(https://google.com/racialequity)

Improve code inspection with annotations

Using code inspections tools such as Lint (/studio/write/lint) can help you find problems and improve your code, but inspection tools can only infer so much. Android resource IDs, for example, use an `int` to identify strings, graphics, colors, and other resource types, so inspection tools cannot tell when you have specified a string resource where you should have specified a color. This situation means that your app may render incorrectly or fail to run at all, even if you use code inspection.

Annotations allow you to provide hints to code inspections tools like Lint, to help detect these more subtle code problems. They are added as metadata tags that you attach to variables, parameters, and return values to inspect method return values, passed parameters, local variables, and fields. When used with code inspections tools, annotations can help you detect problems, such as null pointer exceptions and resource type conflicts.

Android supports a variety of annotations through the Annotations Support Library (/topic/libraries/support-library/features#annotations). You can access the library through the `android.support.annotation` (/reference/android/support/annotation/package-summary) package.

Note: If a module has a dependency on an annotation processor, you must use the `annotationProcessor` dependency configuration to add that dependency. To learn more, read Use the annotation processor dependency configuration (/studio/build/gradle-plugin-3-0-0-migration#annotationProcessor_config).

Add annotations to your project

To enable annotations in your project, add the `support-annotations` dependency to your library or app. Any annotations you add then get checked when you run a code inspection or `lint` task.

Add the support annotations library dependency

The Support Annotations library is published on Google's Maven Repository (/studio/build/dependencies#google-maven). To add the Support Anotations library to your project, include the following line in the `dependencies` block of your `build.gradle` file:

```
dependencies {
    implementation 'com.android.support:support-annotations:28.0.0'
}
```

Then, in the toolbar or sync notification that appears, click **Sync Now**.

If you use annotations in your own library module, the annotations are included as part of the Android Archive (AAR) artifact in XML format in the `annotations.zip` file. Adding the `support-annotations` dependency does not introduce a dependency for any downstream users of your library.

Note: If you're using the `appcompat` (/reference/android/support/v7/appcompat/package-summary) library, you do not need to add the `support-annotations` dependency. Because the `appcompat` library already depends on the annotations library, you have access to the annotations.

For a complete list of annotations included in the support repository, either examine the Support Annotations library reference (/reference/android/support/annotation/package-summary) or use the auto-complete feature to display the available options for the `import android.support.annotation.` statement.

Run code inspections

To start a code inspection from Android Studio, which includes validating annotations and automatic Lint checking, select **Analyze** > **Inspect Code** from the menu bar. Android Studio displays conflict messages to flag potential problems where your code conflicts with annotations and to suggest possible resolutions.

You can also enforce annotations by running the `lint` task using the command line (/studio/write/lint#commandline). Although this may be useful for flagging problems with a continuous integration server, note that the `lint` task does not enforce nullness annotations (only Android Studio does). For more information on enabling and running Lint inspections, see Improving Your Code with Lint (/tools/debugging/improving-w-lint).

Note that although annotation conflicts generate warnings, these warnings do not prevent your app from compiling.

Nullness annotations

Add `@Nullable` (/reference/androidx/annotation/Nullable) and `@NonNull` (/reference/androidx/annotation/NonNull) annotations to check the nullness of a given variable, parameter, or return value. The `@Nullable` annotation indicates a variable, parameter, or return value that can be null while `@NonNull` indicates a variable, parameter, or return value that cannot be null.

For example, if a local variable that contains a null value is passed as a parameter to a method with the `@NonNull` annotation attached to that parameter, building the code generates a warning indicating a non-null conflict. On the other hand, attempting to reference the result of a method marked by `@Nullable` without first checking if the result is null generates a nullness warning. You should only use `@Nullable` on a method's return value if every use of the method should be explicitly null-checked.

The following example attaches the `@NonNull` annotation to the `context` and `attrs` parameters to check that the passed parameter values are not null. It also checks that the `onCreateView()` method itself does not return null. Please note that with Kotlin, we do not need to use the `@NonNull` annotation because it will be automatically added to the generated bytecode when we specify a non-nullable type:

KOTLIN (#KOTLIN)JAVA

```
import android.support.annotation.NonNull;
...

    /** Add support for inflating the <fragment> tag. **/
    @NonNull
    @Override
    public View onCreateView(String name, @NonNull Context context,
      @NonNull AttributeSet attrs) {
      ...
      }
...
```

Nullability analysis

Android Studio supports running a nullability analysis to automatically infer and insert nullness annotations in your code. A nullability analysis scans the contracts throughout the method hierarchies in your code to detect:

- Calling methods that can return null
- Methods that should not return null
- Variables, such as fields, local variables, and parameters, that can be null
- Variables, such as fields, local variables, and parameters, that cannot hold a null value

The analysis then automatically inserts the appropriate null annotations in the detected locations.

