TEST YOUR ANDROID APP !

MEDERIC.HURIER@UNI.LU

OUTLINE

- Motivation
- Unit Tests
- UI/Activity Tests
- Tutorial: Favorite

MOTIVATION

WHAT'S YOUR OPINION ?

- In which case should you test a software ?

  For every piece of code someone else will use or maintain

- When should you start writing your test ?

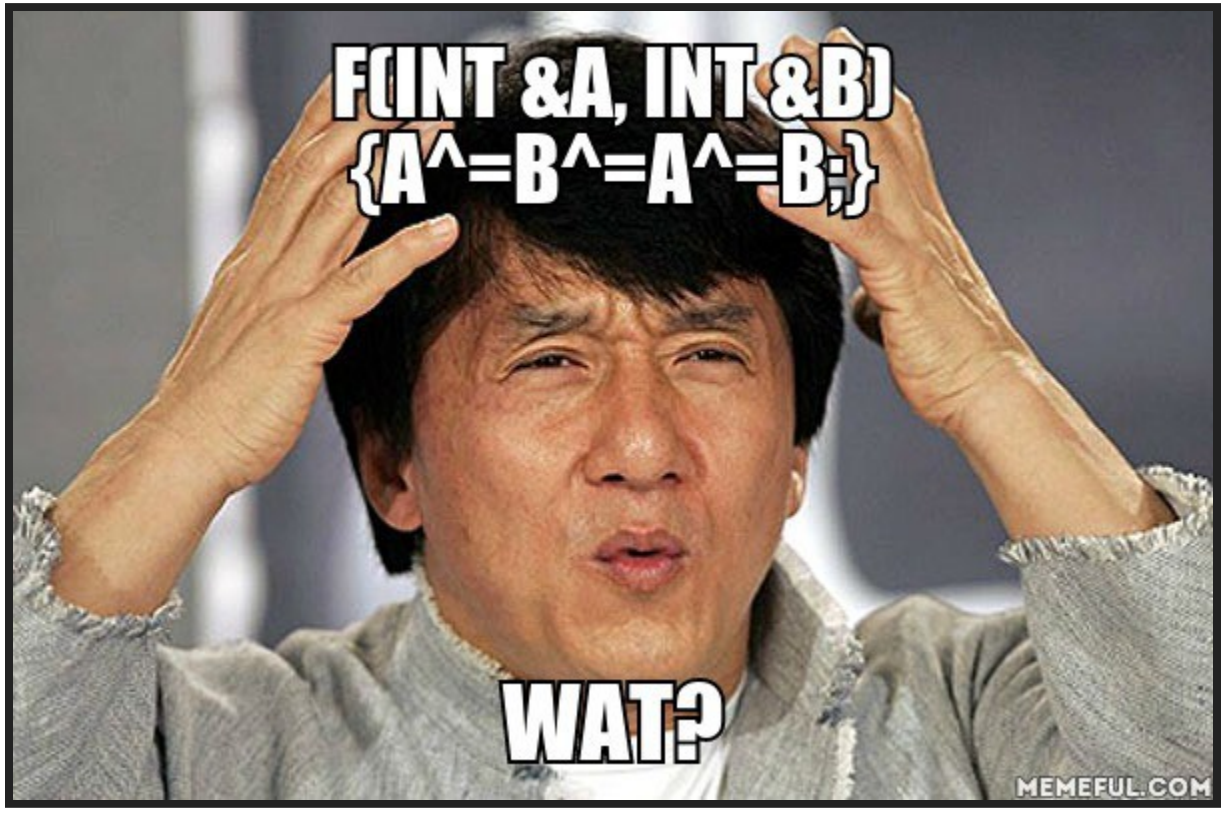
  Before your start coding (ideally) or the second after

- Who should test your app ?

  Yourself (unit test), your team (integration test)
  beta users (validation test) **but not your Final User !**

WHY ARE TEST SO IMPORTANT ?

WHAT DOES THIS FUNCTION DO (C) ?



FOLLOW BEST PRACTICES

Well-written tests can help you to catch bugs early in development and give you confidence in your code.

