JUnit &

Mockito

**GitHub Repository**

*https://github.com/in28minutes?tab=repositories*

*https://github.com/in28minutes/MockitoTutorialForBeginners*

*https://github.com/in28minutes/JUnitIn28Minutes*

Junit

<https://github.com/in28minutes/JUnitIn28Minutes>

## Get started with Junit

1. [Course Overview](https://github.com/in28minutes/JUnitIn28Minutes" \l "topics)
   1. [Section 1 Foundation](https://github.com/in28minutes/JUnitIn28Minutes" \l "section-1-foundation)
   2. [Section 2 First JUnit Example](https://github.com/in28minutes/JUnitIn28Minutes" \l "section-2-first-junit-example)
   3. [Section 3 Second JUnit Example](https://github.com/in28minutes/JUnitIn28Minutes" \l "section-3-second-junit-example)
   4. [Section 4 JUnit Simple Test Scenarios](https://github.com/in28minutes/JUnitIn28Minutes" \l "section-4-junit-simple-test-scenarios)
   5. [Section 5 JUnit Intermediate Test Scenarios](https://github.com/in28minutes/JUnitIn28Minutes" \l "section-5-junit-intermediate-test-scenarios)
   6. [Section 6 JUnit Best Practices](https://github.com/in28minutes/JUnitIn28Minutes" \l "section-6-junit-best-practices)
   7. [Starting Examples](https://github.com/in28minutes/JUnitIn28Minutes" \l "starting-examples)
2. [About in28Minutes](https://github.com/in28minutes/JUnitIn28Minutes" \l "about-in28minutes)
   1. [Our Beliefs](https://github.com/in28minutes/JUnitIn28Minutes" \l "our-beliefs)
   2. [Our Approach](https://github.com/in28minutes/JUnitIn28Minutes" \l "our-approach)
   3. [Find Us](https://github.com/in28minutes/JUnitIn28Minutes" \l "useful-links)
   4. [Other Courses](https://github.com/in28minutes/JUnitIn28Minutes" \l "other-courses)

## Topics

### Section 1 Foundation

1. What is JUnit?
2. Why Unit Testing?

### Section 2 First JUnit Example

@Test Annotation

1. Running JUnit
2. No Failure = Success
3. Basic Assert methods

### Section 3 Second JUnit Example

* assertTrue and assertFalse methods
* @Before @After annotations
* @BeforeClass @AfterClass annotations

### Section 4 JUnit Simple Test Scenarios

* Comparing Arrays
* Testing Exceptions
* Performance Unit Tests

### Section 5 JUnit Intermediate Test Scenarios

1. Parameterized Tests
2. Test Suites

### Section 6 JUnit Best Practices

* Naming Test Methods
* Highlight Important Values in Tests
* Handle Exceptions Properly
* Readable Assert Statements

