To track memory allocation of objects:

1. Start your app as described in [Run Your App in Debug Mode](https://developer.android.com/studio/debug/index.html#runDebug).
2. Click **Android** https://developer.android.com/images/tools/as-android.png to open the **Android DDMS** tool window.
3. On the **Android DDMS** tool window, select the **Devices | logcat tab**.
4. Select your device from the dropdown list.
5. Select your app by its package name from the list of running apps.
6. Click **Start Allocation Tracking** https://developer.android.com/images/tools/as-allocstart.png
7. Interact with your app on the device.
8. Click **Stop Allocation Tracking** https://developer.android.com/images/tools/as-allocstop.png

Android Studio shows the objects that the system allocated with the following information:

* Allocation order
* Allocated class
* Allocation size
* Thread ID
* Allocation method, class, and line number
* Stack trace at the point of allocation



**Figure 6.** Object allocation tracking in Android Studio.

Analyze Runtime Metrics to Optimize your App

Even if your application does not generate runtime errors, this does not mean it is free of problems. You should also consider the following issues:

* Does your app use memory efficiently?
* Does your app generate unnecessary network traffic?
* What methods should you focus your attention on to improve the performance of your app?
* Does your app behave properly when the user receives a phone call or a message?

The Android Device Monitor is a stand-alone tool with a graphical user interface for serveral Android application debugging and analysis tools, including the Dalvik Debug Monitor Server (DDMS). You can use the Android Device Monitor to analyze memory usage, profile methods, monitor network traffic and simulate incoming calls and messages.

To open the Android Device Monitor from Android Studio, click **Monitor** https://developer.android.com/images/tools/as-monitorbutton.png on the toolbar. The Android Device Monitor opens in a new window.

For more information about the Android Device Monitor and DDMS, see [Device Monitor](https://developer.android.com/tools/help/monitor.html) and [Using DDMS](https://developer.android.com/tools/debugging/ddms.html).

Capture Screenshots and Videos

Android Studio enables you to capture a screenshot or a short video of the device screen while your app is running. Screenshots and videos are useful as promotional materials for your app, and you can also attach them to bug reports that you send to your development team.

To take a screenshot of your app:

1. Start your app as described in [Run your App in Debug Mode](https://developer.android.com/studio/debug/index.html#runDebug).
2. Click **Android Monitor** https://developer.android.com/images/tools/as-android.png.
3. Click **Screen Capture** https://developer.android.com/images/tools/as-capture.png on the left.
4. Optional: To add a device frame around your screenshot, click **Frame screenshot**.
5. Click **Save**.

To take a video recording of your app:

1. Start your app as described in [Run your App in Debug Mode](https://developer.android.com/studio/debug/index.html#runDebug).
2. Click **Android Monitor** https://developer.android.com/images/tools/as-android.png.
3. Click **Screen Record** https://developer.android.com/images/tools/as-record.png on the left.
4. Click **Start Recording**.
5. Interact with your app.
6. Click **Stop Recording**.
7. Enter a file name for the recording and click **OK**.