Automated Testing
In Android

236503
Dr. Oren Mishali
Based on the Official Android Development Guide

Android testing is based on JUnit

• JUnit is a testing framework for Java initially developed by Kent Beck and Erich Gamma
• JUnit is linked as a JAR at compile-time (junit.framework for JUnit 3.8, org.junit for JUnit 4 and later)
• (We will use JUnit 4)
• A survey performed in 2013 across 10,000 GitHub projects found JUnit as the mostly commonly included library (30.7% of the projects)
• JUnit is strongly related with test-driven development (TDD)
JUnit annotations

- JUnit uses Java annotations to mark specialized methods
- `@Test` marks a test method (test)
- `@Before` and `@After` mark methods that execute before/after each test
- `@BeforeClass` and `@AfterClass` mark methods that execute only once: before/after all tests
- `@Ignore` allows to disable a test or a group of tests
- Some annotations may have parameters, e.g., `@Test(timeout=100)` causes a test to fail if it takes longer than 100 milliseconds

Note: JUnit 3 did not use annotations and was based on naming conventions for specialized methods and on extending a `TestCase` class

From https://en.wikipedia.org/wiki/JUnit

@Test annotation and assertions

- `@Test` marks a test method
- A single test class usually contains multiple test methods
- Within a test method, you make assertions about the correctness of your code
- The JUnit `Assert` class provides many static assertion methods that you will use a lot
- An assertion method compares an actual value to an expected value, and throws an `AssertionException` if the comparison fails
- For better code clarity, use static import: `import static org.junit.Assert.*;`

Some popular assertion methods:

- `assertEquals(String expected, String actual)`
- `assertNull(Object object)`
- `assertTrue(boolean condition)`

** Note that each assertion method is overloaded and has many parameter variations.

** See the list of all methods [here](https://en.wikipedia.org/wiki/JUnit)
An example unit test

This is the code to be tested...

```java
public class Calculator {
    public int evaluate(String expression) {
        int sum = 0;
        for (String summand: expression.split("\+"))
            sum += Integer.valueOf(summand);
        return sum;
    }
}
```

From Code is taken from https://github.com/junit-team/junit4/wiki/Getting-started

And here is a possible unit test

```java
import static org.junit.Assert.assertEquals;
import org.junit.Test;

public class CalculatorTest {
    @Test
    public void evaluatesExpression() {
        Calculator calculator = new Calculator();
        int sum = calculator.evaluate("1+2+3");
        assertEquals(6, sum);
    }
}
```

- In TDD, the unit test should be created **before** the code

Common types of tests for android

- **Local unit tests**
  - Run on the local machine only
  - The fastest approach
  - Use this approach to run unit tests that have no dependencies on the Android framework
  - Or have dependencies that mock objects can satisfy

- **Instrumented unit tests**
  - Run on an Android device or emulator
  - Use this approach to run unit tests that have Android dependencies, e.g., need the Context

- **Instrumented integration (functional) testing**
  - The test verifies that the target app behaves as expected when a user performs a specific action
  - UI testing frameworks like Espresso allow you to programmatically simulate user actions
Android basic test organization

- Within an app, Android Studio creates two test folders: `src/test/` and `src/androidTest/`
- Local unit test should be placed in the `src/test/` folder
- Instrumented tests should be placed within `src/androidTest/`
- Gradle uses `src/androidTest/` when generating the test APK that is used to test the app
- See here for more details and possible configurations

Running local unit tests with JUnit & Mockito

- Use this option if your unit test has no dependencies or only has simple dependencies on Android
- This testing approach is efficient and execution time is greatly reduced
Writing local unit tests

• Local unit test are written as regular JUnit tests
• But first configure the testing dependencies in the app’s top-level build.gradle file:

```groovy
dependencies {
    // Required -- JUnit 4 framework
testCompile 'junit:junit:4.12'
    // Optional -- Mockito framework
testCompile 'org.mockito:mockito-core:1.10.19'
}
```

Mockito is a mocking framework for Java (see next slide)

Recall: local unit tests should be created under the “test” package

Recall: when you change the build file, you should sync your project by clicking Sync Now in the notification bar, or from the toolbar icon

Why mock objects are needed?

• By default, Android executes local unit tests against a modified version of android.jar
• The modified jar does not contain any actual code, and method calls to Android classes from your unit test throw an exception
• By using a mocking framework (e.g., Mockito), stubs may replace those external dependencies
• In other words, (simple) Android dependencies may be replaced with mock objects
How to define mock objects in Mockito?

