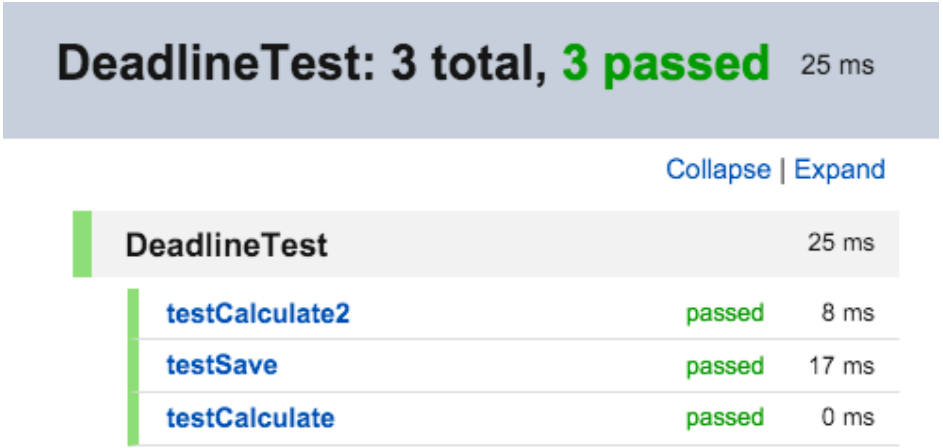
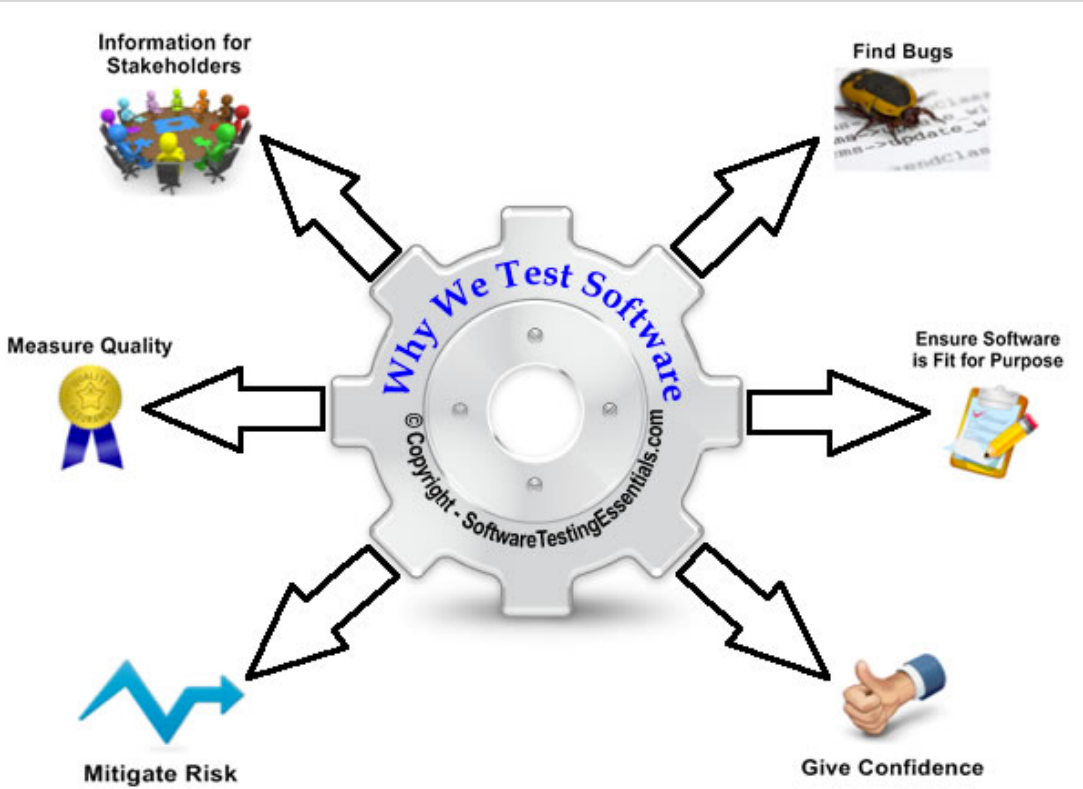
Testing

