ITW202: Mobile Application

Unit IV: Developing for Android

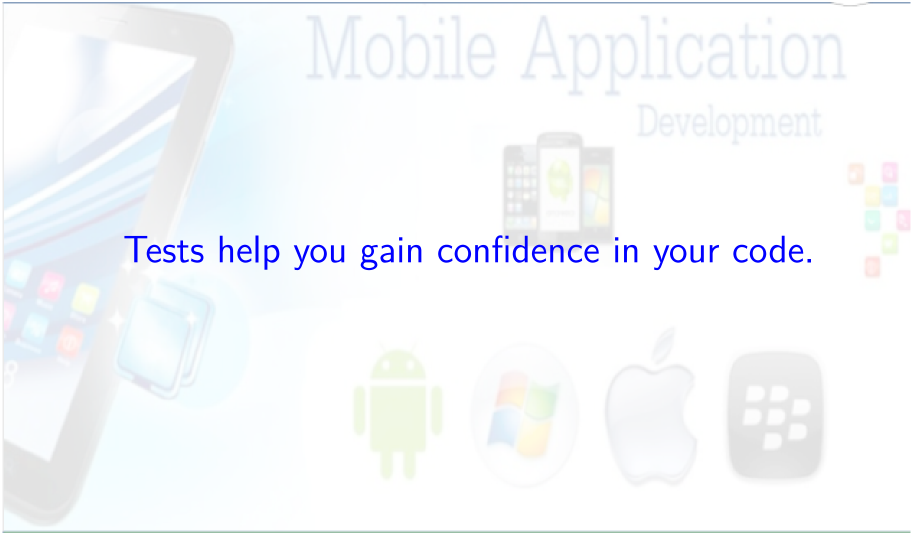
Ms. Sonam Wangmo

Gyalpozhing College of Information Technology
Royal University of Bhutan

May 20, 2021

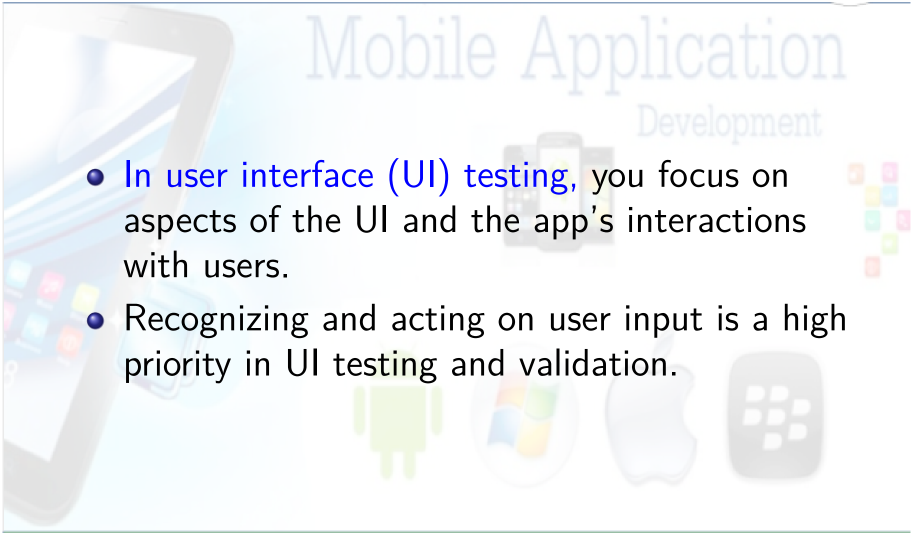
UI testing

UI testing



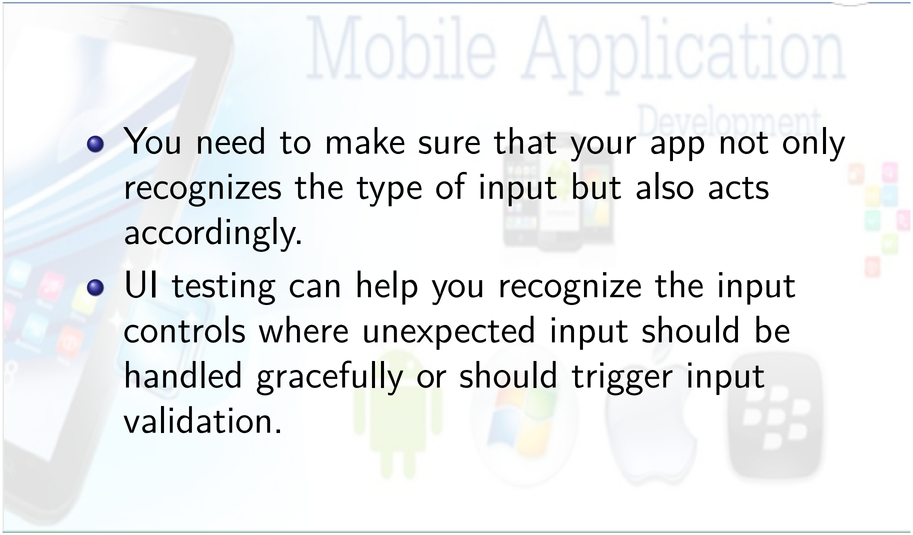
Tests help you gain confidence in your code.

UI testing



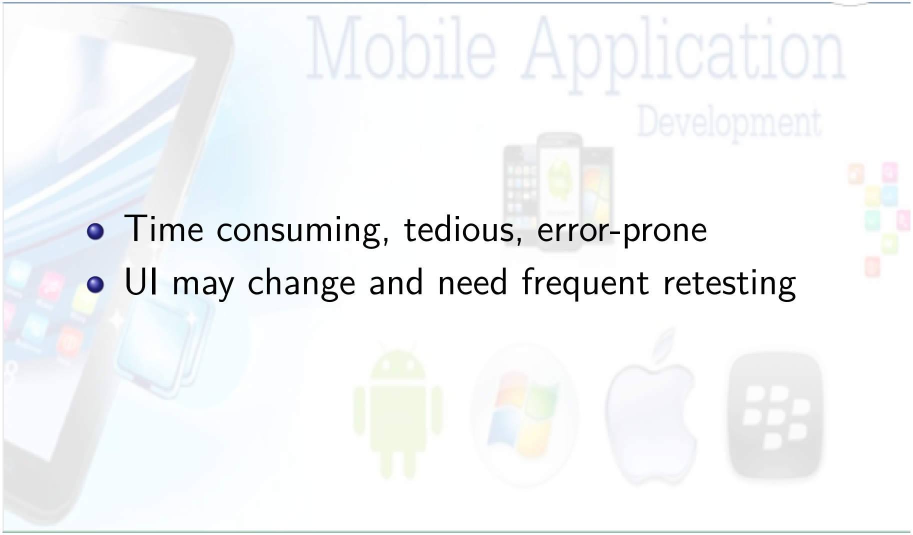