### Starting Examples

*package com.in28minutes.junit.helper;*

*public class StringHelper {*

*public String truncateAInFirst2Positions(String str) {*

*if (str.length() <= 2)*

*return str.replaceAll("A", "");*

*String first2Chars = str.substring(0, 2);*

*String stringMinusFirst2Chars = str.substring(2);*

*return first2Chars.replaceAll("A", "")*

*+ stringMinusFirst2Chars;*

*}*

*public boolean areFirstAndLastTwoCharactersTheSame(String str) {*

*if (str.length() <= 1)*

*return false;*

*if (str.length() == 2)*

*return true;*

*String first2Chars = str.substring(0, 2);*

*String last2Chars = str.substring(str.length() - 2);*

*return first2Chars.equals(last2Chars);*

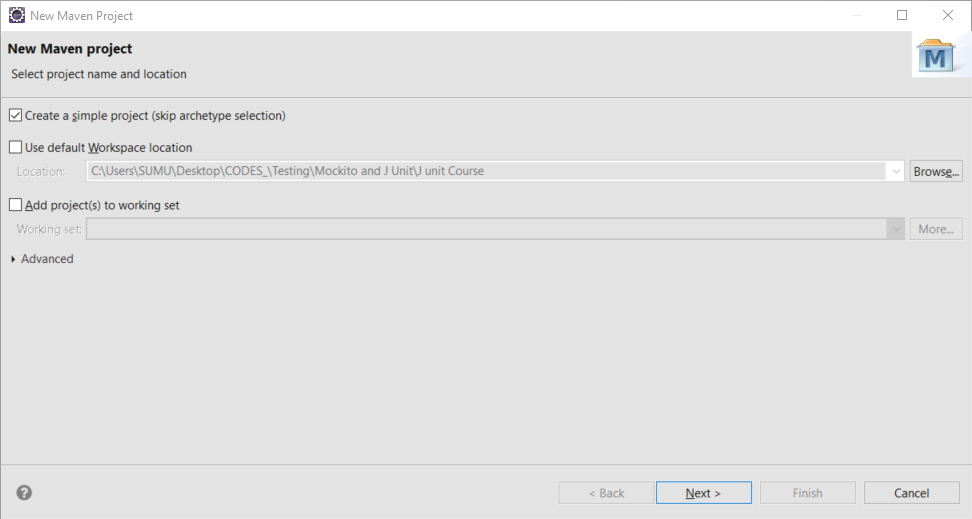
*}*

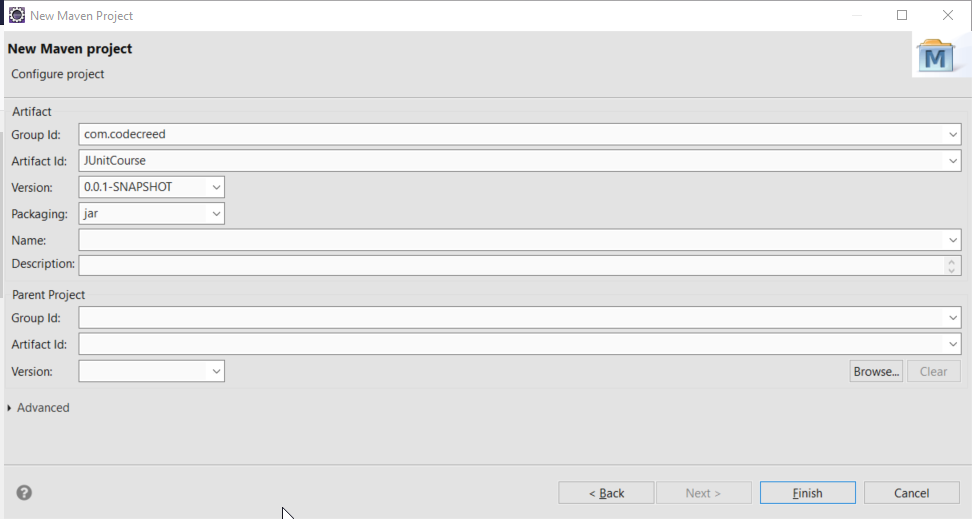
*}*

## Foundation

The advantage of unit testing is that I can specially check for a part of code (ie methods) within milliseconds.

Writing Great Unit Tests distinguishes Good Programmers from Great Programmers.





Starter example code

https://github.com/in28minutes/JUnitIn28Minutes

# Foundation

**package** com.in28minutes.junit.helper;

**public** **class** StringHelper {

**public** String truncateAInFirst2Positions(String str) {

**if** (str.length() <= 2)

**return** str.replaceAll("A", "");

String first2Chars = str.substring(0, 2);

String stringMinusFirst2Chars = str.substring(2);

**return** first2Chars.replaceAll("A", "")

+ stringMinusFirst2Chars;

}

**public** **boolean** areFirstAndLastTwoCharactersTheSame(String str) {

**if** (str.length() <= 1)

**return** **false**;

**if** (str.length() == 2)

**return** **true**;

String first2Chars = str.substring(0, 2);

String last2Chars = str.substring(str.length() - 2);

**return** first2Chars.equals(last2Chars);

}

}

### In Monochrome

package com.in28minutes.junit.helper;

public class StringHelper {

public String truncateAInFirst2Positions(String str) {

if (str.length() <= 2)

return str.replaceAll("A", "");

String first2Chars = str.substring(0, 2);

String stringMinusFirst2Chars = str.substring(2);

return first2Chars.replaceAll("A", "")