- Agile Software Development
- Test Driven Development (TDD)
- Android Best Practices for Test

During this course, we will review the last one together.

DEFINITIONS

Unit Tests generally test the smallest possible unit of code without dependencies on system or network resources.

Test Cases define a set of objects and methods to run multiple tests independently from each other.

Test cases can be organized into **Test Suites** and run programmatically, in a repeatable manner, with a **Test Runner** provided by a **Testing Framework**.

Test Fixtures consist of objects that must be initialized for running one or more tests

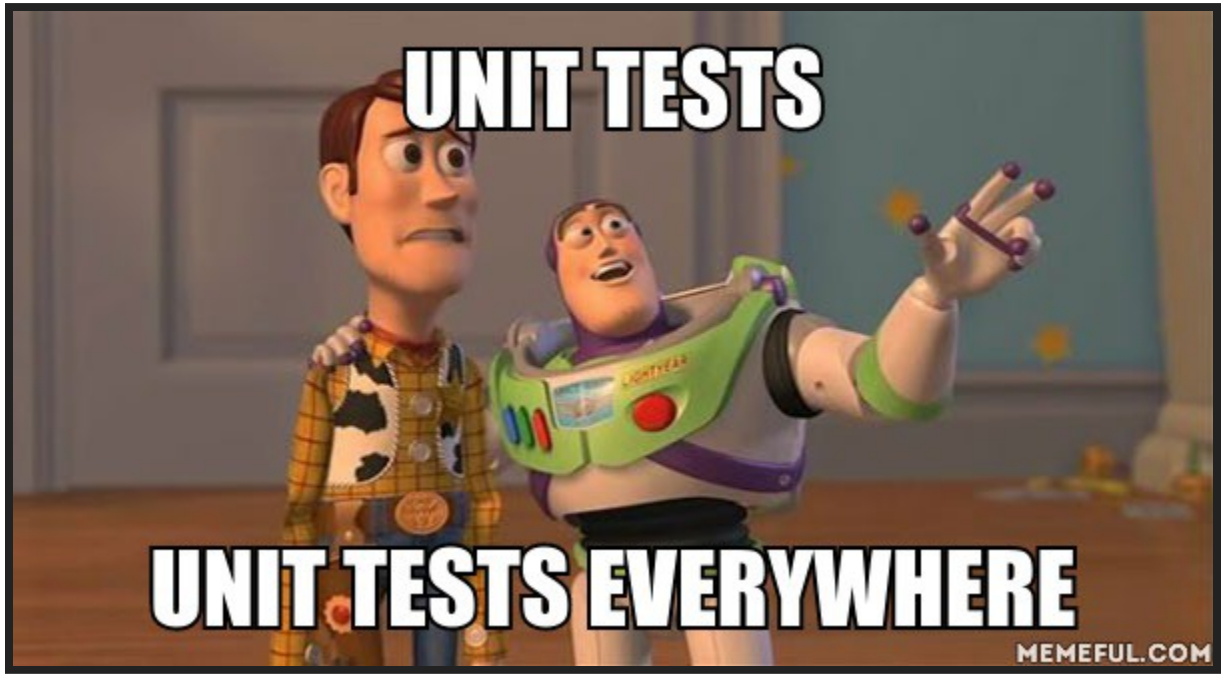
PRINCIPLES

In order to verify that there are no regressions in the functional behavior of your application, it's important to **create a test for each important Class of your project**.

For each test, you need to create the individual parts of a test case, including the *test fixture*, *preconditions test method*, and *test methods*.

You can then run your test to get a **Test Report**.

If any test method fails, this might indicate a defect.


UNIT TESTS
UNIT TESTS EVERYWHERE
MEMEFUL.COM

CONVENTIONS

Tests are written in a structured way:

- create a class per test case, and a test case for each class
- put your tests in a separate package, distinct from the code under test (e.g. androidTest or test)
- **package name** should have the same name as the package you want to test, suffixed with ".tests"
- **test case name** should have the same name as the class you want to test, suffixed with "Test"

ANDROID TOOLS

- **Android Testing Support Library**: provides a set of APIs that allow you to build and run test code for your apps.
  - **AndroidJUnitRunner**: JUnit 4-compatible test runner
  - **Espresso**: UI testing framework; use a programmatic approach for the functional testing within an app
  - **UI Automator**: UI testing framework; suitable for cross-app testing across system and installed apps
- **Monkey**: generates pseudo-random streams of user events such as clicks, touches, or gestures.