1. Include the Mockito library dependency in build.gradle
2. Add the `@RunWith(MockitoJUnitRunner.class)` annotation to the test class
3. To create a mock object, add the `@Mock` annotation before the field declaration
4. Specify the stubs using the methods `when()` and `thenReturn()`

An example unit test that uses a mock object

```java
import static org.hamcrest.MatcherAssert.assertThat;
import static org.hamcrest.Matchers.*;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.mockito.Mock;
import org.mockito.runners.MockitoJUnitRunner;
import android.content.SharedPreferences;

@RunWith(MockitoJUnitRunner.class)
public class UnitTestSample {
    private static final String FAKE_STRING = "HELLO WORLD";

    @Mock
    Context mMockContext;

    @Test
    public void readStringFromContext_LocalizedString() {
        // Given a mocked Context injected into the object under test...
        when(mMockContext.getString(R.string.hello_world)).thenReturn(FAKE_STRING);
        ClassUnderTest myObjectUnderTest = new ClassUnderTest(mMockContext);

        // ...when the string is returned from the object under test...
        String result = myObjectUnderTest.getString();

        // ...then the result should be the expected one.
        assertThat(result, is(FAKE_STRING));
    }
}
```

- In the example, the object that we would like to test must be created with a `Context`
- Calling the “real” Context will result in an Exception
- We therefore define a mock Context object

For more details, see the Mockito API reference
Running local unit tests

- Right-click the unit test class and select Run
- To run all tests in a directory, right-click the directory and click Run
- Tests located under src/test may be executed by gradle’s test task

Automating User Interface Tests with JUnit & Espresso

https://developer.android.com/training/testing/ui-testing/index.html
One approach to UI testing is manual testing

• I.e., 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 automated UI tests, allowing you to run your tests quickly and reliably in a repeatable manner

In any case, do not underestimate manual testing!

Two types of automated UI tests

• UI tests for a single app
  • We will focus on this type of tests
  • We will use the Espresso framework to programmatically simulate user actions and test complex user interactions
• UI tests that span multiple apps
  • E.g., to test that your camera app shares images correctly with a 3rd-party social media app
  • The UI Automator framework allows to create tests for such scenarios

We will cover the basics and leave the rest for self reading
Configuring the project

- Test source code should be placed in `src/androidTest`
- **Setup the Android Testing Support Library**
  - Use the SDK manager for that as explained in the link
  - The library includes a JUnit 4 test runner and APIs for functional UI tests ([Espresso](https://developer.android.com/training/testing/espresso) and [UI Automator](https://developer.android.com/training/testing/automator))
- Configure your Android testing dependencies in your app’s top-level `build.gradle` (next slides)
- Specify `AndroidJUnitRunner` as the default test instrumentation runner (next slides)

### Configuring build.gradle

```groovy
dependencies {
    androidTestCompile 'com.android.support:support-annotations:23.0.1'
    androidTestCompile 'com.android.support.test:runner:0.4.1'
    androidTestCompile 'com.android.support.test:rules:0.4.1'
    // Optional -- Hamcrest library
    androidTestCompile 'org.hamcrest:hamcrest-library:1.3'
    // Optional -- UI testing with Espresso
    androidTestCompile 'com.android.support.test.espresso:espresso-core:2.2.1'
    // Optional -- UI testing with UI Automator
    androidTestCompile 'com.android.support.test.uiautomator:uiautomator-v18:2.1.1'
}

android {
    defaultConfig {
        testInstrumentationRunner "android.support.test.runner.AndroidJUnitRunner"
    }
}
```

Note: the inclusion of the [Hamcrest](https://hamcrest.org/) library is recommended. The library allows for creating more flexible assertions using the Hamcrest matcher APIs.

Specifying `AndroidJUnitRunner` as the default test instrumentation runner
Using Espresso – additional setup

• leaving system animations turned on in the test device might cause unexpected results or may lead to test failures
• Therefore you should **turn off animations** on your test device
• Open your device’s Setting => Developer Options, and turn the following options off
  • Window animation scale
  • Transition animation scale
  • Animator duration scale

An example Espresso test

*ActivityTestRule* – the framework launches the activity under test before each test method, and shut it down after the test finishes

Here the actual **test scenario** takes place. Key players are methods **onView**, **perform**, and **check**

```java
@RunWith(AndroidJUnit4.class)
@LargeTest
class ChangeTextBehaviorTest {

    private String mStringToBeTyped;

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

    @Before
    public void initValidString() {
        // Specify a valid string.
        mStringToBeTyped = "Espresso";
    }

    @Test
    public void changeText_sameActivity() {
        // Type text and then press the button.
        onView(withId(R.id.editTextUserInput)).
          perform(typeText(mStringToBeTyped), closeSoftKeyboard());
        onView(withId(R.id.changeTextBtn)).perform(click());

        // Check that the text was changed.
        check(matches(withText(mStringToBeTyped)));
    }
    }
}
Espresso – self reading

• The Espresso framework has great documentation
• In addition, take a look at an answer from stackoverflow that discusses the use of Espresso for testing multiple activities