+ stringMinusFirst2Chars;

}

public boolean areFirstAndLastTwoCharactersTheSame(String str) {

if (str.length() <= 1)

return false;

if (str.length() == 2)

return true;

String first2Chars = str.substring(0, 2);

String last2Chars = str.substring(str.length() - 2);

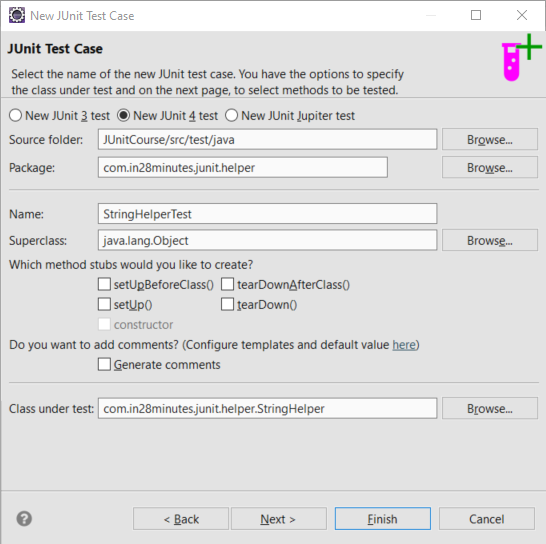
return first2Chars.equals(last2Chars);

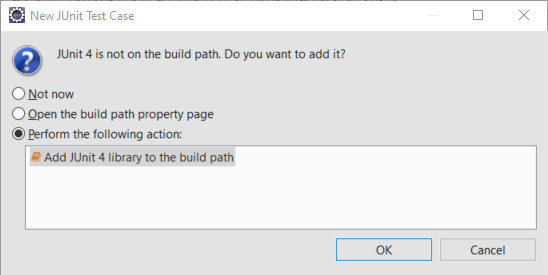
}

}

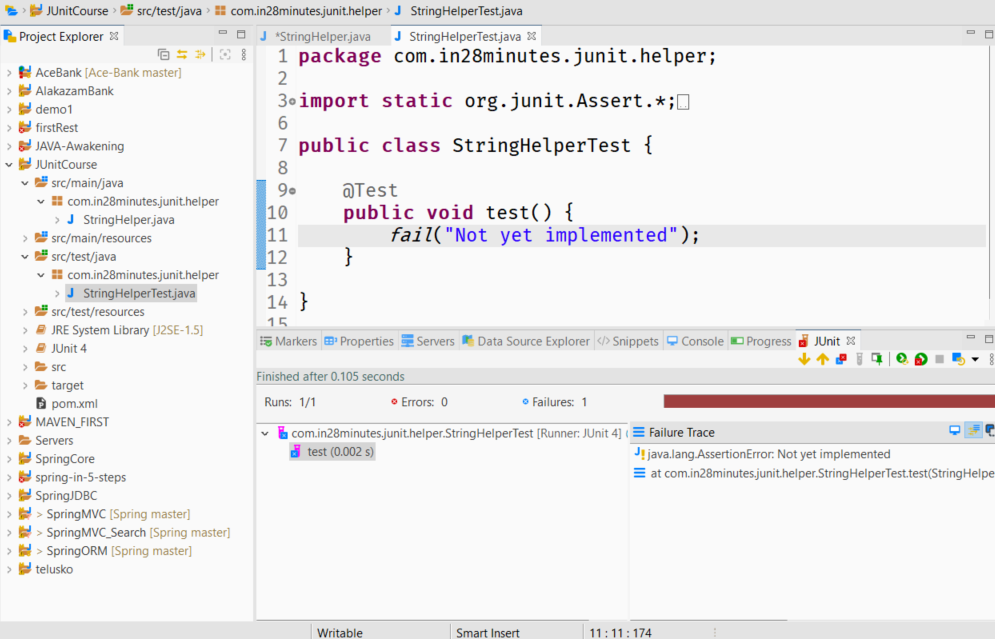
### First JUnit Example

* paste in src/main/java
* and automagically alll the packages and classes will be creates
* create a new J Unit Test class in the */src/test/java*