UNIT TESTS

TEST CASES

You should build unit tests when you need to verify the logic of some specific and independent code in your app.

For example, if you are unit testing a class, your test might check that the class is in the right state after a method call

Unit tests are not suitable for testing complex scenario, such as interaction with the User Interface !

Instead, you should use the UI testing frameworks

TEST CATEGORIES

Local tests: units tests that run on your local machine only to minimize execution time and resource usage.

Use this approach to run unit tests that have no dependencies on the Android framework or have dependencies that can be filled by using mock objects.

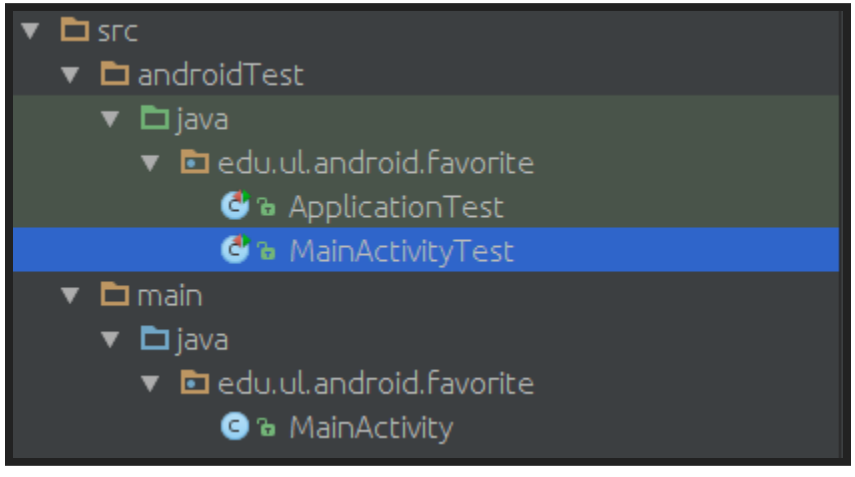
Instrumented tests: unit tests that run on an Android device or emulator. These tests have access to instrumentation information, such as the Context for the app under test.

Use this approach to run tests that have Android dependencies which cannot be filled by using mock objects.

TEST CREATION

You need to create a *Java Class* inside your test package.

Here is an example with *MainActivity*:



TEST DEFINITION

Use the decorator @Test to annotate **Test Methods**

```
import org.junit.*;

@RunWith(AndroidJUnit4.class)
public class TestFoobar {
    @Test
    public void testOneThing() {
        // Code that tests one thing
    }

    @Test
    public void testAnotherThing() {
        // Code that tests another thing
    }
}
```

@Test(timeout=<milliseconds>): Specifies a timeout period for the test. If the test starts but does not complete within the given timeout period, it automatically fails

PRE/POST EXECUTION

- **@Before**: will be invoked before each test
  - you can have multiple @Before methods
- **@After**: will be called after every test method
  - you can define multiple @After methods
- **@BeforeClass**: will be invoked once at initialization
  - your can have a single @BeforeClass methods
- **@AfterClass**: will be invoked once at finalization
  - your can have a single @AfterClass methods

Note: the order in which methods are called is not fixed.

```
import org.junit.*;

@RunWith(AndroidJUnit4.class)
public class TestFoobar {
    @BeforeClass
    public static void setUpClass() {}

    @Before
    public void setUp() {}

    @After
    public void tearDown() {}

    @AfterClass
    public static void tearDownClass() {}
}
```

PRECONDITION TEST

As a sanity check, it is good practice to verify that the test fixture has been set up correctly, and the objects that you want to test have been correctly instantiated or initialized.

```
public void testPreconditions() {
        assertNotNull(“activity is null”, activity);
        assertNotNull(“text is null”, text);
    }
```

By convention, the method for verifying your test fixture is called *testPreconditions()*

TEST ASSERTIONS

You can use **Test Assertions** to verify a specific condition.

