Android Developers

# Create an Android Library

In this document


- Create a library module
- Add your library as a dependency
- Publish non-default variants of your library
- Choose resources to make public
- Development considerations
- Anatomy of an AAR file


An Android library is structurally the same as an Android app module. It can include everything needed to build an app, including source code, resource files, and an Android manifest. However, instead of compiling into an APK that runs on a device, an Android library compiles into an Android Archive (AAR) file that you can use as a dependency for an Android app module. Unlike JAR files, AAR files can contain Android resources and a manifest file, which allows you to bundle in shared resources like layouts and drawables in addition to Java classes and methods.

A library module is useful in the following situations:

- When you're building multiple apps that use some of the same components, such as activities, services, or UI layouts.
- When you're building an app that exists in multiple APK variations, such as a free and paid version and you need the same core components in both.

In either case, simply move the files you want to reuse into a library module then add the library as a dependency for each app module. This page teaches you how to do both.

## Create a library module

To create a new library module in your project, proceed as follows:

1. Click **File > New > New Module**.
2. In the **Create New Module** window that appears, click **Android Library**, then click **Next**.

   There's also an option to create a **Java Library**, which builds a traditional JAR file. While a JAR file is useful for many projects—especially when you want to share code with other platforms—it does not allow you to include

Android resources or manifest files, which is very useful for code reuse in Android projects. So this guide focuses on creating Android libraries.

3. Give your library a name and select a minimum SDK version for the code in the library, then click **Finish**.

Once the Gradle project sync completes, the library module appears in the **Project** panel on the left. If you don't see the new module folder, make sure it's displaying the Android view (https://developer.android.com/studio/projects/index.html#ProjectFiles).

## Convert an app module to a library module

If you have an existing app module with all the code you want to reuse, you can turn it into a library module as follows:

1. Open the `build.gradle` file for the existing app module. At the top, you should see the following:

```
apply plugin: 'com.android.application'
```

2. Change the plugin assignment as shown here:

```
apply plugin: 'com.android.library'
```

3. Click **Sync Project with Gradle Files**.

That's it. The entire structure of the module remains the same, but it now operates as an Android library and the build will now create an AAR file instead of an APK.

# Add your library as a dependency

To use your Android library's code in another app module, proceed as follows:

1. Add the library to your project in one of two ways (if you created the library module within the same project, then it's already there and you can skip this step):
   - Add the compiled AAR (or JAR) file:
     1. Click **File > New Module**.
     2. Click **Import .JAR/.AAR Package** then click **Next**.
     3. Enter the location of the AAR or JAR file then click **Finish**.
   - Import the library module to your project:
     1. Click **File > New > Import Module**.
     2. Enter the location of the library module directory then click **Finish**.

   The library module is copied to your project, so you can actually edit the library code. If you want to maintain a single version of the library code, then this is probably not what you want and you should instead import the

compiled AAR file as described above.

2. Make sure the library is listed at the top of your `settings.gradle` file, as shown here for a library named "my-library-module":

```
include ':app', ':my-library-module'
```

3. Open the app module's `build.gradle` file and add a new line to the `dependencies` block as shown in the following snippet:

```
dependencies {
    compile project(":my-library-module")
}
```

4. Click **Sync Project with Gradle Files**.

In this example above, the `compile` configuration adds the library named `my-library-module` as a build dependency for the entire app module. If you instead want the library only for a specific build variant (https://developer.android.com/studio/build/build-variants.html), then instead of `compile`, use ***`buildVariantName`***`Compile`. For example, if you want to include the library only in your "pro" product flavor, it looks like this:

```
productFlavors {
    pro { ... }
}
dependencies {
    proCompile project(":my-library-module")
}
```

Any code and resources in the Android library is now accessible to your app module, and the library AAR file is bundled into your APK at build time.

However, if you want to share your AAR file separately, you can find it in ***`project-name`***`/`***`module-name`***`/build/outputs/aar/` and you can regenerate it by clicking **Build > Make Project**.