To run a nullability analysis in Android Studio, select **Analyze** > **Infer Nullity**. Android Studio inserts the Android `@Nullable` (/reference/androidx/annotation/Nullable) and `@NonNull` (/reference/androidx/annotation/NonNull) annotations in detected locations in your code. After running a null analysis, it's good practice to verify the injected annotations.

Note: When adding nullness annotations, autocomplete may suggest the IntelliJ `@Nullable and @NotNull` (https://www.jetbrains.com/help/idea/2020.1/nullable-and-notnull-annotations.html) annotations instead of the Android null annotations and may auto-import the corresponding library. However, the Android Studio Lint checker only looks for the Android null annotations. When verifying your annotations, confirm that your project uses the Android null annotations so the Lint checker can properly notify you during code inspection.

Resource annotations

Validating resource types can be useful because Android references to resources, such as drawable (/guide/topics/resources/drawable-resource) and string (/guide/topics/resources/string-resource) resources, are passed as integers. Code that expects a parameter to reference a specific type of resource, for example Drawables, can be passed the expected reference type of `int`, but actually reference a different type of resource, such as an `R.string` resource.

For example, add `@StringRes` (/reference/androidx/annotation/StringRes) annotations to check that a resource parameter contains an `R.string` reference, as shown here:

KOTLIN (#KOTLIN)JAVA

```
public abstract void setTitle(@StringRes int resId)
```

During code inspection, the annotation generates a warning if an `R.string` reference is not passed in the parameter.

Annotations for the other resource types, such as `@DrawableRes` (/reference/androidx/annotation/DrawableRes), `@DimenRes` (/reference/androidx/annotation/DimenRes), `@ColorRes` (/reference/androidx/annotation/ColorRes), and `@InterpolatorRes` (/reference/androidx/annotation/InterpolatorRes) can be added using the same annotation format and run during the code inspection. If your parameter supports multiple resource types, you can put more than one of these annotations on a given parameter. Use `@AnyRes` (/reference/androidx/annotation/AnyRes) to indicate that the annotated parameter can be any type of `R` resource.

Although you can use `@ColorRes` (/reference/androidx/annotation/ColorRes) to specify that a parameter should be a color resource, a color integer (in the `RRGGBB` or `AARRGGBB` format) is not recognized as a color resource. Instead, use the `@ColorInt` (/reference/androidx/annotation/ColorInt) annotation to indicate that a parameter must be a color integer. The build tools will flag incorrect code that passes a color resource ID such as `android.R.color.black`, rather than a color integer, to annotated methods.

Thread annotations

Thread annotations check if a method is called from a specific type of thread (/guide/components/processes-and-threads). The following thread annotations are supported:

- `@MainThread` (/reference/androidx/annotation/MainThread)
- `@UiThread` (/reference/androidx/annotation/UiThread)
- `@WorkerThread` (/reference/androidx/annotation/WorkerThread)
- `@BinderThread` (/reference/androidx/annotation/BinderThread)
- `@AnyThread` (/reference/androidx/annotation/AnyThread)

Note: The build tools treat the `@MainThread` and `@UiThread` annotations as interchangeable, so you can call `@UiThread` methods from `@MainThread` methods, and vice versa. However, it's possible for a UI thread to be different from the main thread in the case of system apps with multiple views on different threads. Therefore, you should annotate methods associated with an app's view hierarchy with `@UiThread` and annotate only methods associated with an app's lifecycle with `@MainThread`.

If all methods in a class share the same threading requirement, you can add a single thread annotation to the class to verify that all methods in the class are called from the same type

of thread.

A common use of the thread annotation is to validate method overrides in the AsyncTask (/reference/android/os/AsyncTask) class because this class performs background operations and publishes results only on the UI thread.

Value constraint annotations

Use the `@IntRange` (/reference/androidx/annotation/IntRange), `@FloatRange` (/reference/androidx/annotation/FloatRange), and `@Size` (/reference/androidx/annotation/Size) annotations to validate the values of passed parameters. Both `@IntRange` and `@FloatRange` are most useful when applied to parameters for which users are likely to get the range wrong.

