(https://google.com/racialequity)

Improve your code with lint checks

In addition to ensuring your app meets its functional requirements by building tests, it's important that you also ensure your code has no structural problems by running the code through lint. The lint tool helps find poorly structured code that can impact the reliability and efficiency of your Android apps and make your code harder to maintain.

For example, if your XML resource files contain unused namespaces, this takes up space and incurs unnecessary processing. Other structural issues, such as use of deprecated elements or API calls that are not supported by the target API versions, might lead to code failing to run correctly. Lint can help you clean up these issues.

To further improve linting performance, you should also add annotations to your code (/studio/write/annotations).

Overview

Android Studio provides a code scanning tool called lint that can help you to identify and correct problems with the structural quality of your code without your having to execute the app or write test cases. Each problem detected by the tool is reported with a description message and a severity level, so that you can quickly prioritize the critical improvements that need to be made. Also, you can lower the severity level of a problem to ignore issues that are not relevant to your project, or raise the severity level to highlight specific problems.

The lint tool checks your Android project source files for potential bugs and optimization improvements for correctness, security, performance, usability, accessibility, and internationalization. When using Android Studio, configured lint and IDE inspections run whenever you build your app. However, you can manually run inspections (#manuallyRunInspections) or run lint from the command line (#commandline).

Note: When your code is compiled in Android Studio, additional IntelliJ code inspections (https://www.jetbrains.com/help/idea/2020.1/code-inspection.html) run to streamline code review.

Figure 1 shows how the lint tool processes the application source files.

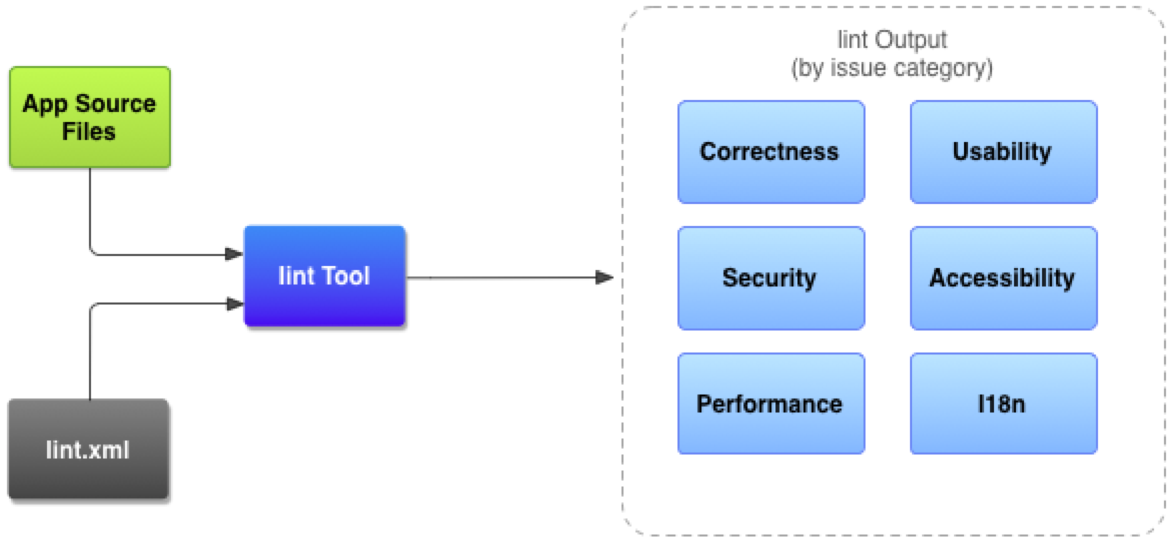


Figure 1. Code scanning workflow with the lint tool

Application source files

The source files consist of files that make up your Android project, including Java, Kotlin, and XML files, icons, and ProGuard configuration files.

The `lint.xml` file

A configuration file that you can use to specify any lint checks that you want to exclude and to customize problem severity levels.

The lint tool

A static code scanning tool that you can run on your Android project either from the command line or in Android Studio (see Manually run inspections (#manuallyRunInspections)). The lint tool checks for structural code problems that could affect the quality and performance of your Android application. It is strongly recommended that you correct any errors that lint detects before publishing your application.

Results of lint checking

You can view the results from lint either in the console or in the **Inspection Results** window in Android Studio. See Manually run inspections (#manuallyRunInspections).

Run lint from the command line

If you're using Android Studio or Gradle, you can use the Gradle wrapper (https://docs.gradle.org/current/userguide/gradle_wrapper.html) to invoke the `lint` task for your project by entering one of the following commands from the root directory of your project:

- On Windows:

  ```
  gradlew lint
  ```

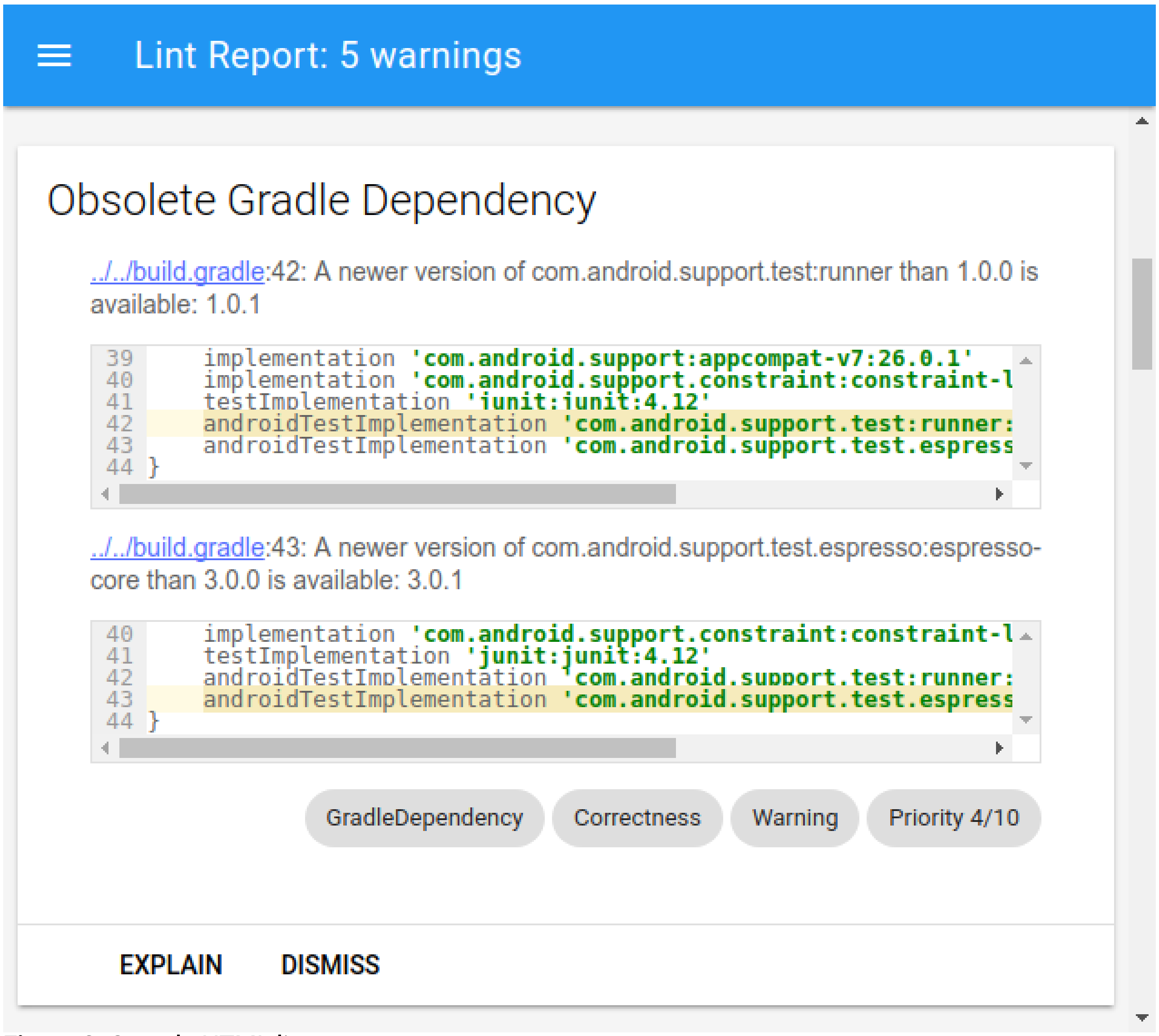
- On Linux or Mac:

  ```
  ./gradlew lint
  ```

You should see output similar to the following:

```
> Task :app:lint
Ran lint on variant release: 5 issues found
Ran lint on variant debug: 5 issues found
Wrote HTML report to file:<path-to-project>/app/build/reports/lint-results.htr
Wrote XML report to file:<path-to-project>/app/build/reports/lint-results.xml
```

When the lint tool completes its checks, it provides paths to the XML and HTML versions of the lint report. You can then navigate to the HTML report and open it in your browser, as shown in figure 2.


Lint Report: 5 warnings

Obsolete Gradle Dependency

../../build.gradle:42: A newer version of com.android.support.test:runner than 1.0.0 is available: 1.0.1

```
    implementation 'com.android.support:appcompat-v7:26.0.1'
    implementation 'com.android.support.constraint:constraint-l
    testImplementation 'junit:junit:4.12'
    androidTestImplementation 'com.android.support.test:runner:
    androidTestImplementation 'com.android.support.test.espress
}
```

../../build.gradle:43: A newer version of com.android.support.test.espresso:espresso-core than 3.0.0 is available: 3.0.1

```
    implementation 'com.android.support.constraint:constraint-l
    testImplementation 'junit:junit:4.12'
    androidTestImplementation 'com.android.support.test:runner:
    androidTestImplementation 'com.android.support.test.espress
}
```

Figure 2. Sample HTML lint report

If your project includes build variants (/studio/build/build-variants), and you instead want to run the `lint` task for only a specific build variant, you must capitalize the variant name and prefix it with `lint`.

```
gradlew lintDebug
```

To learn more about running Gradle tasks from the command line, read Build Your App from the Command Line (/studio/build/building-cmdline).

Run lint using the standalone tool

If you're not using Android Studio or Gradle, you can use the standalone lint tool after you install the Android SDK Tools (/studio/command-line#tools-sdk) from the SDK Manager

(/studio/command-line/sdkmanager). You can then locate the lint tool in the *android_sdk*/tools/ directory.

To run lint against a list of files in a project directory, use the following command:

```
lint [flags] <project directory>
```

For example, you can issue the following command to scan the files under the `myproject` directory and its subdirectories. The issue ID `MissingPrefix` tells lint to only scan for XML attributes that are missing the Android namespace prefix.

```
lint --check MissingPrefix myproject
```

To see the full list of flags and command-line arguments supported by the tool, use the following command:

```
lint --help
```

The following example shows the console output when the lint command is run against a project called Earthquake.

```
$ lint Earthquake

Scanning Earthquake: .......................................................
Scanning Earthquake (Phase 2): .......
AndroidManifest.xml:23: Warning: <uses-sdk> tag appears after <application> ta
  <uses-sdk android:minSdkVersion="7" />
  ^
AndroidManifest.xml:23: Warning: <uses-sdk> tag should specify a target API le
  <uses-sdk android:minSdkVersion="7" />
  ^
res/layout/preferences.xml: Warning: The resource R.layout.preferences appears
res: Warning: Missing density variation folders in res: drawable-xhdpi [IconMi
0 errors, 4 warnings
```

The output above lists four warnings and no errors: three warnings (`ManifestOrder`, `UsesMinSdkAttributes`, and `UnusedResources`) in the project's `AndroidManifest.xml` file, and one warning (`IconMissingDensityFolder`) in the `Preferences.xml` layout file.

Configure lint to suppress warnings

By default when you run a lint scan, the tool checks for all issues that lint supports. You can also restrict the issues for lint to check and assign the severity level for those issues. For example, you can suppress lint checking for specific issues that are not relevant to your project and, you can configure lint to report non-critical issues at a lower severity level.

You can configure lint checking for different levels:

- Globally (entire project)
- Project module
- Production module
- Test module
- Open files
- Class hierarchy
- Version Control System (VCS) scopes

Configure lint in Android Studio

The built-in lint tool checks your code while you're using Android Studio. You can view warnings and errors in two ways:

- As pop-up text in the Code Editor. When lint finds a problem, it highlights the problematic code in yellow, or for more serious issues, it underlines the code in red.
- In the lint **Inspection Results** window after you click **Analyze > Inspect Code**. See Manually run inspections (#manuallyRunInspections).

Configure the lint file

You can specify your lint checking preferences in the `lint.xml` file. If you are creating this file manually, place it in the root directory of your Android project.