Creating a Junit test case in /src/test/java



### fail()

run as > junit test case

red bar becaus of **fail()** method

### **Basic Starter Unit TestCase**

*package com.in28minutes.junit.helper;*

*import static org.junit.Assert.\*;*

*import org.junit.Test;*

*public class StringHelperTest {*

*@Test*

*public void testTruncateAInFirst2Positions() {*

*fail("Not yet implemented");*

*}*

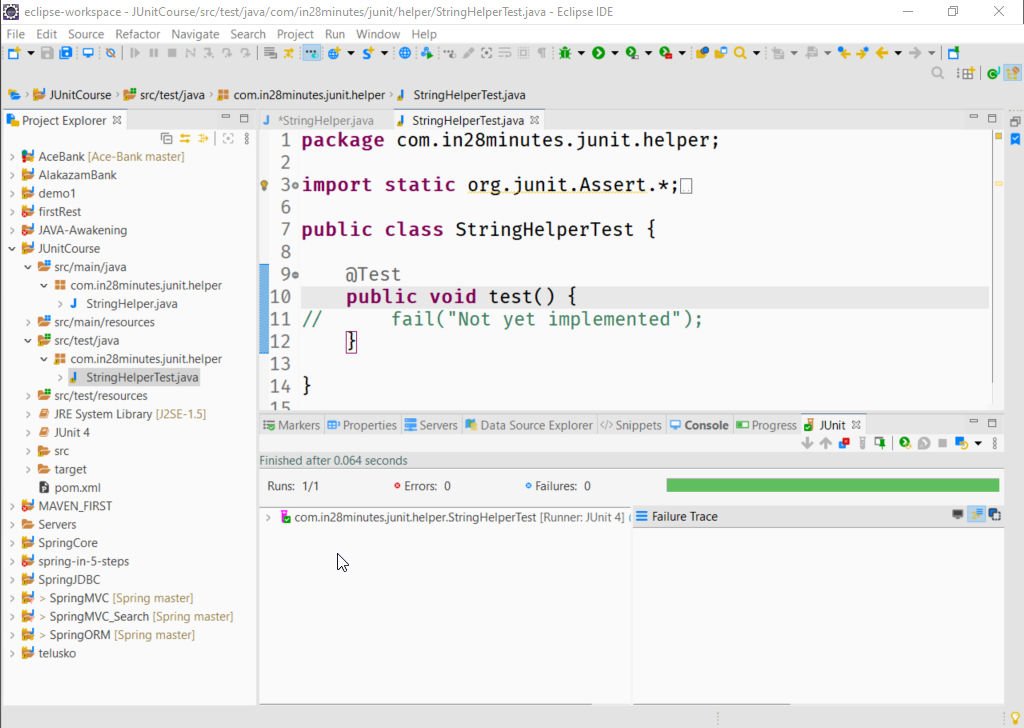
*@Test*

*public void testAreFirstAndLastTwoCharactersTheSame() {*

*fail("Not yet implemented");*

*}*

*}*



***if there are no failures or expectation then a test is said to be a success.***

***So if we don’t test anything the test will always pass and that's not a good idea.***

So if we comment the fail() call the test is success {the **green** bar}

fail|() is one of the method in a series of test methods and it can fail a test.

### assertEquals()

To check two values are equal or not.

package com.in28minutes.junit.helper;

import static org.junit.Assert.\*;

import org.junit.Test;

public class StringHelperTest {

@Test

public void test() {

// fail("Not yet implemented");

assertEquals("ABC", "ABCD");

}

}

assertEquals("ABC", "ABCD");

//**expected, actual**

org.junit.ComparisonFailure: expected:<ABC[]> but was:<ABC[D]>

at org.junit.Assert.assertEquals(Assert.java:117)

....

assertEquals("ABCD", "ABCD");

package com.in28minutes.junit.helper;

import static org.junit.Assert.\*;

import org.junit.Test;

public class StringHelperTest {

@Test

**public void** test() {

// fail("Not yet implemented");

// assertEquals("ABCD", "ABCD");

StringHelper helper = new StringHelper();

String actual = helper.truncateAInFirst2Positions("AACD");

String expected = "CD";

assertEquals(**expected** ,**actual**);

