


Android Testing

Basics and Beyond




Agenda


- Why testing ?
 - Unit testing.
 - Test Doubles.
 - Instrumented unit test.
 - Integration test (Espresso).
 - UI Automator.
 - Demos (wherever needed)
- 
- A decorative orange wavy bar at the bottom of the slide.

Why testing ?

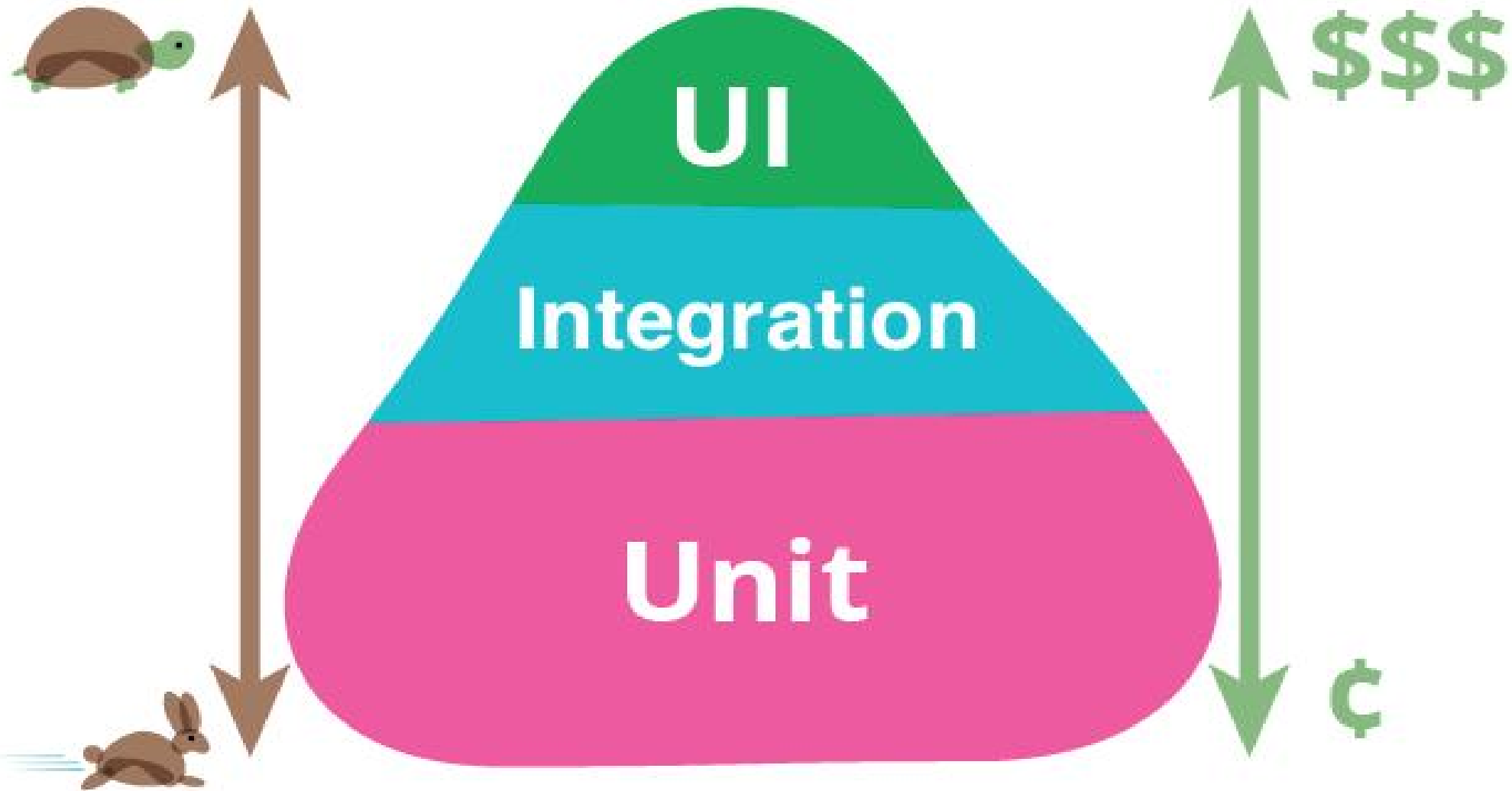
Why testing?