Lab 1

- Why testing? Types of tests
- Auto-generated tests
- The ‘Deadline’ app
- Junit, Mockito and Hamcrest



Why testing?



http://softwaretestingessentials.com/what-is-software-testing/

Junit 4 tests example

```
public class Calculator {
  public int evaluate(String expression) {
    int sum = 0;
    for (String summand: expression.split("\\+"))
      sum += Integer.valueOf(summand);
    return sum;
  }
}
```

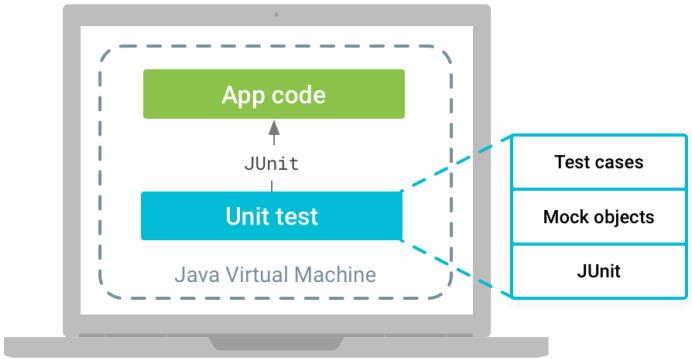
```
import static org.junit.Assert.assertEquals;
import org.junit.Test;

public class CalculatorTest {
  @Test
  public void evaluatesExpression() {
    Calculator calculator = new Calculator();
    int sum = calculator.evaluate("1+2+3");
    assertEquals(6, sum);
  }
}
```

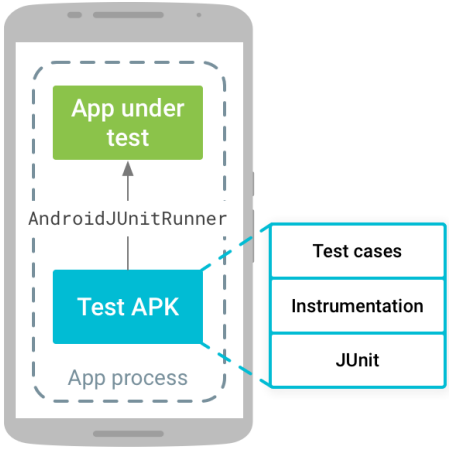
https://github.com/junit-team/junit4/wiki/Getting-started

Types of tests

Type	Subtype
Unit tests	Local Unit Tests
	Instrumented unit tests
Integration Tests	Components within your app only
	Cross-app Components



