Android Unit Testing

Max Wayne

Useful Guides

- Local Unit Tests
  - https://developer.android.com/training/testing/unit-testing/local-unit-tests#java
  - Mock test sample code: https://github.com/android/testing-samples/tree/main/unit/BasicSample
- Instrumented Unit Tests
  - https://developer.android.com/training/testing/unit-testing/instrumented-unit-tests
  - Espresso UI Unit Tests
    - https://developer.android.com/training/testing/espresso/basics
    - https://developer.android.com/training/testing/ui-testing/espresso-testing
- Github link to demo project: https://github.com/maxwell-wayne-17/Android_Testing_Project

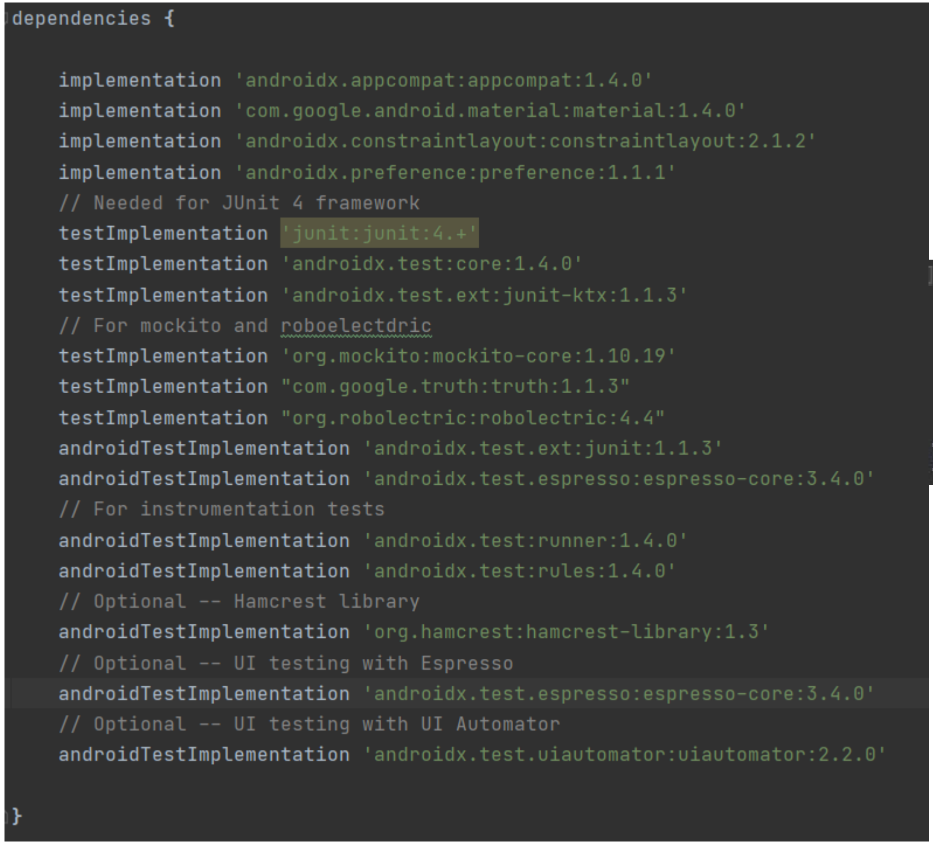
Types of Unit Tests

- Local Unit Tests
  - Used for testing business logic
  - Much faster
  - Runs locally, not on emulator or test device

- Instrumentation Tests
  - Run on emulator or test device
  - Capable of testing framework dependencies
  - Much slower

- By default, Android Studio will provide a package and example of each test in the project