- In user interface (UI) testing, you focus on aspects of the UI and the app's interactions with users.
- Recognizing and acting on user input is a high priority in UI testing and validation.

UI testing



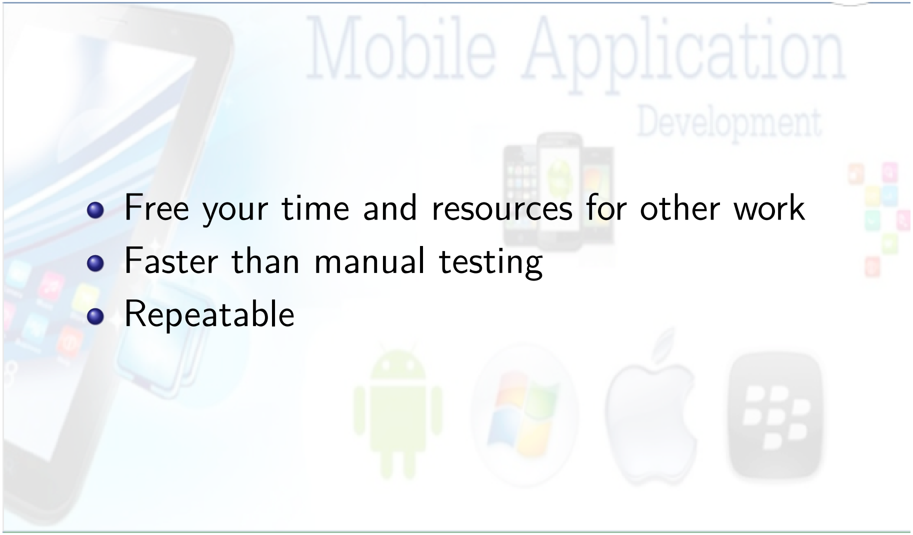
- You need to make sure that your app not only recognizes the type of input but also acts accordingly.
- UI testing can help you recognize the input controls where unexpected input should be handled gracefully or should trigger input validation.