}

}

//refactoring the variable to inline

package com.in28minutes.junit.helper;

import static org.junit.Assert.\*;

import org.junit.Test;

public class StringHelperTest {

@Test

**public void** test() {

// fail("Not yet implemented");

// assertEquals("ABCD", "ABCD");

StringHelper helper = new StringHelper();

assertEquals(helper.truncateAInFirst2Positions("AACD"), "CD");

}

}

**EA**

1. its never a good idea to test 2 or more conditions in the same @Test method
2. runs => no of test cases
3. failures => no of test which didn't ran from
4. when case of multiple @Test methods if we choose to run only a single 2test method -> highlight the method and run as Junit test case.
5. We can also put the helper object at class level instead of method level.
6. The visiblity should be public to be visible to other frameworks

**package** com.in28minutes.junit.helper;

**import** **static** org.junit.Assert.\*;

**import** org.junit.Test;

**public** **class** StringHelperTest {

StringHelper helper = **new** StringHelper();

@Test

**public** **void** testtruncateAInFirst2Positions\_ConditionsOne() {

// fail("Not yet implemented");

// assertEquals("ABCD", "ABCD");

// StringHelper helper = new StringHelper();

// assertEquals("CD",helper.truncateAInFirst2Positions("AACD"));

// its not a good idea to perform to add 2 or more checks in the same unit test

*assertEquals*( "CD",helper.truncateAInFirst2Positions("AACD"));

}

@Test

**public** **void** testtruncateAInFirst2Positions\_ConditionsTwo() {

// public void -> must

// it should be visible to other frameworks which may call

// StringHelper helper = new StringHelper();

*assertEquals*("CD",helper.truncateAInFirst2Positions("AACD"));

}

}



### package com.in28minutes.junit.helper;

import static org.junit.Assert.\*;

import org.junit.Test;

**StringHelperTest.java**

public class StringHelperTest {

/\*

\* AACD -> CD ACD -> CD CDEF -> CDEF CDAA -. CDAA

\*/

**StringHelper helper = new StringHelper();**

@Test

**public void test**TruncateAInFirst2Positions**\_AinFirst2Position**() {

String expected = "CD";

assertEquals(expected, helper.truncateAInFirst2Positions("AACD"));

}

@Test

public void testAreFirstAndLastTwoCharactersTheSame\_AinFirstPosition() {

String expected = "CD";

assertEquals(expected, helper.truncateAInFirst2Positions("ACD"));

}

}

### **Second JUnit Example**

### **assertFalse()**

Checks for false, test passes if the output is false

### **assertTrue()**

test passes if the output is true

\*both the above methods can be overloaded with a String parameter in the beginning

**package** com.in28minutes.junit.helper;

**import** **static** org.junit.Assert.*assertEquals*;

**import** **static** org.junit.Assert.*assertFalse*;

**import** **static** org.junit.Assert.*assertTrue*;

**import** org.junit.Test;

**public** **class** StringHelperTest {

StringHelper helper = **new** StringHelper();

// AACD => CD, ACD => CD ,CDEF=> CDAA=> CDAA

@Test

**public** **void** testTruncateAInFirst2Positions\_ConditionsOne() {

// fail("Not yet implemented");

// assertEquals("ABCD", "ABCD");

// StringHelper helper = new StringHelper();

// assertEquals("CD",helper.truncateAInFirst2Positions("AACD"));

// its not a good idea to perform to add 2 or more checks in the same unit test

*assertEquals*( "CD",helper.truncateAInFirst2Positions("AACD"));

}

@Test

**public** **void** testTruncateAInFirst2Positions\_ConditionsTwo() {

// public void -> must

// public because, it should be visible to other frameworks which may call

// StringHelper helper = new StringHelper();

*assertEquals*("CD",helper.truncateAInFirst2Positions("AACD"));

}

// ABCD => false ; ABAB => true ; AB => true ; A = false

@Test

**public** **void** testAreFirstAndLastTwoCharactersTheSame\_BasicNegativeScenario() {

// assertEquals(false, helper.areFirstAndLastTwoCharactersTheSame("ABC"));

*assertFalse*("MSG: failed for 'false'",helper.areFirstAndLastTwoCharactersTheSame("AB"));

