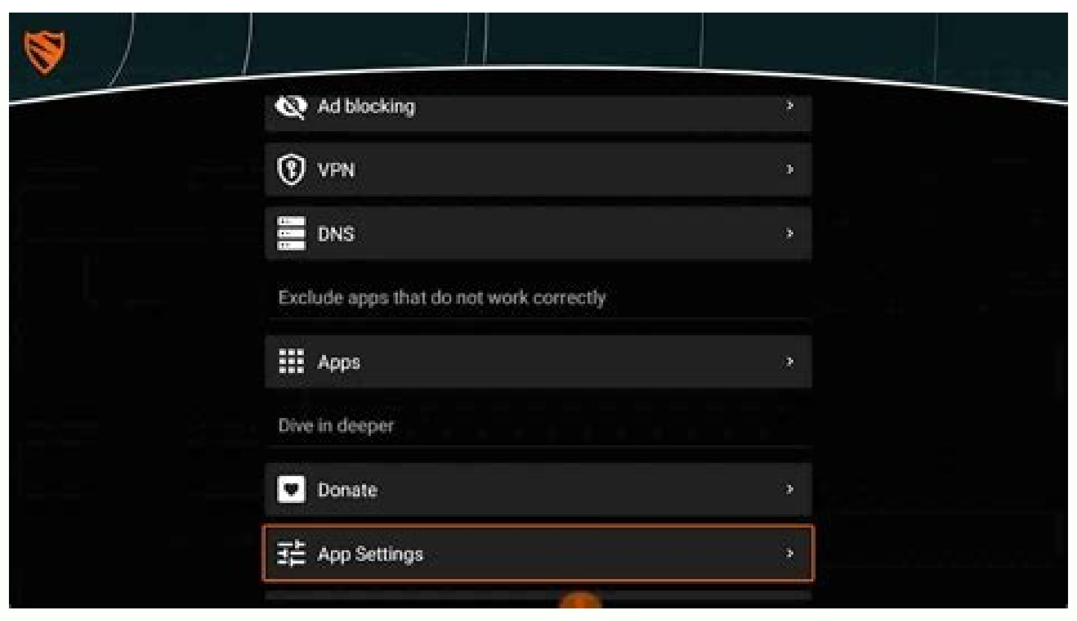
Block ads on android tv

Continue



Block ads on sony android tv. How to block youtube ads on android tv box. How to stop ads on android tv. How do i block ads on youtube android tv. How to block all ads on android tv. How to block ads on youtube app android tv. Block ads on android tv box. Block ads on android tv youtube.

(Pocket-lint) - No one likes to be interrupted by annoying ads, but it has become a common occurrence on our phones. Most people know they can block ads on their computer or laptop, but our phones usually suffer from these things. Long gone are the days when you could install Ad Block Plus from the Play Store, Google no longer allows system-wide ad blocking. So how do you block ads? Fortunately, it's very easy and we've covered everything in this tutorial. But first, a few things need to be mentioned. It's important to remember that advertising helps keep websites (including this one) running and is also important for content creators. Although they can be annoying, in some cases it's worth allowing ads to support content you like. We also focus on blocking ads in the browser. Note that these solutions do not block ads in other applications such as games. With that, let's move on to the guide. How to block pop-ups and annoying ads in Chrome Chrome is the default browser on most Android phones, and as such the vast majority of people stick with it. However, you don't need to switch to another browser to block ads. Fortunately, Chrome has some useful built-in tools to help you with this. The only caveat is that Chrome doesn't block all ads, but only pop-ups and ads it deems intrusive or misleading. How to activate features: Pocket-lint Open Chrome on your Android phone. Click on the three dots in the upper right corner. Click on "Settings". Scroll down to "Site Settings" and select a page. Click "Advertise" and do the same. Chrome will now prevent most pop-ups from loading and completely block ads on sites with misleading and intrusive ads. In a way, it's the best of both worlds, as it allows you to support the content you care about without suffering much.(Pocket Patch) - Nobody likes to be interrupted by annoying ads, but it has become commonplace on our phones. Most people know that they can block ads on their desktops or laptops, but our phones usually suffer from it. Long gone are the days when you could install Ad Block Plus from the Play Store, Google no longer allows system-wide ad blockers. How do you block ads? Luckily, it's very easy and this guide has it all covered. But first, a few things should be mentioned. It's important to remember that ads help keep websites (including this one) running and are also important for content creators. While they can be annoying, there are times when it's worth allowing ads to support the content you offer. Love. Also, we focus on blocking ads in the browser. So remember that these solutions do not block ads in other apps like games. With that said, let's dive into the guide. How to Block Popups and Annoying Ads in Chrome Chrome is the default browser on most Android phones and hence most people follow it. However, you don't have to switch to a different browser to block ads. Luckily, Chrome has some handy tools built in to help you with that. The only caveat is that Chrome doesn't block all ads, only pop-ups and ads that it finds intrusive or misleading. To enable the features: Pocket Clips Open Chrome on your Android phone. Click on the three dots in the top right. Click on "Settings". Scroll down to "Site Settings" and select it. Click on "Pop-ups and Redirects". Make sure the slider is moved to the left. Back to previous. pageClick on the ad and do the same. Chrome now prevents most pop-ups from loading and completely blocks ads on websites with misleading and intrusive advertising. In a way, it's the best of both worlds, as it allows you to support content you care about without horrible suffering.But what if you want to go ahead and block everything? Read more. How to block ads in different browsers If you're ready to ditch Chrome, there are plenty of browsers that offer more effective ad blocking options. Just keep in mind that if you use this, you won't be able to sync your history and bookmarks with Chrome on your desktop. Pocket-lint Our favorite option is Firefox, which lets you install add-ons that act as Chrome extensions on your desktop. uBlock Origin is a powerful free and open source ad blocker that can be easily added to Firefox for Android and will block almost anything. Another great and full-featured browser is Opera, which has a powerful ad blocker built in and even a free VPN. There's even a simple Adblock browser if you just want to block ads. How to block ads in apps Browsers have it figured out, but what if you want to block ads in other apps? As we mentioned earlier, Google removed ad-blocking apps from the Play Store a long time ago, but that doesn't mean you can't block ads elsewhere. You'll need to download these apps, which can be a little tricky if you've never done it before, but that doesn't mean the apps aren't trustworthy — it's just that Google isn't interested in you using them. . Which is understandable given its business model. The Best iPhone Apps of 2022: The Complete Guide Maggie Tillman April 30, 2022 These are the absolute best iPhone apps available right now, from productivity apps to travel, reading, music, and more. Apps like AdGuard and AdLock are developed by well-known cybersecurity brands, so they should be completely safe to use. The downside is that they're subscription-based, so you'll have to pay a monthly fee after the trial period ends. Written by Luke Baker. Android Studio's Gradle build makes it easy to include external binaries or other library modules as dependencies in your build. Dependencies may be on your computerin the remote repository, and any transitive dependencies they declare are automatically included as well. This page describes how dependencies are used in an Android project, including details on how the Android Gradle plugin works and how to configure it. For more detailed conceptual guidance on Gradle dependencies, see the Gradle Dependency Management Guide. However, note that your Android project should only use the dependency configurations defined on this page. Note. Don't use dynamic version numbers like "com.android.tools.build:gradle:3.