(https://google.com/racialequity)

Android Debug Bridge (adb)

Android Debug Bridge (adb) is a versatile command-line tool that lets you communicate with a device. The adb command facilitates a variety of device actions, such as installing and debugging apps, and it provides access to a Unix shell that you can use to run a variety of commands on a device. It is a client-server program that includes three components:

- **A client**, which sends commands. The client runs on your development machine. You can invoke a client from a command-line terminal by issuing an adb command.
- **A daemon (adbd)**, which runs commands on a device. The daemon runs as a background process on each device.
- **A server**, which manages communication between the client and the daemon. The server runs as a background process on your development machine.

`adb` is included in the Android SDK Platform-Tools package. You can download this package with the SDK Manager (/studio/intro/update#sdk-manager), which installs it at ***`android_sdk`*`/platform-tools/`**. Or if you want the standalone Android SDK Platform-Tools package, you can download it here (/studio/releases/platform-tools).

For information on connecting a device for use over ADB, including how to use the Connection Assistant to troubleshoot common problems, see Run apps on a hardware device (/studio/run/device).

How adb works

When you start an adb client, the client first checks whether there is an adb server process already running. If there isn't, it starts the server process. When the server starts, it binds to local TCP port 5037 and listens for commands sent from adb clients—all adb clients use port 5037 to communicate with the adb server.

The server then sets up connections to all running devices. It locates emulators by scanning odd-numbered ports in the range 5555 to 5585, the range used by the first 16 emulators. Where the server finds an adb daemon (adbd), it sets up a connection to that port. Note that each

emulator uses a pair of sequential ports — an even-numbered port for console connections and an odd-numbered port for adb connections. For example:

Emulator 1, console: 5554
Emulator 1, adb: 5555
Emulator 2, console: 5556
Emulator 2, adb: 5557
and so on...

As shown, the emulator connected to adb on port 5555 is the same as the emulator whose console listens on port 5554.

Once the server has set up connections to all devices, you can use adb commands to access those devices. Because the server manages connections to devices and handles commands from multiple adb clients, you can control any device from any client (or from a script).

Enable adb debugging on your device

To use adb with a device connected over USB, you must enable **USB debugging** in the device system settings, under **Developer options**.

On Android 4.2 and higher, the Developer options screen is hidden by default. To make it visible, go to **Settings > About phone** and tap **Build number** seven times. Return to the previous screen to find **Developer options** at the bottom.

On some devices, the Developer options screen might be located or named differently.

You can now connect your device with USB. You can verify that your device is connected by executing `adb devices` from the ***`android_sdk`***`/platform-tools/` directory. If connected, you'll see the device name listed as a "device."

Note: When you connect a device running Android 4.2.2 or higher, the system shows a dialog asking whether to accept an RSA key that allows debugging through this computer. This security mechanism protects user devices because it ensures that USB debugging and other adb commands cannot be executed unless you're able to unlock the device and acknowledge the dialog.

For more information about connecting to a device over USB, read Run Apps on a Hardware Device (/studio/run/device).

Connect to a device over Wi-Fi (Android 11+)

Android 11 and higher support deploying and debugging your app wirelessly from your workstation using Android Debug Bridge (adb). For example, you can deploy your debuggable app to multiple remote devices without physically connecting your device via USB. This eliminates the need to deal with common USB connection issues, such as driver installation.

To use wireless debugging, you need to pair your device to your workstation using a pairing code. Your workstation and device must be connected to the same wireless network. To connect to your device, follow these steps:

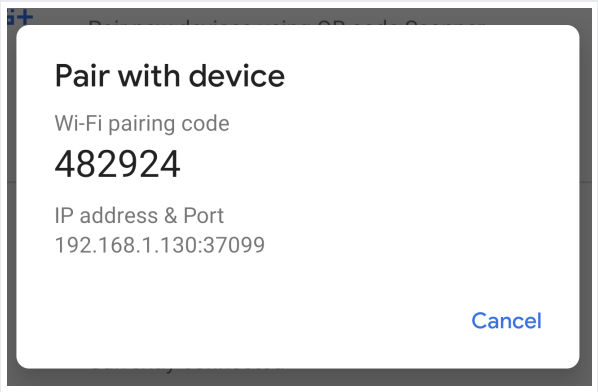

Figure 1. Wireless ADB pairing dialog.

1. On your workstation, update to the latest version of the SDK Platform-Tools (/studio/releases/platform-tools).
2. On the device, enable developer options (/studio/debug/dev-options).
3. Enable the **Wireless debugging** option.
4. On the dialog that asks **Allow wireless debugging on this network?**, click **Allow**.
5. Select **Pair device with pairing code**. Take note of the pairing code, IP address, and port number displayed on the device (see image).
6. On your workstation, open a terminal and navigate to ***android_sdk*`/platform-tools`**.
7. Run `adb pair` ***ipaddr*`:`*****port***. Use the IP address and port number from step 5.
8. When prompted, enter the pairing code that you received in step 5. A message indicates that your device has been successfully paired.

```
none
Enter pairing code: 482924
Successfully paired to 192.168.1.130:37099 [guid=adb-235XY]
```

9. (For Linux or Microsoft Windows only) Run `adb connect ipaddr:port`. Use the IP address and port under **Wireless debugging**.