Problems with testing manually



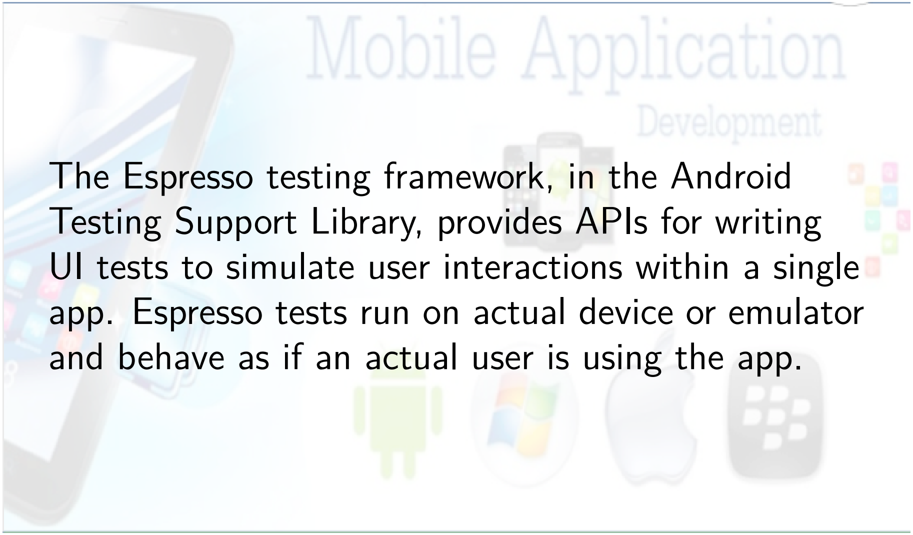
- Time consuming, tedious, error-prone
- UI may change and need frequent retesting

Benefits of testing automatically



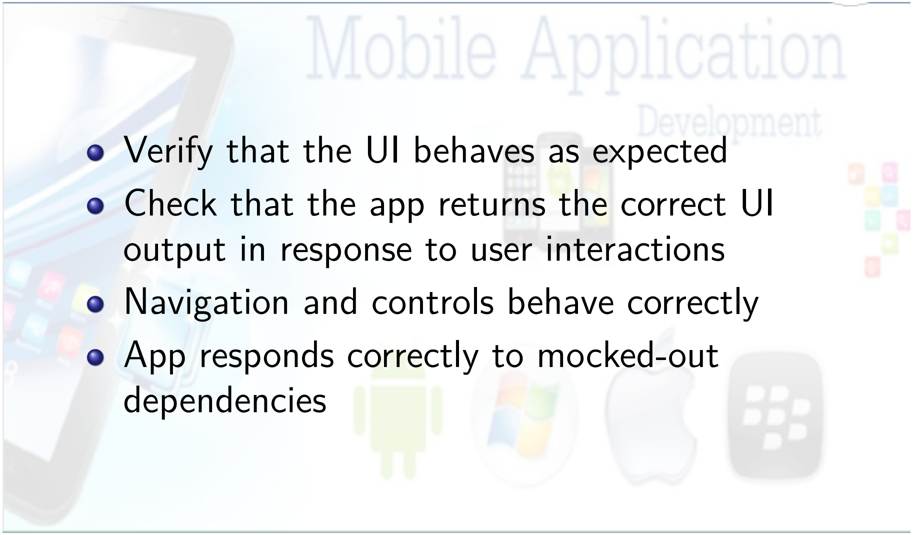
- Free your time and resources for other work
- Faster than manual testing
- Repeatable

Espresso for single app testing



The Espresso testing framework, in the Android Testing Support Library, provides APIs for writing UI tests to simulate user interactions within a single app. Espresso tests run on actual device or emulator and behave as if an actual user is using the app.