- If the condition is true, **the test passes**.
- If the condition is false, the assertion method throws an **AssertionFailedError Exception**.
- In both cases, the test runner proceeds to run the other test methods in the test case.

Note: you can provide a string in the first argument of your assertion method to give some contextual details

ASSERT METHODS 1/2

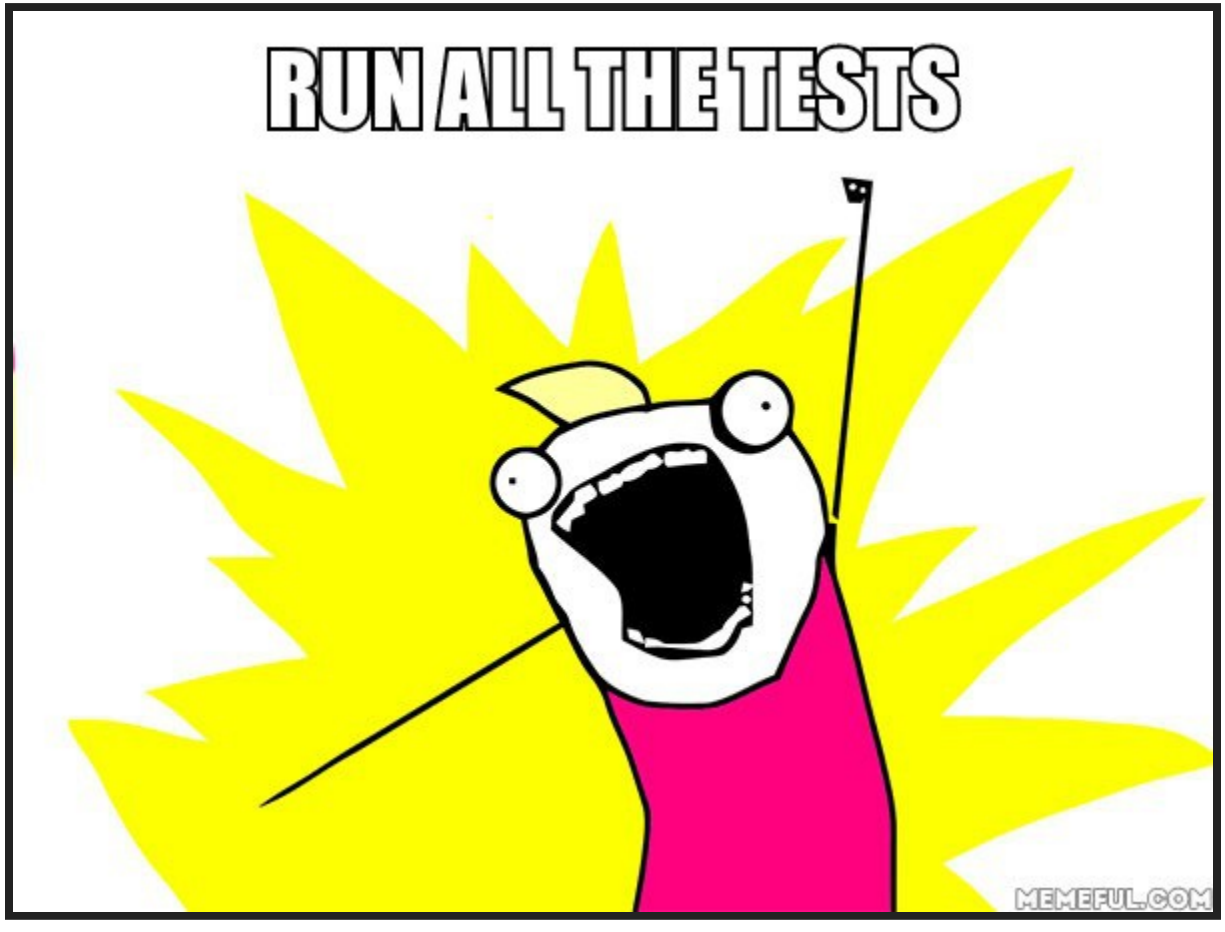
1. **void assertEquals(boolean expected, boolean actual)**
    - check that two primitives/Objects are equal
2. **void assertTrue(boolean expected, boolean actual)**
    - check that a condition is true
3. **void assertFalse(boolean condition)**
    - check that a condition is false
4. **void assertNotNull(Object object)**
    - check that an object isn't null.