- According to Cambridge University research in 2012, the global cost of debugging the software has risen to \$312 billion annually.
 - The research found that, on average, software development spend 50% of their programming time finding and fixing bugs
- 

Why testing?

- Deeper and better understanding of requirement
 - Increase reliability and robustness
 - Customer satisfaction
 - Reduce maintenance cost
- 

Testing Pyramid




Unit test

Using JUnit and Mockito



Unit testing

Unit testing is a software development process in which the smallest testable parts of an application, called units, are individually and independently scrutinized for proper operation.




Benefits of Unit testing

- **Find problems early**
 - **Promote change**
 - **Simplifies integration**
 - **Documentation**
 - **Design**
- 


JUnit

- JUnit is a simple framework to write repeatable tests. It is an instance of the xUnit architecture for unit testing frameworks.

```
@Test  
public void newArrayListsHaveNoElements ()  
{  
    assertThat (new ArrayList ().size (), is (0));  
}
```



JUnit basic annotations

- **@BeforeClass** – Run once before any of the test methods in the class, public static void
 - **@AfterClass** – Run once after all the tests in the class have been run, public static void
 - **@Before** – Run before @Test, public void
 - **@After** – Run after @Test, public void
 - **@Test** – This is the test method to run, public void
- 

JUnit

Gradle command to run unit test: `./gradlew test`

Demo




Test doubles

Generic not specific to android



Test doubles

- **A test double is an object that can stand in for a real object in a test**
 - It is similar to how a stunt double stands in for an actor in a movie.
 - These are sometimes all commonly referred to as “mocks”
 - **The most common types of test doubles are**
 1. Dummy
 2. Stub
 3. Spy
 4. Fake.
 5. Mock
- 

Test doubles

Lets take one example to illustrate all the test doubles

- We have a EmployeeServcie class which will do some action like creating customers on basis of their authorizations

Pojo	DAO	Service
Customer	IEmployeeDAO	EmployeeService
Employee	EmployeeDAO	


Dummy

Dummy object used when a object is needed for the tested method but without actually needing to use the object.




Stub

A **stub** has no logic, and only returns what you tell it to return. **Stubs can be used when you need an object to return specific values in order to get your code under test into a certain state.**




Spy

Spies are stubs that also record some information based on how they were called. One form of this might be an email service that records how many messages it was sent.




Mock

- **Mocks** are pre-programmed with expectations which form a specification of the calls they are expected to receive.
 - They can be checked during verification to ensure they got all the calls they were expecting.
- 


Fake

Fake objects actually have working implementations, but usually take some shortcut which makes them not suitable for production (an `InMemoryTestDatabase` is a good example).



Mockito

Mockito is an open source testing framework for Java. The framework allows the creation of test double objects that is used for effective unit testing of Java applications.