The `lint.xml` file consists of an enclosing `<lint>` parent tag that contains one or more children `<issue>` elements. Lint defines a unique `id` attribute value for each `<issue>`.

```
<?xml version="1.0" encoding="UTF-8"?>
    <lint>
```

```
        <!-- list of issues to configure -->
</lint>
```

You can change an issue's severity level or disable lint checking for the issue by setting the severity attribute in the `<issue>` tag.

Tip: For a full list of lint-supported issues and their corresponding issue IDs, run the `lint --list` command.

Sample lint.xml file

The following example shows the contents of a `lint.xml` file.

```
<?xml version="1.0" encoding="UTF-8"?>
<lint>
    <!-- Disable the given check in this project -->
    <issue id="IconMissingDensityFolder" severity="ignore" />

    <!-- Ignore the ObsoleteLayoutParam issue in the specified files -->
    <issue id="ObsoleteLayoutParam">
        <ignore path="res/layout/activation.xml" />
        <ignore path="res/layout-xlarge/activation.xml" />
    </issue>

    <!-- Ignore the UselessLeaf issue in the specified file -->
    <issue id="UselessLeaf">
        <ignore path="res/layout/main.xml" />
    </issue>

    <!-- Change the severity of hardcoded strings to "error" -->
    <issue id="HardcodedText" severity="error" />
</lint>
```

Configure lint checking for Java, Kotlin, and XML source files

You can disable lint from checking your Java, Kotlin, and XML source files.

Tip: You can manage the lint checking feature for your Java, Kotlin, or XML source files in the **Default Preferences** dialog. Select **File > Other Settings** > **Default Settings**, and then in the left pane of the **Default Preferences** dialog, select **Editor > Inspections**.

Configuring lint checking in Java or Kotlin

To disable lint checking specifically for a class or method in your Android project, add the `@SuppressLint` annotation to that code.

The following example shows how you can turn off lint checking for the `NewApi` issue in the `onCreate` method. The lint tool continues to check for the `NewApi` issue in other methods of this class.

KOTLIN (#KOTLIN)JAVA

```
@SuppressLint("NewApi")
@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);
```

The following example shows how to turn off lint checking for the `ParserError` issue in the `FeedProvider` class:

KOTLIN (#KOTLIN)JAVA

```
@SuppressLint("ParserError")
public class FeedProvider extends ContentProvider {
```

To suppress checking for all lint issues in the file, use the `all` keyword, like this:

KOTLIN (#KOTLIN)JAVA

```
@SuppressLint("all")
```

Configuring lint checking in XML

You can use the `tools:ignore` attribute to disable lint checking for specific sections of your XML files. Put the following namespace value in the `lint.xml` file so the lint tool recognizes the attribute:

```
namespace xmlns:tools="http://schemas.android.com/tools"
```

The following example shows how you can turn off lint checking for the `UnusedResources` issue in the `<LinearLayout>` element of an XML layout file. The `ignore` attribute is inherited by the children elements of the parent element in which the attribute is declared. In this example, the lint check is also disabled for the child `<TextView>` element.

```
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    tools:ignore="UnusedResources" >

    <TextView
        android:text="@string/auto_update_prompt" />
</LinearLayout>
```

To disable more than one issue, list the issues to disable in a comma-separated string. For example:

```
tools:ignore="NewApi,StringFormatInvalid"
```

To suppress checking for all lint issues in the XML element, use the `all` keyword, like this:

```
tools:ignore="all"
```

Configure lint options with Gradle

The Android plugin for Gradle allows you to configure certain lint options, such as which checks to run or ignore, using the `lintOptions {}` (https://google.github.io/android-gradle-dsl/current/com.android.build.gradle.internal.dsl.LintOptions.html#com.android.build.gradle.internal.dsl.LintOptions)
block in your module-level `build.gradle` file. The following code snippet shows you some of the properties you can configure:

```
android {
  ...
  lintOptions {
    // Turns off checks for the issue IDs you specify.
    disable 'TypographyFractions','TypographyQuotes'
    // Turns on checks for the issue IDs you specify. These checks are in
    // addition to the default lint checks.
    enable 'RtlHardcoded','RtlCompat', 'RtlEnabled'
    // To enable checks for only a subset of issue IDs and ignore all others,
    // list the issue IDs with the 'check' property instead. This property ov
    // any issue IDs you enable or disable using the properties above.
    check 'NewApi', 'InlinedApi'
    // If set to true, turns off analysis progress reporting by lint.
    quiet true
    // if set to true (default), stops the build if errors are found.
    abortOnError false
    // if true, only report errors.
    ignoreWarnings true
  }
}
...
```

Create warnings baseline

You can take a snapshot of your project's current set of warnings, and then use the snapshot as a baseline for future inspection runs so that only new issues are reported. The baseline snapshot lets you start using lint to fail the build without having to go back and address all existing issues first.

To create a baseline snapshot, modify your project's `build.gradle` file as follows.

```
android {
  lintOptions {
    baseline file("lint-baseline.xml")
  }
}
```

When you first add this line, the `lint-baseline.xml` file is created to establish your baseline. From then on, the tools only read the file to determine the baseline. If you want to create a new baseline, manually delete the file and run lint again to recreate it.

Then, run lint from the IDE (**Analyze > Inspect Code**) or from the command line as follows. The output prints the location of the `lint-baseline.xml` file. The file location for your setup might be different from what is shown here.

```
$ ./gradlew lintDebug

...

Wrote XML report to file:///app/lint-baseline.xml
Created baseline file /app/lint-baseline.xml
```

Running `lint` records all of the current issues in the `lint-baseline.xml` file. The set of current issues is called the *baseline*, and you can check the `lint-baseline.xml` file into version control if you want to share it with others.

Customize the baseline

If you want to add some issue types to the baseline, but not all of them, you can specify the issues to add by editing your project's `build.gradle`, as follows.

```
android {
  lintOptions {
    check 'NewApi', 'HandlerLeak'
    baseline file("lint-baseline.xml")
   }
}
```

After you create the baseline, if you add any new warnings to the codebase, lint lists only the newly introduced bugs.

Baseline warning

When baselines are in effect, you get an informational warning that tells you that one or more issues were filtered out because they were already listed in the baseline. The reason for this warning is to help you remember that you have configured a baseline, because ideally, you would want to fix all of the issues at some point.

