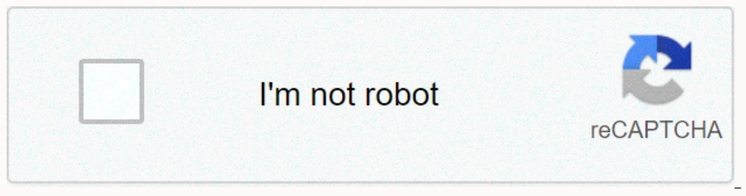

Continue

Android studio debug without build

Content: In this chapter you'll learn about debugging your apps in Android Studio. About Debugging debugging is the process of finding and fixing bugs or unexpected behavior in your code. All code has bugs, from incorrect behavior in your application, to behaviors that consume excessive memory or network resources, to freezing or crashing real applications. Bugs can result for many reasons: Errors in your design or implementation Limitations of Android frames (or bugs) Missing requirements or assumptions of how the app should work Device Limitations (or bugs) Use the debugging, testing, and profiling features in Android Studio to help you reproduce, find, and resolve all of these issues. These features include: The Logcat panel for log messages The Debugger panel for viewing frames, threads, and variables Debug Mode to run applications with breakpoints Test frameworks like JUnit or Dalvik Debug Monitor Server (DDMS) Espresso, to track resource usage In this chapter you learn how to debug your application with Android Studio debugger, set, and preview breakpoints , step through your code and examine variables. Running the debugger running an application in debug mode is similar to running the application. You can run an application in debug mode or attach the debugger to an already running application. Run your application in debug mode To start debugging, click Debug in the toolbar. Android Studio builds an APK, signs it with a debug key, installs it on your selected device, runs it, and opens the Debug panel with the Debugger and Console tabs. The figure above shows the Debug panel with the Debugger and Console tabs. The Debugger tab is selected, showing the Debugger panel with the following characteristics: Frames Tab: Click to show the Frames panel with the current execution stack frames for a given segment. The execution stack shows each class and method that were called in your app and at android runtime, with the latest method at the top. Click the Threads tab to replace the Frames panel with the Threads panel. Clocks button: Click to show the Clocks panel within the Variables panel, which shows the values for any variable clocks that you have set. Clocks allow you to track a specific variable in your program and see how that variable changes as your program runs. Variable panel: Shows the variables in the current scope and their values. Each variable in this pane has an expansion icon to expand the list of properties from the object to the variable. Try expanding a variable to explore its properties. Debug a running app If the app is already running on a device or emulator, start debugging that app with Steps: Select Run > Connect debugger to android process or click the Attach icon in the toolbar. In the Choose Process dialog box, select the process to which you want to attach the debugger. By default, the debugger shows the device and application process for the current project, as well as any connected hardware devices or virtual devices on your computer. Check out everyone's Show option to show all processes on all devices. Click OK. The Debug panel appears as before. Resume or stop debugging To resume running an application after debugging it, select Run > Resume Program or click the Resume icon. To stop debugging your app, select Run > Stop or click the Stop icon in the toolbar. Using breakpoints Android Studio supports various types of breakpoints that trigger different debugging actions. The most common type is a breakpoint that pauses the execution of your application on a specified line of code. While paused, you can examine variables, evaluate expressions, and then continue execution line by line to determine the causes of runtime errors. You can set a breakpoint on any line of executable code. Add breakpoints To add a breakpoint to a line in your code, use these steps: Find the line of code where you want to pause execution. Click the left gutter of the editor panel on this line, next to the line numbers. A red dot appears on this line, indicating a breaking point. The red dot includes a check mark if the application is already running in debug mode. Alternatively, you can choose Run > alternate line breakpoint or press Control-F8 (Command-F8 on a Mac) to set or clear a breakpoint on a line. If your app is already running, you don't need to update it to add the breakpoint. If you click a breakpoint by mistake, you can undo it by clicking the breakpoint. If you clicked a line of code that is not executable, the red dot includes an x and a warning appears that the line of code is not executable. When your code execution reaches the breakpoint, Android Studio pauses the execution of your app. You can then use the tools in the Debug panel to view the state of the application and debug that application as it runs. View and configure breakpoints To view all breakpoints that you have set and configure breakpoint settings, click the 'View breakpoints' icon at the left edge of the Debug panel. The Breakpoints window appears. In this window, all defined breakpoints appear in the left pane and can turn each breakpoint on or off with the check boxes. If a breakpoint is disabled, Android Studio does not