Mockito basic annotation

- **@Mock** : It is used to create dummy instance

@Mock

```
List<String> mockedList;
```

- **@Spy** : It is used to spy on an existing instance

@Spy

```
List<String> spiedList = new ArrayList<String>();
```

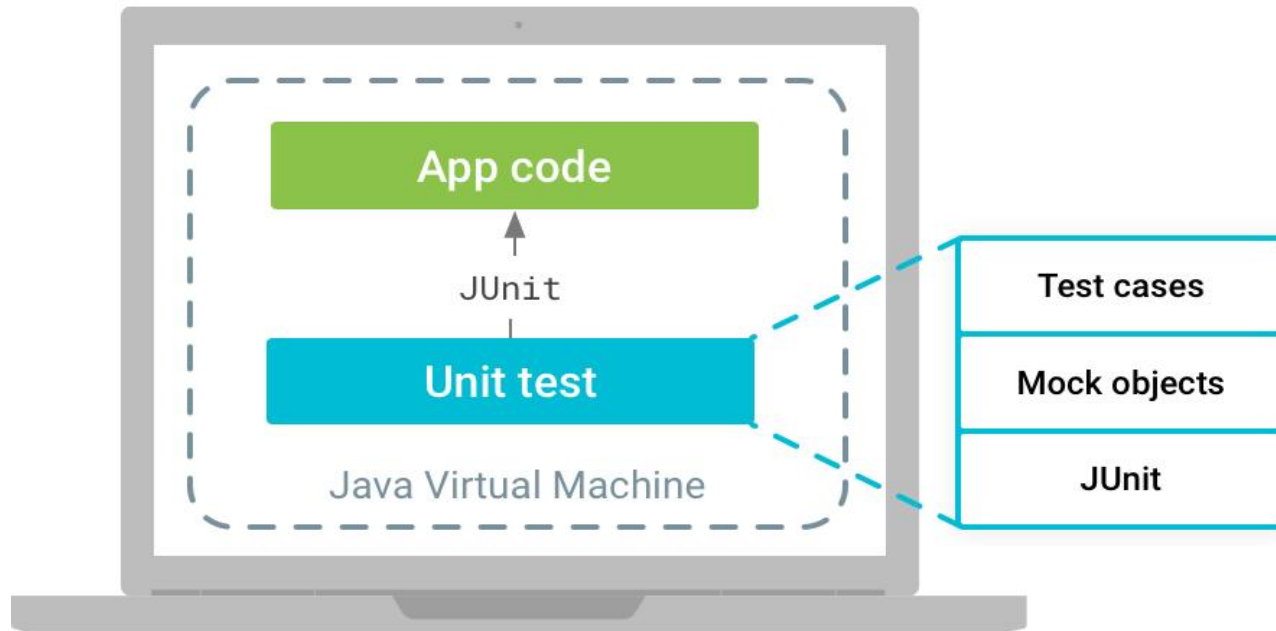
Test double with Mockito

Demo

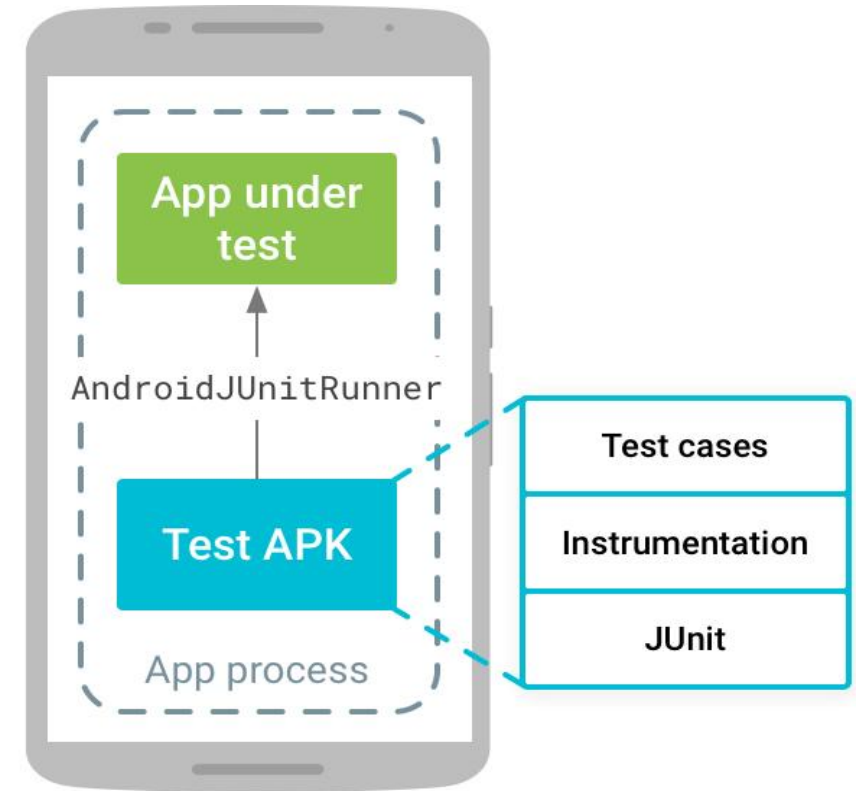


Instrumented Unit test






Local unit test
`src/test/java/`



Instrumented test
`src/androidTest/java/`

Instrumented unit test

- Instrumented unit tests are tests that run on physical devices and emulators
 - It can take advantage of the Android framework APIs and supporting APIs, such as the Android Testing Support Library.
 - We should create instrumented unit tests if our tests require the real implementation of an Android framework component (such as a Parcelable, SharedPreference).
- 

Instrumented unit test

Demo



Integration Testing

Espresso




Espresso



- Espresso is testing framework for integration testing to simulate user interactions within a single target app
- Requires Android 2.2 (API level 8) or higher.

Espresso packages


- **espresso-core** - Contains core and basic View matchers, actions, and assertions.
 - **espresso-web** - Contains resources for WebView support.
 - **espresso-idling-resource** - Espresso's mechanism for synchronization with background jobs.
 - **espresso-contrib** - External contributions that contain DatePicker, RecyclerView and Drawer actions, Accessibility checks, and CountingIdlingResource.
 - **espresso-intents** - Extension to validate and stub Intents, for hermetic testing.
- 

Espresso setup instructions

Setup your test environment

To avoid flakiness, we highly recommend that you turn off system animations on the virtual or physical device(s) used for testing.

On your device, under Settings->Developer options disable the following 3 settings:

- Window animation scale
 - Transition animation scale
 - Animator duration scale
- 

Espresso setup instructions cont.

Download Espresso

- Make sure you have installed the latest Android Support Repository under Extras in SDK Manager.
- Open your app's build.gradle file.
