Android Testing

William Royall Drumheller, Eric Tanner Reed

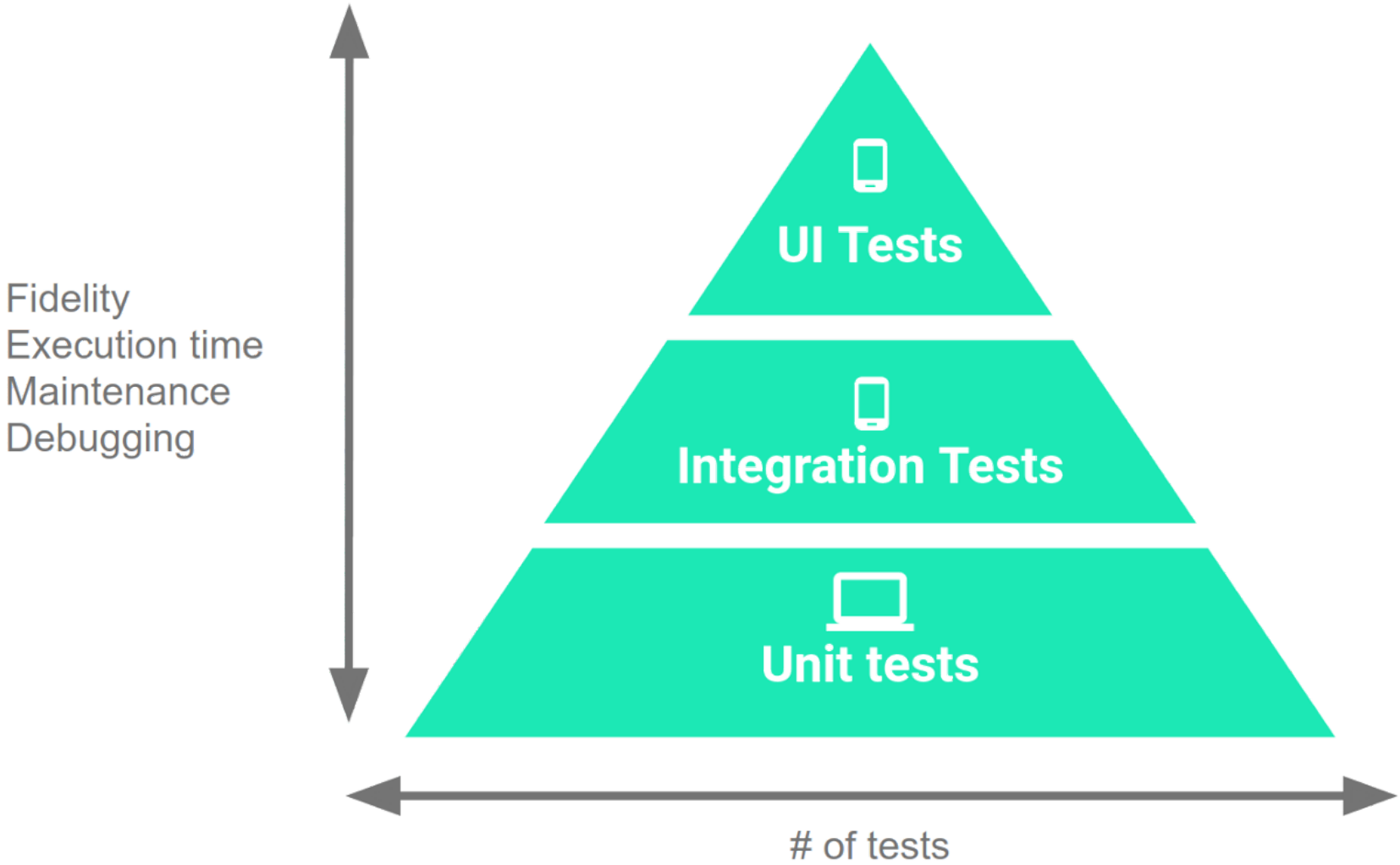
16APR2019

Proudly created using LaTeX

Testing Structure

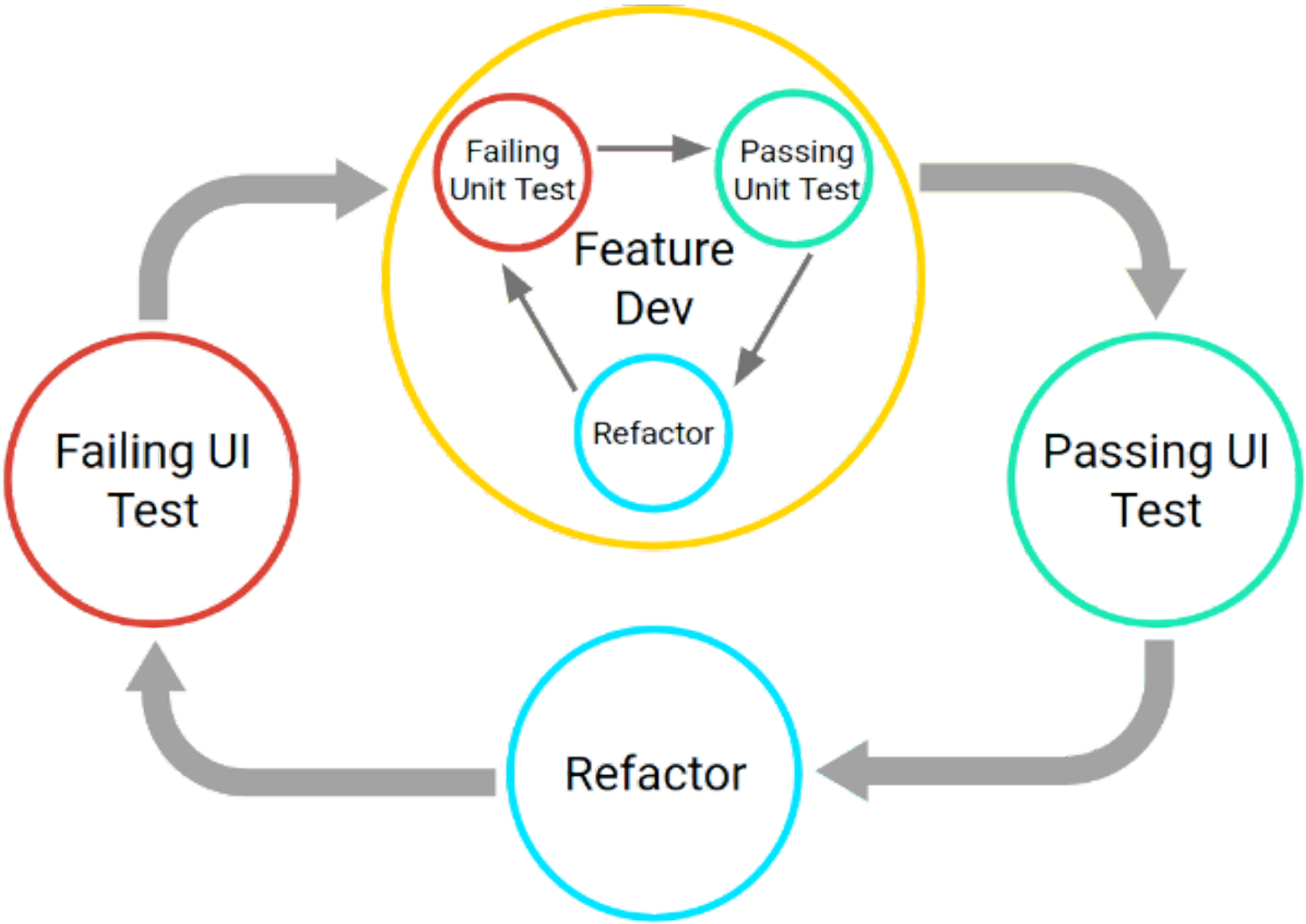
- Small Tests
  - small scale Unit tests
  - e.g. testing that a counter increments correctly
  - run locally on a development machine, without an emulator
- Medium Tests
  - test the integration between multiple components or classes
  - e.g. testing classes that access a file system or network
  - typically need Android environment resources
  - need to run on an emulator or physical device
- Large Tests
  - deal with system level and end user components
  - e.g. testing the UI functionality, including the *MainActivity*
  - typically need Android environment resources
  - need to run on an emulator or physical device