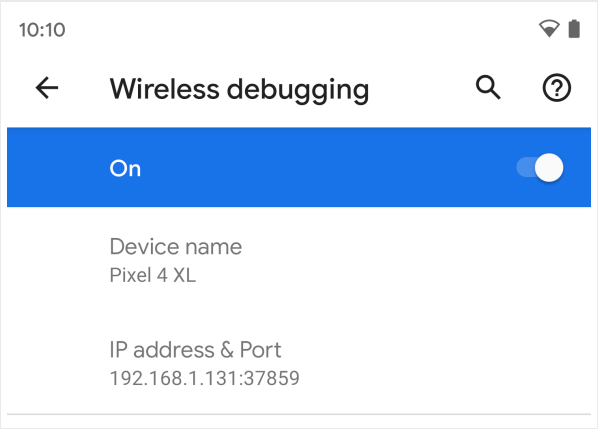

Figure 2. Wireless adb IP and port number.

Connect to a device over Wi-Fi (Android 10 and lower)

adb usually communicates with the device over USB, but you can also use adb over Wi-Fi after some initial setup over USB, as described below. If you're developing for Wear OS, however, you should instead see the guide to debugging a Wear OS app (/training/wearables/apps/debugging), which has special instructions for using adb with Wi-Fi and Bluetooth.

1. Connect your Android device and adb host computer to a common Wi-Fi network accessible to both. Beware that not all access points are suitable; you might need to use an access point whose firewall is configured properly to support adb.
2. If you are connecting to a Wear OS device, turn off Bluetooth on the phone that's paired with the device.
3. Connect the device to the host computer with a USB cable.
4. Set the target device to listen for a TCP/IP connection on port 5555.

```
adb tcpip 5555
```

5. Disconnect the USB cable from the target device.
6. Find the IP address of the Android device. For example, on a Nexus device, you can find the IP address at **Settings** > **About tablet** (or **About phone**) > **Status** > **IP address**. Or, on a Wear OS device, you can find the IP address at **Settings** > **Wi-Fi Settings** > **Advanced** > **IP address**.
7. Connect to the device by its IP address.

   ```
   adb connect device_ip_address
   ```

8. Confirm that your host computer is connected to the target device:

   ```
   $ adb devices
   List of devices attached
   device_ip_address:5555 device
   ```

You're now good to go!

If the adb connection is ever lost:

1. Make sure that your host is still connected to the same Wi-Fi network your Android device is.
2. Reconnect by executing the `adb connect` step again.
3. Or if that doesn't work, reset your adb host:

   ```
   adb kill-server
   ```

   Then start over from the beginning.

Query for devices

Before issuing adb commands, it is helpful to know what device instances are connected to the adb server. You can generate a list of attached devices using the `devices` command.

```
adb devices -l
```

In response, adb prints this status information for each device:

- Serial number: A string created by adb to uniquely identify the device by its port number. Here's an example serial number: `emulator-5554`
- State: The connection state of the device can be one of the following:
  - `offline`: The device is not connected to adb or is not responding.
  - `device`: The device is now connected to the adb server. Note that this state does not imply that the Android system is fully booted and operational because the device connects to adb while the system is still booting. However, after boot-up, this is the normal operational state of an device.
  - `no device`: There is no device connected.
- Description: If you include the `-l` option, the `devices` command tells you what the device is. This information is helpful when you have multiple devices connected so that you can tell them apart.

The following example shows the `devices` command and its output. There are three devices running. The first two lines in the list are emulators, and the third line is a hardware device that is attached to the computer.

```
$ adb devices
List of devices attached
emulator-5556 device product:sdk_google_phone_x86_64 model:Android_SDK_built_for
emulator-5554 device product:sdk_google_phone_x86 model:Android_SDK_built_for_x8
0a388e93      device usb:1-1 product:razor model:Nexus_7 device:flo
```

Emulator not listed

The `adb devices` command has a corner-case command sequence that causes running emulator(s) to not show up in the `adb devices` output even though the emulator(s) are visible on your desktop. This happens when *all* of the following conditions are true:

1. The adb server is not running, and
2. You use the `emulator` command with the `-port` or `-ports` option with an odd-numbered port value between 5554 and 5584, and
3. The odd-numbered port you chose is not busy so the port connection can be made at the specified port number, or if it is busy, the emulator switches to another port that meets the requirements in 2, and
4. You start the adb server after you start the emulator.