- Add the following lines inside dependencies:
 1. androidTestCompile 'com.android.support.test.espresso:espresso-core:2.2.2'
 2. androidTestCompile 'com.android.support.test:runner:0.5'

Espresso setup instructions cont.

Set the instrumentation runner

Add to the same build.gradle file the following line in android.defaultConfig:

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

Espresso setup instructions cont.

Running tests

In Android Studio

Create a test configuration

Open Run menu -> Edit Configurations
Add a new Android Tests configuration
Choose a module

Add a specific instrumentation runner:
`android.support.test.runner.AndroidJUnitRunner`

Run the newly created configuration.

From command-line via Gradle

Execute


`./gradlew connectedAndroidTest`

Espresso Basics

Main components of Espresso

```
onView(Matcher)  
    .perform(ViewAction)  
    .check(ViewAssertion);
```

Main components of Espresso

- **Espresso** – Entry point to interactions with views (via onView and onData).
 - **ViewMatchers** – A collection of objects that implement Matcher<? super View> interface (like 'withId').
 - **ViewActions** – A collection of ViewActions that can be passed to the ViewInteraction.perform() method (like 'click').
 - **ViewAssertions** – A collection of ViewAssertions that can be passed the ViewInteraction.check() method (like 'matches').
- 

Espresso Basic test

Click on the button

The first step is to look for a property that helps to find the button.

```
onView(withId(R.id.button_simple))
```

Now to perform the click:

```
onView(withId(R.id.button_simple)).perform(click());
```



Espresso Basic test cont.

Check that the TextView now contains “**Hello Espresso!**”

The TextView with the text to verify has a unique R.id too:

```
onView(withId(R.id.text_simple))
```

Now to verify the content text:

```
onView(withId(R.id.text_simple)).check(matches(withText("Hello Espresso!")));
```



Espresso Intents

- Espresso-Intents is an extension to Espresso, which enables validation and stubbing of Intents sent out by the application under test. It's like Mockito, but for Android Intents.