pause your app when execution reaches this breakpoint. Select a breakpoint from the list to configure your settings. You can configure a breakpoint to be disabled at the beginning and have the system enable it after a different breakpoint is found. You can also if a breakpoint should be disabled after it has been reached. To set a breakpoint for any exception, select Exception Breakpoints from the breakpoint list. Turning off (mute) all breakpoints Disabling a breakpoint allows you to temporarily mute that breakpoint without removing it from your code. If you remove a breakpoint completely, you also lose any conditions or other features that you have created for that breakpoint, so disabling it may be a better choice. To mute all breakpoints, click the Mute Breakpoints icon. Click the icon again to enable (helpless) all breakpoints. Use conditional breakpoints Breakpoints are breakpoints that only stop running your application if the test in the condition is true. To set a test for a conditional breakpoint, use these steps: Right-click (or control-click) on a breakpoint and enter a test in the Condition field. The test you type in this field can be any Java expression as long as it returns a Boolean value. You can use variable names of your application as part of the expression. You can also use the Breakpoints window to enter a breakpoint condition. Run your application in debug mode. The execution of your application stops at the conditional breakpoint if the condition evaluates the truth. By going through the code After your application's execution has stopped because a breakpoint has been reached, you can run your code from that point one line at a time with the Step Over, Step Into, and Step Out functions. To use any of the step functions: Start debugging your application. Pause the execution of your application with a breakpoint. Application execution stops, and the Debugger panel shows the current state of the application. The current line is highlighted in your code. Click the Step Up icon, select Run > step up or press F8. Step Over executes the next line of code in the current class and method, executing all method calls on that line and remaining in the same file. Click the Step icon, select Run > enter or press F7. Step Into jumps to the execution of a method call on the current line (compared to just executing that method and remaining on the same line). The Frames panel (which you'll learn in the next section) updates to show the new stack frame (the new method). If the method call is contained in another class, the file for that class opens and the current line in that file is highlighted. You can continue to go over lines in this new method call, or dive into other methods. Click the Exit icon, select Run > exit, or press Shift-F8. Step Out finishes running the current method and returns to the point at which this method was called. To resume normal application execution, select Run > Resume Program or click the Resume icon. View execution stack frames The Frame panel of the Debug panel allows you to inspect the execution stack and the specific frame that caused the current breakpoint to be reached. The execution stack shows all the classes and methods (frames) that are running up to this point in the application, in reverse order (most recent frame first). As the execution of a specific frame ends, that frame is overflowed from the stack and execution returns to the next frame. Clicking a line for a frame in the Frames panel opens the associated font in the editor and the line where this frame was initially executed. Variable panels and clocks also update to reflect the state of the execution environment when that frame was last inserted. Inspecting and modifying variables The Variable Panel of the Debugger panel allows you to inspect the variables available in the current when the system stops your application at a breakpoint. Variables that save objects or collections, such as arrays, can be expanded to visualize their components. The variables panel also allows you to evaluate expressions on the fly using static methods or variables available in the selected frame. If the Variables pane is not visible, click the Restore Variables view icon. To modify variables in your application as it runs: Right-click (or control-click) on any variable in the variables panel, and select Set value. You can also press F2. Enter a new value for the variable and press Return. The value you type must be of the appropriate type for this variable, or Android Studio returns a type mismatch error. Setting clocks The clock panel provides functionality similar to the variable panel, except that the expressions added to the clock panel persist between debugging sessions. Add clocks for variables and fields that you frequently access or that provide a state that is useful for the current debugging session. To use watches: Start debugging your app. Click the Show Watches icon. The clock panel appears next to the variable panel. In the Clocks panel, click the plus button (+). In the text box that appears, type the name of the variable or expression you want to watch, and press Enter. Remove an item from the Watches list by selecting the item, and then clicking the minus button (–) Change the order of the elements