Android Testing

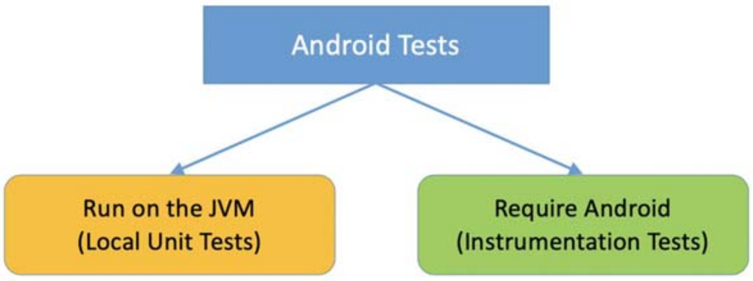
(Unit/Instrumentation/UI) Tests

Course Contents

Android Testing

(Unit/Instrumentation/UI) Tests

@SmallTest @MediumTest @LargeTest


Android Tests
Run on the JVM (Local Unit Tests)
Require Android (Instrumentation Tests)

JUnit

mockito

espresso

Testing app is an integral part of the app development process … but

Several excuses for not testing Apps

- *“Mobile apps are frontend apps, the real logic is in the backend, so backend apps should be tested instead.”*
- *“Mobile apps are difficult to unit test, because most of the logic is done in the UI. At most, you should only care about UI tests.”*
- *“Mobile apps are “simple” or “tiny” compared to backend apps. Thus, effort should be put in the features instead of wasting time making tests.”*

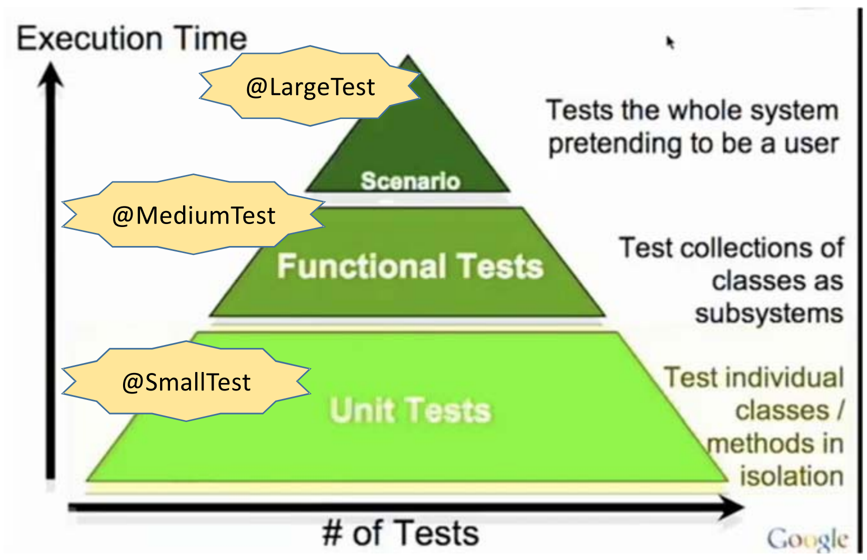
Types of Software Testing

- Integration Testing
- Load Testing
- Acceptance Testing
- Monkey Testing
- Block Box Testing
- White Box Testing
- Performance Testing
- Function Testing
- Alpha Testing
- Beta Testing

