Overview

- Understand the need for writing tests
- Understand the different types of tests
- How to set up a project for Unit testing
- How to add Local Unit tests to your app
- Running Unit tests and viewing test coverage reports
- How to handle dependencies in Unit tests

Advantages of Writing Automated Tests

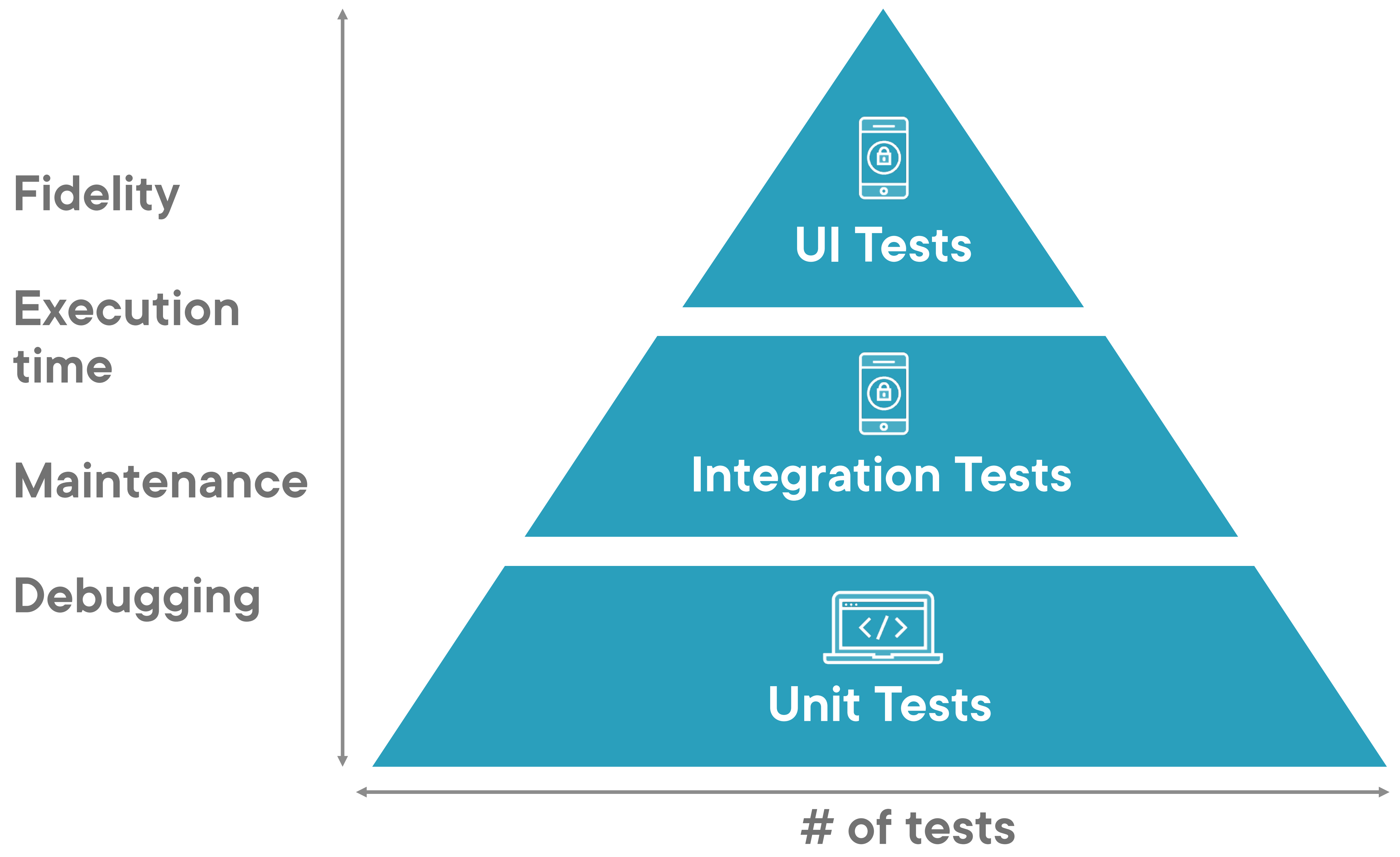
1 Detect failures as they happen

2 Easier code refactoring

3 Consistent development speed

4 No regressions

Testing Pyramid



Small Tests

01

Tests that validate one class at a time

02

Should exhaust all possibilities for the class

03

Very fast to execute

04

JUnit Tests

Instrumented Unit Tests

Medium Tests

01

Tests that validate interaction between modules

02

Should exhaust all interactions between modules

03

Slower than Small Tests

04

Espresso Tests

Large Tests

01