This informational warning does not only tell you the exact number of errors and warnings that were filtered out, it also keeps track of issues that are not reported anymore. This information lets you know if you have actually fixed issues, so you can optionally re-create the baseline to prevent the error from coming back undetected.

Note: Baselines are enabled when you run inspections in batch mode in the IDE, but they are ignored for the in-editor checks that run in the background when you are editing a file. The reason is that baselines are intended for the case where a codebase has a massive number of existing warnings, but you do want to fix issues locally while you touch the code.

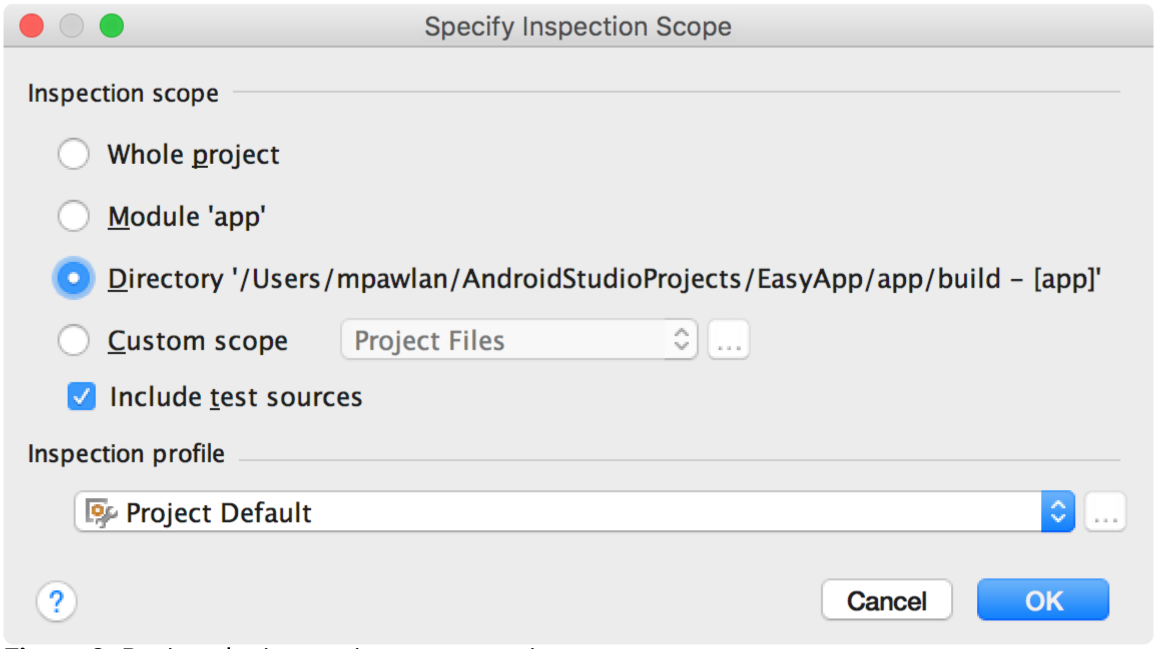
Manually run inspections

You can manually run configured lint and other IDE inspections by selecting **Analyze > Inspect Code**. The results of the inspection appear in the **Inspection Results** window.

Set the inspection scope and profile

Select the files you want to analyze (inspection scope) and the inspections you want to run (inspection profile), as follows:

1. In the **Android** view, open your project and select the project, a folder, or a file that you want to analyze.
2. From the menu bar, select **Analyze > Inspect Code**.
3. In the **Specify Inspection Scope** dialog, review the settings.


Specify Inspection Scope

Inspection scope

- ○ Whole project
- ○ Module 'app'
- ● Directory '/Users/mpawlan/AndroidStudioProjects/EasyApp/app/build – [app]'
- ○ Custom scope: Project Files
- ☑ Include test sources

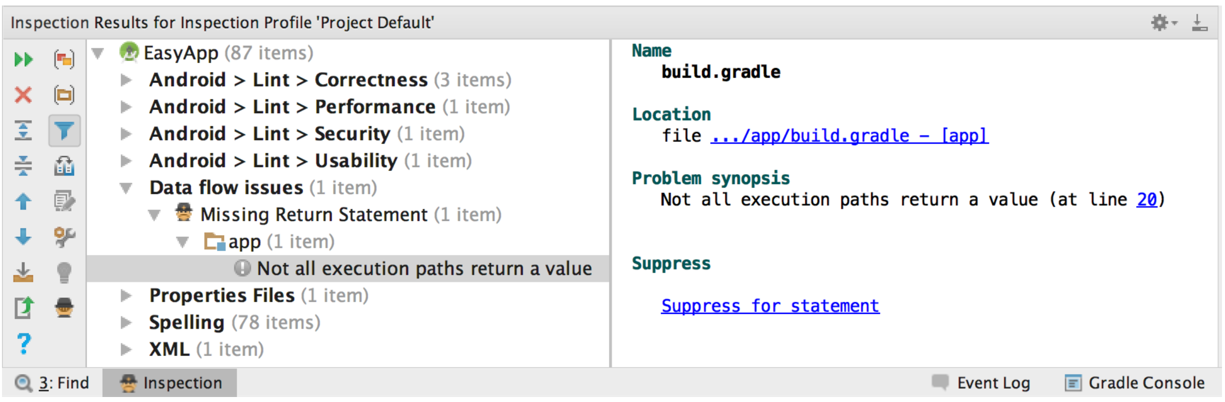
Inspection profile: Project Default

Figure 3. Review the inspection scope settings

The combination of options that appear in the **Specify Inspection Scope** dialog varies depending on whether you selected a project, folder, or file. You can change what to inspect by selecting one of the other radio buttons. See Specify inspection scope (https://www.jetbrains.com/help/idea/2020.1/specify-inspection-scope-dialog.html) dialog for a description of all of the possible fields on the **Specify Inspection Scope** dialog.

- When you select one project, file, or directory, the **Specify Inspection Scope** dialog displays the path to the **Project**, **File**, or **Directory** you selected.
- When you select more than one project, file, or directory, the **Specify Inspection Scope** dialog displays a checked radio button for **Selected files**.