ASSERT METHODS 2/2

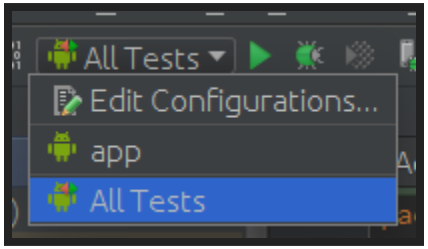
5. **void assertNull(Object object)**
    - check that an object is null
6. **void assertSame(boolean condition)**
    - check if two references point to the same object
7. **void assertNotSame(boolean condition)**
    - check if two references not point to the same object
8. **void assertArrayEquals(expectedArray, resultArray);**
    - check whether two arrays are equal to each other.

TEST EXECUTION

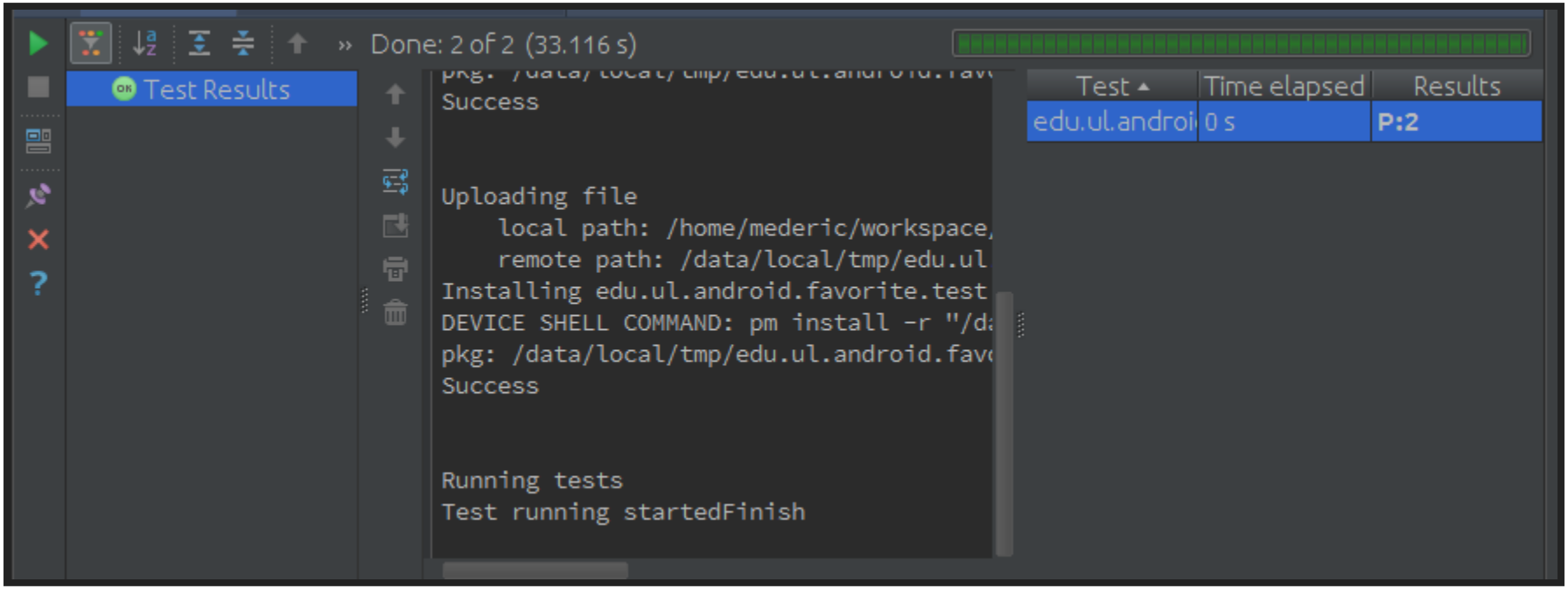
Right click on your test folder > Run tests in ...