Espresso for single app testing

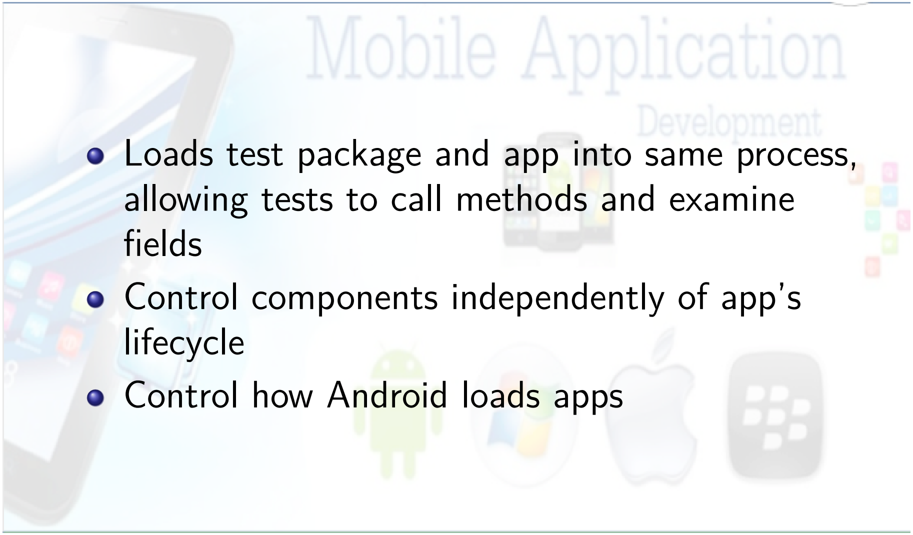


- Verify that the UI behaves as expected
- Check that the app returns the correct UI output in response to user interactions
- Navigation and controls behave correctly
- App responds correctly to mocked-out dependencies

What is instrumentation?

- Android instrumentation is a set of control methods, or hooks, in the Android system, which control Android components and how the Android system loads apps.

What is instrumentation?



- Loads test package and app into same process, allowing tests to call methods and examine fields
- Control components independently of app's lifecycle
- Control how Android loads apps

Benefits of instrumentation

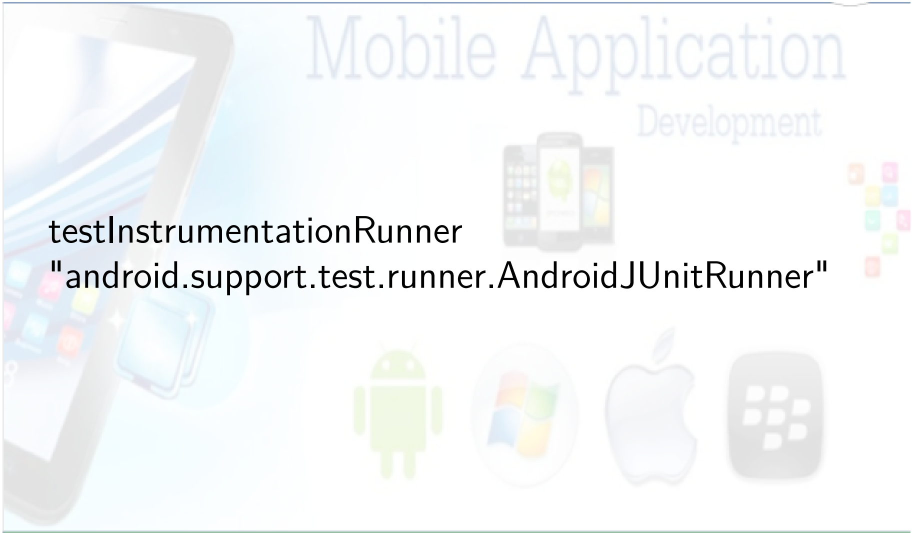
- Tests can monitor all interaction with Android system
- Tests can invoke methods in the app
- Tests can modify and examine fields in the app independent of the app's lifecycle

