Mobile and Ubiquitous Computing

Labsheet 3 - Testing with Android

This labsheet builds directly on Labsheet 2. Therefore, make sure that you have completed Labsheet 2 before you attempt to work on this labsheet.

In this labsheet you will:

1. Be introduced to tests and understand why they can be useful
2. Understand that tests in Android let you test both logic and presentation aspects
3. Adjust an existing test to see when it passes and fails
4. Develop your own tests

Understanding tests is important and we will return to them throughout future labsheets. The focus in future labsheets will generally be on writing code that fits with existing tests, but it is very helpful to understand how they work as part of this.

Note – we're using quite a few imported classes for this labsheet. You will need to import them. The easiest way to do this is when you get red underlining in Android Studio, right click the object for which an import seems to be missing, click "Show Context Actions" and then click the import option. Check the class code if you can't figure out what you need to import.

What are tests and why would we use them?

One way to test your code works as expected is to sit there tapping all kinds of things and checking to see what happens. This doesn't scale very well and it's not terribly reliable either, especially as programs become more complicated and the possible number of states your app can exist in increases.

A better way is to use some kind of formalised approach to testing. This allows you to carefully define the expected behaviour of an app and notice if a new change breaks something unexpectedly. Android lets us write unit tests both for 'normal' code, but also UI elements. In this way we can ensure compliance of both the logic and presentation components of our work.

In Android we have *instrumented* tests. These tests involve running code on an actual device (real or emulated) to see what happens. These tests are slower, but let you test certain functionality as it would run on an actual device. These tests are slow to run because they effectively involve running the actual app. *Non-instrumented* tests are run in the JVM on your development machine. They run much much faster but are of course substantially more limited because they aren't running in a full environment like instrumented tests do.

Creating tests for the code we have so far

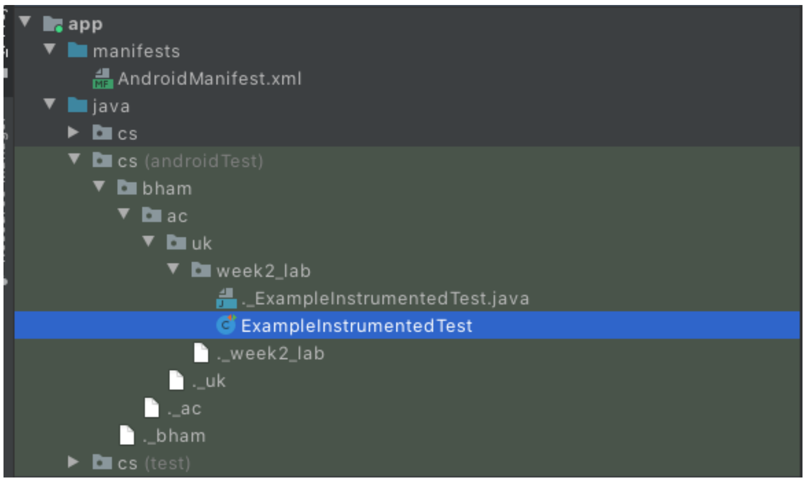
There’s an idea that development should be ‘test-driven’, which is that you write the tests for your code *first* and then you build your functionality to pass your tests. In future weeks we’ll return to this idea and you’ll be trying to complete labs with tests that I have already coded. However, at this stage of your learning, it can be difficult to do test-driven development when you don’t understand very much about the environment you’re working in.

Therefore, this week we will be taking a look at our code from Week 2, adapting it with a few tests the check the correctness of our UI and logic.

The first thing to do is to duplicate our complete working code from Week 2. We’re going to use this as the environment for creating our first tests.

Finding and renaming our test

The project you have should, by default already be set-up for tests. In the project pane, which show the files associated with your project, expand the `app` tree, then the `java` tree. You should see three namespaces here (in my code it’s `cs` because the namespace is `cs.bham.ac.uk`, which is actually backwards). One is labelled `(androidTest)` and the other is labelled `(test)`. Expand out the `(androidTest)` tree all the way down and you should get to a Java class file called `ExampleInstrumentedTest.java`. Open this, then right click on the class name in the source code and select Refactor → Rename… change the name of the class to `SpinnerSelectTest`. Now we’re going to need to actually give our test something to do!