Testing Structure



https://developer.android.com/training/testing/fundamentals

Development and Testing Workflow

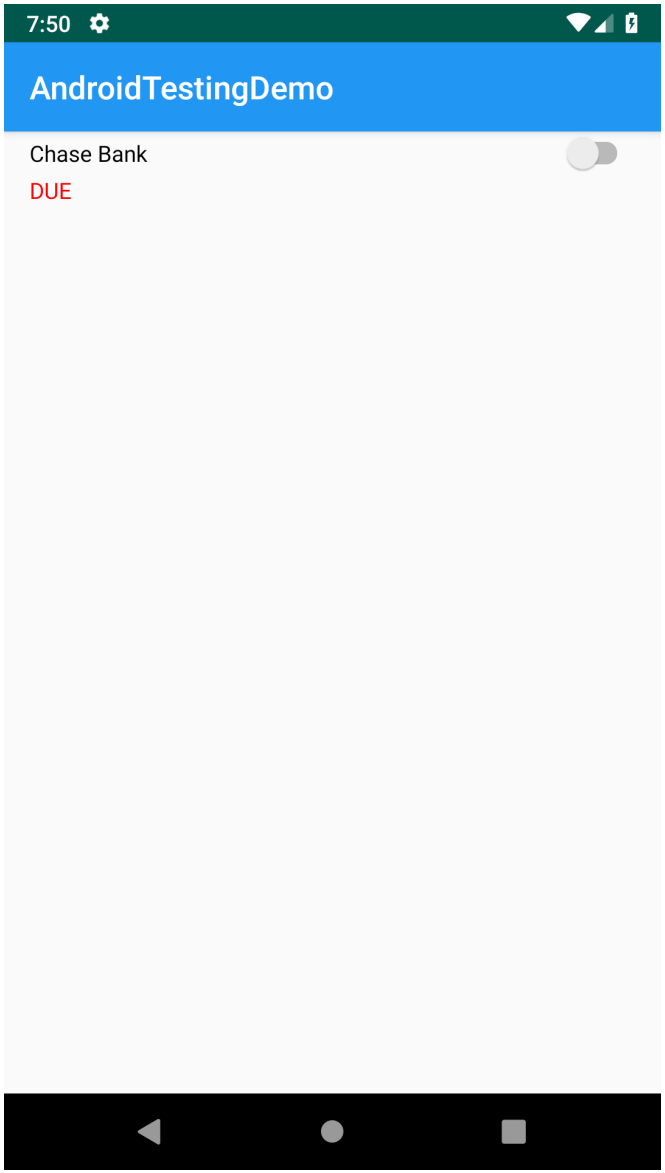


https://developer.android.com/training/testing/fundamentals

Why Do I Care?

- Test Driven Development (TDD)
  - Start with a failing test, implement the bare minimum to make it pass, repeat
  - Reduce code bloat and feel confident that when all tests pass, the job is done
  - Legacy code is not sustainable, so writing well-tested code is key
- Problem! Testing can be very, very slow!!!
  - We need to optimize our workflow by running tests as quickly as possible
  - Seconds of delay or interruption can hinder productivity
  - Small (unit) tests should be run frequently
  - Large (UI) tests should be run sparingly, since they require an emulator

So... What Do We Feel Like Building?



What Do We Feel Like Building?

What Do We Feel Like Building?

- Demo of Application
- Feel free to clone and follow along!
- https://github.com/RoyalIDesigns/AndroidTestingDemo

Tools of the Trade

We are going to explore the following Testing Tools needed in our project:

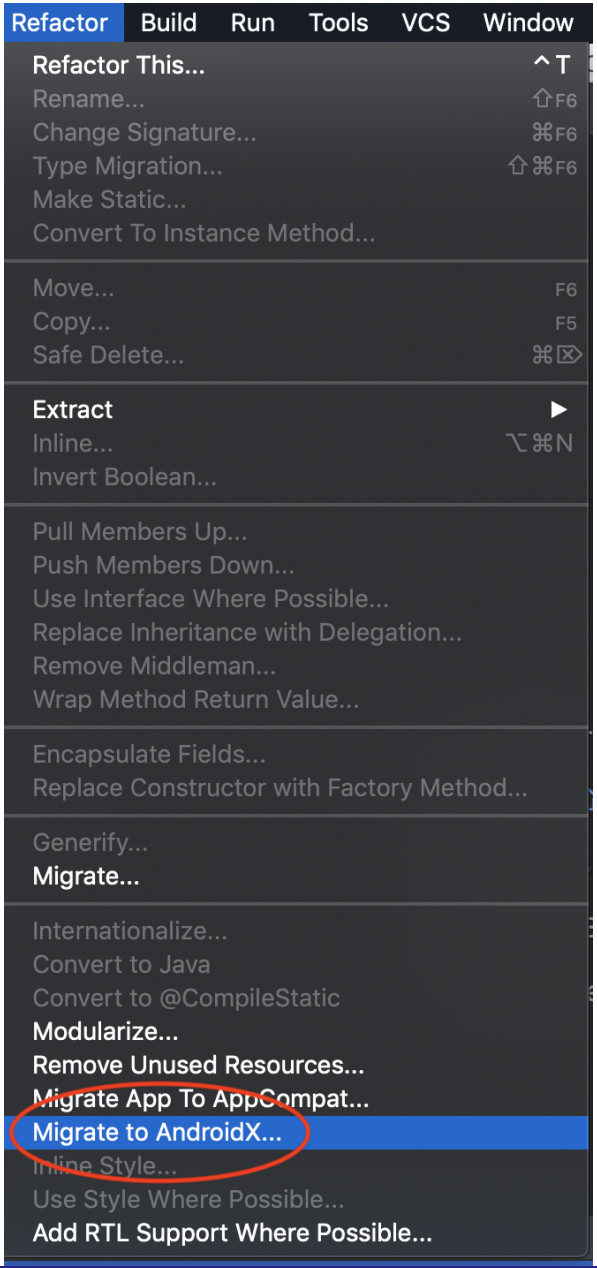
- *AndroidX* and *Espresso*
- *mockito*
- *Robolectric*

AndroidX and Espresso

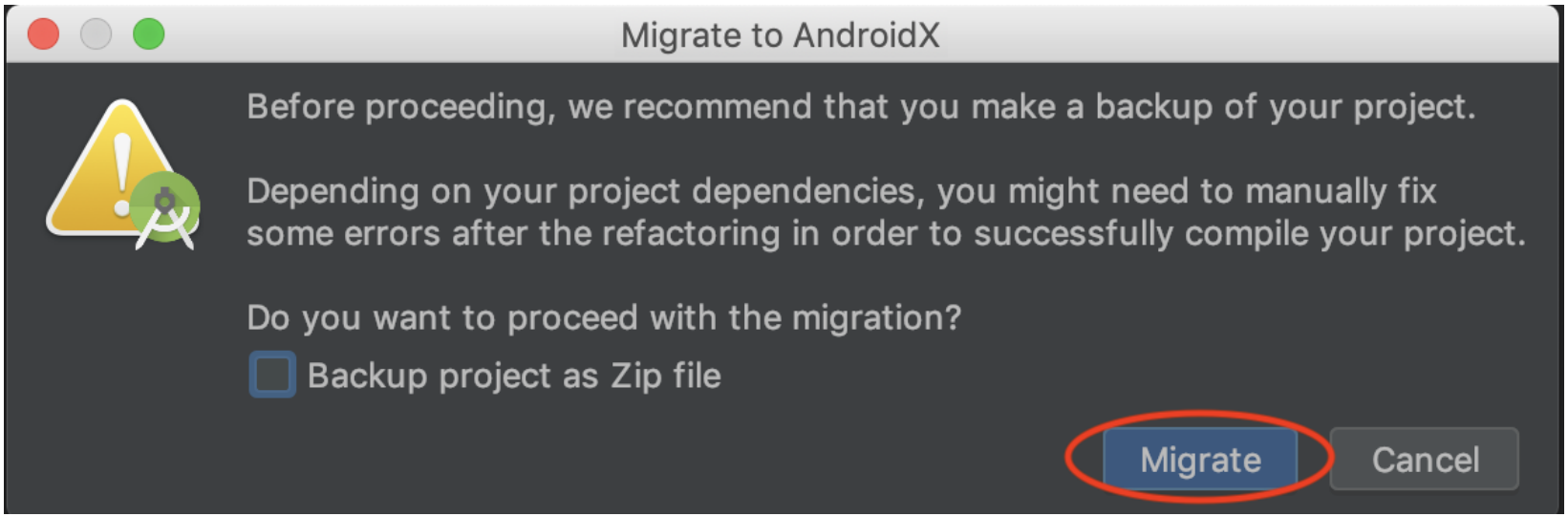
- *AndroidX* provides testing libraries that can be used with *JUnit*
- *AndroidX* also provides *Espresso*
- *Espresso* provides a library for automated UI testing
- Automated tests of the UI are much more scalable than manual testing