Tests that validate end to end user flows

02

Should cover all major end to end user flows

03

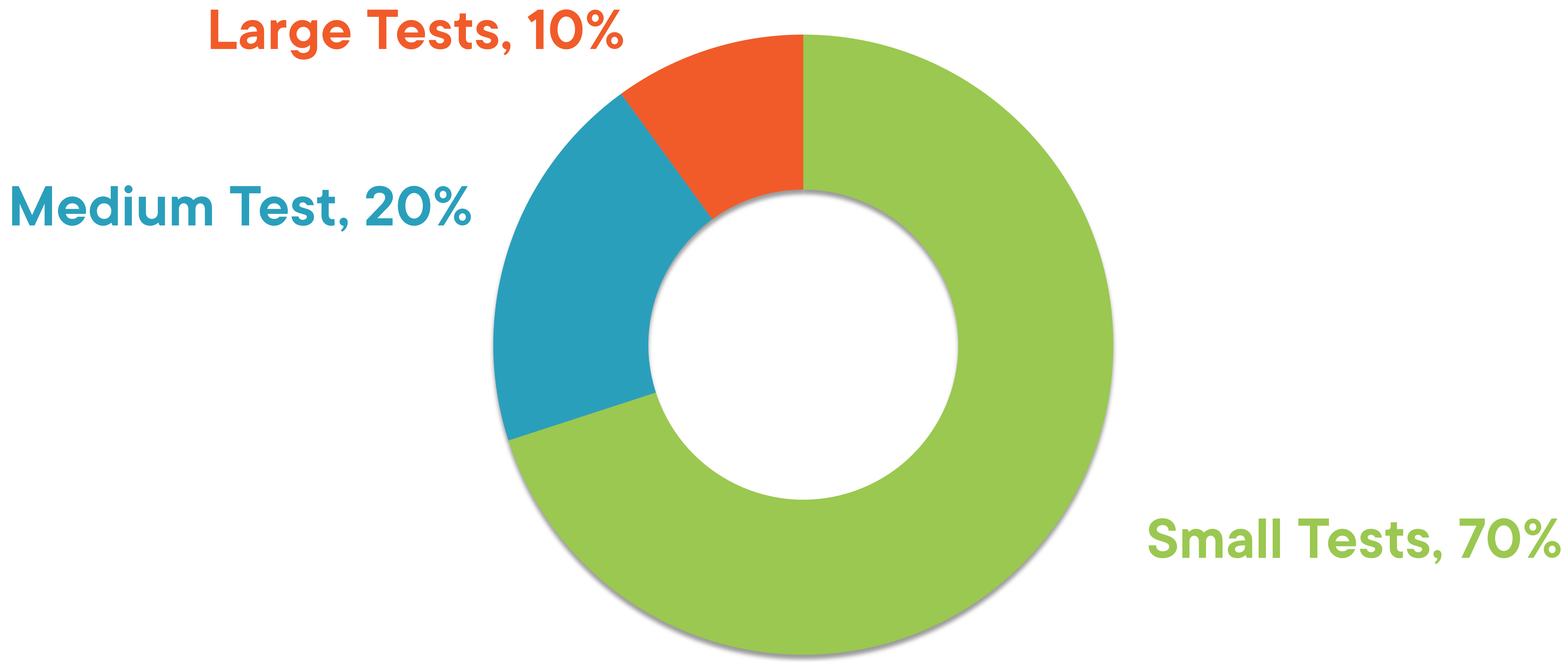
Slowest to execute

04

Espresso Tests

UI Automator Tests

Recommended Proportion of Different Tests



Setting up a Project for Unit Testing

Making Your Code More Testable

1 Think of code in terms of modules

2 Each module has a specific focus and set of things to do

3 Clearly defined interfaces and boundaries for each module

Key Libraries for Writing Unit Tests

Unit testing framework	Framework dependencies	Mock dependencies	Assertion library
JUnit4	Robolectric	Mockito	Truth

```
dependencies {

    testImplementation 'junit:junit:4.12'

    testImplementation 'androidx.test:core:1.0.0'

    testImplementation 'org.mockito:mockito-core:1.10.19’

    androidTestImplementation 'com.google.truth:truth:0.42'
}
```