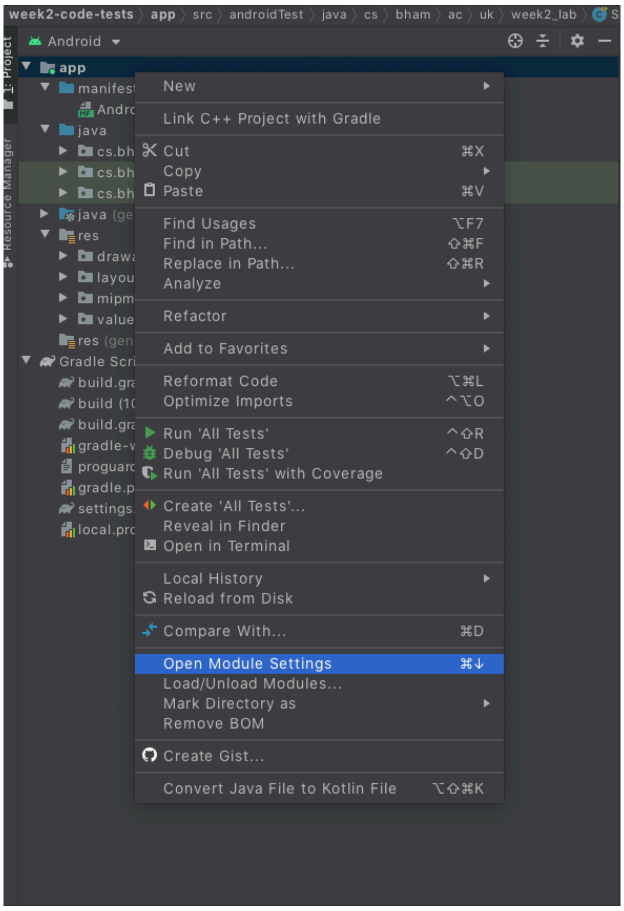


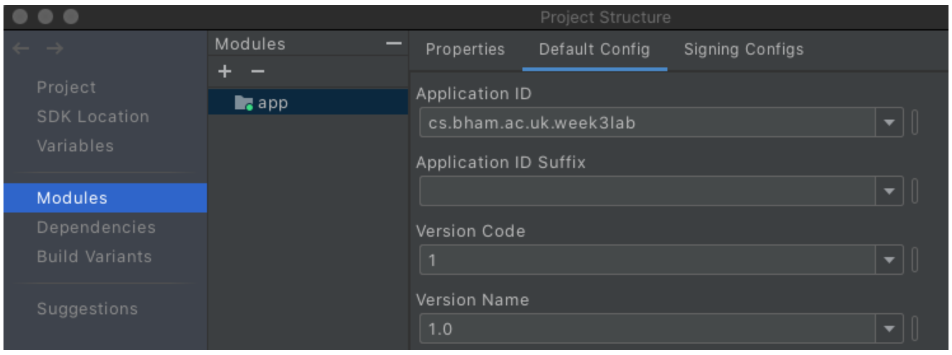
```
▼ app
  ▼ manifests
      AndroidManifest.xml
  ▼ java
    ▶ cs
    ▼ cs (androidTest)
      ▼ bham
        ▼ ac
          ▼ uk
            ▼ week2_lab
                ._ExampleInstrumentedTest.java
                ExampleInstrumentedTest
              ._week2_lab
            ._uk
          ._ac
        ._bham
    ▶ cs (test)
```

Adjusting other names

One of the challenges of using Android Studio is that refactoring isn’t always super reliable or complete. We’re going to see this first of all with an existing test that should be there as an example: `useAppContext()`. If you’ve followed the instructions in Week 2 precisely (open the sample code, if not), then you can see that this method is calling `assertEquals` to test whether the package name specified is equal to `cs.bham.ac.uk.week2_lab`. “Asserts” are very common in testing; they are a way for you to say “this is how things should look”. If they don’t look like that then the test fails.

To cause this test to fail, let's rename our project. The easiest way to do this is to right click '`app`' in the root of your project viewer pane on the left hand side, and select '`Open Module Settings…`'. You should see "Application ID" as one of the options. Adjust this to `cs.bham.ac.uk.week3lab`. And hit Apply.





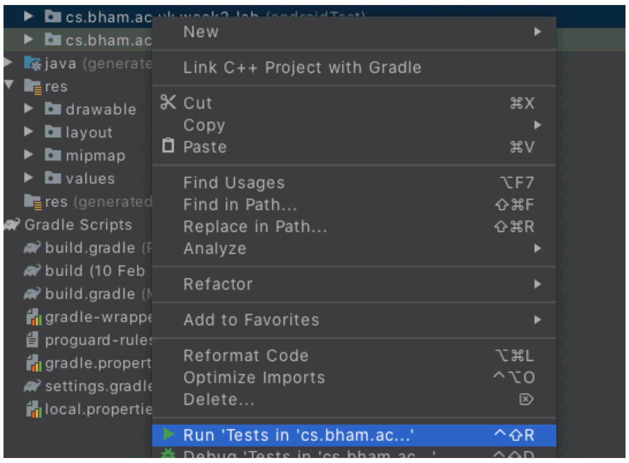
Application ID: cs.bham.ac.uk.week3lab
Application ID Suffix:
Version Code: 1
Version Name: 1.0

You'll notice that the string in the assertEquals() has not automatically been refactored. When this test runs, it should find that there is not a match between the package name (now `cs.bham.ac.uk.week3lab`) and what you have asserted it should be (`cs.bham.ac.uk.week2_lab`). To run the tests, right click on the cs (androidTest) part of the tree we opened up earlier and select `Run 'Tests in 'cs.bham.ac…'`.