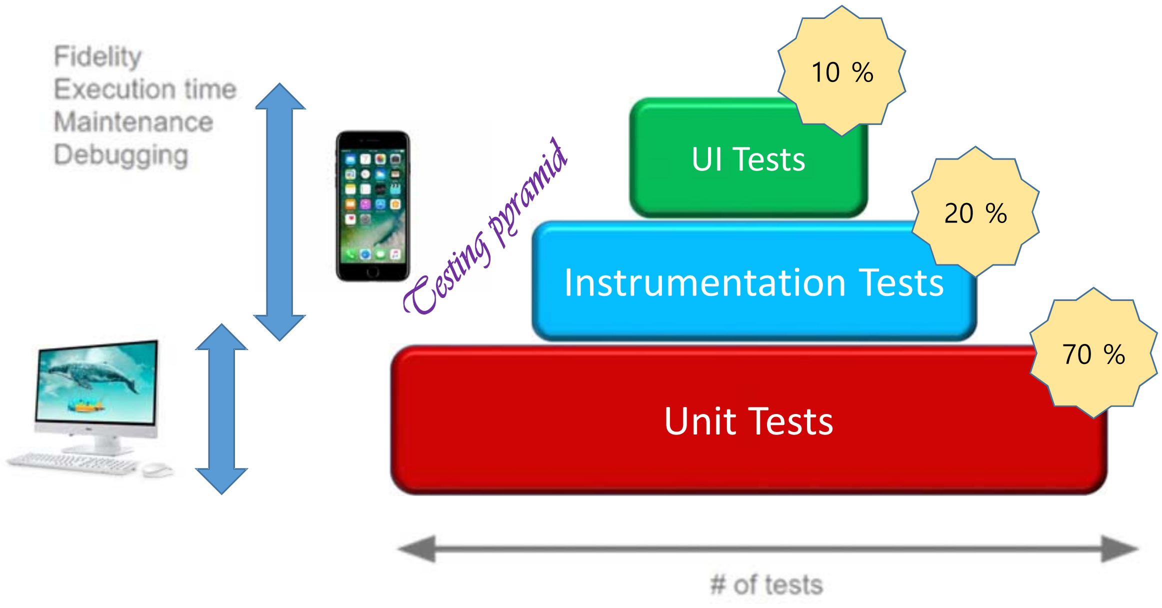
- System Testing
- Gorilla Testing
- Acceptance Testing
- Security Testing
- Compatibility Testing
- White Box Testing
- Regression Testing
- Sanity Testing
- Usability Testing
- Negative Testing

- End-to-End Testing
- Stress Testing
- Property-based Testing
- Security Testing
- Volume Testing
- Mutation Testing
- …

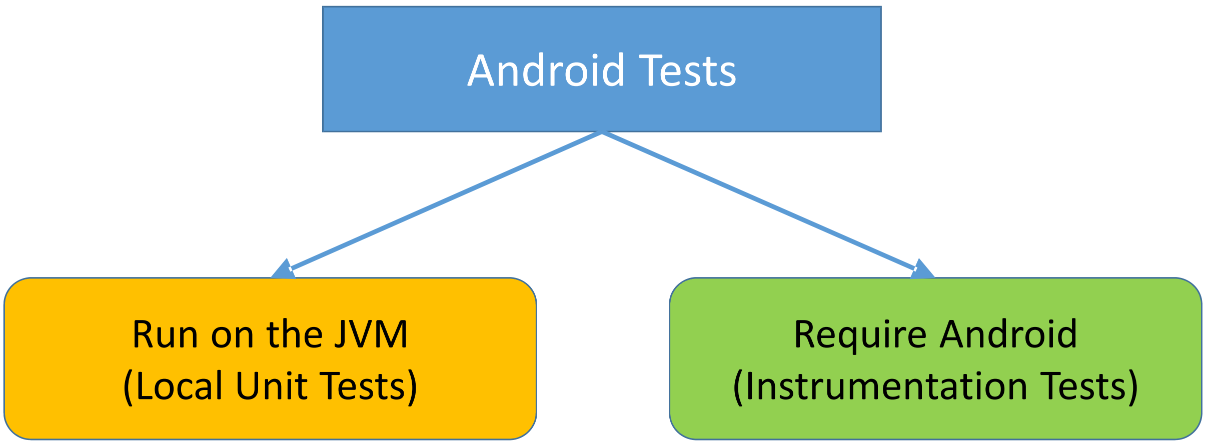
Different Kinds of Tests from Google: Before


Execution Time
@LargeTest
Scenario
Tests the whole system pretending to be a user
@MediumTest
Functional Tests
Test collections of classes as subsystems
@SmallTest
Unit Tests
Test individual classes / methods in isolation
# of Tests
Google