in the Watch panel list by selecting an item, and then clicking the icons up or down. Evaluating expressions Use the Evaluation Expression to explore the state of variables and objects in your application, including calling methods on those objects. To evaluate an expression: Click the Evaluate expression icon or select Run > Evaluate expression. The Evaluate Code Fragment window is displayed. You can also right-click on any variable and choose Evaluate expression. Type any Java expression in the top field of the Evaluate Code Fragment window and click Evaluate. The Result field shows the result of this expression. Note that the result you receive when evaluating an expression is based on the current state of the application. Depending on the values of the variables in your application at the time you evaluate expressions, you can get different results. Changing the values of variables in their expressions also changes the current execution state of the application. Android Studio and the Android SDK include a number of other tools to help you find and fix problems in your code. Log log (Logcat panel) As you learned in another chapter, you can use the Log class to send messages to the Android system registry and view those messages in Android Studio in the Logcat panel. To write code, use the Log class. Log messages help you understand the execution flow by collecting system debug output while you interact with your application. Log messages can tell you which part of your application failed. For more information about registration, see Read and Write Records. Tracking and registration registration Traces allow you to see how much time is spent on certain methods, and which are taking the longer times. To create the trace files, include the Debug class and call one of the startMethodTracing() methods. In the call, you specify a base name for the trace files that the

system generates. To stop tracking, call stopMethodTracing(). These methods start and stop crawling the method across the virtual machine. For example, you can call the startMethodTracing() in the onCreate() method of your activity and call stopMethodTracing() in the onDestroy() method of that activity. Android Debug Bridge (ADB) is a command-line tool that lets you communicate with an emulator instance or device connected with Android. Android Profiler provides real-time data for your app's CPU, memory, and network activity. You can perform sample-based method tracing to time your code execution, capture stack dumps, view memory allocations, and inspect the details of networked files. CPU Profiler helps you inspect your application's CPU usage and thread activity in real time and record method traces, so you can optimize and debug your application's code. Network Profiler displays real-time network activity on a timeline, showing data sent and received, as well as the current number of connections. This allows you to examine how and when your application transfers data and optimizes the underlying code accordingly. Tracking log and the AndroidManifest file.xml There are several types of debugging available to you besides setting breakpoints and passing through code. You can also use logging and tracking to find problems with your code. When you have a trace log file (generated by adding trace code to your application), you can upload the log files to Traceview, which displays the log data in two panes: A timeline pane describes when each thread and method started and stopped. A profile panel provides a summary of what happened within a method. Similarly, to make the app depurable even when the app is running on a device in user mode, you can set android:depurable in the AndroidManifest <application> tag.xml file to true. By default, the depurable value is set to false. You can create and configure build types in the module-level build.gradle file within the android block {}. When you create a new module, Android Studio creates the build debug and debug types for you. Although the debug build type does not appear in the build configuration file, Android Studio configures it with debuggable truth. This allows you to debug the app on android-safe devices and the APK login with a generic debug key store. If you want to add or change certain settings, add the debug build type to your configuration. When you prepare your application for release, you must remove all the extra code in your source files that you wrote for testing purposes. In addition to preparing the code itself, there are a few other tasks you need to complete to get your app ready for </application> </application> These include: Removal of registration claims. Remove all calls to show toast messages. Disable debugging in the AndroidManifest file.xml removing android:debugable attribute from <application> tag or setting android:debugable to fake. Remove all debugging trace calls from your source code files, such as startMethodTracing() and stopMethodTracing(). All of these changes made to debugging must be removed from your code before release because they can impact execution code and performance production. The related practice is 3.1: The debugger. Learn more documentation from Android Studio: Other: Video: Debugging and Testing at Android Studio Studio </application>