4. Under **Inspection profile**, keep the default profile (**Project Default**).
5. Click **OK** to run the inspection. Figure 4 shows lint and other IDE inspection results from the **Inspect Code** run:


Inspection Results for Inspection Profile 'Project Default'

- EasyApp (87 items)
  - Android > Lint > Correctness (3 items)
  - Android > Lint > Performance (1 item)
  - Android > Lint > Security (1 item)
  - Android > Lint > Usability (1 item)
  - Data flow issues (1 item)
    - Missing Return Statement (1 item)
      - app (1 item)
        - Not all execution paths return a value
  - Properties Files (1 item)
  - Spelling (78 items)
  - XML (1 item)

```
Name
   build.gradle

Location
   file .../app/build.gradle - [app]

Problem synopsis
   Not all execution paths return a value (at line 20)

Suppress

   Suppress for statement
```

Figure 4. Select an issue to see its resolution

6. In the left pane tree view, view the inspection results by expanding and selecting error categories, types, and issues.
   The right pane displays the inspection report for the selected error category, type, or issue and provides the name and location of the error. Where applicable, the inspection report displays other information such as a problem synopsis to help you correct the problem.
7. In the left pane tree view, right-click a category, type, or issue to display the context menu.
   Depending on the context, you can do all or some of the following: jump to source, exclude and include selected items, suppress problems, edit settings, manage inspection alerts, and rerun an inspection.

For descriptions of the left-side Toolbar buttons, context menu items, and inspection report fields, see Inspection Tool Window (https://www.jetbrains.com/help/idea/2020.1/inspection-tool-window.html).

Use a custom scope

You can use one of the custom scopes provided in Android Studio, as follows:

1. In the **Specify Inspection Scope** dialog, click **Custom scope**.
2. Click the **Custom scope** drop-down list to display your options.

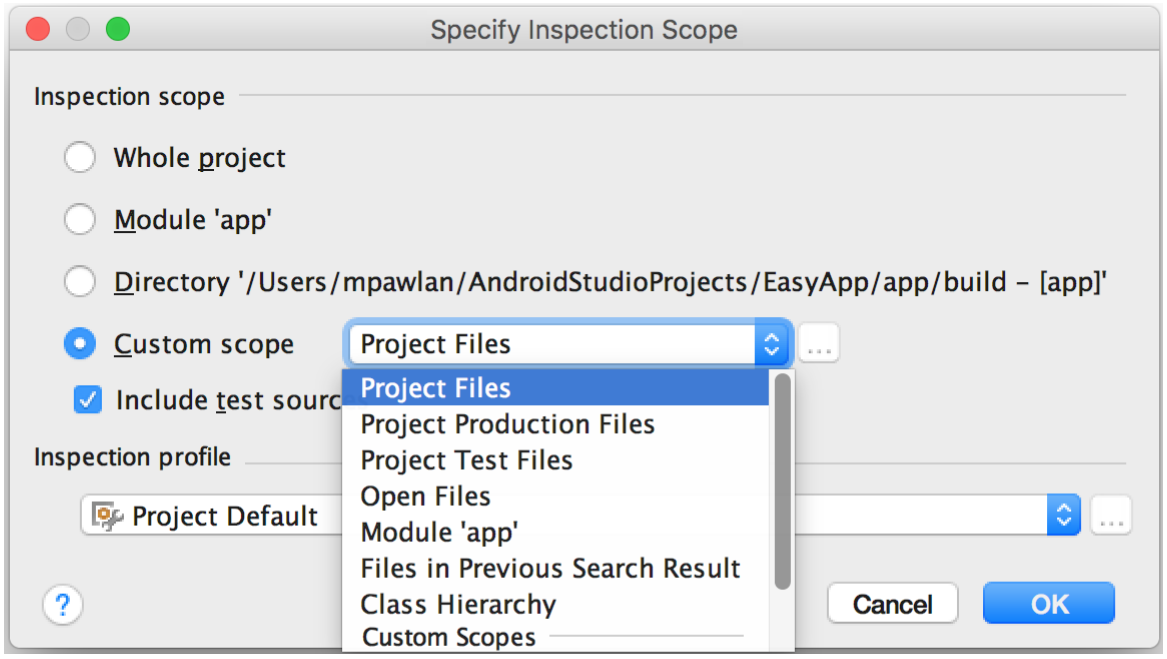


Figure 5. Select the custom scope you want to use

- **Project Files:** All of the files in the current project.
- **Project Production Files:** Only the production files in the current project.
- **Project Test Files:** Only the test files in the current project. See Test types and location (/studio/test#test_types_and_location).
- **Open Files:** Only the files you have open in the current project.
- **Module <your-module>:** Only the files in the corresponding module folder in your current project.
- **Current File:** Only the current file in your current project. Appears when you have a file or folder selected.
- **Class Hierarchy:** When you select this one and click **OK**, a dialog appears with all of the classes in the current project. Use the **Search by Name** field in the dialog to filter and select the classes to inspect. If you do not filter the classes list, code inspection inspects all of the classes.

3. Click **OK.**

Create a custom scope

When you want to inspect a selection of files and directories that is not covered by any of the currently available custom scopes, you can create a custom scope.

1. In the **Specify Inspection Scope** dialog, select **Custom scope**.
2. Click the three dots after the **Custom Scope** drop-down list.