Local unit test
`src/test/java/`

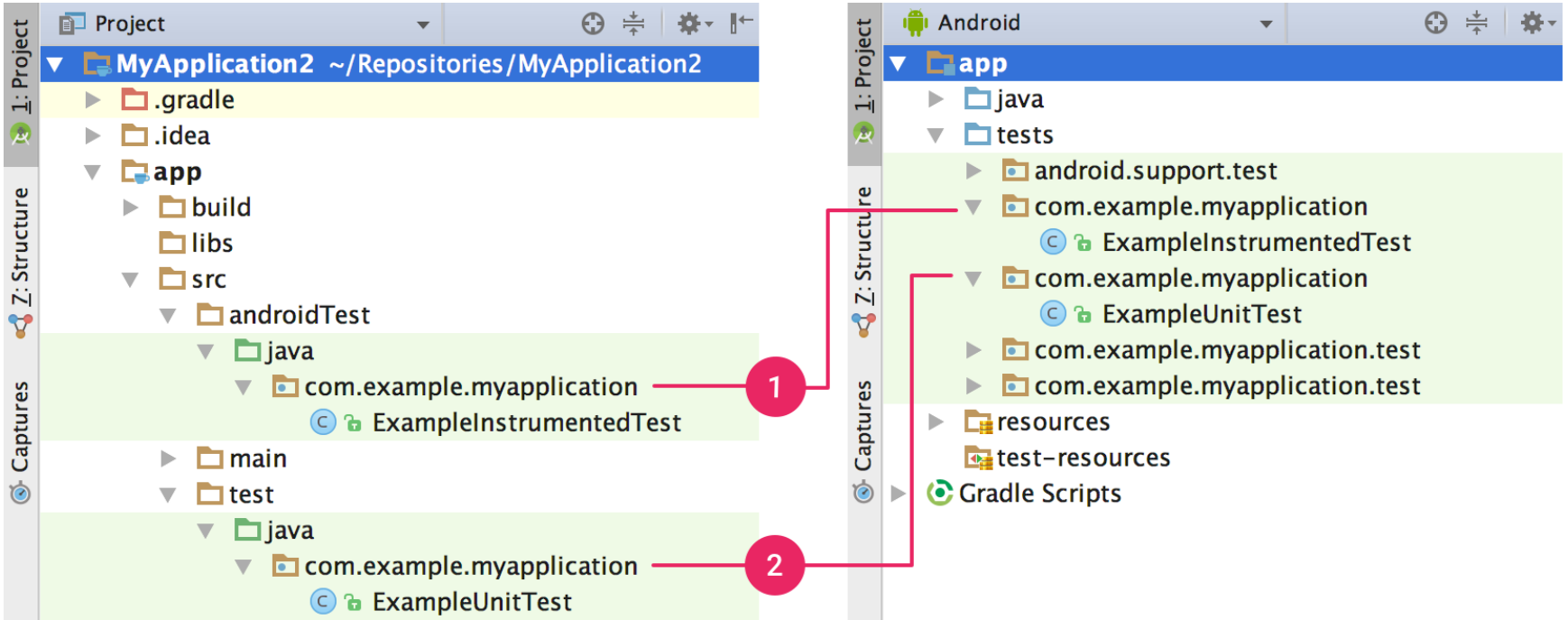


Instrumented test
`src/androidTest/java/`

https://developer.android.com/training/testing/start/index.html

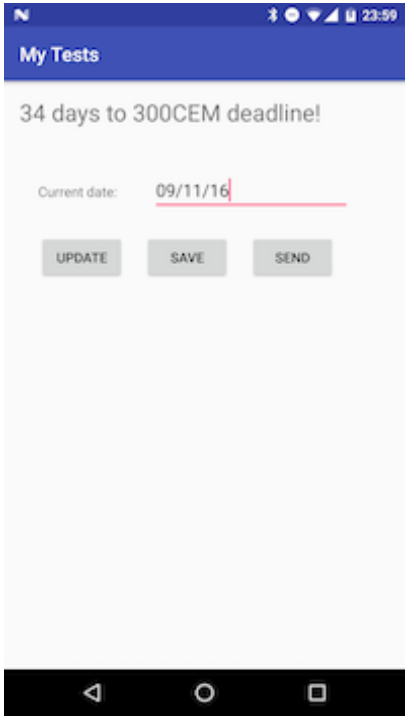
https://developer.android.com/training/testing/start/index.html

