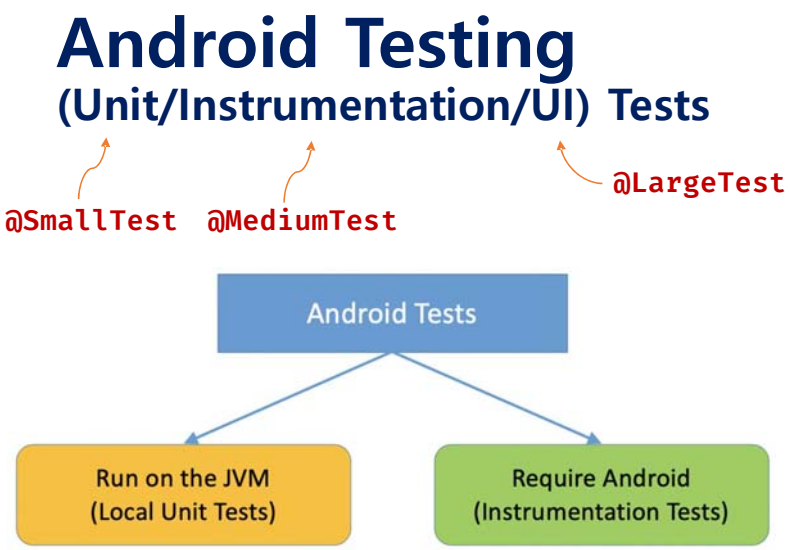
Android Testing

(Unit/Instrumentation/UI) Tests

Course Contents



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

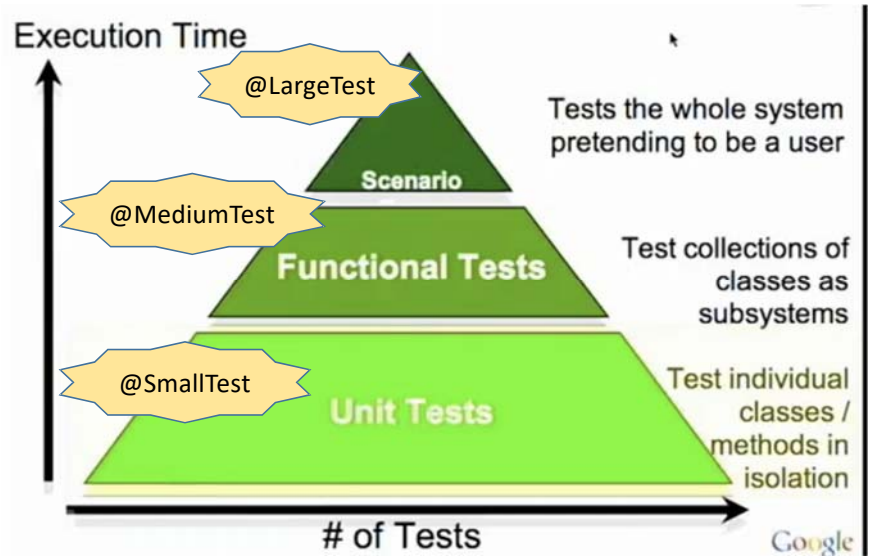
Several excuses for not testing Apps

- *“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.”*

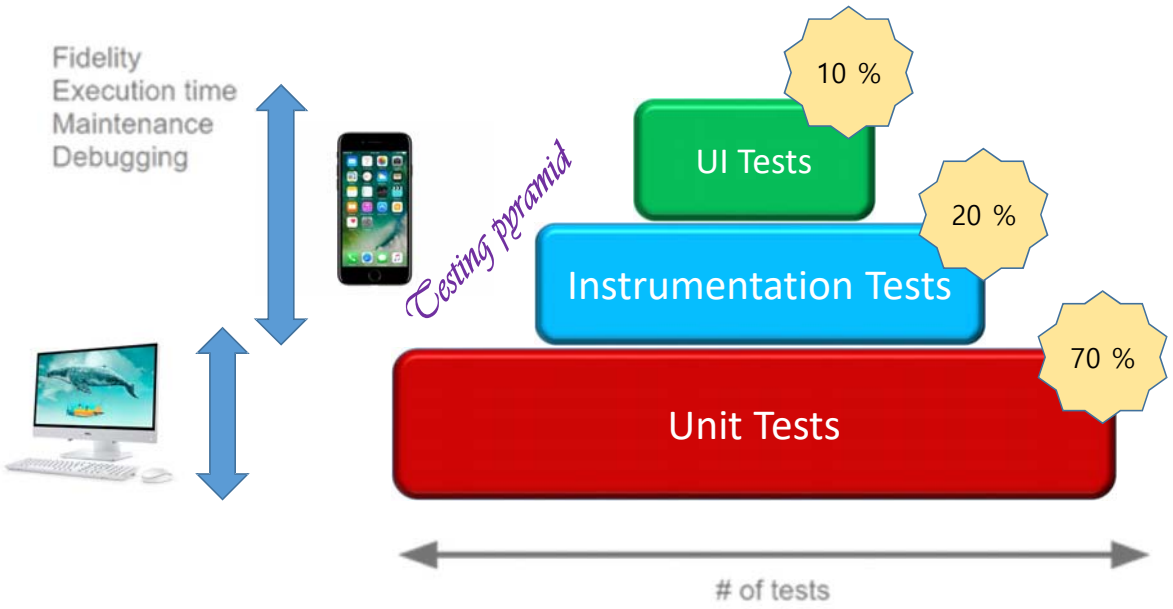
Types of Software Testing

- 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
- …

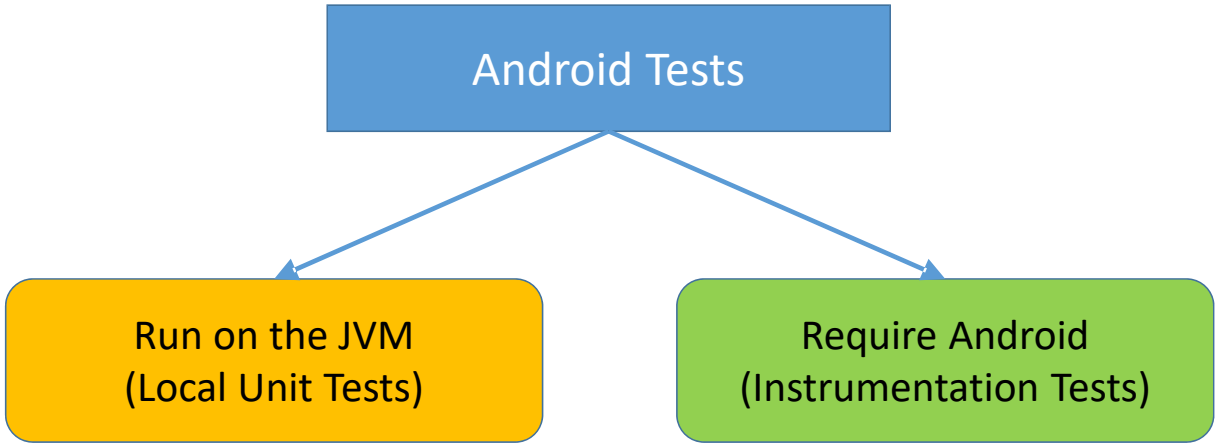
Different Kinds of Tests from Google: Before



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.

Unit Tests

- "local tests" or "local unit tests"
- Can run only on JVM
- Cannot test the code (e.g. UI) with Android Framework dependencies, such as activities, views, contexts, etc.
- If minimal dependencies on Android framework, you can use mocking framework.

Instrumentation Tests

- Run on a device or an emulator
- In the background, your app and a testing app that will control your app will be installed. The testing app lunches your app and runs UI tests as needed
- Can also be used to test none UI logic with a dependency on a context

"Instrumented Unit Tests"

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

AndroidX

- **AndroidX** is a major improvement to the **Android Support Library**.

- **AndroidX** fully replaces the Support Library.

Support Library class	AndroidX class
android.support.annotation.AnimatorRes	androidx.annotation.AnimatorRes
android.support.annotation.AnimRes	androidx.annotation.AnimRes

AndroidX includes the following features

- All packages in `AndroidX` live in a namespace starting with the `androidx`.
  - Support Library packages mapped into corresponding `androidx.*` packages.
  - See `https://developer.android.com/jetpack/androidx/migrate`
- Unlike the Support Library, AndroidX packages are separately maintained and updated.
  - Use strict Semantic Versioning starting with version 1.0.0.
  - Can update AndroidX libraries in your project independently.
- All new Support Library development will occur in the AndroidX library.
  - This includes maintenance of the original Support Library artifacts and introduction of new Jetpack components.