One way to avoid this situation is to let the emulator choose its own ports, and don't run more than 16 emulators at once. Another way is to always start the adb server before you use the `emulator` command, as explained in the following examples.

Example 1: In the following command sequence, the `adb devices` command starts the adb server, but the list of devices does not appear.

Stop the adb server and enter the following commands in the order shown. For the avd name, provide a valid avd name from your system. To get a list of avd names, type `emulator -list-avds`. The `emulator` command is in the ***`android_sdk`***`/tools` directory.

```
$ adb kill-server
$ emulator -avd Nexus_6_API_25 -port 5555
$ adb devices

List of devices attached
* daemon not running. starting it now on port 5037 *
* daemon started successfully *
```

Example 2: In the following command sequence, `adb devices` displays the list of devices because the adb server was started first.

To see the emulator in the `adb devices` output, stop the adb server, and then start it again after using the `emulator` command and before using the `adb devices` command, as follows:

```
$ adb kill-server
$ emulator -avd Nexus_6_API_25 -port 5557
$ adb start-server
$ adb devices

List of devices attached
emulator-5557 device
```

For more information about emulator command-line options, see Using Command Line Parameters (/studio/run/emulator-commandline#startup-options).

Send commands to a specific device

If multiple devices are running, you must specify the target device when you issue the adb command. To specify the target, use the `devices` command to get the serial number of the target. Once you have the serial number, use the `-s` option with the adb commands to specify the serial number. If you're going to issue a lot of adb commands, you can set the `$ANDROID_SERIAL` environment variable to contain the serial number instead. If you use both `-s` and `$ANDROID_SERIAL`, `-s` overrides `$ANDROID_SERIAL`.

In the following example, the list of attached devices is obtained, and then the serial number of one of the devices is used to install the `helloWorld.apk` on that device.

```
$ adb devices
List of devices attached
emulator-5554 device
emulator-5555 device

$ adb -s emulator-5555 install helloWorld.apk
```

Note: If you issue a command without specifying a target device when multiple devices are available, adb generates an error.

If you have multiple devices available, but only one is an emulator, use the `-e` option to send commands to the emulator. Likewise, if there are multiple devices but only one hardware device attached, use the `-d` option to send commands to the hardware device.

Install an app

You can use adb to install an APK on an emulator or connected device with the `install` command:

```
adb install path_to_apk
```

You must use the `-t` option with the `install` command when you install a test APK. For more information, see -t (#-t-option).

For more information about how to create an APK file that you can install on an emulator/device instance, see Build and Run Your App (/studio/run).

Note that, if you are using Android Studio, you do not need to use adb directly to install your app on the emulator/device. Instead, Android Studio handles the packaging and installation of the app for you.

Set up port forwarding

You can use the `forward` command to set up arbitrary port forwarding, which forwards requests on a specific host port to a different port on a device. The following example sets up forwarding of host port 6100 to device port 7100:

```
adb forward tcp:6100 tcp:7100
```

The following example sets up forwarding of host port 6100 to local:logd:

```
adb forward tcp:6100 local:logd
```

Copy files to/from a device

Use the **`pull`** and **`push`** commands to copy files to and from an device. Unlike the **`install`** command, which only copies an APK file to a specific location, the **`pull`** and **`push`** commands let you copy arbitrary directories and files to any location in a device.

To copy a file or directory and its sub-directories *from* the device, do the following:

```
adb pull remote local
```

To copy a file or directory and its sub-directories *to* the device, do the following:

```
adb push local remote
```

Replace ***`local`*** and ***`remote`*** with the paths to the target files/directory on your development machine (local) and on the device (remote). For example:

```
adb push foo.txt /sdcard/foo.txt
```

Stop the adb server

In some cases, you might need to terminate the adb server process and then restart it to resolve the problem (e.g., if adb does not respond to a command).

To stop the adb server, use the **`adb kill-server`** command. You can then restart the server by issuing any other adb command.

Issuing adb commands

You can issue adb commands from a command line on your development machine or from a script. The usage is:

```
adb [-d | -e | -s serial_number] command
```

If there's only one emulator running or only one device connected, the adb command is sent to that device by default. If multiple emulators are running and/or multiple devices are attached, you need to use the `-d`, `-e`, or `-s` option to specify the target device to which the command should be directed.

You can see a detailed list of all supported adb commands using the following command:

```
adb --help
```

Issue shell commands

You can use the `shell` command to issue device commands through adb, or to start an interactive shell. To issue a single command use the `shell` command like this:

```
adb [-d |-e | -s serial_number] shell shell_command
```

To start an interactive shell on a device use the `shell` command like this:

```
adb [-d | -e | -s serial_number] shell
```

To exit an interactive shell, press Control + D or type `exit`.