+" when specifying dependencies. Using this feature can result in unexpected version updates and difficulty resolving version differences. Dependency Types To add a dependency to your project, specify the dependency configuration, e.g. B. Implementation, in the dependency block of the module's build.gradle file. For example, this application module file build.gradle has three different types of dependencies: plugins { id 'com.android.application' } android { ... }dependencies { // library module implementation dependency (':mylibrary') // Depends on local binary implementation fileTree(dir: 'libs', include: ['*.jar']) // Depends on remote binary implementation 'com.example.android:app-magic:12.3' } plugins { id ("com .android .application") } android { ... }dependencies { // Dependency on local library module implementation (project(":mylibrary")) // Dependency on local binary implementation (fileTree (mapOf("dir") on " libs" , "include " to listOf("*.jar"))))) // Dependency on remote binary implementation ("com.example.android:app-magic:12.3") } Each requires a different type of library dependency as follows: Dependency implementation of the local library module project(':mylibrary') Implementation(project(":mylibrary")) Explain e the conformance of the Android library module named "mylibrary" (this name must match the defined library nameinclude: in settings.gradle). When building an application, the build system compiles the library module and packages the resulting compiled content into the application. local binary dependency implementation fileTree(dir: 'libs', include: ['*.jar']) implementation(fileTree(mapOf("dir" to "libs", "include" to listOf("*.jar")))) Gradle declares dependencies on JAR files in your project's module_name/libs/ directory (because Gradle reads paths relative to build.gradle file). You can also specify individual files as follows: implementation files('libs/foo.jar', 'libs/bar.jar') Binary dependency implementation "com.example.android:app-magic:12.3" ("com.example. android:app-magic:12.3") This is actually short for: implementation group: "com.example.android", name: "app -magic", version: implementation "12.3" (group = "com.example.android", name = "app-magic", version = "12.3") This declares a dependency on the 12.3 version of the "app-magic" library in the namespace group " com.example.android". Note. Such remote dependencies must be declared by the corresponding remote repositories, where Gradle should look for the library. If the library doesn't already exist locally, Gradle will fetch it from the remote site when it's needed for a build (for example, when you click Sync project with Gradle files or when running a build). If you rely on an AGP dependency at runtime build, be sure to add it as an explicit dependency uses the API/implementation configuration internally, some artifacts may be removed from the build classpath and the build classpath may change. Native dependencies Starting with the Android Gradle 4.0 plugin, native dependencies can also be imported as described in this document. Depending on the AAR that provides the native libraries, it will automatically make them available to the build system via externalNativeBuild. To have an accessyou must reference it from your code in your native build scripts. For more information, see the Using Local Dependencies section of this document. Dependency Configuration In the Dependencies block, you can declare a library dependency using one of several different dependency configurations (such as the implementation shown above). Each dependency configuration gives Gradle different instructions for using the dependency. This table describes all the configurations you can use for an Android project dependency. The table also compares these configurations to those deprecated by Android Gradle plugin 3.0.0. Gradle's configuration behavior implementation adds the dependency to the compilation classpath and includes the dependency in the build output. However, when a module configures an implementation dependency, it tells Gradle that you don't want the module to propagate the dependency to other modules at compile time. This means that the dependency is only available to other modules at runtime. Using this dependency configuration instead of api or compile (deprecated) can significantly reduce build time by reducing the number of modules the build system needs to recompile. For example, if an implementation dependency changes its API, Gradle recompiles only that dependency and the modules that directly depend on it. Most applications and test modules should use this configuration. The gradle api adds a dependency on compiling the classpath and creating the output. If a module has an API dependency, it tells Gradle that the module wants to temporarily export that dependency to other modules so that it is available at runtime and compile time. This configuration works like compile (which is now deprecated), but use it carefully and only with dependencies that need to