This will start running the tests we've defined, which at the moment is the default one. You should get a failure with a complaint that our assertion about the name of the package has failed:

```
org.junit.ComparisonFailure: expected:<cs.bham.ac.uk.week[2_]lab> but
was:<cs.bham.ac.uk.week[3]lab>
```

See how it tells us how it has failed, indicating the differences with [square brackets]. Let's adjust our test so that we're asserting that the returned value of `appContext.getPackageName()` is equal to `cs.bham.ac.uk.week3lab`. This should be successful, your tests should run to completion. Happy days. You've already seen the power of testing, finding that unexpected change in package name!



Writing your first test

OK now you've updated and run a test, so now you're going to write your own. This first test is going to do something really simple, check that our `Spinner` called `spinner` appears on the `MainActivity` (the only `Activity` we have for this app) once it has loaded. Tests let us programmatically control aspects of the interface and then 'read-back' the results and check whether things look as they should.

Create a new test by defining a method `spinnerExists()` that you label `@Test` so Android knows how to deal with it:

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

Because we need to have a running activity to whether our spinner is displayed on it, we need to add to our test class something to create an activity that our test will run against. Just like the `@Test` annotation signifies ones of the test methods we can use the `@Rule` annotation to define bits of code that will be run before any `@Test` method. In this instance we need some code that will create an `Activity` that we can test against.

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

What we've done here is to create a new instance of `ActivityScenarioRule`. This accepts an implementation of `Activity` as a generic type. In this case, we only have one activity in our application, `MainActivity`, but in a more complex application we might have a variety of activities that we'd want to write tests for.

We first need to identify the `View` that we want Android to 'focus' its attention on. The first thing we're going to do is to create a test to see if your `Spinner` has been declared with the correct name `spinner`. Why would we want to test something so simple? Well, someone might change the name of the spinner accidentally which could, for instance, break certain kinds of runtime usage that the compiler wouldn't detect.

Now we have an activity in place, we can update our `spinnerExists @Test` method:

```java
@Test
public void spinnerExists(){
    onView(withId(R.id.spinner)).check(matches(isDisplayed()));
}
```

What does this do? Well we first call `onView()`. This method tells Android "Hey we're going to be performing some action on this particular view". We supply it with an instance of `ViewMatcher` by calling `withId()`, which itself takes a resource as a parameter (in this case, our instance of `Spinner`). Once we've done that, we can start doing things to the `View`. We might `perform()` an action like a `click()`, but here we just want to `check()` something. In this case, we're checking that it `isDisplayed()`. The `matches()` method tests whether the view exists and that its state matches the supplied parameter. In this case it's whether it is displayed.

Run your tests again. Does it pass? It should! How do we know if our test is actually doing anything, though? Easy. Navigate to `activity_main.xml`. Make sure you're in the Design view where you can see the layout of the Views on the screen. Select the `Spinner` we have on the screen, then in the Attributes pane (the big list on the right hand side), scroll too, or search for `visibility`. Set this to `invisible`. Run your test again. What happens? It should now fail. Be sure to remove the `invisible` attribute before continuing.

Adding another test for layout

Most of the tests you're going to encounter on the module will be UI-focused. That's because most of our applications are going to be high on interface and low on application logic. That's for the simple reason that the kinds of steps you need for application logic should be familiar to you from previous programming modules. The application logic is often a little more generic, whereas the presentation layer is highly Android-specific.

The next test we're going to implement is going to **check** that our `Spinner` appears above the `Button` we have on the screen. The `Spinner` has the id `spinner` and the `Button` has the id `button`. Where we used `isDisplayed()` in the previous test, here the `isCompletelyAbove()`

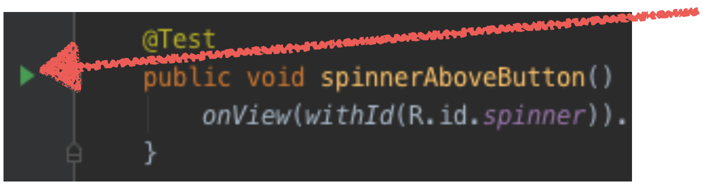
method might be useful. Have a go at creating a new test, `spinnerAboveButton()`. It should be a one line method, so see if you can write it yourself before looking at the solution on the next page.