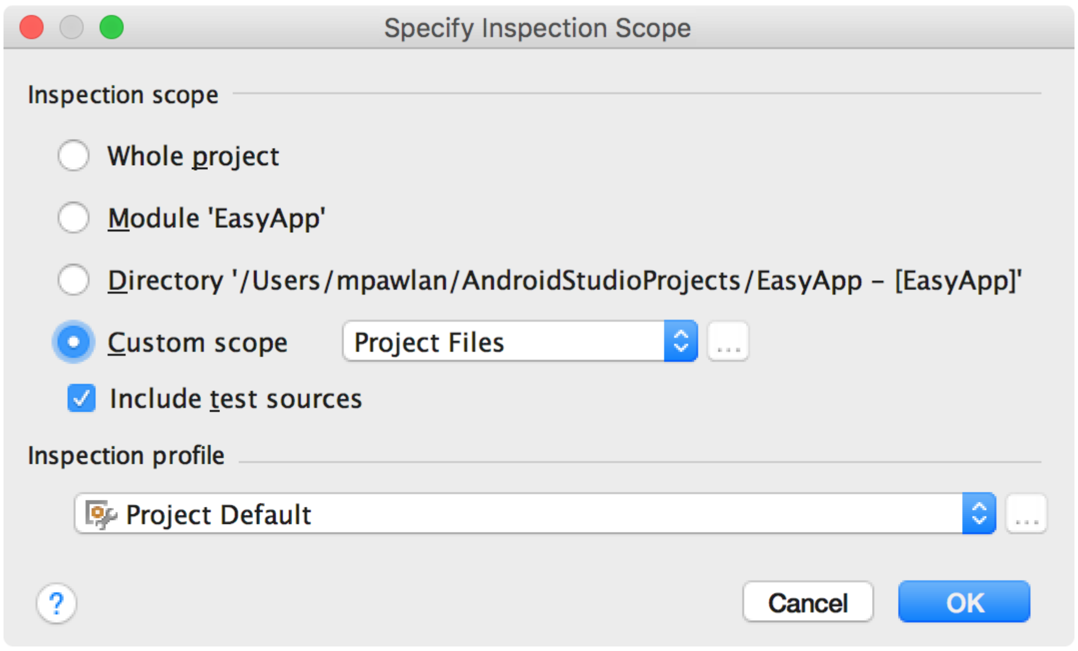


Figure 6. Specify Inspection Scope dialog

The **Scopes** dialog appears.

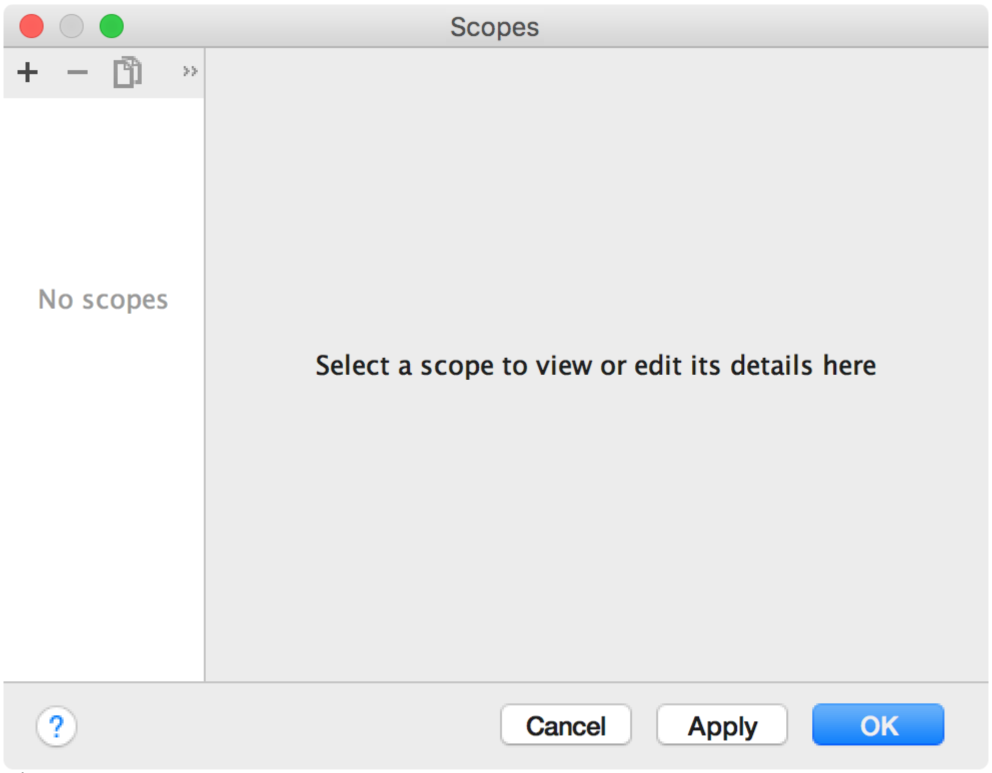
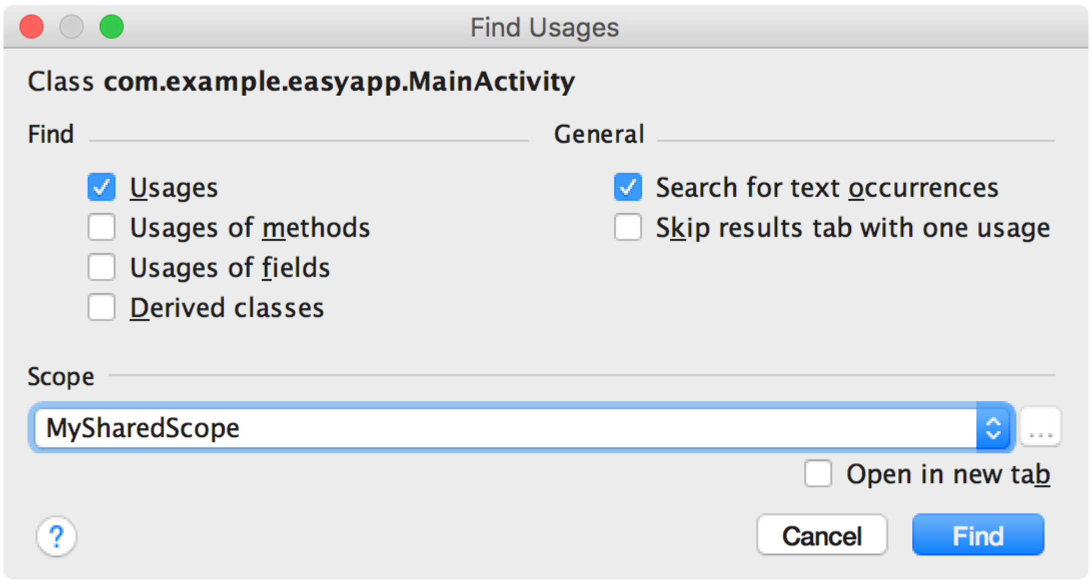


Figure 7. Create a custom scope

3. Click **Add**
   ➕
   to define a new scope.
4. In the resulting **Add Scope** drop-down list, select **Local**.

   Both the local and shared scopes are used within the project for the **Inspect Code** feature. A **Shared** scope can also be used with other project features that have a scope field. For example, when you click **Edit Settings**

   to change the settings for **Find Usages**, the resulting dialog has a **Scope** field where you can select a shared scope.

   Find Usages

   Class **com.example.easyapp.MainActivity**

   Find
   - [x] Usages
   - [ ] Usages of methods
   - [ ] Usages of fields
   - [ ] Derived classes

   General
   - [x] Search for text occurrences
   - [ ] Skip results tab with one usage

   Scope: MySharedScope

   - [ ] Open in new tab

   Figure 8. Select a shared scope from the **Find Usages** dialog

5. Give the scope a name and click **OK**.

   The right pane of the **Scopes** dialog populates with options that enable you to define the custom scope.
6. From the drop-down list, select **Project**.