be temporarily exported to other upstream consumers. This is because when an API dependency changes its external API, Gradleall modules that have access to this dependency at compile time. Therefore, a large number of API dependencies can significantly increase compilation time. Unless you plan to expose API dependencies to a separate module, library modules should use implementation dependencies instead. CompileOnly Gradle only adds a dependency to the build classpath (i.e. it's not added to the build output). It's useful when you're building an Android module and need a compile-time dependency, but its presence at runtime is optional. When using this setup, the library module needs to include a runtime condition to check if the dependency is available, then neatly modify its behavior to continue working even if not deployed. This helps to reduce the size of the final application by not adding non-critical transient dependencies. This configuration behaves exactly as stated (which is now deprecated). Note: You cannot use build-only configurations with AAR dependencies. runtimeOnly Gradle will only add a dependency to the build output for use at runtime. This means it is not added to the build classpath. This setup behaves exactly like the apk (which is now deprecated). annotationProcessor To add a dependency on an annotation processor library, add it to the annotation processor class using the annotationProcessor configuration. This is because using this configuration improves build performance by separating the build classpath from the annotation processor classpath. If Gradle finds annotation processors in the build's classpath, build avoidance is disabled, which negatively affects build time (Gradle 5.0 and above ignores annotation processors found in the build's classpath). The Android Gradle plugin considers a dependency as an annotation processor if its JAR contains the following file: META-INF/services/javax.annotation.processing.Processor If the plugin detects an annotation processorCompiling the classpath will cause a build error. Note. Kotlin projects should use capt to declare annotation processor dependencies. lintChecks Use this configuration to include lint checks that Gradle should run when building a project. Note. If you're using Android Gradle plugin 3.4.0 and later, this dependency configuration no longer includes thread checks in your Android library projects. To include the patch check dependencies in your AAR libraries, use the lintPublish configuration described below. lintPublish Use this configuration in your Android library projects to include the patch checks you want Gradle to compile into a lint.jar file and package into your AAR. Thus, projects that consume your AAR will also use these thread checks. If you previously used the lintChecks dependency configuration to include lint checks in your published AAR, you must migrate those dependencies to use the lintPublish configuration instead. Dependencies { // Run thread check from project ":checks" during build. lintChecks project(':checks') // Compiles the lint checks from ':checks-to-publish' // into lint.jar and publishes it to your Android library. lintPublish project(':checks-to-publish') }dependencies { // Run the lint check from project ":checks" during build. lintChecks(project(":checks")) // Compiles the collection checks from ":checks-to-publish" // into a lint.jar file and publishes it to your Android library. lintPublish(project(":test-publish")) } The Gradle APK only adds a dependency to create the output for use at runtime. That is, it is not added to the compilation classpath. This configuration is deprecated (available in AGP 1.0-4.2). Compile Gradle adds the dependency to the compiler classpath and build output and exports the dependency to other modules. This configuration is deprecated (available in AGP 1.0-4.2). provided that Gradle only adds the dependency to the compilation classpath (ie it is not added to the buildThis configuration is deprecated (available in AGP 1.0-4.2). All of the above configurations apply dependencies to all build variants. If instead you want to declare a dependency only on a specific build variant source set or test source set, you must capitalize the configuration name and precede it with the build variant or test source set name. For example, to add an implementation dependency to only the "free" product variant (using a remote binary dependency), it would look like this: dependencies { freeImplementation 'com.google.firebase:firebase-ads:9.8.0' } dependencies { freeImplementation(" com. google.firebase:firebase-ads:9.8.0") } However, if you want to add a dependency for a variant that combines a product variant and a build type, you must initialize the config name in config block. The following example adds a runtimeOnly dependency (using a local binary dependency) to the freeDebug build variant. configurations { // Initializes the dependency configuration placeholder freeDebugRuntimeOnly. freeDebugRuntimeOnly {} }dependencies { freeDebugRuntimeOnly fileTree(dir: 'libs', include: ['*.jar']) } // Initializes the freeDebugRuntimeOnly dependency configuration placeholder. val freeDebugRuntimeOnly with config. Create dependencies { freeDebugRuntimeOnly(fileTree(mapOf("dir" to "libs", "include" to listOf("*.jar")))) } To add implementation dependencies for local tests and instrument tests, it looks like this: Dependencies { // Add remote binary dependency for local testing only. testImplementation 'junit:junit:4.12' // Add remote binary dependency for instrumented test APK only. androidTestImplementation 'androidx.test.espresso:espresso-core:3.0.2' }dependencies { // Adds a remote binary dependency for local testing only. testImplementation("junit:junit:4.12") // Adds a remote binary dependency for the instrumented test APK only.