The `@IntRange` annotation validates that an integer or long parameter value is within a specified range. The following example ensures that the `alpha` parameter contains an integer value from 0 to 255:

KOTLIN (#KOTLIN)JAVA

```
public void setAlpha(@IntRange(from=0,to=255) int alpha) { ... }
```

The `@FloatRange` annotation checks that a float or double parameter value is within a specified range of floating point values. The following example ensures that the `alpha` parameter contains a float value from 0.0 to 1.0:

KOTLIN (#KOTLIN)JAVA

```
public void setAlpha(@FloatRange(from=0.0, to=1.0) float alpha) {...}
```

The `@Size` annotation checks the size of a collection or array, as well as the length of a string. The `@Size` annotation can be used to verify the following qualities:

- Minimum size (such as `@Size(min=2)`)
- Maximum size (such as `@Size(max=2)`)
- Exact size (such as `@Size(2)`)

- A number of which the size must be a multiple (such as `@Size(multiple=2)`)

For example, `@Size(min=1)` checks if a collection is not empty, and `@Size(3)` validates that an array contains exactly three values. The following example ensures that the `location` array contains at least one element:

KOTLIN (#KOTLIN)**JAVA**

```
void getLocation(View button, @Size(min=1) int[] location) {
    button.getLocationOnScreen(location);
}
```

Permission annotations

Use the `@RequiresPermission` (/reference/androidx/annotation/RequiresPermission) annotation to validate the permissions of the caller of a method. To check for a single permission from a list the valid permissions, use the `anyOf` attribute. To check for a set of permissions, use the `allOf` attribute. The following example annotates the `setWallpaper()` method to ensure that the caller of the method has the `permission.SET_WALLPAPERS` permission:

KOTLIN (#KOTLIN)**JAVA**

```
@RequiresPermission(Manifest.permission.SET_WALLPAPER)
public abstract void setWallpaper(Bitmap bitmap) throws IOException;
```

This example requires the caller of the `copyImageFile()` method to have both read access to external storage and read access to location metadata in the copied image:

KOTLIN (#KOTLIN)**JAVA**

```
@RequiresPermission(allOf = {
    Manifest.permission.READ_EXTERNAL_STORAGE,
    Manifest.permission.ACCESS_MEDIA_LOCATION})
public static final void copyImageFile(String dest, String source) {
    //...
}
```

For permissions on intents, place the permission requirement on the string field that defines the intent action name:

KOTLIN (#KOTLIN)JAVA

```
@RequiresPermission(android.Manifest.permission.BLUETOOTH)
public static final String ACTION_REQUEST_DISCOVERABLE =
            "android.bluetooth.adapter.action.REQUEST_DISCOVERABLE";
```

For permissions on content providers for which you need separate permissions for read and write access, wrap each permission requirement in an `@RequiresPermission.Read` (/reference/androidx/annotation/RequiresPermission.Read) or `@RequiresPermission.Write` (/reference/androidx/annotation/RequiresPermission.Write) annotation:

KOTLIN (#KOTLIN)JAVA

```
@RequiresPermission.Read(@RequiresPermission(READ_HISTORY_BOOKMARKS))
@RequiresPermission.Write(@RequiresPermission(WRITE_HISTORY_BOOKMARKS))
public static final Uri BOOKMARKS_URI = Uri.parse("content://browser/bookmark
```

Indirect permissions

When a permission depends on the specific value supplied to a method's parameter, use `@RequiresPermission` on the parameter itself, without listing the specific permissions. For example, the `startActivity(Intent)` (/reference/android/app/Activity#startActivity(android.content.Intent)) method uses an indirect permission on the intent passed to the method:

KOTLIN (#KOTLIN)JAVA

```
public abstract void startActivity(@RequiresPermission Intent intent, @Nullab
```

When you use indirect permissions, the build tools perform data flow analysis to check if the argument passed into the method has any `@RequiresPermission` annotations. They then enforce any existing annotations from the parameter on the method itself. In the `startActivity(Intent)` example, annotations in the `Intent`

(/reference/android/content/Intent) class cause the resulting warnings on invalid uses of `startActivity(Intent)` when an intent without the appropriate permissions is passed to the method, as shown in figure 1.