Note: With **Android Platform-Tools 23** and higher, adb handles arguments the same way that the `ssh(1)` command does. This change has fixed a lot of problems with command injection (https://en.wikipedia.org/wiki/Code_injection#Shell_injection) and makes it possible to now safely execute commands that contain shell metacharacters (https://en.wikipedia.org/wiki/Metacharacter), such as **`adb install Let\'sGo.apk`**. But, this change means that the interpretation of any command that contains shell metacharacters has also changed. For example, the **`adb shell setprop foo 'a b'`** command is now an error because the single quotes (`'`) are swallowed by the local shell, and the device sees **`adb shell`**

`setprop foo a b`. To make the command work, quote twice, once for the local shell and once for the remote shell, the same as you do with **`ssh(1)`**. For example, **`adb shell setprop foo "'a b'"`**.

Android provides most of the usual Unix command-line tools. For a list of available tools, use the following command:

```
adb shell ls /system/bin
```

Help is available for most of the commands via the **`--help`** argument. Many of the shell commands are provided by toybox (http://landley.net/toybox/). General help applicable to all toybox commands is available via **`toybox --help`**.

See also Logcat Command-Line Tool (/studio/command-line/logcat) which is useful for monitoring the system log.

Call activity manager (`am`)

Within an adb shell, you can issue commands with the activity manager (**`am`**) tool to perform various system actions, such as start an activity, force-stop a process, broadcast an intent, modify the device screen properties, and more. While in a shell, the syntax is:

```
am command
```

You can also issue an activity manager command directly from adb without entering a remote shell. For example:

```
adb shell am start -a android.intent.action.VIEW
```

Table 2. Available activity manager commands

Command	Description
`start [options] intent`	Start an `Activity` (/reference/android/app/Activity) specified by *`intent`*.

	See the Specification for intent arguments (#IntentSpec). Options are: • `-D`: Enable debugging. • `-W`: Wait for launch to complete. • `--start-profiler` ***`file`***: Start profiler and send results to ***`file`***. • `-P` ***`file`***: Like `--start-profiler`, but profiling stops when the app goes idle. • `-R` ***`count`***: Repeat the activity launch ***`count`*** times. Prior to each repeat, the top activity will be finished. • `-S`: Force stop the target app before starting the activity. • `--opengl-trace`: Enable tracing of OpenGL functions. • `--user` ***`user_id`*** | `current`: Specify which user to run as; if not specified, then run as the current user.
`startservice [`***`options`***`]` ***`intent`***	Start the `Service` (/reference/android/app/Service) specified by ***`intent`***. See the Specification for intent arguments (#IntentSpec). Options are: • `--user` ***`user_id`*** | `current`: Specify which user to run as; if not specified, then run as the current user.
`force-stop` ***`package`***	Force stop everything associated with ***`package`*** (the app's package name).
`kill [`***`options`***`]` ***`package`***	Kill all processes associated with ***`package`*** (the app's package name). This command kills only processes that are safe to kill and that will not impact the user experience. Options are: • `--user` ***`user_id`*** | `all` | `current`: Specify user whose processes to kill; all users if not specified.
`kill-all`	Kill all background processes.
`broadcast [`***`options`***`]` ***`intent`***	Issue a broadcast intent. See the Specification for intent arguments (#IntentSpec).

	Options are: • `[--user user_id | all | current]`: Specify which user to send to; if not specified then send to all users.
`instrument [options] component`	Start monitoring with an `Instrumentation` (/reference/android/app/Instrumentation) instance. Typically the target *component* is the form *test_package*/*runner_class*. Options are: • `-r`: Print raw results (otherwise decode *report_key_streamresult*). Use with `[-e perf true]` to generate raw output for performance measurements. • `-e name value`: Set argument *name* to *value*. For test runners a common form is `-e testrunner_flag value[,value...]`. • `-p file`: Write profiling data to *file*. • `-w`: Wait for instrumentation to finish before returning. Required for test runners. • `--no-window-animation`: Turn off window animations while running. • `--user user_id | current`: Specify which user instrumentation runs in; current user if not specified.
`profile start process file`	Start profiler on *process*, write results to *file*.
`profile stop process`	Stop profiler on *process*.
`dumpheap [options] process file`	Dump the heap of *process*, write to *file*. Options are: • `--user [user_id | current]`: When supplying a process name, specify user of process to dump; uses current user if not specified. • `-n`: Dump native heap instead of managed heap.
`set-debug-app [options] package`	Set app *package* to debug. Options are: • `-w`: Wait for debugger when app starts.