} However, some configurations do not make sense in this situation. For example, since other modules cannot depend on AndroidTest, if you use the androidTestApi configuration, you will get the following warning: WARNING. The configuration "androidTestApi" is deprecated and has been replaced by "androidTestImplementation". Adding annotation handlers When you add annotation handlers to the build classpath, you get an error message similar to the following: Error: Annotation handlers must now be explicitly declared. To avoid this error, add annotation processors to the project by configuring a dependency with the annotationProcessor tool as shown below: dependencies { // Adds the annotation-defining libraries to the build classpath only. compileOnly "com.google.dagger:dagger:version-number" // Add the annotation processor dependency to the annotation processor classpath. annotationProcessor 'com.google.dagger:dagger-compiler:version-number' }dependencies { // Add the annotation-defining library to the build classpath only. compileOnly("com.google.dagger:dagger:version-number") // Add annotation processor dependency to annotation processor classpath. annotationProcessor("com.google.dagger:dagger-compiler:version-number") } Note. Android Plugin Gradle 3.0.0+ no longer supports the android-apt plugin. Passing Arguments to Annotation Processors If you want to pass arguments to an annotation processor, you can do so using the AnnotationProcessorOptions module's build configuration block. For example, if you want to pass primitive data types as key-value pairs, you can use the argument property as shown below: android { ... defaultConfig { ... javaCompileOptions { annotationProcessorOptions { argument "key1", "value1" argument "key2" , "value2" } } } } Android { ... defaultConfig { ... javaCompileOptions { annotationProcessorOptions { argument + = mapOf ("key1" to "value1", "key2" to "value2") } } } } with the Android Gradle plugin version 3.2.0 or higheryou need to pass cpu arguments representing files or directories using the Gradle CommandLineArgumentProvider interface. Using the CommandLineArgumentProvider allows you or the author of the annotation processor to improve the correctness and performance of incremental pure assemblies and cached pure assemblies by applying incremental annotations of the assembly property type to each argument. For example, the following class implements CommandLineArgumentProvider and annotates each argument to the processor. The example also uses the Groovy syntax and is included directly in the build.gradle file of the module. Note. Annotation processor authors typically provide this class or instructions for writing such a class. This is because each argument must specify a valid assembly property type annotation for it to work correctly. class MyArgsProvider Implements CommandLineArgumentProvider { // Annotates each directory as input or output to the // annotation processor. @InputFiles // Using this annotation helps Gradle determine which part of the // file path to consider in current checks. @PathSensitive(PathSensitivity.RELATIVE) FileCollection inputDir @OutputDirectory File outputDir // The class constructor sets the input and output directory paths. MyArgsProvider(FileCollection input, File output) { inputDir = input outputDir = output } // Specify each directory as a command line argument to the processor. // The Android plugin uses this method to pass arguments // to the annotation processor. @Override Iterable asArguments() { // Use the "-Akey[=value]" form to pass parameters to the Java compiler. ["-AinputDir=${inputDir.singleFile.absolutePath}", "-AoutputDir=${outputDir.absolutePath}"] } } android {...} class MyArgsProvider(// Mark each directory as input or output for // @get:InputFiles annotation processor // Using this annotation will help Gradle determine which part of the file path // to considercurrent tests. @get:PathSensitive(PathSensitivity.RELATIVE) val inputDir: FileCollection, @get:OutputDirectory val outputDir: File) : CommandLineArgumentProvider { // Specify each directory as a processor command line argument. // The Android plugin uses this method to pass arguments to the // annotation processor. override fun asArguments(): Iterable { // Use the "-Akey[=value]" form to pass your options to the Java compiler. return listOf("-AinputDir=${inputDir.singleFile.absolutePath}", "-AoutputDir=${outputDir.absolutePath}") } } android {...} After defining a class that implements CommandLineArgumentProvider, you must and instantiate it pass it to the android plugin using the annotationProcessorOptions.compilerArgumentProvider method as shown below. // This is in your module's build.gradle file. android { defaultConfig { javaCompileOptions { annotationProcessorOptions { // Creates a new MyArgsProvider object, specifies the input and // output paths of the constructor, and passes the // object to the Android plugin. CompilerArgumentProvider new MyArgsProvider(files("input/path"), new File("output/path")) } } } } // This is in your module's build.gradle file. android { defaultConfig { javaCompileOptions { annotationProcessorOptions { // Creates a new MyArgsProvider object, specifies the input and // output paths of the constructor, and passes the // object to the Android plugin. CompilerArgumentProvider(MyArgsProvider(files("input/path"), file("output/path"))) } } } } To learn more about how the CommandLineArgumentProvider implementation helps improve build performance, see Caching Java projects. Disable annotation processor error checking If you have dependencies on your build