Android Tests: Now

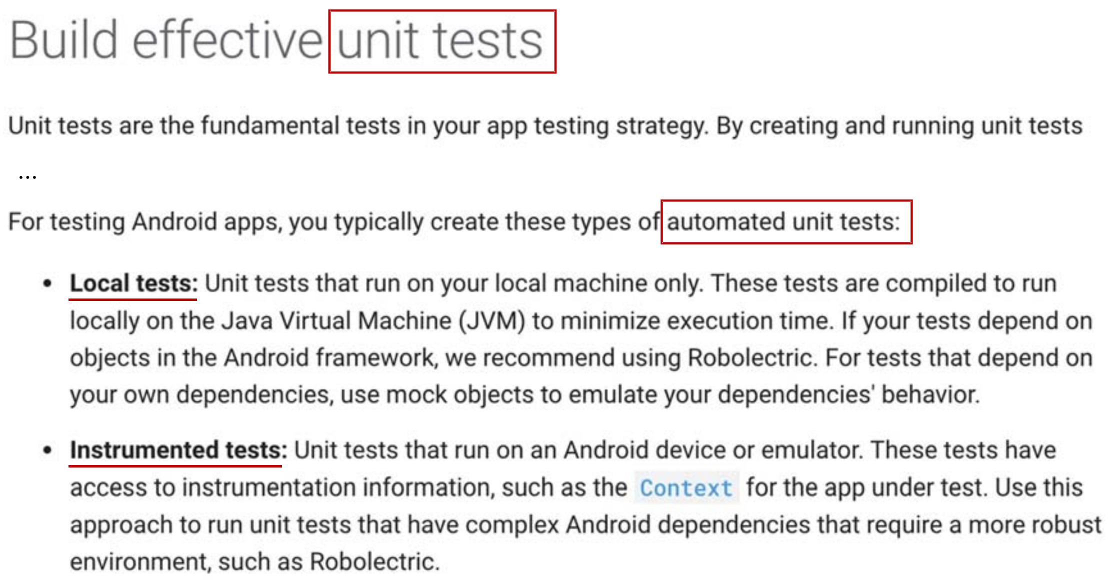


Categories of Android Tests



Confusing terminology

https://developer.android.com/training/testing/unit-testing



Build effective unit tests

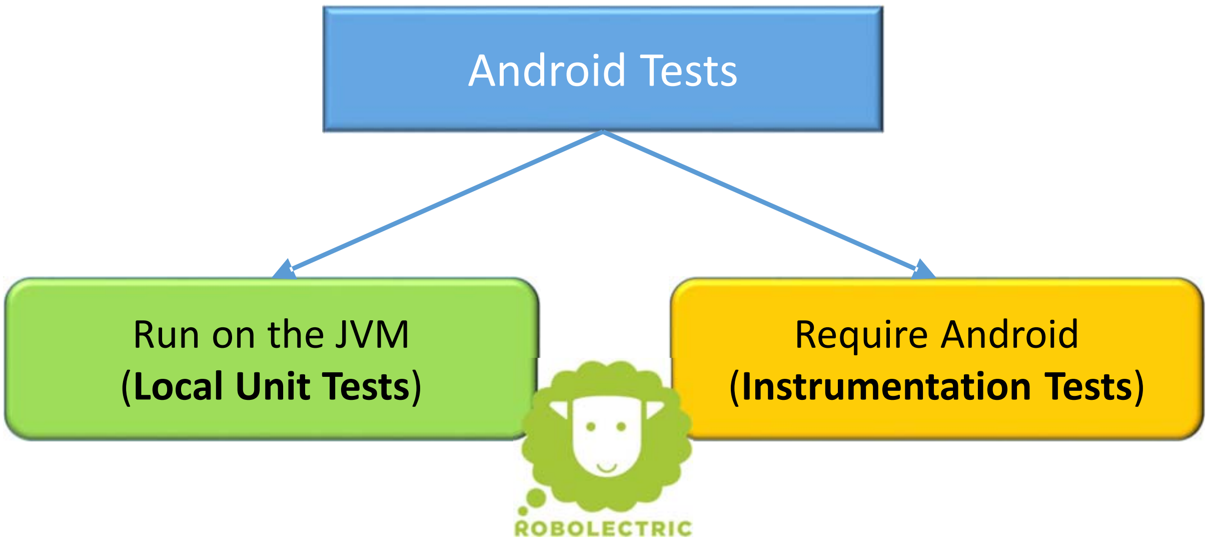
Unit tests are the fundamental tests in your app testing strategy. By creating and running unit tests

...