   A list of available projects appears.

★ **Note:** You can create the custom scope for projects or packages. The steps are the same either way.

7. Expand the project folders, select what you want to add to the custom scope, and click one of the buttons on the right.

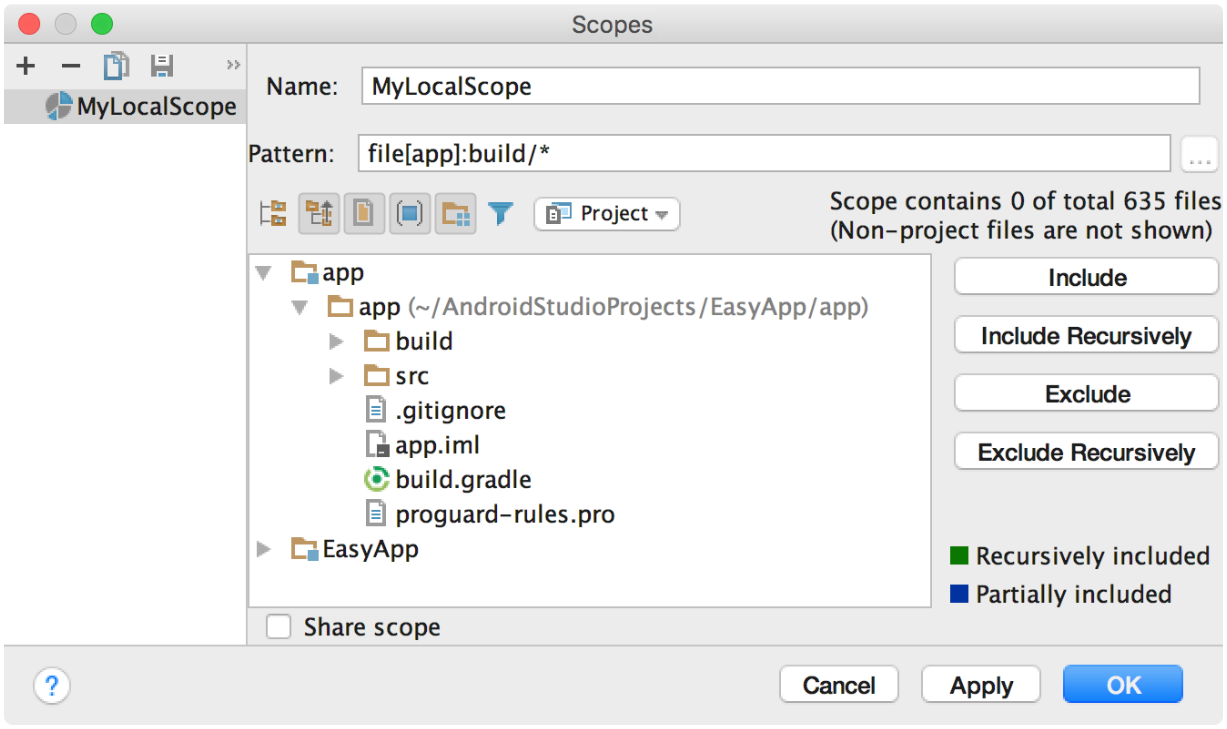


Figure 9. Define a custom scope

- **Include**: Include this folder and its files, but do not include any of its subfolders.
- **Include Recursively**: Include this folder and all of its files and subfolders and their files.
- **Exclude**: Exclude this folder and its files, but do not exclude any of its subfolders.
- **Exclude Recursively**: Exclude ths folder and all of its files and subfolders and their files.

Figure 10 shows that the **main** folder is included, and that the **java** folder is included recursively. The blue indicates partially included folders and the green indicates recursively included folders and files.