	• `--persistent`: Retain this value.
`clear-debug-app`	Clear the package previous set for debugging with `set-debug-app`.
`monitor [`***`options`***`]`	Start monitoring for crashes or ANRs. Options are: • `--gdb`: Start gdbserv on the given port at crash/ANR.
`screen-compat {on | off)` ***`package`***	Control screen compatibility (/guide/practices/screen-compat-mode) mode of ***`package`***.
`display-size [reset |` ***`width`***`x`***`height`***`]`	Override device display size. This command is helpful for testing your app across different screen sizes by mimicking a small screen resolution using a device with a large screen, and vice versa. Example: `am display-size 1280x800`
`display-density` ***`dpi`***	Override device display density. This command is helpful for testing your app across different screen densities on high-density screen environment using a low density screen, and vice versa. Example: `am display-density 480`
`to-uri` ***`intent`***	Print the given intent specification as a URI. See the Specification for intent arguments (#IntentSpec).
`to-intent-uri` ***`intent`***	Print the given intent specification as an `intent:` URI. See the Specification for intent arguments (#IntentSpec).

Specification for intent arguments

For activity manager commands that take an ***`intent`*** argument, you can specify the intent with the following options:

Show all

`-a` ***`action`***

Specify the intent action, such as `android.intent.action.VIEW`. You can declare this only once.

`-d` ***`data_uri`***

Specify the intent data URI, such as `content://contacts/people/1`. You can declare this only once.

`-t` ***`mime_type`***

Specify the intent MIME type, such as `image/png`. You can declare this only once.

`-c` ***`category`***

Specify an intent category, such as `android.intent.category.APP_CONTACTS`.

`-n` ***`component`***

Specify the component name with package name prefix to create an explicit intent, such as `com.example.app/.ExampleActivity`.

`-f` ***`flags`***

Add flags to the intent, as supported by `setFlags()` (/reference/android/content/Intent#setFlags(int)).

`--esn` ***`extra_key`***

Add a null extra. This option is not supported for URI intents.

`-e | --es` ***`extra_key extra_string_value`***

Add string data as a key-value pair.

`--ez` ***`extra_key extra_boolean_value`***

Add boolean data as a key-value pair.

`--ei` ***`extra_key extra_int_value`***

Add integer data as a key-value pair.

`--el` ***`extra_key extra_long_value`***

Add long data as a key-value pair.

`--ef` ***`extra_key extra_float_value`***

Add float data as a key-value pair.

`--eu` ***`extra_key extra_uri_value`***

Add URI data as a key-value pair.

`--ecn` ***`extra_key extra_component_name_value`***

Add a component name, which is converted and passed as a ComponentName (/reference/android/content/ComponentName) object.

`--eia` ***`extra_key extra_int_value`***`[,`***`extra_int_value...`***`]`

Add an array of integers.

`--ela` ***`extra_key extra_long_value`***`[,`***`extra_long_value...`***`]`

Add an array of longs.

`--efa` ***`extra_key extra_float_value`***`[,`***`extra_float_value...`***`]`

Add an array of floats.

`--grant-read-uri-permission`

Include the flag FLAG_GRANT_READ_URI_PERMISSION (/reference/android/content/Intent#FLAG_GRANT_READ_URI_PERMISSION).

`--grant-write-uri-permission`

Include the flag FLAG_GRANT_WRITE_URI_PERMISSION (/reference/android/content/Intent#FLAG_GRANT_WRITE_URI_PERMISSION).

`--debug-log-resolution`

Include the flag FLAG_DEBUG_LOG_RESOLUTION (/reference/android/content/Intent#FLAG_DEBUG_LOG_RESOLUTION).

`--exclude-stopped-packages`

Include the flag FLAG_EXCLUDE_STOPPED_PACKAGES (/reference/android/content/Intent#FLAG_EXCLUDE_STOPPED_PACKAGES).

`--include-stopped-packages`

Include the flag FLAG_INCLUDE_STOPPED_PACKAGES (/reference/android/content/Intent#FLAG_INCLUDE_STOPPED_PACKAGES).

`--activity-brought-to-front`

Include the flag FLAG_ACTIVITY_BROUGHT_TO_FRONT (/reference/android/content/Intent#FLAG_ACTIVITY_BROUGHT_TO_FRONT).

`--activity-clear-top`

Include the flag FLAG_ACTIVITY_CLEAR_TOP (/reference/android/content/Intent#FLAG_ACTIVITY_CLEAR_TOP).

`--activity-clear-when-task-reset`

Include the flag FLAG_ACTIVITY_CLEAR_WHEN_TASK_RESET (/reference/android/content/Intent#FLAG_ACTIVITY_CLEAR_WHEN_TASK_RESET).

`--activity-exclude-from-recents`

Include the flag FLAG_ACTIVITY_EXCLUDE_FROM_RECENTS (/reference/android/content/Intent#FLAG_ACTIVITY_EXCLUDE_FROM_RECENTS).

`--activity-launched-from-history`

Include the flag FLAG_ACTIVITY_LAUNCHED_FROM_HISTORY (/reference/android/content/Intent#FLAG_ACTIVITY_LAUNCHED_FROM_HISTORY).

`--activity-multiple-task`

Include the flag FLAG_ACTIVITY_MULTIPLE_TASK (/reference/android/content/Intent#FLAG_ACTIVITY_MULTIPLE_TASK).