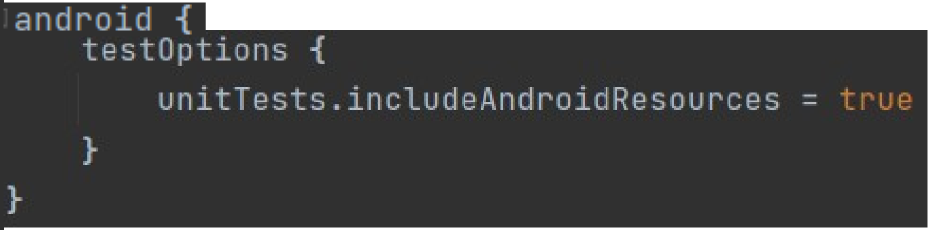
Dependencies (gradle app file)



```
dependencies {

    implementation 'androidx.appcompat:appcompat:1.4.0'
    implementation 'com.google.android.material:material:1.4.0'
    implementation 'androidx.constraintlayout:constraintlayout:2.1.2'
    implementation 'androidx.preference:preference:1.1.1'
    // Needed for JUnit 4 framework
    testImplementation 'junit:junit:4.+'
    testImplementation 'androidx.test:core:1.4.0'
    testImplementation 'androidx.test.ext:junit-ktx:1.1.3'
    // For mockito and roboelectdric
    testImplementation 'org.mockito:mockito-core:1.10.19'
    testImplementation "com.google.truth:truth:1.1.3"
    testImplementation "org.robolectric:robolectric:4.4"
    androidTestImplementation 'androidx.test.ext:junit:1.1.3'
    androidTestImplementation 'androidx.test.espresso:espresso-core:3.4.0'
    // For instrumentation tests
    androidTestImplementation 'androidx.test:runner:1.4.0'
    androidTestImplementation 'androidx.test:rules:1.4.0'
    // Optional -- Hamcrest library
    androidTestImplementation 'org.hamcrest:hamcrest-library:1.3'
    // Optional -- UI testing with Espresso
    androidTestImplementation 'androidx.test.espresso:espresso-core:3.4.0'
    // Optional -- UI testing with UI Automator
    androidTestImplementation 'androidx.test.uiautomator:uiautomator:2.2.0'

}
```

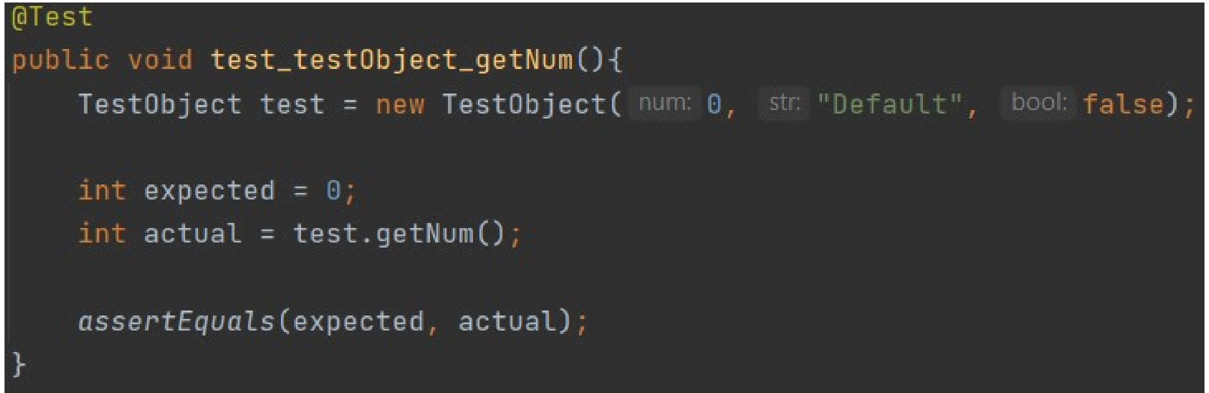


```
android {
    testOptions {
        unitTests.includeAndroidResources = true
    }
}
```

Basic Unit Test Recipe

1. Create class to hold unit tests that includes required imports

2. Declare methods that return void with no parameters and have @Test annotation
   a. Declare expected value
   b. Retrieve actual value from business logic
   c. Assert that actual value is what it is expected to be



```java
@Test
public void test_testObject_getNum(){
    TestObject test = new TestObject( num: 0,  str: "Default",  bool: false);

    int expected = 0;
    int actual = test.getNum();

    assertEquals(expected, actual);
}
```

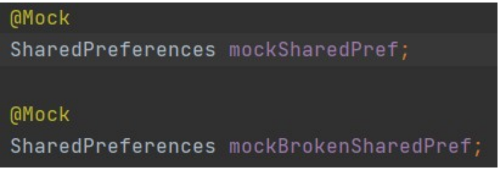
Local Mock Tests

- Allows testing of specific objects, including Android dependencies
- Isolates tests from the rest of the Android system
- Can verify the correct methods in those dependencies are called
- Essentially enables fake method calls from an uninstantiated object
    - Hard code the value you expect to be returned from a method
        - Get expected result, without actually doing any work

Mock Object Recipe Using Mockito Framework

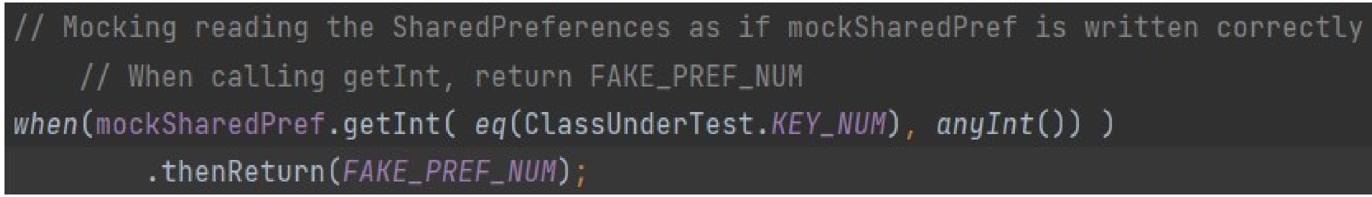
- Declare object field with @Mock annotation
    - Usually some object that has Android dependency (SharePreferences, Context, etc…)

