IT2010 – Mobile Application Development
BSc (Hons) in Information Technology
2nd Year
Faculty of Computing
SLIIT
2023 - Tutorial

Mobile App Testing

it's important to note that testing is an integral part of software development, especially in the mobile app development industry, where user experience plays a significant role. Mobile application testing ensures the quality and reliability of the app, which ultimately impacts user satisfaction and retention.
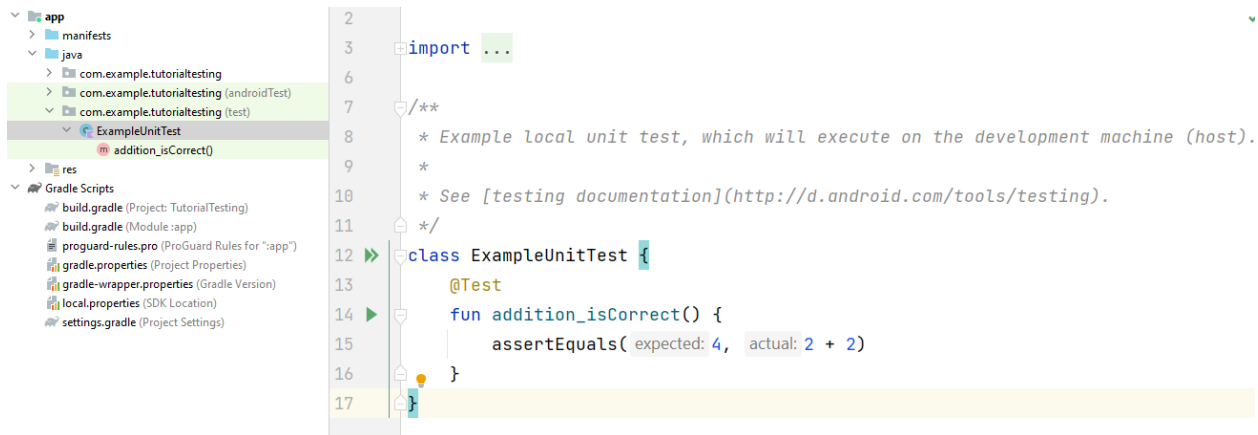
There are different types of testing methods in Android, such as unit testing, integration testing, functional testing, and acceptance testing. In this tutorial, we will cover some basic testing techniques using Kotlin and Android Studio.

1. Start a new project in Android Studio
2. In the app Gradle file make sure that following lines are there. Usually they are added by default when you create a project

```
testImplementation 'junit:junit:4.13.2'
androidTestImplementation 'androidx.test.ext:junit:1.1.5'
androidTestImplementation 'androidx.test.espresso:espresso-core:3.5.1'
```

Unit tests

1. Observe the Example Unit test that is given in the project by default


```kotlin
import ...

/**
 * Example local unit test, which will execute on the development machine (host).
 *
 * See [testing documentation](http://d.android.com/tools/testing).
 */
class ExampleUnitTest {
    @Test
    fun addition_isCorrect() {
        assertEquals(expected: 4, actual: 2 + 2)
    }
}
```

2. Create a new class named Calculations in the package you can find the MainActivity
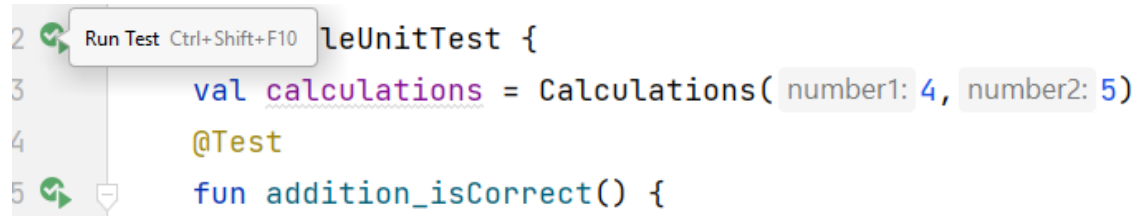
3. Implement the following functions

```
class Calculations(
    private val number1: Int,
    private val number2: Int
) {

    fun addition() = number1 + number2;
    fun subtraction() = number1 - number2;
    fun multiplication() = number1 * number2;
    fun division() = number1 / number2;
}
```