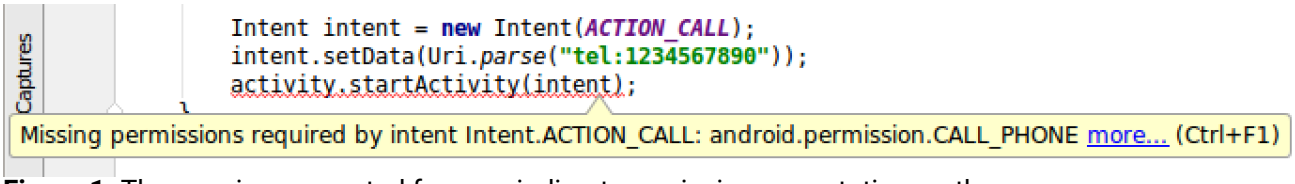

Figure 1. The warning generated from an indirect permissions annotation on the `startActivity(Intent)` method.

The build tools generate the warning on `startActivity(Intent)` from the annotation on the corresponding intent action name in the `Intent` (/reference/android/content/Intent) class:

KOTLIN (#KOTLIN)JAVA

```
@SdkConstant(SdkConstantType.ACTIVITY_INTENT_ACTION)
@RequiresPermission(Manifest.permission.CALL_PHONE)
public static final String ACTION_CALL = "android.intent.action.CALL";
```

If necessary, you can substitute `@RequiresPermission` for `@RequiresPermission.Read` and/or `@RequiresPermission.Write` when annotating a method's parameter. However, for indirect permissions `@RequiresPermission` should not be used in conjunction with either of the read or the write permissions annotations.

Return value annotations

Use the `@CheckResult` (/reference/androidx/annotation/CheckResult) annotation to validate that a method's result or return value is actually used. Instead of annotating every non-void method with `@CheckResult`, add the annotation to clarify the results of potentially confusing methods. For example, new Java developers often mistakenly think that `<String>.trim()` removes whitespace from the original string. Annotating the method with `@CheckResult` flags uses of `<String>.trim()` where the caller does not do anything with the method's return value.

The following example annotates the `checkPermissions()` (/reference/android/content/pm/PackageManager#checkPermission(java.lang.String, java.lang.String)) method to ensure the return value of the method is actually referenced. It also names the `enforcePermission()` (/reference/android/content/ContextWrapper#enforcePermission) method as a method to be suggested to the developer as a replacement:

KOTLIN (#KOTLIN)**JAVA**

```
@CheckResult(suggest="#enforcePermission(String,int,int,String)")
public abstract int checkPermission(@NonNull String permission, int pid, int
```

CallSuper annotations

Use the `@CallSuper` (/reference/androidx/annotation/CallSuper) annotation to validate that an overriding method calls the super implementation of the method. The following example annotates the `onCreate()` method to ensure that any overriding method implementations call `super.onCreate()`:

KOTLIN (#KOTLIN)**JAVA**

```
@CallSuper
protected void onCreate(Bundle savedInstanceState) {
}
```

Typedef annotations

Use the `@IntDef` (/reference/androidx/annotation/IntDef) and `@StringDef` (/reference/androidx/annotation/StringDef) annotations so you can create enumerated annotations of integer and string sets to validate other types of code references. Typedef annotations ensure that a particular parameter, return value, or field references a specific set of constants. They also enable code completion to automatically offer the allowed constants.

Typedef annotations use `@interface` to declare the new enumerated annotation type. The `@IntDef` and `@StringDef` annotations, along with `@Retention`, annotate the new annotation and are necessary in order to define the enumerated type. The `@Retention(RetentionPolicy.SOURCE)` annotation tells the compiler not to store the enumerated annotation data in the `.class` file.

The following example illustrates the steps to create an annotation that ensures a value passed as a method parameter references one of the defined constants:

KOTLIN (#KOTLIN)JAVA

```
import android.support.annotation.IntDef;
//...
public abstract class ActionBar {
    //...
    // Define the list of accepted constants and declare the NavigationMode a
    @Retention(RetentionPolicy.SOURCE)
    @IntDef({NAVIGATION_MODE_STANDARD, NAVIGATION_MODE_LIST, NAVIGATION_MODE_
    public @interface NavigationMode {}

    // Declare the constants
    public static final int NAVIGATION_MODE_STANDARD = 0;
    public static final int NAVIGATION_MODE_LIST = 1;
    public static final int NAVIGATION_MODE_TABS = 2;

    // Decorate the target methods with the annotation
    @NavigationMode
    public abstract int getNavigationMode();

    // Attach the annotation
    public abstract void setNavigationMode(@NavigationMode int mode);
}
```

When you build this code, a warning is generated if the `mode` parameter does not reference one of the defined constants (`NAVIGATION_MODE_STANDARD`, `NAVIGATION_MODE_LIST`, or `NAVIGATION_MODE_TABS`).