Espresso Intents cont.

Download Espresso-Intents

Add the following line inside dependencies:

```
androidTestCompile 'com.android.support.test.espresso:espresso-intents:2.2.2'
```

Espresso-Intents is only compatible with Espresso 2.1+ and the testing support library 0.3 so make sure you update those lines as well:

```
androidTestCompile 'com.android.support.test:runner:0.5'
```


```
androidTestCompile 'com.android.support.test:rules:0.5'
```

```
androidTestCompile 'com.android.support.test.espresso:espresso-core:2.2.2'
```



Espresso Intents cont.

IntentsTestRule

- Use IntentsTestRule instead of ActivityTestRule when using Espresso-Intents.
 - IntentsTestRule makes it easy to use Espresso-Intents APIs in functional UI tests.
 - This class is an extension of ActivityTestRule.
- 

Espresso Intents cont.

Demo



Espresso Intents cont.

Intent validation

- Espresso-Intents records all intents that attempt to launch activities from the application under test.
- Using the intended API (cousin of Mockito.verify), you can assert that a given intent has been seen.

An example test that simply validates an outgoing intent:

```
intended(toPackage("com.android.phone"));
```

Espresso Intents cont.

Intent stubbing

- Using the intending API (cousin of Mockito.when), you can provide a response for activities that are launched with startActivityForResult


An example test with intent stubbing:

```
intending(toPackage("com.android.contacts")).  
respondWith(new ActivityResult());
```

Espresso Intents cont.


Intent matchers

These are the matchers used in intending and intended methods.

- hasAction
 - hasCategories
 - hasExtras
 - toPackage
- 

UI Automator


UI Automator

- The UI Automator testing framework provides a set of APIs to build UI tests that perform interactions on user apps and system apps.
 - The UI Automator APIs allows you to perform operations such as opening the Settings menu or the app launcher in a test device.
 - The UI Automator testing framework is well-suited for writing *black box*-style automated tests.
- 

UI Automator key features

- A viewer to inspect layout hierarchy.
- An API to retrieve state information and perform operations on the target device.
- APIs that support cross-app UI testing.

Requires Android 4.3 (API level 18) or higher.



UI Automator setup

In the build.gradle file of your Android app module, you must set a dependency reference to the UI Automator library:

```
dependencies {
```

```
    ...
```

```
    androidTestCompile 'com.android.support.test.uiautomator:uiautomator-v18:2.1.1'
```

```
}
```




UI Automator

Demo



UI Automator key features cont.


UI Automator Viewer

- The [uiautomatorviewer](#) tool provides a convenient GUI to scan and analyze the UI components currently displayed on an Android device.
 - It is used to view the properties of UI components that are visible on the foreground of the device.
 - The [uiautomatorviewer](#) tool is located in the [<android-sdk>/tools/](#) directory.
- 

UI Automator key features cont.

Access to device state


This framework provides a **UiDevice** class to access and perform operations on the device on which the target app is running.

- Change the device rotation
 - Press a D-pad button
 - Press the Back, Home, or Menu buttons
 - Open the notification shade
 - Take a screenshot of the current window
- 

UI Automator key features cont.

UI Automator APIs

The UI Automator APIs allow you to write robust tests without needing to know about the implementation details of the app that you are targeting.

- **UiObject**: Represents a UI element that is visible on the device.
 - **UiScrollable**: Provides support for searching for items in a scrollable UI container.
 - **UiSelector**: Represents a query for one or more target UI elements on a device.
- 

Monkey

- The Monkey generates pseudo-random streams of user events such as clicks, touches, or gestures, as well as a number of system-level events.
- We can use the Monkey to stress-test applications that you are developing, in a random yet repeatable manner.
- The basic syntax is:

```
$ adb shell monkey [options] <event-count>
```

Example :

```
$ adb shell monkey -p your.package.name -v 500
```



- Q & A

[illegible]