For testing Android apps, you typically create these types of automated unit tests:

- **Local tests:** Unit tests that run on your local machine only. These tests are compiled to run locally on the Java Virtual Machine (JVM) to minimize execution time. If your tests depend on objects in the Android framework, we recommend using Robolectric. For tests that depend on your own dependencies, use mock objects to emulate your dependencies' behavior.
- **Instrumented tests:** Unit tests that run on an Android device or emulator. These tests have access to instrumentation information, such as the `Context` for the app under test. Use this approach to run unit tests that have complex Android dependencies that require a more robust environment, such as Robolectric.

Categories of Android Tests



Unit Tests vs. Integration Tests in General

- **Unit testing** is a method that instantiates a small part of our code and verifies its behavior ***independently*** from other parts of the project.
- **Integration testing** focuses on testing and observing different software modules as a group.
- It is typically performed after unit testing is complete and precedes validation testing.

Unit Testing Tools for Android App

- **JUnit4/JUnit5**
  - normal test assertions
- **Mockito/MockK**
  - mocking out other classes that are not under test
- **PowerMock**
  - mocking out static classes
- **Robolectric**
  - simulate Android Framework

Instrumented & UI testing Tools for Android

- **Espresso**
  - Used for testing within your app, selecting items, making sure something is visible, etc.

- UIAutomator
  - Used for testing interaction between different apps.

- Other tools
  - Appium, Calabash, Robotium, etc.

Why Write Tests?

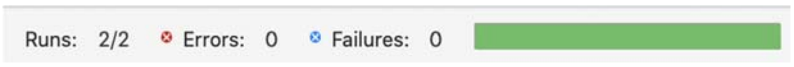
- To find the bug that may exist in our code? ✗

- To test the functionality of our code, i.e., whether our code is working as expected or not. ✓

- To refactor the code for evolution ✓

More on Why Testing?

- Testing forces you to think in a different way and implicitly makes your code cleaner in the process.
- You feel more ***confident*** about your code if it has tests.
- Regression testing is made a lot easier, as automated tests would pick up the bugs first.
- Executable *live* documents!
- Shiny green status bars and cool coverage reports are added bonus!



```
Runs: 2/2   Errors: 0   Failures: 0
```

Advantages of Testing

Testing also provides you with the following advantages:

- **Rapid feedback** on failures.
- **Early failure detection** in the development cycle.
- **Safer code refactoring**, letting you optimize code without worrying about regressions.
- **Stable development velocity**, helping you minimize technical debt.

"If it is not tested, it's broken”

– Bruce Eckel

“Legacy code is code without tests”

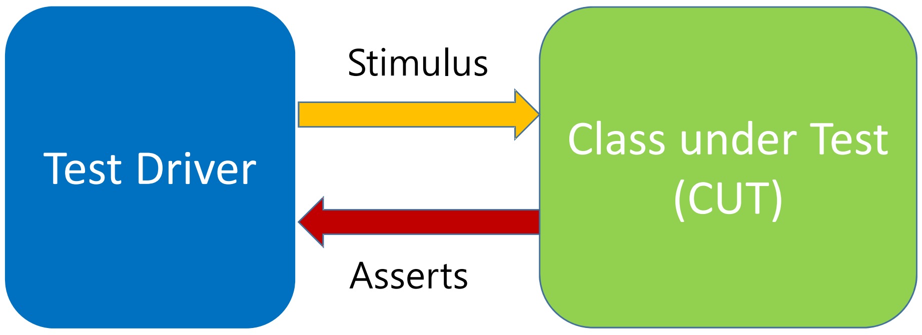
“Code without tests is bad code”