You also can combine `@IntDef` (/reference/androidx/annotation/IntDef) and `@IntRange` (/reference/androidx/annotation/IntRange) to indicate that an integer can be either a given set of constants or a value within a range.

Enable combining constants with flags

If users can combine the allowed constants with a flag (such as `|`, `&`, `^`, and so on), you can define an annotation with a `flag` attribute to check if a parameter or return value references a valid pattern. The following example creates the `DisplayOptions` annotation with a list of valid `DISPLAY_` constants:

KOTLIN (#KOTLIN)JAVA

```
import android.support.annotation.IntDef;
...

@IntDef(flag=true, value={
        DISPLAY_USE_LOGO,
        DISPLAY_SHOW_HOME,
        DISPLAY_HOME_AS_UP,
        DISPLAY_SHOW_TITLE,
        DISPLAY_SHOW_CUSTOM
})
@Retention(RetentionPolicy.SOURCE)
public @interface DisplayOptions {}

...
```

When you build code with an annotation flag, a warning is generated if the decorated parameter or return value does not reference a valid pattern.

Keep annotation

The `@Keep` (/reference/androidx/annotation/Keep) annotation ensures that an annotated class or method is not removed when the code is minified at build time. This annotation is typically added to methods and classes that are accessed through reflection to prevent the compiler from treating the code as unused.

Caution: The classes and methods that you annotate using `@Keep` always appear in your app's APK, even if you never reference these classes and methods within your app's logic.

To keep your app's size small, consider whether it's necessary to preserve each `@Keep` annotation in your app. If you use reflection to access an annotated class or method, use an `-if` (https://www.guardsquare.com/en/products/proguard/manual/usage#if) conditional in your ProGuard rules, specifying the class that makes the reflection calls.

For more information about how to minify your code and specify which code should not be removed, see Shrink Your Code and Resources (/studio/build/shrink-code).

Code visibility annotations

Use the following annotations to denote the visibility of specific portions of code, such as methods, classes, fields, or packages.

Make visible for testing

The `@VisibleForTesting` (/reference/androidx/annotation/VisibleForTesting) annotation indicates that an annotated method is more visible than normally necessary to make the method testable. This annotation has an optional `otherwise` argument that lets you designate what the visibility of the method should have been if not for the need to make it visible for testing. Lint uses the `otherwise` argument to enforce the intended visibility.

In the following example, `myMethod()` is normally `private`, but it is package-private for tests. With the following `VisibleForTesting.PRIVATE` designation, lint displays a message if this method is called from outside the context allowed by `private` access, such as from a different compilation unit.

KOTLIN (#KOTLIN)**JAVA**

```
@VisibleForTesting(otherwise = VisibleForTesting.PRIVATE)
void myMethod() { ... }
```

You can also specify `@VisibleForTesting(otherwise = VisibleForTesting.NONE)` to indicate that a method exists only for testing. This form is the same as using `@RestrictTo(TESTS)`. They both perform the same lint check.

Restrict an API

The `@RestrictTo` (/reference/androidx/annotation/RestrictTo) annotation indicates that access to the annotated API (package, class, or method) is limited as follows.

Subclasses

Use the annotation form `@RestrictTo(RestrictTo.Scope.SUBCLASSES)` to restrict API access to subclasses only.

Only classes that extend the annotated class can access this API. The Java `protected` modifier is not restrictive enough because it allows access from unrelated classes within the same package. Also, there are cases when you want to leave a method `public` for future flexibility because you can never make a previously `protected` and overridden

method `public`, but you want to provide a hint that the class is intended for usages within the class or from subclasses, only.

Libraries

Use the annotation form `@RestrictTo(RestrictTo.Scope.GROUP_ID)` to restrict API access to your libraries only.

Only your library code can access the annotated API. This allows you to not only organize your code in whatever package hierarchy you want, but to also share the code among a group of related libraries. This option is already available to the support libraries that have a lot of implementation code that is not meant for external use, but that has to be `public` to share it across the various complementary support libraries.

Note: The Android suport library (/topic/libraries/support-library) classes and packages are now annotated with `@RestrictTo(GROUP_ID)`, which means that if you accidentally use these implementation classes, lint warns you that this is discouraged.

Testing

Use the annotation form `@RestrictTo(RestrictTo.Scope.TESTS)` to prevent other developers from accessing your testing APIs.

Only testing code can access the annotated API. This prevents other developers from using APIs for development that you intend for testing purposes only.

Last updated 2020-10-12 UTC.