4. Now modify the ExampleUnitTest to test all these functions

```
class ExampleUnitTest {
    val calculations = Calculations(4,5)
    @Test
    fun addition_isCorrect() {
        assertEquals(9, calculations.addition())
    }
}
```
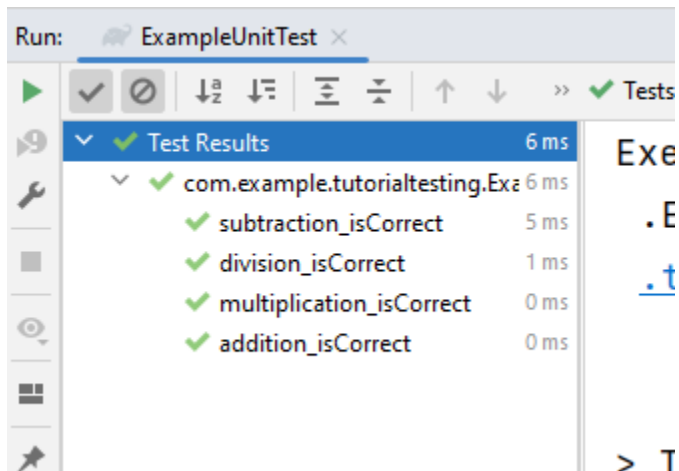
5. Add the other three tests to check the subtraction, multiplication, and division
6. Run the Unit test and observe the output
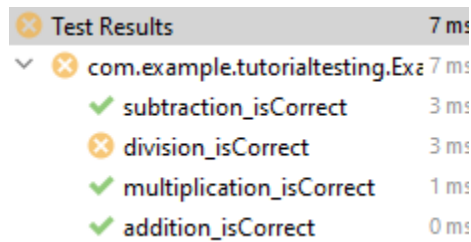
```
2   Run Test Ctrl+Shift+F10  leUnitTest {
3                val calculations = Calculations( number1: 4,  number2: 5)
4            @Test
5            fun addition_isCorrect() {
```

If all the tests passed it should output the following

7. Now do some changes in the Calculation class and observe how Unit tests fail

| Test Results	7 ms
com.example.tutorialtesting.Exa	7 ms
✔ subtraction_isCorrect	3 ms
✘ division_isCorrect	3 ms
✔ multiplication_isCorrect	1 ms
✔ addition_isCorrect	0 ms

In this division was calculated using the %(modulus)

Best practices when writing unit tests

- Keep your tests small and focused: Each test should focus on testing a single unit of functionality. Avoid testing too many things at once, as it can make it difficult to pinpoint the root cause of a test failure.
- Use test doubles: When writing unit tests, use test doubles (ex: mocks, stubs, fakes) to isolate the unit of code being tested from its dependencies. This makes it easier to test the code in isolation and helps to identify problems with specific components.
- Use descriptive test names: Use descriptive test names that accurately describe what the test is testing. This makes it easier to understand the intent of the test and helps to identify issues more quickly.
- Test edge cases: Make sure to test edge cases (ex: empty input, maximum/minimum input values) to ensure that the code works as expected in all scenarios.
- Use assertions: Use assertions to verify that the code being tested behaves as expected. This helps to identify problems with the code quickly and accurately.
- Use a consistent naming convention: Use a consistent naming convention for your tests and test classes to make it easier to navigate your test code.
- Write maintainable tests: Write tests that are easy to maintain, update, and refactor as the codebase evolves. Avoid creating tests that are too tightly coupled to the implementation details of the code being tested.
- Run tests regularly: Run your tests regularly to catch issues early in the development process. Make sure to run your tests after making changes to the code to ensure that everything is still working as expected.