Why Write Tests?

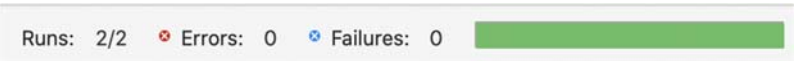
- To find the bug that may exist in our code?

- 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!



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”

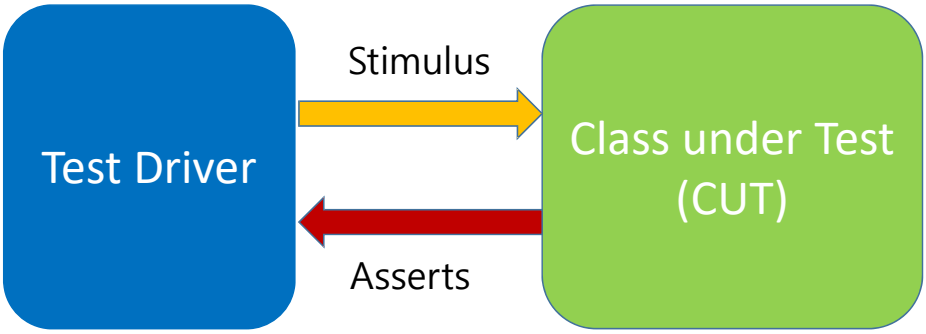
“Code without tests is bad code”

- Michael Feathers

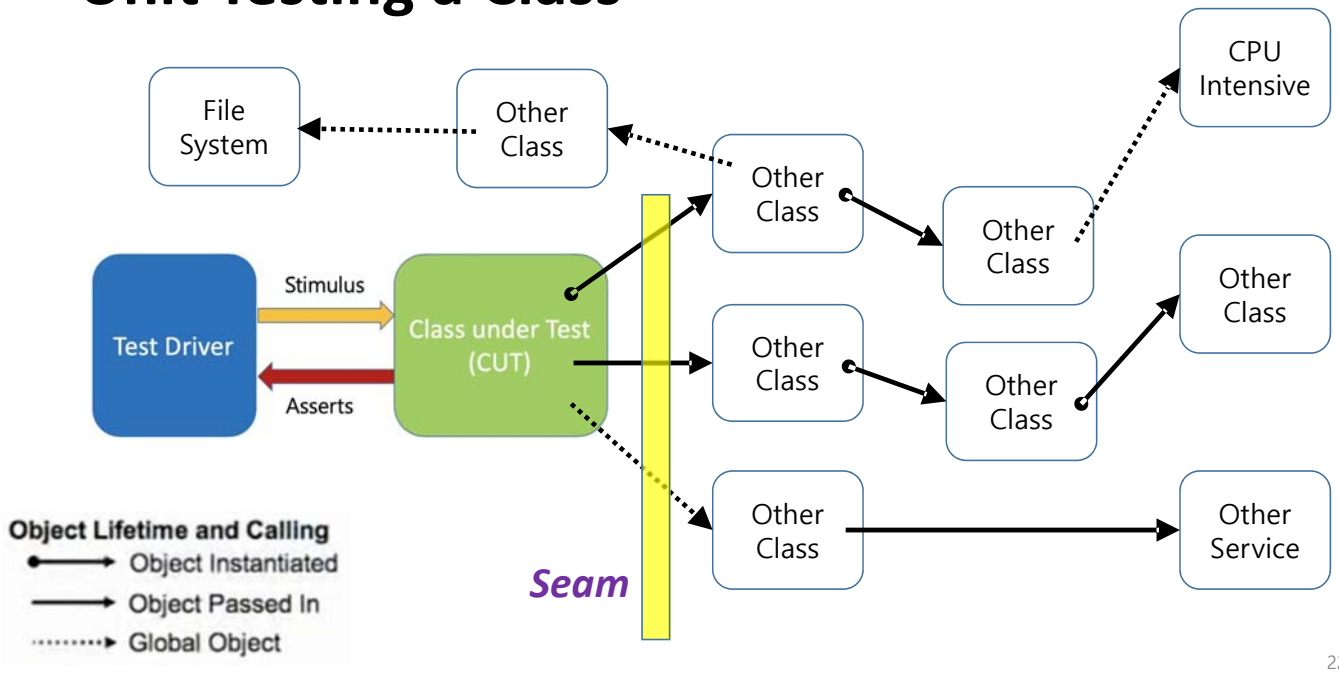
What is a unit?