Add Dependencies to App's build.gradle

Demo

Hydration tracker application

Configure key dependencies for writing Local Unit tests in the app

Anatomy of a JUnit Test

Setup

Create class instances needed to execute the code under test

Action

Invoke code under test

Assert

Validate that the output is as expected

Writing JUnit Tests

01

Add test class to src/test/java/ folder

02

Annotate setup method with @Before

03

Annotate test methods with @Test

```
public class EmailValidatorTest {


    @Before
    fun setUp() {
        // ...
    }

    @Test
    fun emailValidator_CorrectEmailSimple_ReturnsTrue() {
        assertThat(EmailValidator.isValidEmail("name@email.com")).isTrue()
    }
}
```

Sample JUnit Test

Running Local Unit Tests in Android Studio

Single method

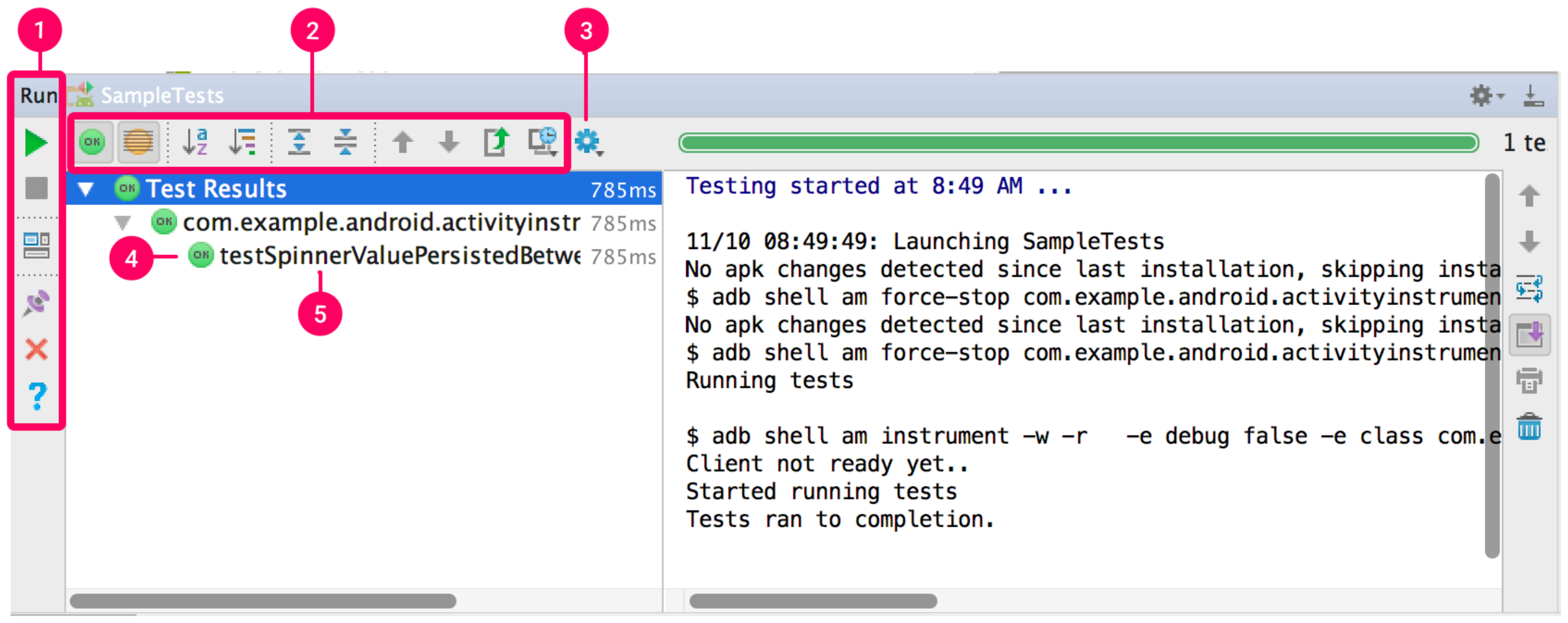
All methods in a class

All classes in a folder

Right click on the test, class or a directory

Click Run

Android Studio will run and display the results


1
2
3
4
5
Run
SampleTests
Test Results 785ms
com.example.android.activityinstr 785ms
testSpinnerValuePersistedBetwe 785ms
1 te

```
Testing started at 8:49 AM ...

11/10 08:49:49: Launching SampleTests
No apk changes detected since last installation, skipping insta
$ adb shell am force-stop com.example.android.activityinstrumen
No apk changes detected since last installation, skipping insta
$ adb shell am force-stop com.example.android.activityinstrumen
Running tests

$ adb shell am instrument -w -r   -e debug false -e class com.e
Client not ready yet..
Started running tests
Tests ran to completion.
```