Writing Instrumentation Tests

1. Modify the app by introducing a button.
2. Change the text view text to "Click the button"
3. Once the user clicks the button, change the text to "Hello, User!"
   a. Button – btnClick
   b. TextView - tvMessage
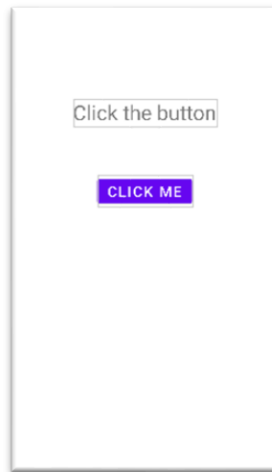
4. Open the ExampleInstumentedTest file and modify as follows

```kotlin
fun useAppContext() {
    // Context of the app under test.
    val appContext = InstrumentationRegistry.getInstrumentation().targetContext
    assertEquals("com.example.tutorialtesting", appContext.packageName)
}

@get:Rule
val activityScenarioRule = ActivityScenarioRule(MainActivity::class.java)

private lateinit var activityScenario: ActivityScenario<MainActivity>

@Before
fun setUp() {
    activityScenario = activityScenarioRule.scenario
    activityScenario.onActivity { activity ->
      // Do any setup you need for your activity here
    }
}

@Test
fun testButton(){
    onView(withId(R.id.btnClick)).perform(click())
    onView(withId(R.id.tvMessage)).check(matches(withText("Hello, User!")))
```

```
  }
  @After
  fun tearDown() {
    activityScenario.close()
  }
```

5. Make sure to import the correct libraries

```
import androidx.test.core.app.ActivityScenario
import androidx.test.espresso.Espresso.onView
import androidx.test.espresso.action.ViewActions.click
import androidx.test.espresso.assertion.ViewAssertions.matches
import androidx.test.espresso.matcher.ViewMatchers.withId
import androidx.test.espresso.matcher.ViewMatchers.withText
import androidx.test.ext.junit.rules.ActivityScenarioRule
import androidx.test.platform.app.InstrumentationRegistry
import androidx.test.ext.junit.runners.AndroidJUnit4
import org.junit.After

import org.junit.Test
import org.junit.runner.RunWith

import org.junit.Assert.*
import org.junit.Before
import org.junit.Rule
```

6. Run the test. You need to open the emulator or connect your test device to test the instrumented tests.
7. If you have used the exact code above, the test will fail. Modify the code with the correct string and run the test again

Output will be like this

Tests	Duration	HMD Global TA-1003
✔ Test Results	3 s	2/2
✔ ExampleInstrumentedTest	3 s	2/2
✔ useAppContext	1 s	✔
✔ testButton	1 s	✔

Best Practices when using Instrumented Testing

- Keep the test code separate from the app code: It's a good practice to keep your test code separate from your app code. You can create a separate directory for your test code and keep your test files there.
- Use the latest version of Android Studio: Make sure you're using the latest version of Android Studio to take advantage of the latest features and improvements.
- Use AndroidX Test: Use the AndroidX Test library for writing your tests. It provides a set of testing APIs that are optimized for Android and can help you write robust and reliable tests.
- Use the correct testing framework: Android provides two testing frameworks: JUnit and Espresso. Use JUnit for unit testing and Espresso for UI testing.
- Use ActivityScenario for testing activities: Use the ActivityScenario API for testing activities. It provides a simple and reliable way to launch and interact with activities in your app.
- Run tests on multiple devices: Test your app on multiple devices to ensure that your app works correctly on different device configurations.
- Use Gradle build scans: Use Gradle build scans to get more insights into your build process and identify any issues that may be affecting your tests.
- Use continuous integration: Set up continuous integration for your tests to ensure that they are run automatically whenever you push new code.
- Write clear and concise test names: Write clear and concise test names that describe what the test is doing. This will make it easier to understand the purpose of the test and help you identify any issues that may arise.
- Regularly run tests: Run your tests regularly to catch any issues early in the development process. This will help you avoid spending time debugging issues later.