App Development
in Android Studio

HÁSKÓLINN Í REYKJAVÍK
REYKJAVIK UNIVERSITY

Lab 6: Testing

December 7, 2017

Jón Steinn Elíasson

jonsteinn@gmail.com

Contents

1 Android testing

This course assumes knowledge of unit testing with JUnit so we will not go over that in much detail. In Android there are two main categories of tests. Those that can run on a JVM (and thus, from our developing machine) and those that require Android to run. Local tests are much faster so the general rule is to only use the Android tests if we have to. You should also see how the app displays on various devices. It is good to deploy the app on a device with the lowest and highest supported Android versions and on devices with low and high resolutions.

The following folder structure for tests is used in Android Studio.

- `app/src/main/java` - The source code of our app.
- `app/src/test/java` - The source code of our tests that run on a JVM.
- `app/src/androidTest/java` - The source code of our tests that run on Android.

There are specific dependencies for both types of tests in our `build.gradle`.

```
testCompile '...'
androidTestCompile '...'
```

The following shows a non instantiable class with a single method.

```
public final class Division {
    private Division() { }
    public static double quotient(int a, int b) {
        if (b == 0) throw new ArithmeticException();
        return a / (double)b;
    }
}
```

A typical test for the class is shown below.

```
public class DivisionTest {
    private static final double delta = 1E-10;

    @Test
    public void quotientTest() throws Exception {
        assertEquals(Arithmetic.quotient(1, 3), 0.3333333333333, delta);
        assertEquals(Arithmetic.quotient(42, 3), 14.0, delta);
        assertEquals(Arithmetic.quotient(-15, -5), 3.0, delta);
        assertEquals(Arithmetic.quotient(66, 5), 13.2, delta);
        assertEquals(Arithmetic.quotient(0, 123124), 0.0, delta);
    }

    @Test(expected = ArithmeticException.class)
    public void quotientExceptionTest() throws Exception {
        Arithmetic.quotient(5, 0);
    }
}
```

Suppose we have a resource string `'Some resource string'` in an id `'test_string'`, then the following (although rather useless) demonstrates a test that requires Android to run.

```
@RunWith(AndroidJUnit4.class)
public class DemoTest {
    @Test
    public void androidTestExample() throws Exception {
        Context appContext = InstrumentationRegistry.getTargetContext();
        String resString = appContext.getResources().getString(R.string.test_string);
        assertEquals("Some resource string", resString);
    }
}
```

We can use ActivityTestRule to test activities, provided by the Android testing support library. In the following example, the activity is lunched before each test and terminate once its complete (or manually terminated during the test).

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

    @Rule
    public final ActivityTestRule<MainActivity> mActivity =
            new ActivityTestRule<>(MainActivity.class, true, false);

    @Before
    public void setUp() {
        mActivity.launchActivity(new Intent());
    }

    @Test
    public void androidTest() throws Exception {
        Activity activity = mActivity.getActivity();
        activity.findViewById(/* Some id */);
        // some assertions
    }
}
```

2 Mockito

Mockito is a very powerful library that provides test doubles so a specific functionality can be isolated and tested in each test. To use it in Android Studio, you should add the following dependencies.

```
testCompile 'org.mockito:mockito-core:2.12.0'
androidTestCompile 'org.mockito:mockito-core:2.12.0'
```

It can even be used to mock the functionality of Android APIs on the tests that run on a JVM.

```
@RunWith(MockitoJUnitRunner.class)
public class MockDemo {
    @Mock
    Context context;
    @Mock
    Resources resource;
    @Test
    public void testDemo() {
        // When getResources() is called for our mocked
        // object, we return our other mocked object
        when(context.getResources()).thenReturn(resource);
        // When getString() is called with this
        // specific id, then "What I want" is returned
        when(resource.getString(R.string.test_string)).thenReturn("What I want");
        assertEquals("What I want",
            context.getResources().getString(R.string.test_string));
    }
}
```

In the provided example we have two classes. One class has a method that takes long to execute and the other class keeps and instance of first one and uses its method in a method. We use Mockito to mock the expensive method when testing the other one. The source code can be found here and a programming session here.

3 Room database testing

Suppose we have a database called `MyDatabase`, then we can create an in memory database for testing our dao queries. The database query tests belong to the tests performed on Android.

```
@RunWith(AndroidJUnit4.class)
public class DaoTest {
    @Rule
    public InstantTaskExecutorRule instantTaskExecutorRule = new
       InstantTaskExecutorRule();

    private MyDatabase mDatabase;

    @Before
    public void initDb() throws Exception {
        mDatabase = Room.inMemoryDatabaseBuilder(
                InstrumentationRegistry.getContext(),
                MyDatabase.class)
                .allowMainThreadQueries()
                .build();
        // TODO: Add fake data
    }

    @After
    public void closeDb() throws Exception {
        mDatabase.close();
    }

    // TODO: add tests
}
```

In the provided example we have a very simple database of users with nothing but an id and name and some possibles queries that we test. The source code is available here and a programming session here.

4 Espresso

Espresso is a UI testing framework where you can automate interaction with views based on ids or text. The following shows an example of a button click on a button with id `my_btn` and matching of text on a text view with id `my_txt`.

```
onView(withId(R.id.my_view)).perform(click());
onView(withId(R.id.my_txt)).check(matches(withText("TEXT")));
```

There are multiple matchers, actions and assertions available but we will suffice providing a link with a cheat sheet here, rather than to go over each individually. We can also write custom matchers.

One problem we might find us in when creating UI tests is that our interaction talks to web services or databases which we do not want during testing. We therefore need to replace these calls with mocked versions. The provided example does this for a web service call, shared preferences, Room database and Firebase database. It also includes one custom matcher. The source code can be found here and here and a programming session here {I - II - III - IV - V}.

5 Assignment

Add tests to this project.

- [25%] Unit test the `BinaryConverter` class. For full points you must have full coverage excluding the private constructor.
- [25%] Test the database queries. For full points, every query must be tested.
- [50%] Test UI. For full points, every possible interaction (including invalid inputs) must be tested and while doing so, you must not use real data from either the database or the web service.