Test environment And Espresso setup

Add dependencies to build.gradle

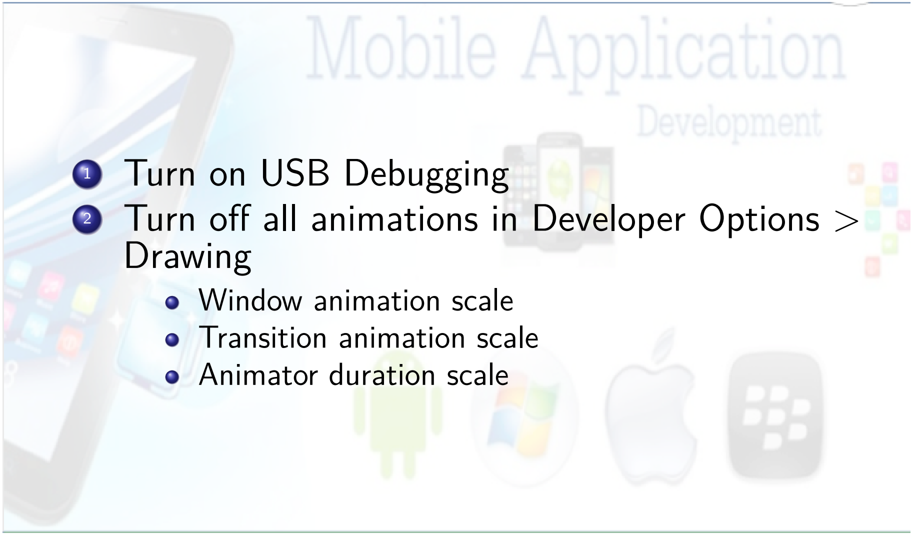
```
testImplementation 'junit:junit:4.12'
androidTestImplementation 'com.android.support.test:runner:1.0.1'
androidTestImplementation
              'com.android.support.test.espresso:espresso-core:3.0.1'
```

Add defaultConfig to build.gradle



```
testInstrumentationRunner
"android.support.test.runner.AndroidJUnitRunner"
```

Prepare your device



1. Turn on USB Debugging
2. Turn off all animations in Developer Options > Drawing
    - Window animation scale
    - Transition animation scale
    - Animator duration scale

Create tests

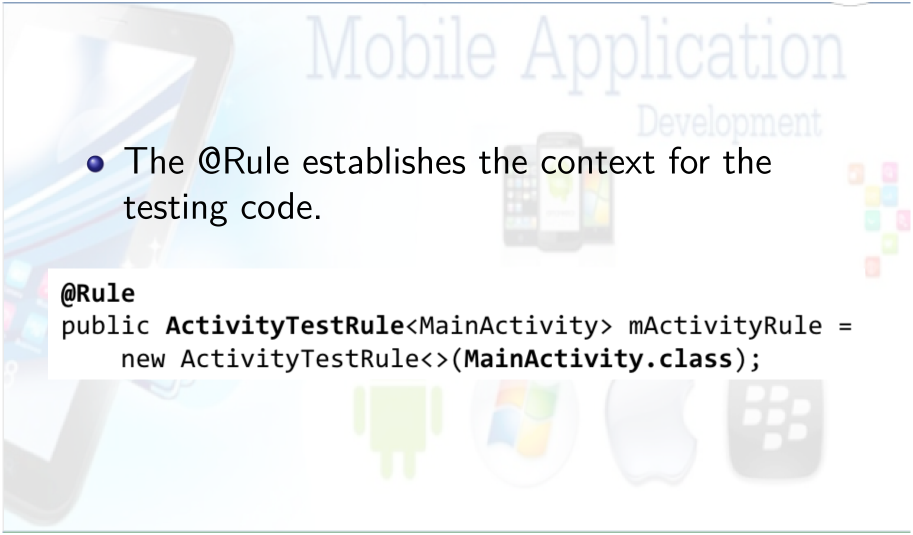
- Store in module-name/src/androidTests/java/
  - In Android Studio: app > java > module-name (androidTest)
- Create tests as JUnit classes