Viewing Code Coverage reports

1 **Right click on method, class, folder**

2 **Click Run with coverage**

3 **Coverage tool window shows code coverage report**

Demo

Hydration tracker application

Write JUnit tests for the water intake model class

Run the JUnit tests and view code coverage reports

Handling Dependencies for Unit Tests

1

Framework dependencies

Complex interaction with Android framework

2

Mock dependencies

Minimal or no interaction with Android framework

Adding Framework Dependencies

1 Use the Robolectric library

2 Provides classes with same name as Android to reduce cognitive load

3 Executes real Android framework code on local JVM

Adding Mock Dependencies

1 Use the Mockito library

2 Allows adding mock objects for dependencies

3 The mocked objects don’t execute real code and instead return a specific preset value when invoked

Demo

Hydration tracker application

Add JUnit tests for the main screen with dependencies on Android framework

Summary

Advantages of writing automated tests

Different types of tests available on Android

Configuring a project for writing Local Unit tests

Adding Local Unit tests to an app

Running Local Unit tests with code coverage reports

How to handle dependencies when writing Local Unit tests

Up Next:
Implementing Instrumented and UI Tests

Overview

Configuring a project for Instrumented tests

Adding an Instrumented test to a project

Running an Instrumented test

Configuring a project for Espresso UI tests

Writing an end-to-end UI test using Espresso

Running an Espresso UI test

Features of Instrumented Unit Tests

Run on physical devices or emulators

Complex interaction with Android frameworks and APIs

Much slower compared to Local Unit Tests

Configuring Project for Instrumented Tests

Add AndroidX dependencies

Configure JUnit Runner

```
dependencies {

    ...Other dependencies

    androidTestImplementation 'androidx.test:core-ktx:1.4.0’
    androidTestImplementation 'androidx.test.ext:junit-ktx:1.1.3’
    androidTestImplementation 'androidx.test:runner:1.4.0'
}
```

Add Dependencies to App's build.gradle

JUnit4 Rules

ActivityScenarioRule

Launching an Activity, state transition and performing actions on an Activity

ServiceTestRule

Start up, shut down and perform action on a Service

Writing Instrumented Unit Tests

01

Add test class to src/androidTest/java/ folder

02

Annotate setup method with @Before

03

Annotate test methods with @Test

```
class MyTestSuite {
    @Test fun testResult() {
        val scenario = launchActivity<MyActivity>()
        onView(withId(R.id.finish_button)).perform(click())

        // Activity under test is now finished.
        val resultData = scenario.result.resultData

       // Do assertions
    }
}
```

Sample Instrumented Unit Test

Features of Effective UI Tests

Tests flows in an app by simulating user actions

Removes need for manual user flow verification by a human tester

Ensures app runs as expected on different devices

Configuring Project for Espresso UI Tests

Add dependency for Espresso

Turn off animations on the test device

```
dependencies {

    ...Other dependencies

    androidTestImplementation 'androidx.test.espresso:espresso-core:3.4.0'
}
```

Add Dependencies to App's build.gradle

Anatomy of an Espresso UI Test

Access

Get hold of a view using id, text, etc

onView() / onData()

Perform

Perform one of the various actions

perform()

Verify

Verify the state of the view

Check()

```
onView(withId(R.id.editTextUserInput))
                .perform(typeText(stringToBetyped),
closeSoftKeyboard());
```

- Type text in the edit text field

```
onView(withId(R.id.changeButton)).perform(click());
```

- Perform click on save button

```
onView(withId(R.id.textToBeChanged))
                .check(matches(withText(stringToBetype
d)));
```

- Verify the typed string is updated in the Label

Summary

Configure a project for Instrumented tests

How to implement and run an Instrumented test

Configure a project for Espresso UI tests

How to implement and run an Espresso UI test

Thank You