`--activity-no-animation`

Include the flag FLAG_ACTIVITY_NO_ANIMATION (/reference/android/content/Intent#FLAG_ACTIVITY_NO_ANIMATION).

`--activity-no-history`

Include the flag FLAG_ACTIVITY_NO_HISTORY (/reference/android/content/Intent#FLAG_ACTIVITY_NO_HISTORY).

`--activity-no-user-action`

Include the flag `FLAG_ACTIVITY_NO_USER_ACTION` (/reference/android/content/Intent#FLAG_ACTIVITY_NO_USER_ACTION).

`--activity-previous-is-top`

Include the flag `FLAG_ACTIVITY_PREVIOUS_IS_TOP` (/reference/android/content/Intent#FLAG_ACTIVITY_PREVIOUS_IS_TOP).

`--activity-reorder-to-front`

Include the flag `FLAG_ACTIVITY_REORDER_TO_FRONT` (/reference/android/content/Intent#FLAG_ACTIVITY_REORDER_TO_FRONT).

`--activity-reset-task-if-needed`

Include the flag `FLAG_ACTIVITY_RESET_TASK_IF_NEEDED` (/reference/android/content/Intent#FLAG_ACTIVITY_RESET_TASK_IF_NEEDED).

`--activity-single-top`

Include the flag `FLAG_ACTIVITY_SINGLE_TOP` (/reference/android/content/Intent#FLAG_ACTIVITY_SINGLE_TOP).

`--activity-clear-task`

Include the flag `FLAG_ACTIVITY_CLEAR_TASK` (/reference/android/content/Intent#FLAG_ACTIVITY_CLEAR_TASK).

`--activity-task-on-home`

Include the flag `FLAG_ACTIVITY_TASK_ON_HOME` (/reference/android/content/Intent#FLAG_ACTIVITY_TASK_ON_HOME).

`--receiver-registered-only`

Include the flag `FLAG_RECEIVER_REGISTERED_ONLY` (/reference/android/content/Intent#FLAG_RECEIVER_REGISTERED_ONLY).

`--receiver-replace-pending`

Include the flag `FLAG_RECEIVER_REPLACE_PENDING` (/reference/android/content/Intent#FLAG_RECEIVER_REPLACE_PENDING).

`--selector`

Requires the use of `-d` and `-t` options to set the intent data and type.

`URI component package`

You can directly specify a URI, package name, and component name when not qualified by one of the above options. When an argument is unqualified, the tool assumes the argument is a URI if it contains a ":" (colon); it assumes the argument is a component name if it contains a "/" (forward-slash); otherwise it assumes the argument is a package name.

Call package manager (`pm`)

Within an adb shell, you can issue commands with the package manager (`pm`) tool to perform actions and queries on app packages installed on the device. While in a shell, the syntax is:

```
pm command
```

You can also issue a package manager command directly from adb without entering a remote shell. For example:

```
adb shell pm uninstall com.example.MyApp
```

Table 3. Available package manager commands.

Command	Description
`list packages [options] filter`	Prints all packages, optionally only those whose package name contains the text in ***`filter`***. Options: • `-f`: See their associated file. • `-d`: Filter to only show disabled packages.

	• `-e`: Filter to only show enabled packages. • `-s`: Filter to only show system packages. • `-3`: Filter to only show third party packages. • `-i`: See the installer for the packages. • `-u`: Also include uninstalled packages. • `--user` ***`user_id`***: The user space to query.
`list permission-groups`	Prints all known permission groups.
`list permissions [`***`options`***`]` ***`group`***	Prints all known permissions, optionally only those in ***`group`***. Options: • `-g`: Organize by group. • `-f`: Print all information. • `-s`: Short summary. • `-d`: Only list dangerous permissions. • `-u`: List only the permissions users will see.
`list instrumentation [`***`options`***`]`	List all test packages. Options: • `-f`: List the APK file for the test package. • ***`target_package`***: List test packages for only this app.
`list features`	Prints all features of the system.
`list libraries`	Prints all the libraries supported by the current device.
`list users`	Prints all users on the system.
`path` ***`package`***	Print the path to the APK of the given ***`package`***.
`install [`***`options`***`]` ***`path`***	Installs a package (specified by ***`path`***) to the system. Options: • `-r`: Reinstall an existing app, keeping its data.