Creating Espresso tests

Test class definition

- @RunWith(AndroidJUnit4.class) — Required annotation for tests
- @LargeTest — Based on resources the test uses and time to run
- @SmallTest — Runs in $<$ 60s and uses no external resources
- @MediumTest — Runs in $<$ 300s, only local network
- @LargeTest — Runs for a long time and uses many resources

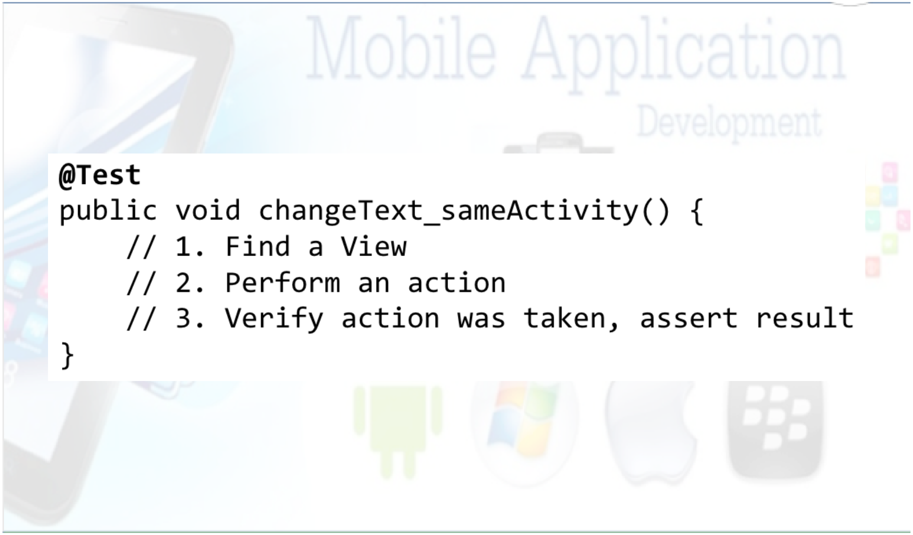
@Rule specifies the context of testing



- The @Rule establishes the context for the testing code.

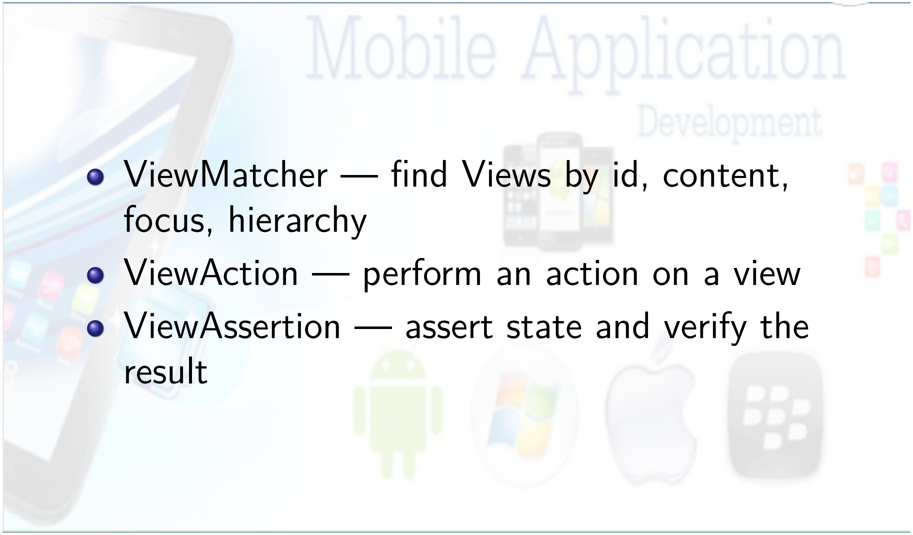
```
@Rule
public ActivityTestRule<MainActivity> mActivityRule =
    new ActivityTestRule<>(MainActivity.class);
```