- “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.

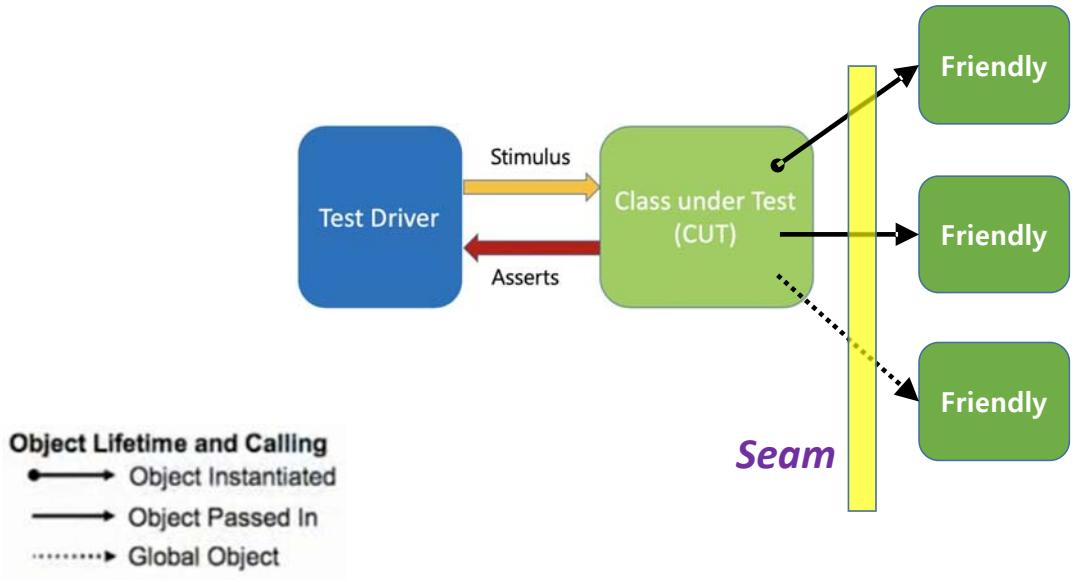
Unit Testing a Class



Unit Testing a Class



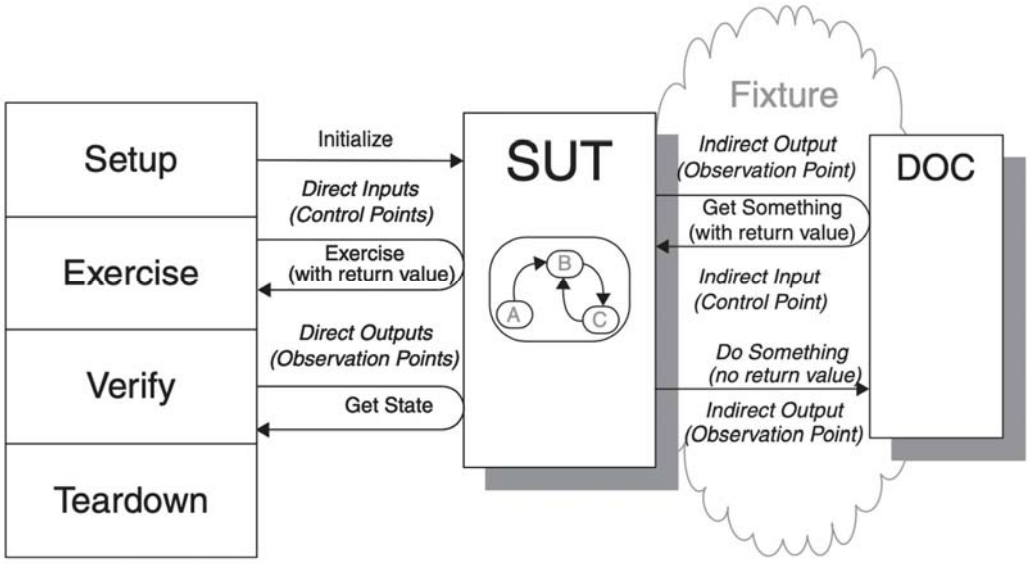
Unit Testing a Class



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)

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

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

```java
// 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'
}
```