classpath that contain annotation processors that you don't need, you can disable error checking by adding the following to your build.gradle file. Note that the annotation processors that you add to the compilation classpath are still not added to the processorandroid { ... defaultConfig { ... javaCompileOptions { annotationProcessorOptions { includeCompileClasspath false } } } } android { ... defaultConfig { ... javaCompileOptions { annotationProcessorOptions { argument ("includeCompileClasspath", "false") } } } } If used Kotlin and kapt: android { ... defaultConfig { ... kapt { includeCompileClasspath false } } } android { ... defaultConfig { ... kapt { includeCompileClasspath = false } } } processor class, you can enable annotation processors in assembly classpath by setting includeCompileClasspath to true. However, setting this property to true is deprecated and this setting will be removed in a future Android plugin update. Eliminating transitive dependencies As an application grows in size, it can contain a number of dependencies, including direct dependencies and transitive dependencies (libraries that imported application libraries depend on). To exclude transitive dependencies that are no longer needed, you can use the Exclude keyword like this: implementation ("some-library") { exclude (group="com.example.imgtools", module="native") } } dependencies from test configurations If you want to exclude certain transitive dependencies from your tests, the code shown in the above example may not work properly. This is because the test configuration (like androidTestImplementation) extends the module's implementation configuration. This means that it always includes implementation dependencies when Gradle resolves the configuration. So in order to exclude transitive dependencies from your tests, you should do it at run time like below: android.testVariants.all { variant -> variant.getCompileConfiguration(). 'group: 'com.jakewharton.threetenabp', module: 'threetenabp' } android.testVariants.all { compileConfiguration.exclude(group = "com.jakewharton.threetenabp", module = "threetenabp") runtimeConfiguration.exclude(group = "com .jakewharton.threetenabp", module = "threetenabp") } Note: You can still use the exclusive keyword in the dependency block, as shown in the original code example of Excluding transitive dependencies, to bypass transitive dependencies that are specific to your test setup and not be included . in other configurations. Configuring Wear OS app dependencies Configuring dependencies for a Wear OS module is similar to any other module; That is, Wear OS modules only use the same dependency configurations as implement and build. Wear modules also support variant-aware dependency management. So if the app's base module has a dependency on the Wear module, each variant of the base module uses the corresponding variant from the Wear module. If you are building a simple app that depends on only one Wear module, where the module configures the same variants as the base module, you must specify the wearApp configuration in the base module's build.gradle file as shown below:dependencies { // If the main modules and app modules only wear these variants, // variant-aware dependency management automatically links // variants of the main application module to the variants of the wear module. wearApp project(':wearable') }dependencies { // If the main application module and the wear module have the same variants, // variant-aware dependency management automatically merges // the main application module variants with the wear module variants. wearApp(project(":wearable")) } If you have multiple wear modules and want to specify a different wear module for each version of the app, you can do that using the FlavorWearApp configuration as follows (but you can't specify other dependencies include that use the wearApp configuration): dependencies { paidWearApp project(':wear1')project(':wear1') freeWearApp project(':wear2') }dependencies { "paidWearApp"(project(":wear1")) "demoWearApp"(project(":wear1")) "freeWearApp"(project(": Wear2")) } Remote repositories If your dependency is not a local library or file tree, Gradle looks for files in the online repositories specified in the DependencyResolutionManagement { repositories {...} } block of your settings.gradle file. The order in which you list each repository determines the order in which Gradle looks for the repositories for each project dependency. For example, if a dependency is available from both repository A and repository B, and you specify A first, Gradle will download the dependency from repository A. By default, new Android Studio projects specify the Google Maven repository and the Maven central repository as repository locations in the settings file .gradle of the project as shown below: DependencyResolutionManagement { repositoriesMode.set(RepositoriesMode.FAIL_ON_PROJECT_REPOS) repositories { google() mavenCentral() } } DependencyResolutionManagement { repositoriesMode.set(RepositoriesMode.FAIL_ON_PROJECT_REPOS) repositories { google() mavenCentral() } Warning: 31 March 2021 The default JCenter repository is read-only. For more information, see Updating the JCenter Service. If you need something from a local repository use mavenLocal():dependencyResolutionManagement { repositoriesMode.set(RepositoriesMode.FAIL_ON_PROJECT_REPOS) repositories { google() mavenCentral() mavenLocal() } }dependencyResolutionManagement { repositoriesMode.set(RepositoriesMode.FAIL_ON_PROJECT_REPOS) repositories { google ( ) mavenCentral() mavenLocal() } } You can also declare specific Maven or Ivy repositories like this: } maven { url 'file://local/repo/' } ivy { url} } }dependencyResolutionManagement { repositoriesMode.set(RepositoriesMode.FAIL_ON_PROJECT_REPOS) Repositories { maven(url = ") maven(url = "file://local/repo/") ivy(url=" } } } See the Gradle Repositories Guide for more information. Google Maven Repository The latest versions of the following Android libraries are available in the Google Maven Repository: You can see all available artifacts in the Google Maven Repository Index (see programmatic access below). To add one of these libraries to your build, add the Google Maven repository to your top-level build.gradle file: dependencyResolutionManagement { repositoriesMode.set(RepositoriesMode.FAIL_ON_PROJECT_REPOS) repositories { google() // If you have a Gradle version lower than 4.1, instead you should use: // maven { // url ' ' // } // Alternative URL is " dl/android/ maven2 / '. } }dependencyResolutionManagement { repositoriesMode.set(RepositoriesMode.FAIL_ON_PROJECT_REPOS) repositories { google() // If you are using a version of Gradle earlier than 4.1, you should use: // maven instead { // url = "https :// maven . google.com" // } // An alternative URL is " . } } Next, add the required library to your module's dependencies block. For example, the Appcompat library looks like this: dependencies { implementation 'androidx.appcompat:appcompat:1.2.0' } dependencies { implementation ("com.an droid.support:appcompat-v7:28.0.0") } However, if for example you try to use an older version of the above libraries and your the dependency fails, it is not available in the Maven repository and you have to get from the offline repository.Programmatic access To access Google Maven artifacts programmatically, you can get a list of XML artifact groups at maven.google.com/master-index.xml.Then you can view their names and versions libraries for each group at:For example, the libraries in the android.arch.lifecycle group are listed at maven.google.com/android/arch/lifecycle/group-index.xml. You can also download the POM and JAR files from maven.google.com/group_path/library/version/library-version.ext, for example: maven.google.com/android/arch/lifecycle/compiler/1.0.0/compiler. - 1. 0.0 assists Offline Repository from SDK Manager For libraries not available from Google Maven repository (usually older versions of libraries), download the Google Offline Repository package from SDK Manager. You can then add these libraries to your dependency block as usual. Offline libraries are stored in the android_sdk/extras/ folder. The native AAR dependencies of the Android Gradle plugin may contain native dependencies that the Android Gradle plugin can use. AGP is also able to create AARs that expose local libraries to their consumers. Using Native Dependencies Starting with the Android Gradle 4.0 plugin, C/C++ dependencies can be imported from AARs that are linked in the build.gradle file. Gradle will automatically make them available to your local build system, but your build system must be configured to use the imported libraries and headers. Since C/C++ dependencies are distributed as AARs, the following links to generic AARs may be useful: Creating an Android library for generic AAR documentation and integrating it into your project, especially if you want to use AAR as native C/C++. addiction. Add Build Dependencies for information about adding dependencies to build.gradle, especially for remote dependencies. This document focuses on how to set up a native build system and assumes that you have already added a C/C++ dependent AAR to your project's Gradle build environment. AAR Native Dependencies AAR dependencies of Gradle modules can provide native libraries for your application to use. Inside the AAR, the prefab directory contains the prefab package that contains the headersown dependency library. Each dependency can provide at most one build package containing one or more modules. A prefabricated module is a standalone library that can be shared, static, or just a header. To use libraries, you need to know the package and module names. By convention, the package name will be the name of the Maven artifact, and the module name will be the name of the C/C++ library, but this is not required. See the dependency documentation to see what names it uses. Build Configuration Android Gradle Plugin 4.0 Android Gradle Plugin 4.1+ Your Android Gradle Module must have build enabled. To do this, add the following to the Android block in the module's build.gradle file: buildFeatures { prefab true } buildFeatures { prefab = true } Further configure the version in the project's gradle.properties file: the default AGP version will suit your needs. You should only choose a different version if there is a bug that needs to be fixed or if you need a new feature. Dependencies imported from AAR are made available to CMake via CMAKE_FIND_ROOT_PATH. This value is automatically set by Gradle when you call CMake, so if your build changes this variable, be sure to attach it instead of assigning it. Each dependency makes the config file package available to your build. They are imported using the find_package command. This command looks for configuration file packages that match the given package name and version, and provides the targets it defines to use in your build. For example, if your application defines a libapp.so file and uses cURL, the CMakeLists.txt file should contain the following text: add_library(app SHARED app.cpp) # Add these two lines. find_package(curl REQUIRED CONFIGURATION) target_link_libraries(app curl::curl) app.cpp can now #include "curl/curl.h", libapp.so will automatically link to libcurl.so on build and libcurl.so willwith application. Publishing local libraries to AAR. The ability to create native AARs was first added in AGP 4.1. To