}

@Test

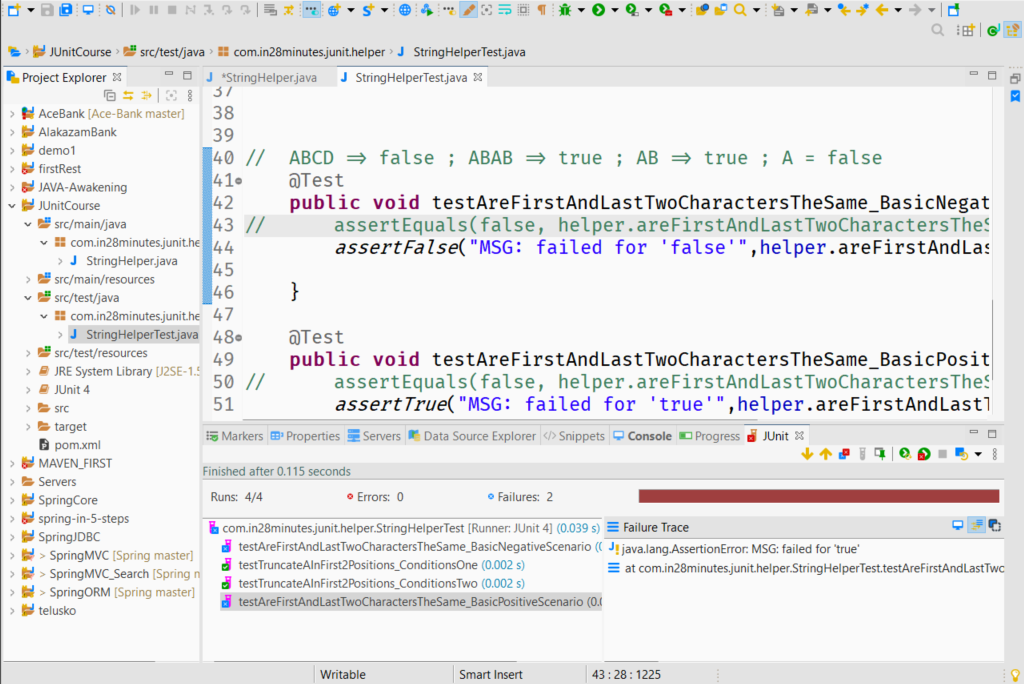
**public** **void** testAreFirstAndLastTwoCharactersTheSame\_BasicPositiveScenario() {

// assertEquals(false, helper.areFirstAndLastTwoCharactersTheSame("ABC"));

*assertTrue*("MSG: failed for 'true'",helper.areFirstAndLastTwoCharactersTheSame("ABC"));

}

}

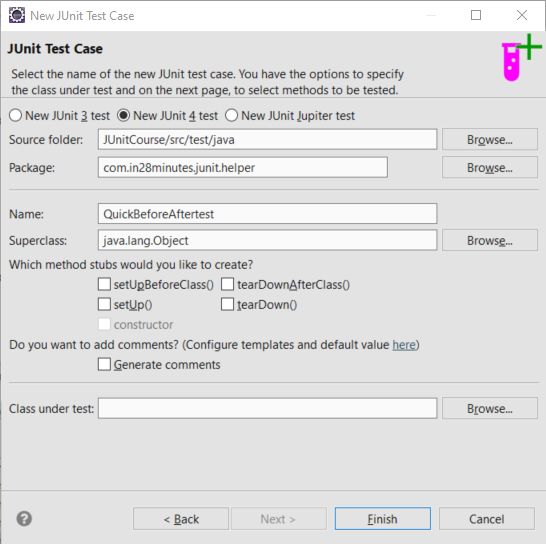


### @Before

This can be used for a method which should be precursor for all the @tests

### @**After**

This can be used for a method which should be successor for all the @tests



**package** com.in28minutes.junit.helper;

**import** org.junit.After;

**import** org.junit.Before;

**import** org.junit.Test;

**public** **class** QuickBeforeAftertest {

@Before

**public** **void** setup() {

System.***out***.println("Before test \"Initializers & Precurssors\"");