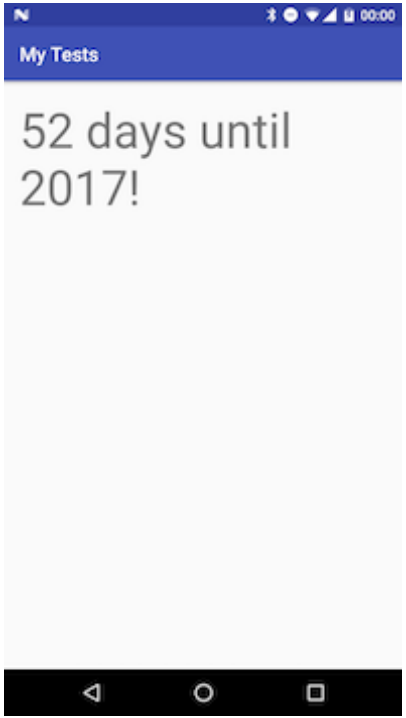
Test locations



https://developer.android.com/studio/test/index.html

The ‘Deadline’ app





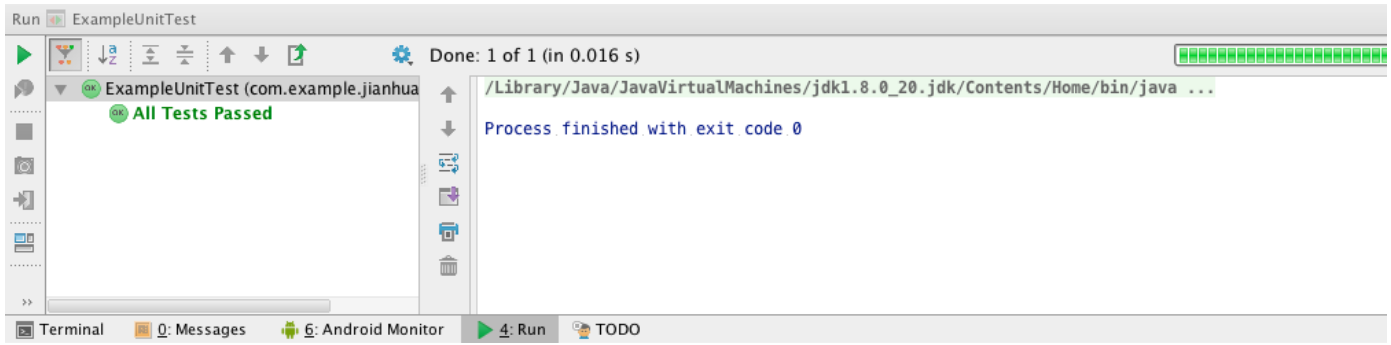
Local unit test

```
public class DeadlineTest {
    @Test
    public void calculate() throws Exception {
        Deadline deadline = new Deadline("12/12/16");
        assertEquals(deadline.calculate(), 1);
    }
}
```

- Naming conventions http://stackoverflow.com/questions/3146821/naming-convention-junit-suffix-or-prefix-test
- Assertion parameter order is expected value followed by actual value. Optionally the first parameter can be a String message that is output on failure. https://github.com/junit-team/junit4/wiki/assertions

Local unit test

- The color of the status bar indicates whether the tests have passed successfully.
- The left-hand pane shows the tree view of all tests
- The toolbar provides controls that enable you to monitor the tests and analyze results.



Local mocked unit tests

Dependencies:

** SmallTest filter

- testCompile 'com.android.support.test:rules:0.5'

\\Mockito runner

- testCompile 'org.mockito:mockito-core:1.10.19'

\\matcher

- testCompile 'org.hamcrest:hamcrest-library:1.3'

Why mock?

At runtime, local unit tests will be executed against a modified version of android.jar where all final modifiers have been stripped off.

https://developer.android.com/studio/test/index.html

```
@SmallTest
@RunWith(MockitoJUnitRunner.class)
public class DeadlineTest {

    @Mock
    Context context;

    @Mock
    Activity activity;

    @Mock
    SharedPreferences sharedPreferences;

    @Mock
    SharedPreferences.Editor editor;
```

Lab sheet example

```
@Before
public void initTests() {
    deadline = new Deadline("10/12/16", activity);
}

@Test
public void testSave() throws Exception {

    when(activity.getPreferences(Context.MODE_PRIVATE)).thenReturn(sharedPreferences);
    when(sharedPreferences.edit()).thenReturn(editor);
    when(editor.commit()).thenReturn(true);
    assertThat(deadline.save(), is(true));
}
```

JUnit

JUnit

Mockito

Hamcrest

http://stackoverflow.com/questions/27256429/is-org-junit-assert-assertthat-better-than-org-hamcrest-matcherassert-assertthat

Lab 2

- Instrumented tests
- UI tests using Espresso

Instrumented tests dependencies

In build.gradle

```
androidTestCompile 'com.android.support:support-annotations:24.2.1'
androidTestCompile 'com.android.support.test:runner:0.5'
androidTestCompile 'com.android.support.test:rules:0.5'
androidTestCompile 'org.hamcrest:hamcrest-library:1.3'
```

Instrumented tests

```
@RunWith(AndroidJUnit4.class)
@SmallTest
public class InstrumentedDeadlineTest {

    private Deadline deadline;

    @Before
    public void initTests() {
        Context context = InstrumentationRegistry.getInstrumentation().getTargetContext();
        deadline = new Deadline("04/12/16", context);
    }

    @Test
    public void testCalculate() {
        Log.d("actual_results", Integer.toString(deadline.calculate()));
        assertThat(deadline.calculate(), is(equalTo(9)));
    }

}
```

Android instrumentation

Android instrumentation is a set of control methods or 'hooks' in the Android system. With Android instrumentation, you can invoke for example activity callback methods in your test code

https://stuff.mit.edu/afs/sipb/project/android/docs/tools/testing/testing_android.html#Instrumentation