# Publish non-default variants of your library

By default, the library module publishes and exposes only the "release" build variant to other Android projects and modules. That is, if an app module consumes the library as a dependency, Gradle targets only the "release" variant of the library even when it's building a debug version of the app. However, you can tell Gradle to target the "debug" build variant by adding the following to the library's `build.gradle` file:

```
android {
  ...
  // Sets the "debug" build variant as the default variant
  // of the library that Gradle should publish.
  defaultPublishConfig "debug"
}
```

Now Gradle always publishes the "debug" variant of the library to other modules. However, if the library module uses product flavors (https://developer.android.com/studio/build/build-variants.html#product-flavors), you must configure the `defaultPublishConfig` property and specify a build variant by its full configuration name (otherwise, Gradle doesn't publish your library because the traditional "release" and "debug" variants no longer exist). The following sample targets a build variant that combines the "demo" product flavor and "debug" build type:

```
android {
  ...
  defaultPublishConfig "demoDebug"
}
```

Keep in mind, if you are building a release version of your app for publication, you need to change `defaultPublishConfig` to use a release variant of the library. Alternatively, you can tell Gradle to publish and expose all available variants of the library and configure each app variant to use only the one it needs. To tell Gradle to publish all variants of the library, include the following line in the library's `build.gradle` file:

```
android {
  ...
  // Tells Gradle to build all variants of the library. Note that this
  // may increase build times because Gradle must build multiple AARs,
  // instead of only one.
  publishNonDefault true
}
```

> **Note:** When publishing all variants of your library to Maven, the Maven publishing plugin (https://docs.gradle.org/current/userguide/publishing_maven.html) publishes the default variant of the library with each additional variant as an additional artifact. Gradle differentiates each of those artifacts by setting its classifier to the variant name, such as "debug". However, this behavior is not fully supported by the Android plugin for Gradle, and you should either publish a single variant of the library to Maven, or set `publishNonDefault` to `true` only when publishing different variants of your library to local Android modules or projects.

Now you can configure the `dependencies` block in the app's `build.gradle` file to target any of the available variants of the library. The following code snippet in the app module's `build.gradle` file instructs Gradle to use the library's "demoDebug" variant when building the "demoDebug" version of the app and to use the library's "fullRelease" variant when building the "fullRelease" version of the app:

```
android {...}
...

// Creates Gradle dependency configurations (https://docs.gradle.org/current/userguide/dependency_mana
configurations {
  // Initializes placeholder configurations that the Android plugin can use when targeting
  // the corresponding variant of the app.
  demoDebugCompile {}
  fullReleaseCompile {}
  ...
}
```

```
dependencies {
  // If the library configures multiple build variants using product flavors,
  // you must target one of the library's variants using its full configuration name.
  demoDebugCompile project(path: ':my-library-module', configuration: 'demoDebug')
  fullReleaseCompile project(path: ':my-library-module', configuration: 'fullRelease')
  ...
}
```

# Choose resources to make public

All resources in a library default to public. To make all resources implicitly private, you must define at least one specific attribute as public. Resources include all files in your project's `res/` directory, such as images. To prevent users of your library from accessing resources intended only for internal use, you should use this automatic private designation mechanism by declaring one or more public resources.

To declare a public resource, add a `<public>` declaration to your library's `public.xml` file. If you haven't added public resources before, you need to create the `public.xml` file in the `res/values/` directory of your library.

The following example code creates two public string resources with the names `mylib_app_name` and `mylib_public_string`:

```
<resources>
    <public name="mylib_app_name" type="string"/>
    <public name="mylib_public_string" type="string"/>
</resources>
```

You should make public any resources that you want to remain visible to developers using your library. For example, although most of the resources in the v7 appcompat library (https://developer.android.com/topic/libraries/support-library/features.html#v7-appcompat) are private, attributes controlling the Toolbar widget are public to support material design (https://developer.android.com/design/material/index.html).

Implicitly making attributes private not only prevents users of your library from experiencing code completion suggestions from internal library resources but also allows you to rename or remove private resources without breaking clients of your library. Private resources are filtered out of code completion and the theme editor (https://developer.android.com/studio/write/theme-editor.html), and Lint (https://developer.android.com/studio/write/lint.html) warns you when you try to reference a private resource.

# Development considerations

As you develop your library modules and dependent apps, be aware of the following behaviors and limitations.

Once you have added references to library modules to your Android app module, you can set their relative priority. At build time, the libraries are merged with the app one at a time, starting from the lowest priority to the highest.

- **Resource merge conflicts**

The build tools merge resources from a library module with those of a dependent app module. If a given resource ID is defined in both modules, the resource from the app is used.