export native libraries, add the following to the Android block in your library project's build.gradle file: buildFeatures { prefabPublishing true } prefab { mylibrary { headers "src/main/cpp/mylibrary/include" } myotherlibrary { headers " src/ main / cpp/myotherbibrary/ include" } } buildFeatures { prefabPublishing = true } prefabs { create("mylibrary") { headers = "src/main/cpp/myotherbibrary/include" } create("myotherbibrary") { headers = "src/main/cpp/ myotherlibrary/include" } } In this example, myotherlibrary and myotherlibrary from your ndk build or external CMake native are packaged into the AAR generated by your build, and each exported with headers from a specified directory to their dependents . Dependency Order The order in which dependencies are listed indicates their priority: the first library has a higher priority than the second, the second has a higher priority than the third, and so on. This order is important when combining features or manifest items from libraries into applications. For example, if your project declares: a dependency on LIB_A and LIB_B (in that order), and LIB_A depends on LIB_C and LIB_D (in that order), and LIB_B also depends on LIB_C, then the unified dependency order is: This ensures that both LIB_A , both LIB_B can overwrite LIB_C; and LIB_D still has higher priority than LIB_B because LIB_A (which depends on it) has higher priority than LIB_B. For more information on merging manifests from different project sources/dependencies, see Merge multiple manifest files. Viewing Module Dependencies Some direct dependencies may have dependencies of their own. These are called transient dependencies. Instead of manually declaring each transient dependency, Gradle automatically compiles and adds them for you. Android pluginGradle provides a task that lists the dependencies that Gradle resolves for a given module. For each module, the report also includes dependencies grouped based on build variant, test source set, and classpath. Below is an example report of the app module runtime classpath in the debug build variant and the build classpath of its instrumented set of test sources. debugRuntimeClasspath - runtime/package dependencies +--- :mylibrary (variant: debug) +--- com.google.android.material:material:1.0.0@aar +--- androidx.appcompat:appcompat:1.0. 2 @ aar +--- androidx.constraintlayout:constraintlayout:1.1.3@aar +--- androidx.fragment:fragment:1.0.0@aar +--- androidx.vectordrawable:vectordrawable-animated:1.0.0@aar + - -- androidx.recyclerview:recyclerview:1.0.0@aar +--- androidx.legacy:legacy-support-core-ui:1.0.0@aar ... debugAndroidTest debugAndroidTestCompileClasspath - Compile dependencies +--- androidx . test.ext:junit:1.1.0@aar +--- androidx.test.espresso:espresso-core:3.1.1@aar +--- androidx.test:runner:1.1.1@aar +--- junit :junit:4.12@jar ... To complete the task, do the following: Select View > Tool Windows > Gradle (or click Gradle in the Tool Windows panel). Expand AppName > Tasks > Android and double-click androidDependencies. Once Gradle finishes the job, you should open a Run window to see the output. For more information on dependency management in Gradle, see the Basics of Dependency Management in the Gradle User Guide. Fixing Dependency Resolution Errors When you add multiple dependencies to an application project, these direct and transitive dependencies can cause conflicts. The Android Gradle plugin tries to neatly resolve these conflicts, but some conflicts can cause compile-time or run-time errors. To find out which dependencies are causing errors, review the application's dependency tree and look for dependencies that appear more than once or have conflicting versions. If you cannot easily identify re-addiction,Use the Android Studio UI to find dependencies that contain a duplicate class as follows. Choose Navigation > Class from the menu bar. In the search dialog box that appears, make sure that "Include non-project items" is checked. Enter the class name shown in the build error. Check the results for class-related dependencies. The following sections describe the different types of dependency resolution errors you may encounter and how to resolve them. Fixing Duplicate Class Errors If a class appears more than once on the classpath, you will receive an error message similar to the following: Program type already exists com.example.MyClass This error typically occurs when one of the following conditions occurs: A binary dependency contains a library that your application is also included as a direct dependency. For example, your application declares a direct dependency on library A and library B, but library A already contains library B in its binary form. To resolve this issue, remove the B library as a direct dependency. Your application has a local binary dependency and a remote binary dependency on the same library. To resolve this issue, remove one of the binary dependencies. Resolving Classpath Conflicts When Gradle resolves the compiler classpath, it first resolves the runtime classpath and uses the result to determine which dependency versions to add to the compiler classpath. In other words, the runtime classpath defines the required version numbers for identical child classpath dependencies. Your app's runtime classpath also defines the version numbers that Gradle needs to match dependencies in the app's