https://developer.android.com/training/testing/espresso

AndroidX Migration



AndroidX Migration



Adding JUnit, AndroidX, and Espresso

Listing 1: build.gradle

```
1     dependencies {
2         implementation fileTree(include: ['*.jar'], dir: 'libs')
3         .
4         .
5         .
6         testImplementation 'junit:junit:4.12'
7         testImplementation 'androidx.test:runner:1.1.0-alpha4'
8         androidTestImplementation 'androidx.test:runner:1.1.0-alpha4'
9         androidTestImplementation 'androidx.test:rules:1.1.0-alpha4'
10        androidTestImplementation 'androidx.test.espresso:espresso-core:3.1.0-alpha4'
11    }
```

Using JUnit, AndroidX, and Espresso

Listing 2: ToggleStateBehaviorTest.java

```
1   @RunWith(AndroidJUnit4.class)
2   @LargeTest
3   public class ToggleStateBehaviorTest {
4
5       @Rule
6       public ActivityTestRule<MainActivity> activityRule =
7           new ActivityTestRule<>(MainActivity.class);
8
9       @Test
10      public void defaultSwitchStateIsOff() {
11          onView(withId(R.id.chase_switch)).check(matches(isNotChecked()));
12      }
13
14      @Test
15      public void defaultTextViewIsDue() {
16          ViewInteraction interaction = onView(withId(R.id.chase_text_view));
17          interaction.check(matches(hasTextColor(R.color.due)));
18          interaction.check(matches(withText("DUE")));
19      }
```

Live Demo Time!

Mockito

- *Mockito* provides the framework to implement mocks
- Mocks are useful for "black-box testing", since they stub out functionality based on the interface
- The tester wants to provide a specific value that a method should return
- The tester defines basic, expected behavior without using a full blown object
- Mocks essentially prevent more overhead and more potential for failure
- Failures could stem from dependencies on implicit interfaces/behavior

https://site.mockito.org/

Adding Mockito

Listing 3: build.gradle

```
1        dependencies {
2            implementation fileTree(include: ['*.jar'], dir: 'libs')
3            .
4            .
5            .
6            testImplementation 'junit:junit:4.12'
7            testImplementation 'org.mockito:mockito-core:2.26.0'
8            androidTestImplementation 'org.mockito:mockito-android:2.26.0'
9            .
10           .
11           .
12       }
```

Using Mockito

Listing 4: StatePersistenceTest.java

```
@RunWith(MockitoJUnitRunner.class)
@MediumTest
public class StatePersistenceTest {

    @Rule
    public TemporaryFolder temporaryLocation = new TemporaryFolder();
    @Mock
    private Switch mockSwitch;
    @Mock
    private Context context;

    @Test
    public void savesTrueSwitchStateToFileSystem() throws IOException {
        initMocks(this);
        when(mockSwitch.isChecked()).thenReturn(true);
        when(context.getFilesDir()).thenReturn(temporaryLocation.newFolder());

        SwitchStatePersistence persistence = new SwitchStatePersistence(context,
            "state.save");
        String savedPath = persistence.saveState(mockSwitch);

        String result = new String(Files.readAllBytes(Paths.get(savedPath)));

        assertTrue(result.equals("true"));
    }
```

Live Demo Time!