- *when*(<MockObjName>.<desiredMethod>(*eq(<specific param>,anyString()..)*
  .thenReturn(<ExpectedReturnValue>)



```
@Mock
SharedPreferences mockSharedPref;

@Mock
SharedPreferences mockBrokenSharedPref;
```



```
// Mocking reading the SharedPreferences as if mockSharedPref is written correctly
    // When calling getInt, return FAKE_PREF_NUM
when(mockSharedPref.getInt( eq(ClassUnderTest.KEY_NUM), anyInt()) )
        .thenReturn(FAKE_PREF_NUM);
```

Instrumented Unit Tests

- Allows developer to legitimately test complex interactions with Android framework
- Allows developer to test against behavior of a real device
- Follows same basic unit test recipe

Test UI Using Espresso Framework

- Requires device behavior -> instrumented test
- Must turn animations off on device or emulator
  - Settings app -> Accessibility -> Toggle “Remove animations”
- Simulates user action on UI
  - Clicks
  - Swipes
  - Enter text
  - Etc...

Espresso Test Recipe

1. Find the UI component you want to test in an *Activity* by calling the onView() method (or the onData() method for *AdapterView*)
   a. Use ActivityScenarioRule to launch corresponding activity
2. Simulate a specific user interaction to perform on that UI component by calling ViewInteraction.Perform() or DataInteraction.perform() method and passing in the user action
   a. Actions can be chained using a comma-separated list in the method argument
3. Repeat steps as necessary
   a. Can simulate user flow across multiple activities in target app
4. Use the *ViewAssertions* methods to check that the UI reflects the expected state or behavior after these user interactions are performed

** Heavily referenced https://developer.android.com/training/testing/ui-testing/espresso-testing

Espresso with ActivityScenerioRule

- Launches the activity before each test method annotated with @Before and @Test, then shuts down activity and runs test methods annotated with @After

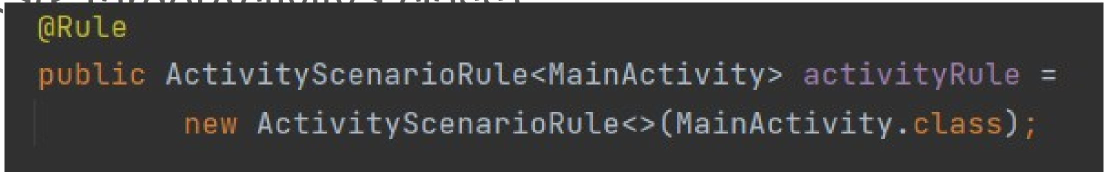
- Test class needs @RunWith(AndroidJUnit4.class) annotation

- To establish ActivityScenarioRule:

@Rule

public ActivitySceneriaRule<TargetActivity> activityRule

= new ActivityScenarioRule<>(<TargetActivity>.class);



```
@Rule
public ActivityScenarioRule<MainActivity> activityRule =
        new ActivityScenarioRule<>(MainActivity.class);
```

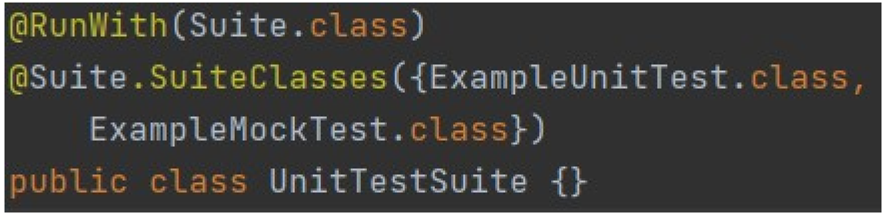
Test Suites

- Allows developer to run multiple test classes at once

1. Create package in testing package with .suite suffix (by convention)

2. Create java class

3. Add @RunWith(Suite.class) annotation
    a. Add @Suite.SuiteClasses({TestClass1.class, TestClass2.class, ... TestClassN.class}) annotation



```
@RunWith(Suite.class)
@Suite.SuiteClasses({ExampleUnitTest.class,
    ExampleMockTest.class})
public class UnitTestSuite {}
```