APK runtime classpath. The classpath hierarchy is shown in Figure 1. Figure 1. The version numbers of dependencies that appear on multiple classpaths must match according to this hierarchy. A conflict can arise when different versions of the same dependency appear in multiple classpaths, for example if yourcontains a dependency version that uses an implementation dependency configuration, and a library module contains another dependency version that uses a runtimeonly configuration. When resolving runtime and classpath dependencies at build time, Android Gradle 3.3.0 and later attempts to resolve some partial conflicts automatically. For example, if the runtime classpath contains library A version 2.0 and the build classpath contains library A version 1.0, the plugin will automatically update the build classpath dependency to version 2.0 of library A to avoid errors. However, if the runtime classpath contains a version 1.0 library and the build classpath contains a version 2.0 library, the plugin will not downgrade the build classpath to version 1.0 of the library and you will still receive an error message similar to the following : Dependency conflict com .example.library:some-lib:2.0 in the my-library project. The allowed versions of runtime classpath (1.0) and build classpath (2.0) are different. To resolve this issue, do one of the following: Inject the required dependency version into the library module as an API dependency. This means that only your library module declares the dependency, but the application module also has transient access to its API. Alternatively, you can declare the dependency in both modules, but you must ensure that each module uses the same version of the dependency. Consider setting the property project-wide to keep versions of each dependency consistent throughout the project. Applying Custom Build Logic This section contains additional topics that may be useful if you want to extend the Android Gradle plugin or write your own plugin. Publishing Variant Dependencies for Custom Logic A library may have functionality that other projects or subprojects may need. Publishing to a library is the process by which a library is made available to users. Libraries can control whichits consumers have access at compile time and at run time. There are two separate configurations that contain transitive dependencies for each classpath that consumers must use to use the library, as described below: variant_nameApiElements: This configuration contains transitive dependencies that are available to consumers at compile time. variant_nameRuntimeElements: This configuration contains temporary dependencies available to consumers at run time. To learn more about the relationship between the various configurations, go to the Java Library Plugin Configurations section. Custom Dependency Handling Strategies A project can depend on two different versions of the same library, which can cause dependency conflicts. For example, if your project depends on version 1 of module A and version 2 of module B, and module A transitions from version 3 of module B, a dependency version conflict occurs. To resolve this conflict, the Android Gradle plugin uses the following dependency resolution strategy: When a plugin detects that there are different versions of the same module in the dependency graph, it defaults to the highest numbered version. However, this strategy may not always work as intended. To configure a dependency resolution strategy, use the following configurations to resolve specific variant dependencies required by a task: variant_nameCompileClasspath: This configuration contains the compilation class path resolution policy for a specific variant. variant_nameRuntimeClasspath: This configuration contains the resolution policy for the runtime classpath of the specified variant. The Android Gradle plugin includes getters that can be used to access per-variant configuration objects. So you can use the variant API to request dependency resolution, as shown in the example below: android { applicationVariants.all { variant -> // Return build configuration objectsvariant.getCompileConfiguration().resolutionStrategy { // Use the Gradle ResolutionStrategy API // to configure how this variant resolves dependencies. ... } // Returns the configuration objects of the runtime variant. variant.getRuntimeConfiguration().resolutionStrategy { ... } // Returns the configuration of the variant annotation processor. variant.getAnnotationProcessorConfiguration().resolutionStrategy { ... } } } android { applicationVariants.all { // Returns configuration objects for variant compilation. compileConfiguration.resolutionStrategy { // Use the Gradle ResolutionStrategy API // to configure how this option resolves dependencies. ... } // Returns the configuration objects of the runtime variant. runtimeConfiguration.resolutionStrategy { ... } // Returns the configuration variant of the annotation processor. annotationProcessorConfiguration.resolutionStrategy { ... } } } When building an application using AGP 4.0.0 and later, the plug-in contains metadata describing the dependencies of the libraries compiled into your application. When your app loads, the Play Console examines this metadata to alert you to known issues with the SDK and dependencies your app uses and, in some cases, provide feedback on how to fix those issues. The data is compressed, encrypted with Google Play's signing key, and stored in your published app's signature block. For a safe and pleasant user experience, we recommend keeping this dependency. However, if you do not want to share this information, you can opt out by including the following dependenciesInfo block in your module's build.gradle file: android {dependenciesInfo { // Disable dependency metadata when building an APK file. includeInApk = false // Disable dependency metadata when creating an Android app bundle. includeInBundle = false } } For more information on our policies and potential dependency issues, visit our support page on using third-party SDKs in your app. Appendix.