– Michael Feathers

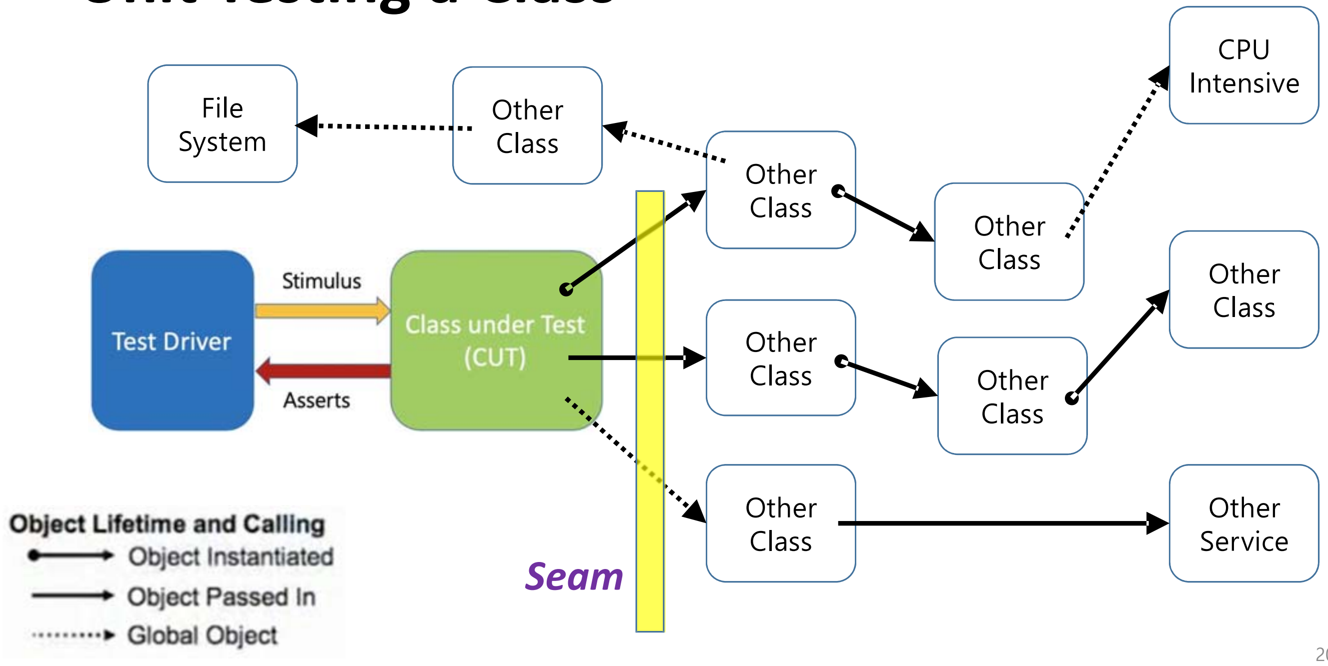
What is a unit?

- “The smallest component that it makes sense to test”
- Unit for testing depends on individual programmers or teams
- Generally, a unit means
  - class or an interface
  - a single method or function.