- `-t`: Allow test APKs to be installed. Gradle generates a test APK when you have only run or debugged your app or have used the Android Studio **Build > Build APK** command. If the APK is built using a developer preview SDK (if the `targetSdkVersion` is a letter instead of a number), you must include the `-t` option (/studio/command-line/adb#-t-option) with the `install` command if you are installing a test APK.
- `-i` ***`installer_package_name`***: Specify the installer package name.
- `--install-location` ***`location`***: Sets the install location using one of the following values:
  - `0`: Use the default install location
  - `1`: Install on internal device storage
  - `2`: Install on external media
- `-f`: Install package on the internal system memory.
- `-d`: Allow version code downgrade.
- `-g`: Grant all permissions listed in the app manifest.
- `--fastdeploy`: Quickly update an installed package by only updating the parts of the APK that changed.
- `--incremental`: Installs enough of the APK to launch the app while streaming the remaining data in the background. To use this feature, you must sign the APK and create an APK Signature Scheme v4 file (/studio/command-line/apksigner#v4-signing-enabled) . This feature is only supported on certain devices. This option forces adb to use the feature or fail if it is not supported (with verbose information on why it failed). Append the `--wait` option to wait until the APK is fully installed before granting access to the APK.

	`--no-incremental` prevents adb from using this feature.
`uninstall [`***`options`***`] `***`package`***	Removes a package from the system. Options: • `-k`: Keep the data and cache directories around after package removal.
`clear `***`package`***	Deletes all data associated with a package.
`enable `***`package_or_component`***	Enable the given package or component (written as "package/class").
`disable `***`package_or_component`***	Disable the given package or component (written as "package/class").
`disable-user [`***`options`***`] `***`package_or_component`***	Options: • `--user `***`user_id`***: The user to disable.
`grant `***`package_name permission`***	Grant a permission to an app. On devices running Android 6.0 (API level 23) and higher, the permission can be any permission declared in the app manifest. On devices running Android 5.1 (API level 22) and lower, must be an optional permission defined by the app.
`revoke `***`package_name permission`***	Revoke a permission from an app. On devices running Android 6.0 (API level 23) and higher, the permission can be any permission declared in the app manifest. On devices running Android 5.1 (API level 22) and lower, must be an optional permission defined by the app.
`set-install-location `***`location`***	Changes the default install location. Location values: • `0`: Auto: Let system decide the best location. • `1`: Internal: install on internal device storage. • `2`: External: on external media. ★ **Note:** This is only intended for debugging; using this can cause apps to break and other undesireable behavior.

`get-install-location`	Returns the current install location. Return values: • `0 [auto]`: Lets system decide the best location • `1 [internal]`: Installs on internal device storage • `2 [external]`: Installs on external media
`set-permission-enforced` ***`permission`*** `[true | false]`	Specifies whether the given permission should be enforced.
`trim-caches` ***`desired_free_space`***	Trim cache files to reach the given free space.
`create-user` ***`user_name`***	Create a new user with the given ***`user_name`***, printing the new user identifier of the user.
`remove-user` ***`user_id`***	Remove the user with the given ***`user_id`***, deleting all data associated with that user
`get-max-users`	Prints the maximum number of users supported by the device.

Call device policy manager (`dpm`)

To help you develop and test your device management (or other enterprise) apps, you can issue commands to the device policy manager (`dpm`) tool. Use the tool to control the active admin app or change a policy's status data on the device. While in a shell, the syntax is:

```
dpm command
```

You can also issue a device policy manager command directly from adb without entering a remote shell:

```
adb shell dpm command
```

Table 4. Available device policy manager commands

Command	Description

`set-active-admin [`***`options`***`] `***`component`***	Sets ***component*** as active admin. Options are: • `--user `***`user_id`***: Specify the target user. You can
`set-profile-owner [`***`options`***`] `***`component`***	Sets ***component*** as active admin and its package as pro Options are: • `--user `***`user_id`***: Specify the target user. You can • `--name `***`name`***: Specify the human-readable organiz
`set-device-owner [`***`options`***`] `***`component`***	Sets ***component*** as active admin and its package as dev Options are: • `--user `***`user_id`***: Specify the target user. You can • `--name `***`name`***: Specify the human-readable organiz
`remove-active-admin [`***`options`***`] `***`component`***	Disables an active admin. The app must declare andro command also removes device and profile owners. Options are: • `--user `***`user_id`***: Specify the target user. You can
`clear-freeze-period-record`	Clears the device's record of previously-set freeze perio developing apps that manage freeze-periods. See Mana Supported on devices running Android 9.0 (API level 28)
`force-network-logs`	Forces the system to make any existing network logs re `onNetworkLogsAvailable()` (/reference/android/app/admin/DeviceAdminReceiver# callback. See Network activity logging (/work/dpc/loggi This command is rate-limited. Supported on devices rur
`force-security-logs`	Forces the system to make any existing security logs av `onSecurityLogsAvailable()` (/reference/android/app/admin/DeviceAdminReceiver# Log enterprise device activity (/work/dpc/security#log_ This command is rate-limited. Supported on devices rur

Take a screenshot

The `screencap` command is a shell utility for taking a screenshot of a device display. While in a shell, the syntax is:

```
screencap filename
```

To use the `screencap` from the command line, type the following:

```
adb shell screencap /sdcard/screen.png
```

Here's an example screenshot session, using the adb shell to capture the screenshot and the `pull` command to download the file from the device:

```
$ adb shell
shell@ $ screencap /sdcard/screen.png
shell@ $ exit
$ adb pull /sdcard/screen.png
```

Record a video

The `screenrecord` command is a shell utility for recording the display of devices running Android 4.4 (API level 19) and higher. The utility records screen activity to an MPEG-4 file. You can use this file to create promotional or training videos or for debugging and testing.