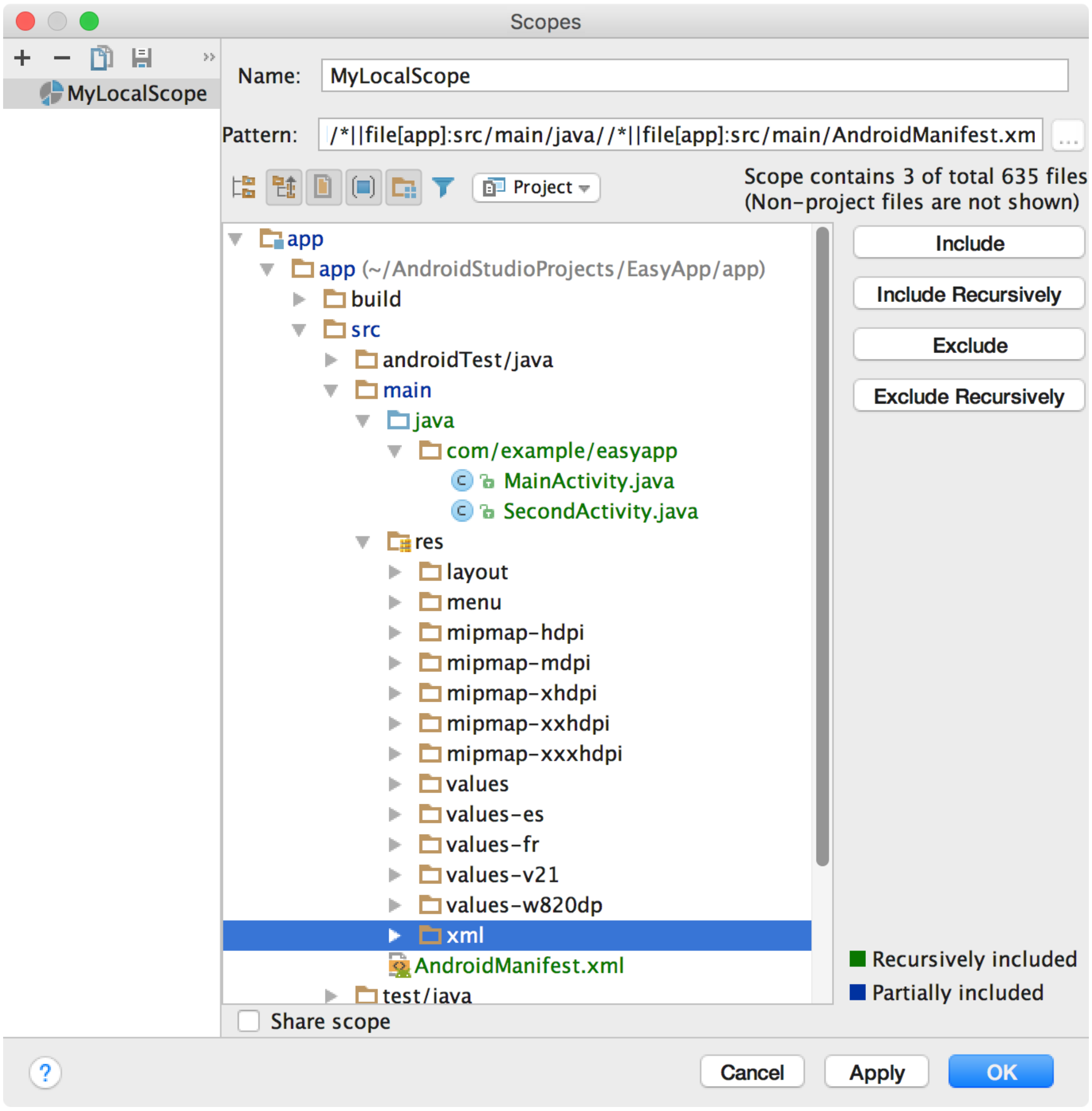


Figure 10. Example pattern for a custom scope

- If you select the **java** folder and click **Exclude Recursively**, the green highlighting goes away on the **java** folder and all folders and files under it.
- If you instead select the green highlighted **MainActivity.java** file and click Exclude, **MainActivity.java** is no longer green highlighted but everything else under the **java** folder is green highlighted.

8. Click **OK**. The custom scope appears at the bottom of the drop-down list.

Review and edit inspection profiles

Android Studio comes with a selection of lint and other inspection profiles that are updated through Android updates. You can use these profiles as is or edit their names, descriptions, severities, and scopes. You can also activate and deactivate entire groups of profiles or individual profiles within a group.

To access the **Inspections** dialog:

1. Select **Analyze > Inspect Code.**
2. In the **Specify Scope** dialog under **Inspection Profile**, click **More**.

   The **Inspections** dialog appears with a list of the supported inspections and their descriptions.

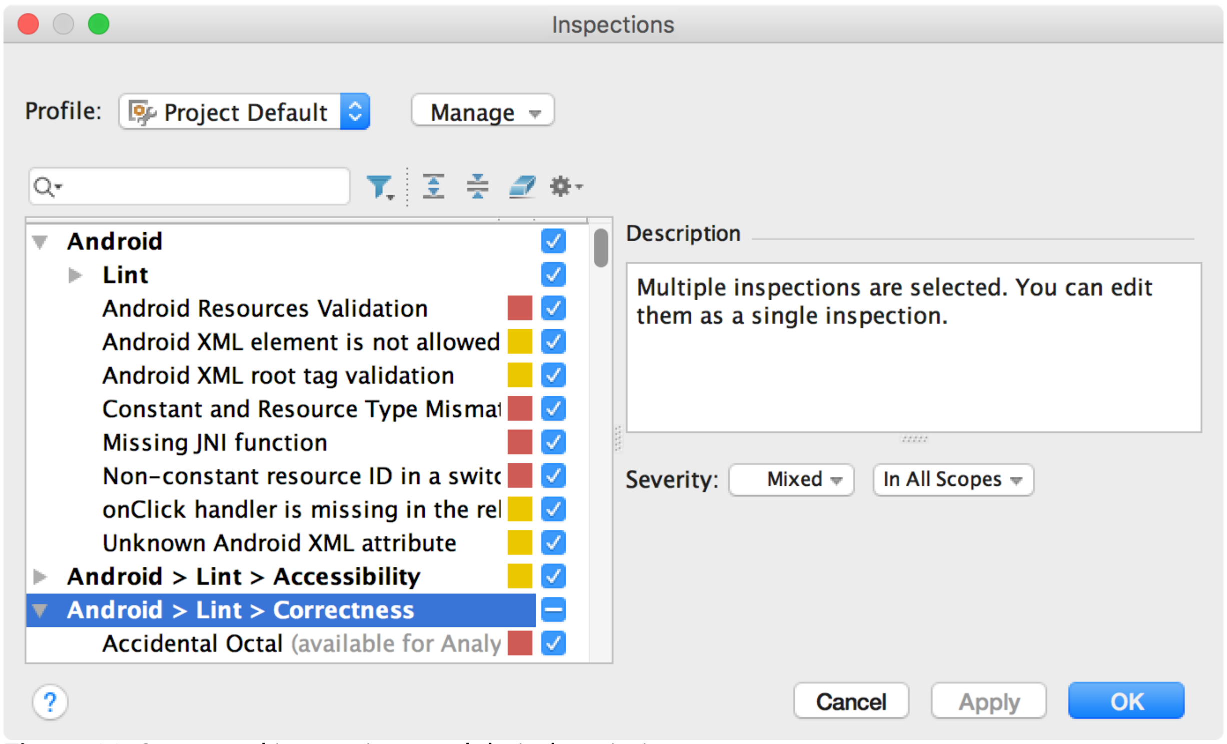

Inspections
Profile: Project Default
Manage
Android
Lint
Android Resources Validation
Android XML element is not allowed
Android XML root tag validation
Constant and Resource Type Mismat
Missing JNI function
Non-constant resource ID in a switc
onClick handler is missing in the rel
Unknown Android XML attribute
Android > Lint > Accessibility
Android > Lint > Correctness
Accidental Octal (available for Analy
Description
Multiple inspections are selected. You can edit them as a single inspection.
Severity: Mixed
In All Scopes
Cancel
Apply
OK

Figure 11. Supported inspections and their descriptions

3. Select the **Profile** drop-down list to toggle between **Default** (Android Studio) and **Project Default** (the active project) inspections. For more information, see this IntelliJ Specify Inspection Scope Dialog (https://www.jetbrains.com/help/idea/2020.1/specify-inspection-scope-dialog.html) page.
4. In the **Inspections** dialog in the left pane, select a top-level profile category, or expand a group and select a specific profile. When you select a profile category, you can edit all of the inspections in that category as a single inspection.
5. Select the **Manage** drop-down list to copy, rename, add descriptions to, export, and import inspections.
6. When you're done, click **OK**.

Last updated 2020-10-12 UTC.