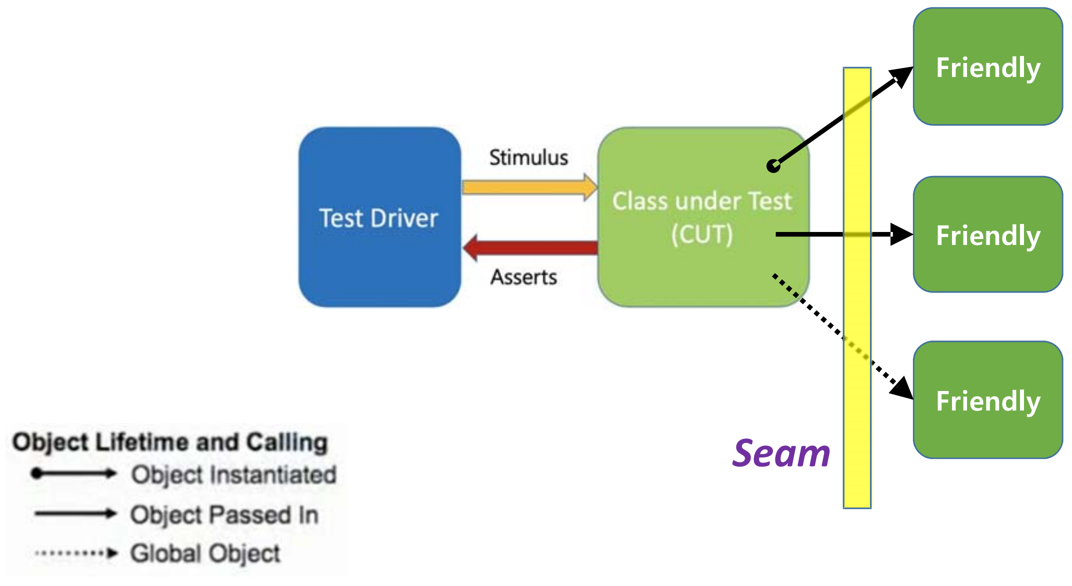
Unit Testing a Class



Unit Testing a Class



Unit Testing a Class



What is unit testing?

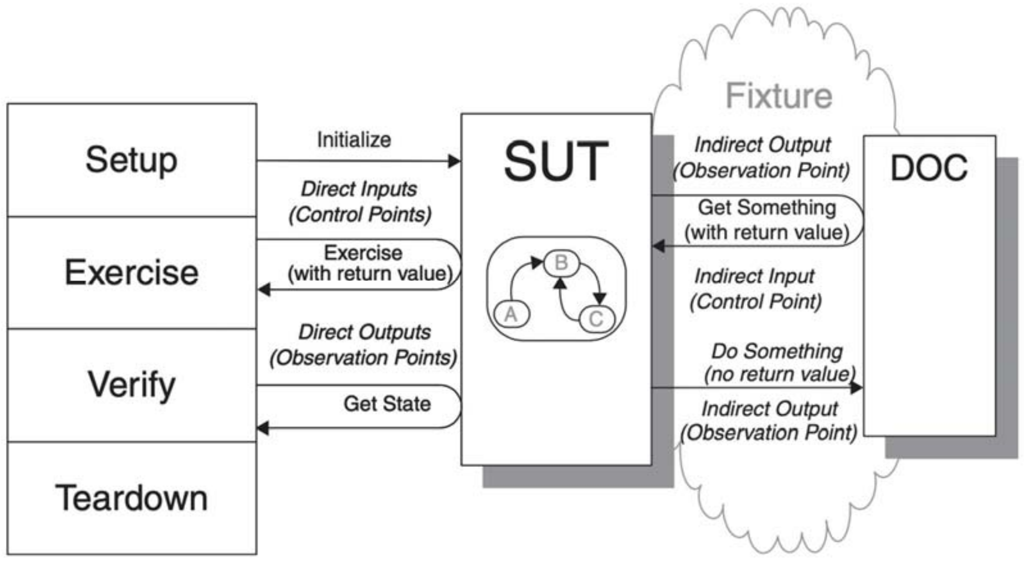
- Unit testing is a method that
  - instantiates a small part of our code (i.e., unit of work) and
  - verifies its behavior
  - ***independently*** from any other parts (Unit, Code etc.) of the project.
- External dependencies are managed by **Test Doubles** (Dummies/Fakes/Mocks/Stubs/Spies)

A Set of Unit Testing Rules

A test is <u>not</u> a unit test if:

- It talks to the database
- It communicates across the network
- It touches the file system
- It can't run at the same time as any of your other unit tests
- You have to do special things to your environment (such as editing config files) to run it.

System Under Test (SUT) vs. Depended-On Objects (DOC)



How to do unit testing?