@Test method structure



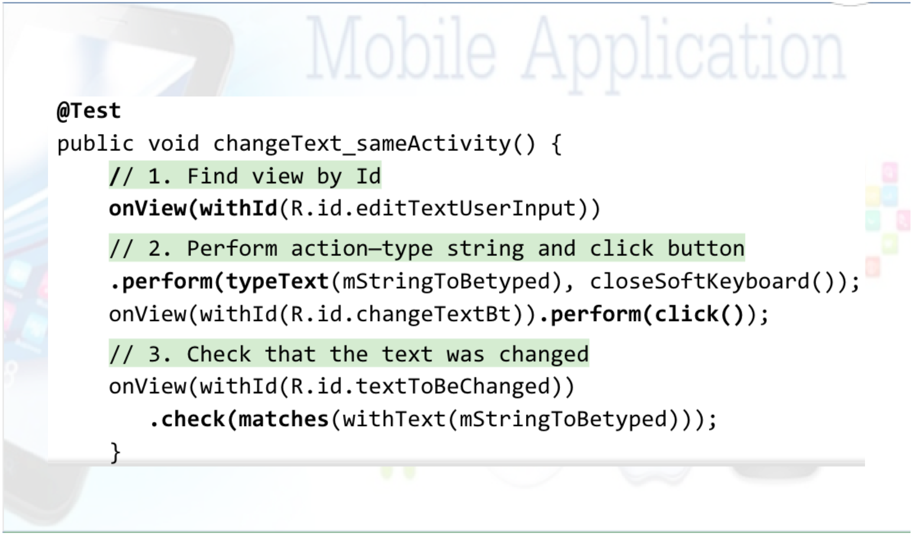
```
@Test
public void changeText_sameActivity() {
    // 1. Find a View
    // 2. Perform an action
    // 3. Verify action was taken, assert result
}
```

Hamcrest Matchers



- ViewMatcher — find Views by id, content, focus, hierarchy
- ViewAction — perform an action on a view
- ViewAssertion — assert state and verify the result

Basic example test


Mobile Application

```
@Test
public void changeText_sameActivity() {
    // 1. Find view by Id
    onView(withId(R.id.editTextUserInput))
    // 2. Perform action—type string and click button
    .perform(typeText(mStringToBetyped), closeSoftKeyboard());
    onView(withId(R.id.changeTextBt)).perform(click());
    // 3. Check that the text was changed
    onView(withId(R.id.textToBeChanged))
        .check(matches(withText(mStringToBetyped)));
    }
```

Recording tests

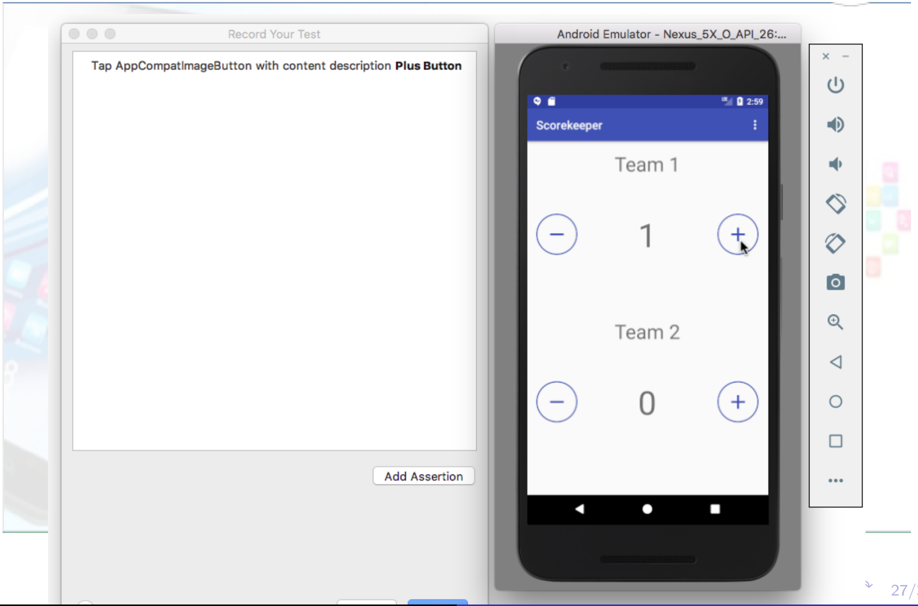
Recording an Espresso test

- Use app normally, clicking through the UI
- Editable test code generated automatically
- Add assertions to check if a view holds a certain value
- Record multiple interactions in one session, or record multiple sessions