Robolectric

- *Robolectric* provides the framework test Android code without using an emulator
- *Robolectric* also allows the user to forgo using mocks or spending overhead attempting to instantiate concrete Android objects
- Instead, the user may use "Shadow" objects, which can be used similarly to mocks
- Allows the user to run tests on the development machine using real Android framework code, as opposed to slowing down tests using an emulator

http://robolectric.org/

Adding Robolectric

Listing 5: build.gradle

```
android {
    .
    .
    .
    testOptions {
        unitTests {
            includeAndroidResources = true
        }
    }
}
```

Adding Robolectric

Listing 6: build.gradle

```
dependencies {
    implementation fileTree(include: ['*.jar'], dir: 'libs')
    .
    .
    .
    testImplementation 'junit:junit:4.12'
    testImplementation 'org.robolectric:robolectric:4.2.1'
    .
    .
    .
}
```

Using Robolectric

Listing 7: ToggleStateBehaviorTestRobolectric.java

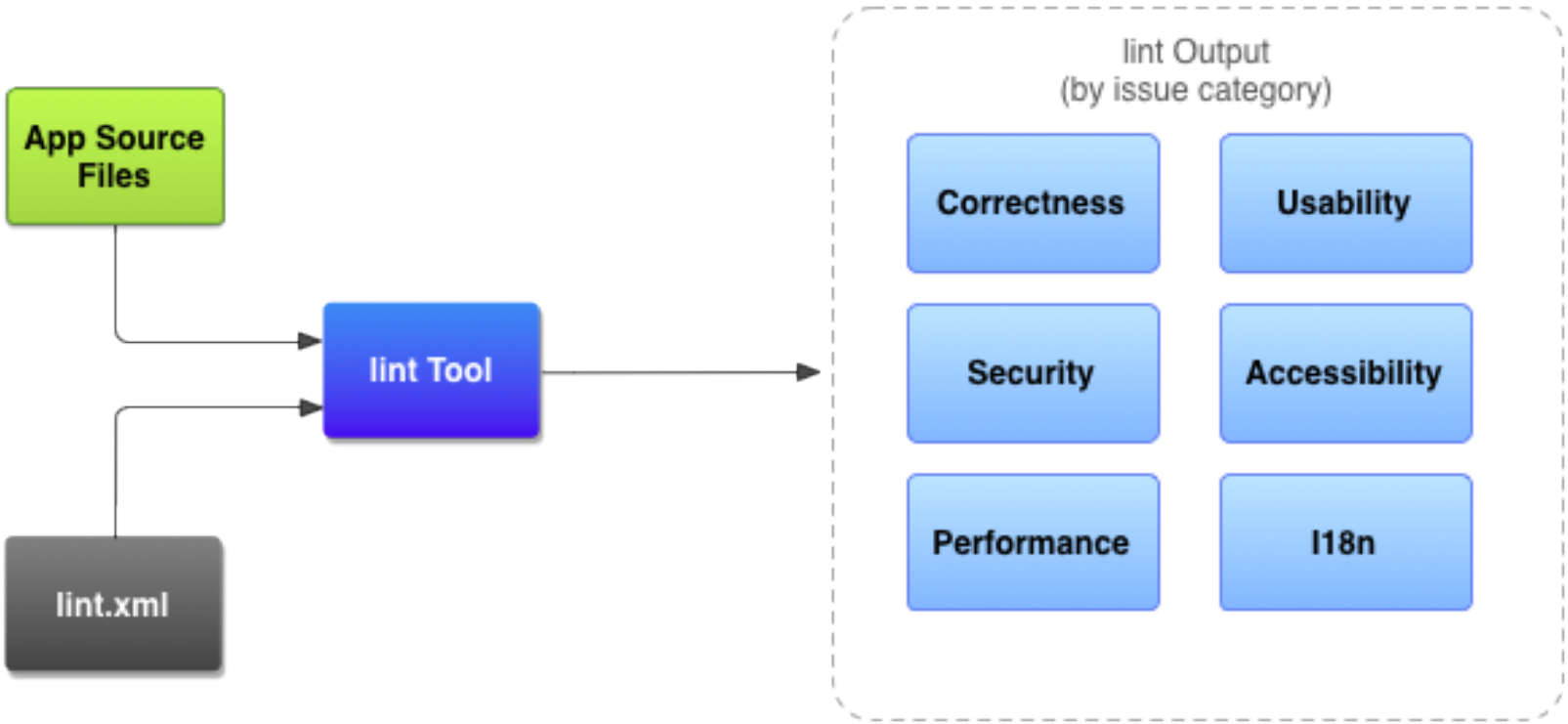
```
1   @RunWith(RobolectricTestRunner.class)
2   public class ToggleStateBehaviorTestRobolectric {
3
4       @Test
5       public void defaultSwitchStateIsOff() {
6           ActivityController<MainActivity> activity =
7               Robolectric.buildActivity(MainActivity.class).setup();
8
9           Switch chaseSwitch = activity.get().findViewById(R.id.chase_switch);
10
11          assertFalse(chaseSwitch.isChecked());
12      }
```