You can then create a configuration to run all the tests.



And view the Test Result at the bottom of Android Studio



UI/ACTIVITY TESTS

WHAT IS UI TESTING ?

User Interface (UI) testing lets you ensure that your app meets its functional requirements and achieves a high standard of quality such that it is more likely to be successfully adopted by users.

ALTERNATIVES

One approach to UI testing is to simply have a human tester perform a set of user operations on the target app and verify that it is behaving correctly. However, this manual approach can be time-consuming, tedious, and error-prone.

A more efficient approach is to write your UI tests such that user actions are performed in an automated way. The automated approach allows you to run your tests quickly and reliably in a repeatable manner.

WHITE BOX VS BLACK BOX TESTING

White-Box Testing: you have the source code for the application that you want to test. The Android Instrumentation framework is suitable for creating white-box tests for UI components within an application.

Black-Box Testing: you may not have access to the application source. This type of testing is useful when you want to test how your app interacts with external apps.

FOR SINGLE APP

This type of test verifies that the target app behaves as expected when a user performs a specific action.

It allows you to check that the app returns the correct output in response to user interactions.

UI testing frameworks like **Espresso** allow you to programmatically simulate user actions.

https://developer.android.com/training/testing/ui-testing/espresso-testing.html

FOR MULTIPLE APPS

This type of test verifies the correct behavior of interactions between different user apps or between apps.

For example, you might want to test that your camera app shares images correctly with a 3rd-party social media app.

UI testing frameworks that support cross-app interactions, such as **UI Automator**, allow you to create such scenarios.

https://developer.android.com/training/testing/ui-testing/uiautomator-testing.html

UNIT TESTS FOR ACTIVITIES

To create a unit test for your Activity, use a @Rule decorator

```
@RunWith(AndroidJUnit4.class)
public class MainActivityTest {

    @Rule
    public ActivityTestRule<MainActivity> activity
        = new ActivityTestRule<>(MainActivity.class);

    @Test
    public void exampleTest() {
    }
}
```

MORE TEST ANNOTATIONS

Android annotations let you classify how long a test should take to determine how frequently the test should run.

@SmallTests (execution time: < 100ms)
these tests should be run very frequently

@MediumTests (execution time: < 2s)
these tests run at least on every check in of your code

@LargeTests (execution time: < 120s)
these tests should be run as often as practical (once a day)

SIMPLE EXAMPLE

You can add a test method like this to test your UI:

```
@Test
public void topAndBottomWidgetsAreDisplayed() {
    onView(withId(R.id.ip_label))
        .check(matches(isDisplayed()));
    onView(withId(R.id.hex_check))
        .check(matches(isDisplayed()));
}
```

This method checks that the two most extreme views are correctly displayed to the user.

TUTORIAL: FAVORITE

SCOPE

Included:

- Unit and UI/Activity Testing
- Mock of Android Framework
- Introduction to Monkey

Excluded:

- Service/Content Provider Testing
- Performance Testing

YOUR CHALLENGE

You have an existing Android App called "Favorite"