Instrumentation class

Instrumentation

```
public class Instrumentation
extends Object
```

java.lang.Object

↳ android.app.Instrumentation

Known Direct Subclasses

InstrumentationTestRunner

Known Indirect Subclasses

MultiDexTestRunner

Base class for implementing application instrumentation code. When running with instrumentation turned on, this class will be instantiated for you before any of the application code, allowing you to monitor all of the interaction the system has with the application.

https://developer.android.com/reference/android/app/Instrumentation.html

Test runner vs @RunWith

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

AndroidJUnitRunner

```
public class AndroidJUnitRunner
extends MonitoringInstrumentation
```

java.lang.Object
  ↳ android.app.Instrumentation
    ↳ android.support.test.runner.MonitoringInstrumentation
      ↳ android.support.test.runner.AndroidJUnitRunner

An Instrumentation that runs JUnit3 and JUnit4 tests against an Android package (application).

https://developer.android.com/reference/android/support/test/runner/AndroidJUnitRunner.html

Test runner vs @RunWith

Aliases the current default Android JUnit 4 class runner, for future-proofing.

AndroidJUnit4

```
public final class AndroidJUnit4
extends BlockJUnit4ClassRunner
```

java.lang.Object
↳ org.junit.runner.Runner
↳ org.junit.runners.ParentRunner<org.junit.runners.model.FrameworkMethod>
↳ org.junit.runners.BlockJUnit4ClassRunner
↳ android.support.test.runner.AndroidJUnit4

https://developer.android.com/reference/android/support/test/runner/AndroidJUnit4.html

Espresso tests

```
@RunWith(AndroidJUnit4.class)
@SmallTest
public class EspressoTest {

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

    @Test
    public void changeText_sameActivity() {
        onView(withId(R.id.editText)).perform(typeText("12/12/16"), closeSoftKeyboard());
        onView(withId(R.id.buttonUpdate)).perform(click());
        onView(withId(R.id.textView)).check(matches(withText("1 days to 300CEM deadline!")));
    }
```

ActivityTestRule

ActivityTestRule

```
public class ActivityTestRule
extends UiThreadTestRule
```

java.lang.Object
↳ android.support.test.rule.UiThreadTestRule
↳ android.support.test.rule.ActivityTestRule<T extends android.app.Activity>

Known Direct Subclasses

IntentsTestRule<T extends Activity>

This rule provides functional testing of a single activity. The activity under test will be launched before each test annotated with Test and before methods annotated with Before.

https://developer.android.com/reference/android/support/test/rule/ActivityTestRule.html

Espresso workflow

1. Find the UI component you want to test in an `Activity` (for example, a sign-in button in the app) by calling the `onView()` method, or the `onData()` method for `AdapterView` controls.

2. Simulate a specific user interaction to perform on that UI component, by calling the `ViewInteraction.perform()` or `DataInteraction.perform()` method and passing in the user action (for example, click on the sign-in button). To sequence multiple actions on the same UI component, chain them using a comma-separated list in your method argument.

3. Repeat the steps above as necessary, to simulate a user flow across multiple activities in the target app.

4. Use the `ViewAssertions` methods to check that the UI reflects the expected state or behavior, after these user interactions are performed.

https://developer.android.com/training/testing/ui-testing/espresso-testing.html

Espresso workflow

```
onView(withId(R.id.my_view))            // withId(R.id.my_view) is a ViewMatcher
        .perform(click())               // click() is a ViewAction
        .check(matches(isDisplayed())); // matches(isDisplayed()) is a ViewAssertion
```

* ViewMatchers: onView(withText("Sign-in")), onView(withId(R.id.button_signin));
* ViewAction:
  - `ViewActions.click()`: Click
  - `ViewActions.typeText()`: C
  - `ViewActions.scrollTo()`: S
    property must be `VISIBLE`. Fo
  - `ViewActions.pressKey()`: P
  - `ViewActions.clearText()`:
* ViewAssertion:
  - `doesNotExist`: Asserts that the
  - `matches`: Asserts that the speci
  - `selectedDescendentsMatch`:

https://developer.android.com/training/testing/ui-testing/espresso-testing.html

Espresso Test Recorder

Still in beta

https://storage.googleapis.com/androiddevelopers/videos/studio/espresso-test-recorder-overview_v2.mp4