Live Demo Time!

What is Linting?

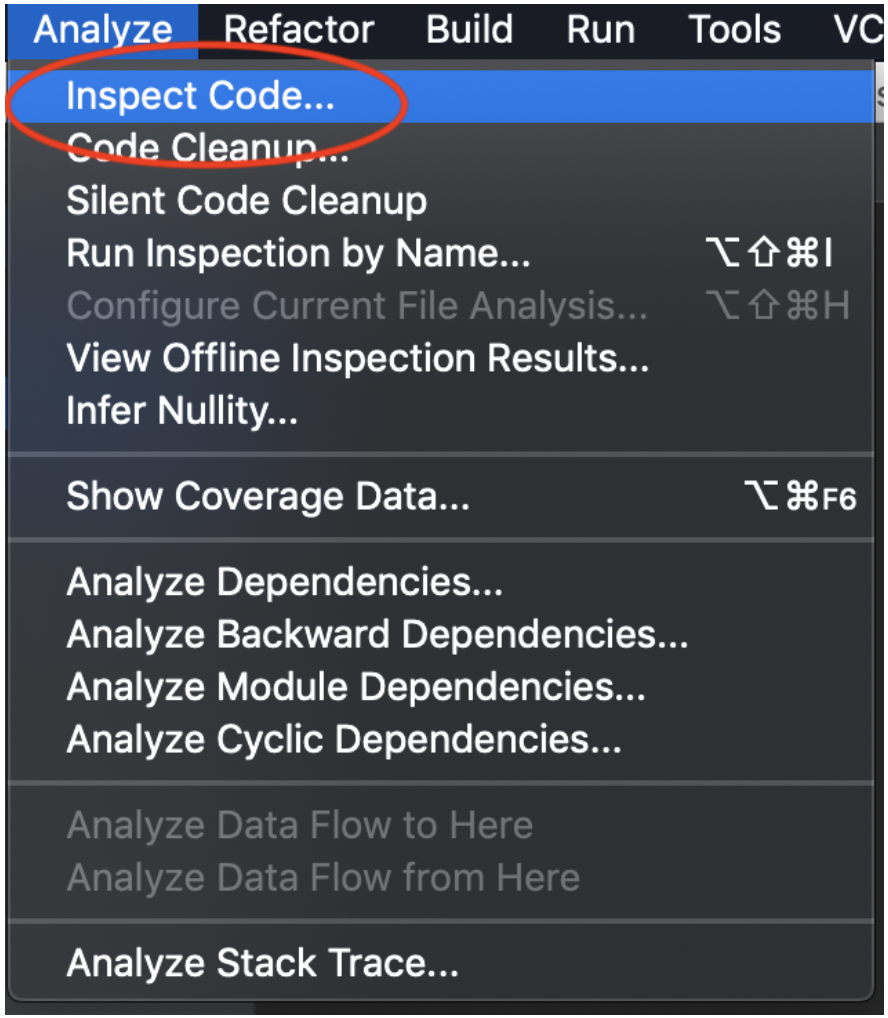
- Now that we are done testing our app, how do we vest confidence in how well we wrote it?
- Well, the IDE tends to lint or check for possible suggestions to our code, as we write it
- Checks include syntax errors, spell checking, API issues, versioning issues, etc.
- Style recommendations are also provided
- Can force the IDE to lint our code on demand