If conflicts occur between multiple AAR libraries, then the resource from the library listed first in the dependencies list (toward the top of the `dependencies` block) is used.

To avoid resource conflicts for common resource IDs, consider using a prefix or other consistent naming scheme that is unique to the module (or is unique across all project modules).

- **A library module can include a JAR library**

  You can develop a library module that itself includes a JAR library; however you need to manually edit the dependent app modules's build path and add a path to the JAR file.

- **A library module can depend on an external JAR library**

  You can develop a library module that depends on an external library. (for example, the Maps external library). In this case, the dependent app must build against a target that includes the external library (for example, the Google APIs Add-On). Note also that both the library module and the dependent app must declare the external library in their manifest files, in a `<uses– library>` (https://developer.android.com/guide/topics/manifest/uses-library-element.html) element.

- **Library modules cannot include raw assets**

  The tools do not support the use of raw asset files (saved in the `assets/` directory) in a library module. Any asset resources used by an app must be stored in the `assets/` directory of the app module itself.

- **The app module's `minSdkVersion` must be equal to or greater than the version defined by the library**

  A library is compiled as part of the dependent app module, so the APIs used in the library module must be compatible with the platform version that the app module supports.

- **Each library module creates its own R class**

  When you build the dependent app modules, library modules are compiled into an AAR file then added to the app module. Therefore, each library has its own `R` class, named according to the library's package name. The `R` class generated from main module and the library module is created in all the packages that are needed including the main module's package and the libraries' packages.

- **Library modules may include their own ProGuard configuration file**

  You can enable code shrinking on your library by adding a ProGuard (https://developer.android.com/studio/build/shrink-code.html) configuration file to your library that includes its ProGuard directives. The build tools embed this file within the generated AAR file for the library module. When you add the library to an app module, the library's ProGuard file gets appended to the ProGuard configuration file (`proguard.txt`) of the app module.

  By embedding a ProGuard file in your library module, you ensure that app modules that depend on your library do not have to manually update their ProGuard files to use your library. When ProGuard runs on the Android app module, it uses the directives from both the app module and the library so you should not run ProGuard on the library alone.

  To specify the name of your library's configuration file, add it to the `consumerProguardFiles` method, inside the `defaultConfig` block of your library's `build.gradle` file. For example, the following snippet sets `lib–`

`proguard-rules.txt` as the library's ProGuard configuration file:

```
android {
    defaultConfig {
        consumerProguardFiles 'lib-proguard-rules.txt'
    }
    ...
}
```

To ensure that your library's ProGuard rules do not apply unwanted shrinking side effects to app modules, only include rules that disable ProGuard features that do not work with your library. Rules that attempt to aid developers can conflict with the existing code in an app module or its other libraries and therefore should not be included. For example, your library's ProGuard file can specify what code needs to be kept (https://developer.android.com/studio/build/shrink-code.html#keep-code) during an app module's minification.

> **Note**: The Jack toolchain (https://source.android.com/source/jack.html#shrinking_and_obfuscation) provides support for only some shrinking and obfuscation options with ProGuard.

- **Testing a library module is the same as testing an app (https://developer.android.com/studio/test/index.html)**

  The main difference is that the library and its dependencies are automatically included as dependencies of the test APK. This means that the test APK includes not only its own code, but also the library's AAR and all its dependencies. Because there is no separate "app under test," the `androidTest` task installs (and uninstalls) only the test APK.

  When merging multiple manifest files (https://developer.android.com/studio/build/manifest-merge.html), Gradle follows the default priority order and merges the library's manifest into the test APK's main manifest.

# Anatomy of an AAR file

The file extension for an AAR file is `.aar`, and the Maven artifact type should be `aar` as well. The file itself is a `zip` file containing the following mandatory entries:

- `/AndroidManifest.xml`
- `/classes.jar`
- `/res/`
- `/R.txt`

Additionally, an AAR file may include one or more of the following optional entries:

- `/assets/`
- `/libs/`***`name`***`.jar`
- `/jni/`***`abi_name`***`/`***`name`***`.so` (where ***`abi_name`*** is one of the Android supported ABIs (https://developer.android.com/ndk/guides/abis.html#sa))

- `/proguard.txt`
- `/lint.jar`