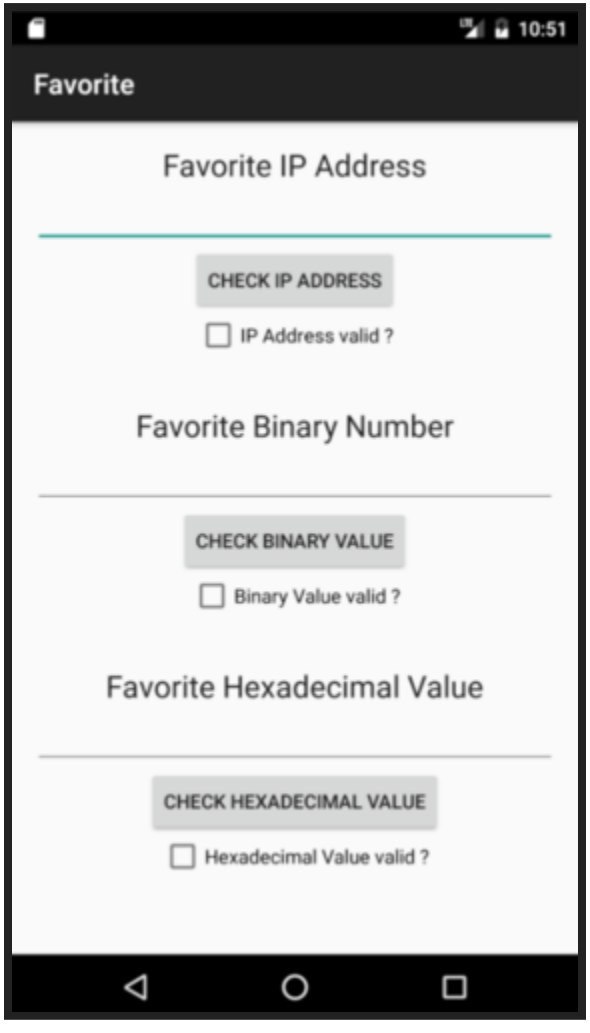
https://github.com/freaxmind/favorite

NOTE: This is an application for geeks only !

You have to register your favorite things, such as IP Address, Binary Numbers, and Hexadecimal Numbers

Unfortunately, there is no tests for this app. You must create them to improve its quality and maintainability

DEMO


10:51
Favorite
Favorite IP Address
CHECK IP ADDRESS
IP Address valid ?
Favorite Binary Number
CHECK BINARY VALUE
Binary Value valid ?
Favorite Hexadecimal Value
CHECK HEXADECIMAL VALUE
Hexadecimal Value valid ?

#1 INSTRUMENTED TESTING

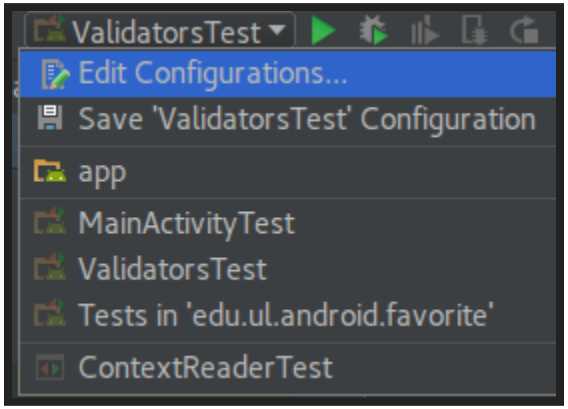
- Fill the methods in main Validators.java
- Complete the units test in ValidatorsTest.java
- Ask me to come and try to mess with your tests ;)

If your computer is not powerful enough to run an emulator, convert them to local tests and run them on your machine

(move the class from app/src/androidTest to app/src/test).

To run Instrumented Unit Tests in Android Studio:

1. Right-click on the file and select "Run ClassTest"
   - this works for the navigation menu and in the editor
   - you can also right click on a test method specifically



2. Then, you can run the same test from the top menu

#2 ACTIVITY TESTING

Open MainActivityTest.java and fill the test methods

- Check that the text inputs are empty by default
- Check that checkboxes are unchecked by default
- Check that checkboxes are not clickable by default
- Check the validation when a user submit an IP Address

#3 LOCAL UNIT TESTING

Open ContextReaderTest.java and fill the test methods

- Mock calls to Context methods from ContextReader
  - if it helps, look at the class imports and this link

Document for Mokito on: site.mockito.org

Migrate your test class for Validators to become local tests

- since the test are local, they should be faster to execute

#4 MONKEY TESTING

Run the following code in your shell:

```
adb shell monkey -p edu.ul.android.favorite -v 1500
```

Give me your feedback ;)

DOCUMENTATIONS

- https://www.udacity.com/courses/android
- https://developer.android.com/studio/test/index.html
- https://developer.android.com/training/testing/index.html