A unit test typically features three different phases (**AAA**):

• **Arrange** (Given) • ***Preparation***	**Act** (When) ***Execution***	**Assert** (Then) ***Assertion***
• An SUT initialization • Stubs/Mocks creation • Stubbing and Injection *"Given your is logged in …"*	• An operation to test in a given test *"When user launches app …"* *"When user clicks Log Out …"*	• Received result verification • Mocks verification (if needed) *"Then user sees logout text"*

Writing Unit Tests

```
// Production Code
public int sum(int a, int b) {
    return a + b;
}
```

```
// Unit Test Code
@Test
void testSum() {
   // Given (Arrange)
   int firstNumber = 15;
   int secondNumber = 27

   // When (Act)
   int result = sum(firstNumber, secondNumber);

   // Then (Assert)
   assertEquals(42, result);

}
```

Build Local Unit Tests

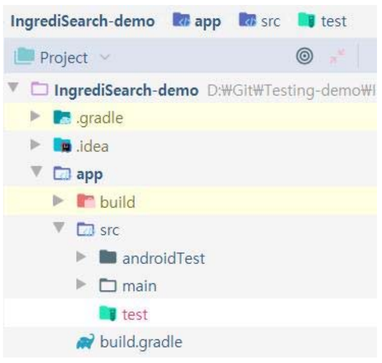
- Store the source files for local unit tests at
  module-name/src/test/java/
- In your app's top-level **build.gradle** file,

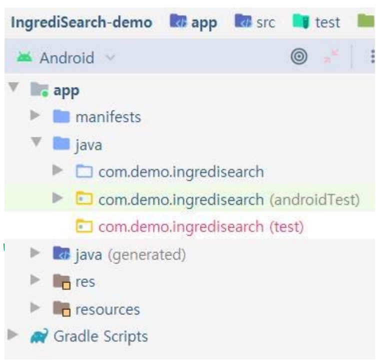
```
dependencies {
    // Required -- JUnit 4 framework
    testImplementation 'junit:junit:4.13'

    // Optional -- Mockito framework
    testImplementation 'org.mockito:mockito-core:3.5.5'
}
```

Build Local Unit Tests

- Store the source files for local unit tests at
  app/src/**test**/java/
- In your app's top-level **build.gradle** file,

```
dependencies {
    // Required -- JUnit 4 framework
    testImplementation 'junit:junit:4.13'

    // Optional -- Mockito framework
    testImplementation 'org.mockito:mockito-core:3.5.5'
}
```


IngrediSearch-demo app src test
Project
IngrediSearch-demo D:₩Git₩Testing-demo₩I
.gradle
.idea
app
build
src
androidTest
main
test
build.gradle


IngrediSearch-demo app src test
Android
app
manifests
java
com.demo.ingredisearch
com.demo.ingredisearch (androidTest)
com.demo.ingredisearch (test)
java (generated)
res
resources
Gradle Scripts

How to write Good Tests?

Unfortunately, well … virtually ***nothing***!

There is no secret knowledge of how to write good tests …

… except for a couple of tips about styles and general tactics …

Trouble writing tests?

- The problem's not in your test suite.
- It's in your code.

Common Warning Signs of Hard to Test Codes

- Static Properties and Fields
- Singletons
- Static Methods
- The `new` Operator
- Work in constructor
- Mixing `new` with logic
- …

After all, writing tests is just like writing production code … But

- Unit testing code is production code that you will need to maintain, refactor and build upon for years to come.

- The rules that apply for writing good production code do NOT always apply to creating a good unit testing.

- Do not fall into the trap of following best practices for writing production code that are not appropriate for writing unit tests.

What makes a good unit test?

- Unit tests are short, quick, and automated tests that make sure a specific part of your program works.
- They test specific functionality of a unit that have a clear **pass**/**fail** condition.
- A "good" unit test follows these rules:
  1. **The test only fails when a new bug is introduced into the system or requirements change**
  2. **When the test fails, it is easy to understand the reason for the failure.**

Traits of Good Unit Tests

- **F**ast
- **I**solated/Independent
- **R**epeatable
- **S**elf-Validating
- **T**horough