Were you able to do it? You should have something like this:

```
@Test
public void spinnerAboveButton() {
onView(withId(R.id.spinner)).check(isCompletelyAbove(withId(R.id.button)
}
```

Run the test. What do you find? Did it pass? What if you go back to `activity_main.xml` and adjust the layout so that `button` appears above `spinner`? What then? Hopefully the test will fail and that you're starting to see why tests can be powerful ways of structuring the development of apps so that certain design rules can be strictly enforced. This is especially important as interfaces become more complex.

Note that you can run individual tests from inside the class file. Simply hit the little run button in the margin and you can run that specific test. This is helpful if you're working on a single test; it saves you having to run all the tests every time.



```
@Test
public void spinnerAboveButton()
    onView(withId(R.id.spinner)).
}
```

This is the little run button in the margin!

Getting more complicated

So far, we've had a go at developing a test to check that a particular view is visible when an activity is started. We've also built a test to enforce restrictions on the layout of `View`s on the screen. We're going to finish this labsheet with a more complex test that is going to test whether the main 'functionality' of our application works as expected.

The app is quote simple. There is a `Spinner` that contains three items. You select an item, hit the 'Update' `Button` and the `TextView` is updated with the content of the Spinner item that you selected. Let's build one final test that checks that this functionality works as it should!

Create a new `@Test` method, `spinnerSelectAndUpdate()` in the same way you've created the other `@Test` methods. The easiest way to explain this to you is for me to tell you what's going to go in this method and then explain, line by line, what exactly is going on.

```
@Test
public void spinnerSelectAndUpdate() {
    Context appContext =
InstrumentationRegistry.getInstrumentation().getTargetContext();
    String[] spinOpts =
appContext.getResources().getStringArray(R.array.spinneroptions);

    String firstItem =  spinOpts[0];
    onView(withId(R.id.spinner)).perform(click());
    onData(allOf(is(instanceOf(String.class)), is(firstItem))).perform(click());
    onView(withId(R.id.button)).perform(click());
    onView(withId(R.id.textView)).check(matches(withText(firstItem)));
}
```

OK, this is quite a lot more complicated than our other tests! What is going on!? Let's go through it line-by line.

The first three lines of the method are all about set-up. Accessing `string-array` resources is a little tricky. We have to great a 'context' (in this case the context is the activity we started with our `@Rule`, `MainActivity`.) This allows us to access the array resource. We take the first element of this list and create a variable `firstItem`. This is the string that we're going to be looking for as we perform different parts of our test.

The first line of the test-proper (and not just set-up), simply clicks the spinner. We're getting to use the `perform()` method now – this lets us manipulate the interface. As you can see the thing we perform is a `click()`, which is passed as the parameter to `perform()`. What do we expect to happen now? Well we expect the spinner to expand letting us click options from it.

What if the spinner is really big though? What if it contains, say, a list of all of the countries in the world? Helpfully, Android lets us act using `onData()`, rather than `onView()`. Using `onData()` lets us act on elements of `View`s that may not even be visible yet. In this case we look the elements for one that meets `allOf` the criteria: the first is that it is an instance of `String`. The second is that the value matches `firstItem`, the value that we got from our `string-array` resource. In other words, "look for something that is a `String` with the value of `firstItem`. If this item is found, then we again `perform()` the `click()` action on it to select it.

Now when we're using the app, we select the element from `spinner`, click `button` and `textView` gets updated. We've done the selecting from `spinner` now, so the next step is to hit `button` (which is labelled 'Update'), which should cause `textView` to update. The next line then, `click()`s `button`.

For the final step we don't need to do anything else, we've done the interaction with the interface. We just need to `check()` that the value of `textView` has changed so that it `matches()` `firstItem`, which we selected on earlier lines. That completes the work we need to do for this test.

Test it. Does it work? How can you break it? What if you comment out the step to click `button`? Does it still work? What if you ask it to, instead of looking for `firstItem` on `spinner`, to look for the string "M&UC"? Does the test still pass? It shouldn't! Spent a moment sabotaging different parts of the test to see what will cause the test to fail.

This is definitely not the perfect test for this. For example it makes the explicit assumption that the first element in `R.array.spinneroptions` exists! This would crash out if we had not elements in the array with an ArrayIndexOutOfBounds error. Try it!

Summary and extension

So how about that? We've written about 10 lines of code in this labsheet, but we've had to learn a huge amount to get to that point. This is so often what developing with Android is like; the actual programming is almost trivial in that you're just calling a few class methods on some resources you created using a GUI. But plumbing all this together is hard, there are lots of concepts to understand . Android feels like a sea of implemented classes sometimes and it can be difficult to keep track of what is going on.

If you've finished the labsheet and want to push yourself further, have a go at developing tests that check UI features or enforce particular rules about interaction with it. You might be surprised by the number of tests that you need to write to enforce the operation of the app exactly as it is specified at the moment.