In a shell, use the following syntax:

```
screenrecord [options] filename
```

To use `screenrecord` from the command line, type the following:

```
adb shell screenrecord /sdcard/demo.mp4
```

Stop the screen recording by pressing Control + C (Command + C on Mac); otherwise, the recording stops automatically at three minutes or the time limit set by `--time-limit`.

To begin recording your device screen, run the `screenrecord` command to record the video. Then, run the `pull` command to download the video from the device to the host computer. Here's an example recording session:

```
$ adb shell
shell@ $ screenrecord --verbose /sdcard/demo.mp4
(press Control + C to stop)
shell@ $ exit
$ adb pull /sdcard/demo.mp4
```

The `screenrecord` utility can record at any supported resolution and bit rate you request, while retaining the aspect ratio of the device display. The utility records at the native display resolution and orientation by default, with a maximum length of three minutes.

Limitations of the `screenrecord` utility:

- Audio is not recorded with the video file.
- Video recording is not available for devices running Wear OS.
- Some devices might not be able to record at their native display resolution. If you encounter problems with screen recording, try using a lower screen resolution.
- Rotation of the screen during recording is not supported. If the screen does rotate during recording, some of the screen is cut off in the recording.

Table 5. `screenrecord` options

Options	Description
`--help`	Displays command syntax and options
`--size` ***`width`***`x`***`height`***	Sets the video size: `1280x720`. The default value is the device's native display resolution (if supported), 1280x720 if not. For best results, use a size supported by your device's Advanced Video Coding (AVC) encoder.
`--bit-rate` ***`rate`***	Sets the video bit rate for the video, in megabits per second. The default value is 4Mbps. You can increase the bit rate to improve video quality, but doing so results in larger movie files. The following example sets the recording bit rate to 6Mbps:

	`screenrecord --bit-rate 6000000 /sdcard/demo.mp4`
`--time-limit` ***`time`***	Sets the maximum recording time, in seconds. The default and maximum value is 180 (3 minutes).
`--rotate`	Rotates the output 90 degrees. This feature is experimental.
`--verbose`	Displays log information on the command-line screen. If you do not set this option, the utility does not display any information while running.

Read ART profiles for apps

Starting in Android 7.0 (API level 24) the Android Runtime (ART) collects execution profiles for installed apps, which are used to optimize app performance. You might want to examine the collected profiles to understand which methods are determined to be frequently executed and which classes are used during app startup.

To produce a text form of the profile information, use the command:

```
adb shell cmd package dump-profiles package
```

To retrieve the file produced, use:

```
adb pull /data/misc/profman/package.txt
```

Reset test devices

If you test your app across multiple test devices, it may be useful to reset your device between tests, for example, to remove user data and reset the test environment. You can perform a factory reset of a test device running Android 10 (API level 29) or higher using the `testharness` adb shell command, as shown below.

```
adb shell cmd testharness enable
```

When restoring the device using `testharness`, the device automatically backs up the RSA key that allows debugging through the current workstation in a persistent location. That is, after the device is reset, the workstation can continue to debug and issue adb commands to the device without manually registering a new key.

Additionally, to help make it easier and more secure to keep testing your app, using the `testharness` to restore a device also changes the following device settings:

- The device sets up certain system settings so that initial device setup wizards do not appear. That is, the device enters a state from which you can quickly install, debug, and test your app.
- Settings:
    - Disables lock screen
    - Disables emergency alerts
    - Disables auto-sync for accounts
    - Disables automatic system updates
- Other:
    - Disables preinstalled security apps

If you app needs to detect and adapt to the default settings of the `testharness` command, you can use the `ActivityManager.isRunningInUserTestHarness()` (/reference/android/app/ActivityManager#isRunningInUserTestHarness()).

sqlite

`sqlite3` starts the sqlite command-line program for examining sqlite databases. It includes commands such as `.dump` to print the contents of a table, and `.schema` to print the `SQL CREATE` statement for an existing table. You can also execute SQLite commands from the command line, as shown below.

```
$ adb -s emulator-5554 shell
$ sqlite3 /data/data/com.example.app/databases/rssitems.db
SQLite version 3.3.12
Enter ".help" for instructions
```

For more information, see the sqlite3 command line documentation (http://www.sqlite.org/cli.html).

Content and code samples on this page are subject to the licenses described in the Content License (/license). Java is a registered trademark of Oracle and/or its affiliates.

Last updated 2020-09-10 UTC.