Start recording an Espresso test

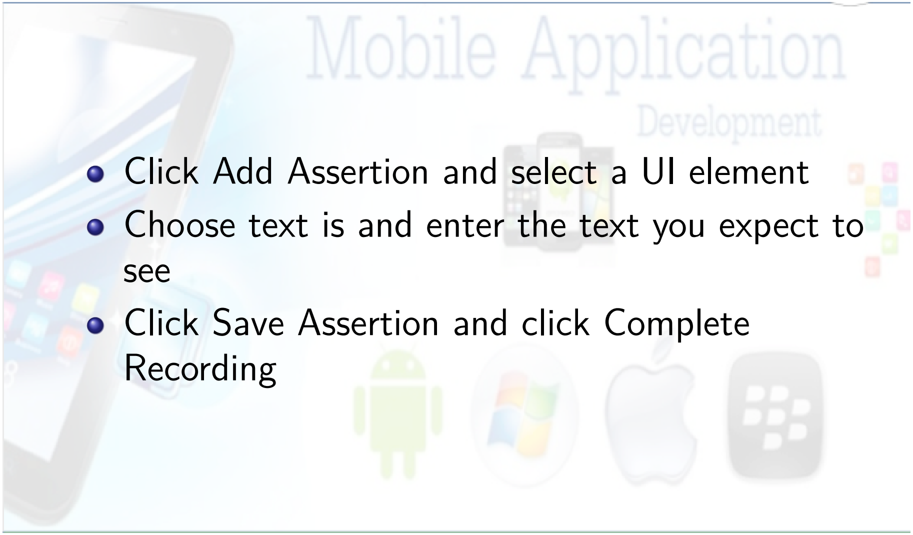
- Run > Record Espresso Test
- Click Restart app, select target, and click OK
- Interact with the app to do what you want to test

Start recording an Espresso test



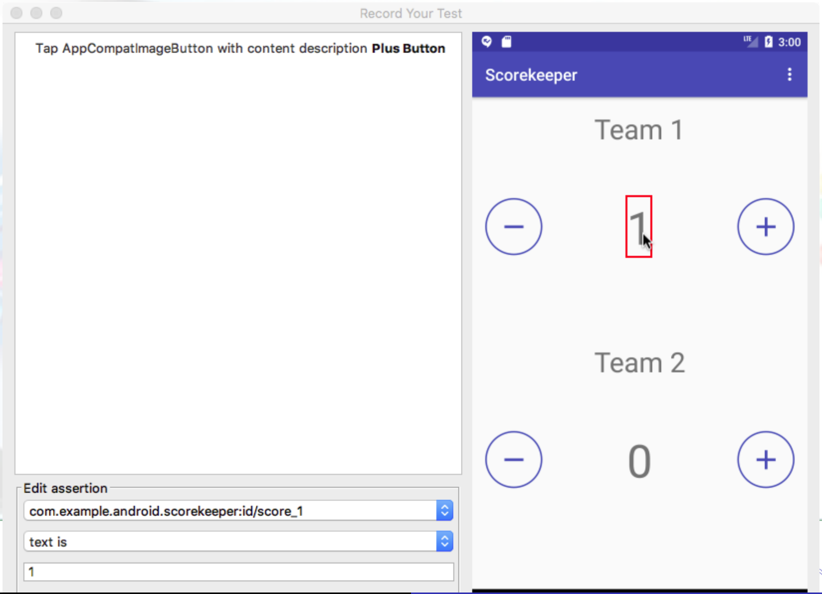
Tap AppCompatImageButton with content description **Plus Button**

Add assertion to Espresso test recording



- Click Add Assertion and select a UI element
- Choose text is and enter the text you expect to see
- Click Save Assertion and click Complete Recording

Add assertion to Espresso test recording


Record Your Test

Tap AppCompatImageButton with content description **Plus Button**

Edit assertion

com.example.android.scorekeeper:id/score_1

text is

1

THANK YOU