Linting

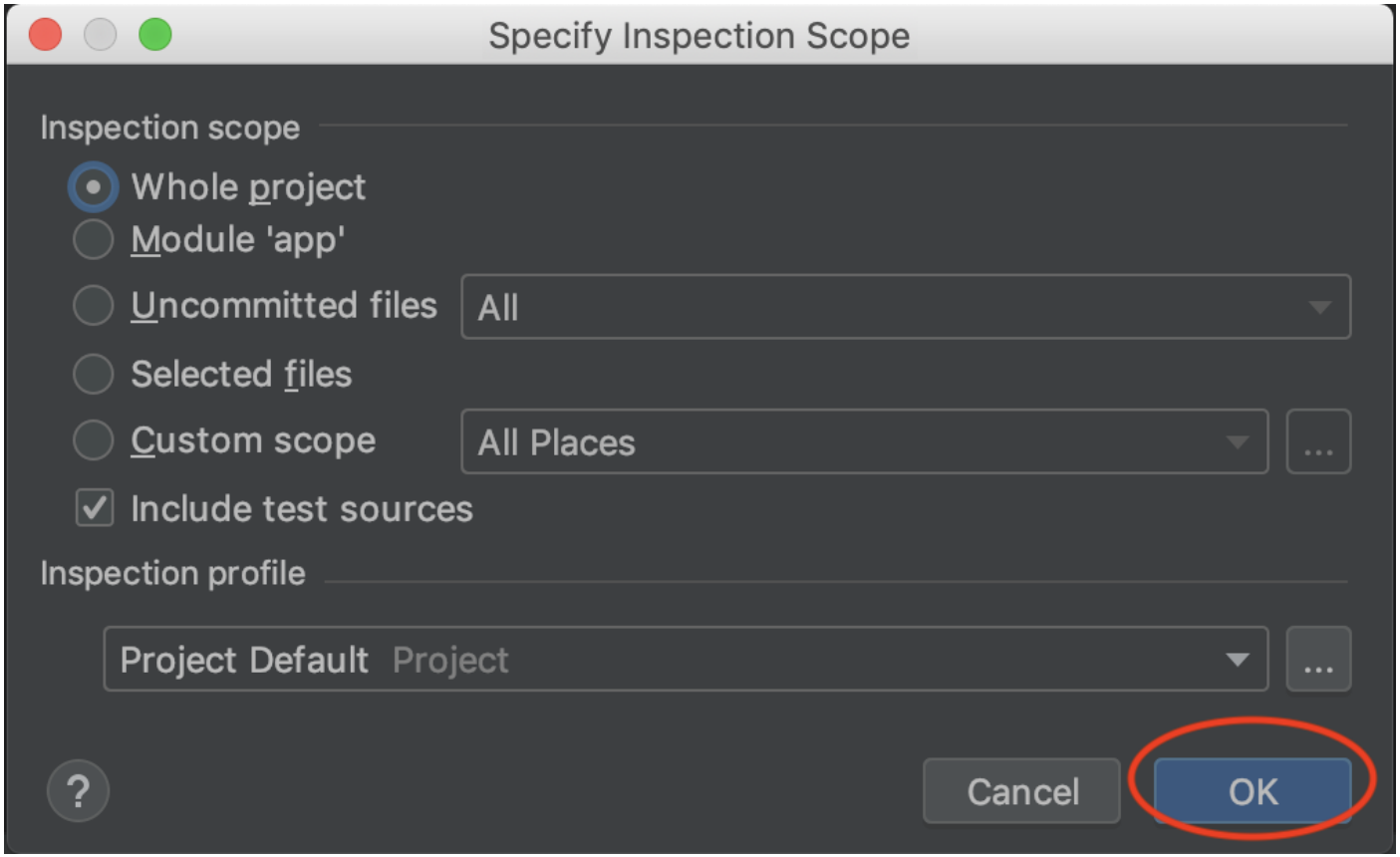


https://developer.android.com/studio/write/lint

Your IDE... can Lint!



Your IDE... can Lint! (Live DEMO Time!)



Questions, Concerns, Comments?

- Anything to ask?
- Anything to be concerned with?
- Thoughts?

References

- https://developer.android.com/training/testing/fundamentals
- https://www.youtube.com/watch?v=pK7W5npkhho
- https://developer.android.com/training/testing/ui-testing/espresso-testing#java
- http://robolectric.org/
- http://robolectric.org/writing-a-test/
- https://developer.android.com/studio/write/lint