}

@After

**public** **void** teardown() {

System.***out***.println("After test \"Dispossal\"");

}

@Test

**public** **void** test1() {

System.***out***.println("test 1 excuted");

}

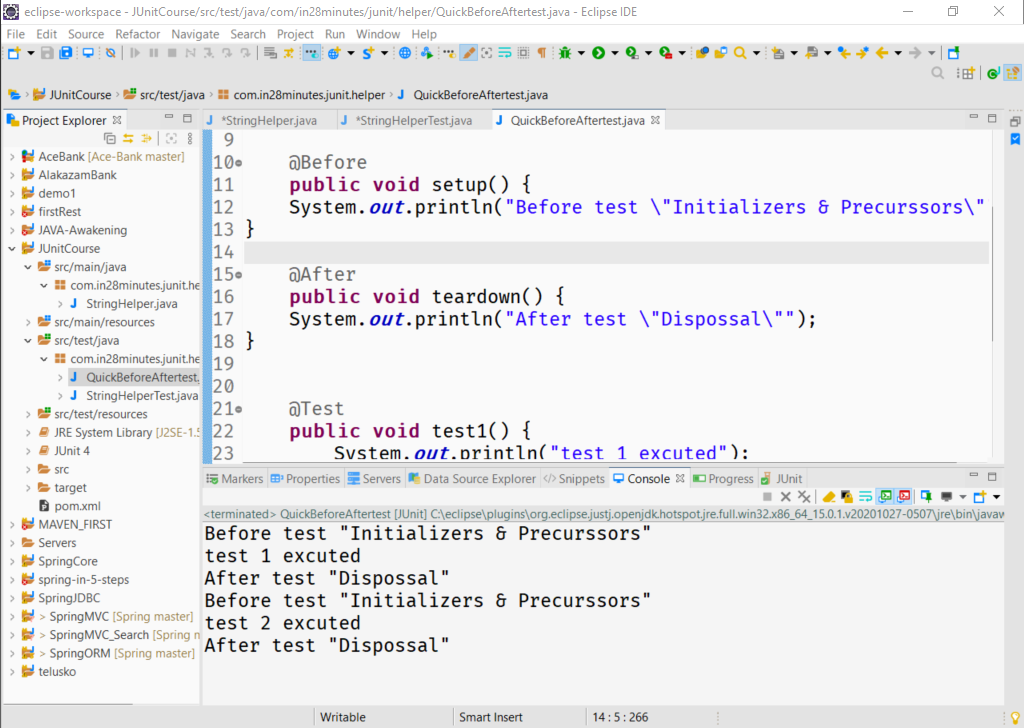
@Test

**public** **void** test2() {

System.***out***.println("test 2 excuted");

}

}



**import** **static** org.junit.Assert.*assertTrue*;

**import** org.junit.Before;

**import** org.junit.Test;

**public** **class** StringHelperTest {

StringHelper helper;

@Before

**public** **void** before() {

helper = **new** StringHelper();

}

// AACD => CD, ACD => CD ,CDEF=> CDAA=> CDAA

@Test

### @**BeforeClass**

This can be used for a method which should be precursor for the @tests

and it executes once only.

The methods should be **public** **static** **void**

### @**AfterClass**

This can be used for a method which should be precursor for the @tests

and it executes once only.

The methods should be **public** **static** **void**

**package** com.in28minutes.junit.helper;

**import** org.junit.After;

**import** org.junit.AfterClass;

**import** org.junit.Before;

**import** org.junit.BeforeClass;

**import** org.junit.Test;

**public** **class** QuickBeforeAftertest {

@Before

**public** **void** initialize() {

System.***out***.println("Before test \"Initializers & Precurssors\"");

}

@After

**public** **void** teardown() {

System.***out***.println("After test \"Dispossal\"");

}

@BeforeClass

**public** **static** **void** setup() {

System.***out***.println("\*\*\*Before execution of class\"Initializers & Precurssors\"");

}

@AfterClass

**public** **static** **void** dispose() {

System.***out***.println("\*\*\*After execution of class \"Dispossal\"");

}

@Test

**public** **void** test1() {

System.***out***.println("test 1 excuted");

}

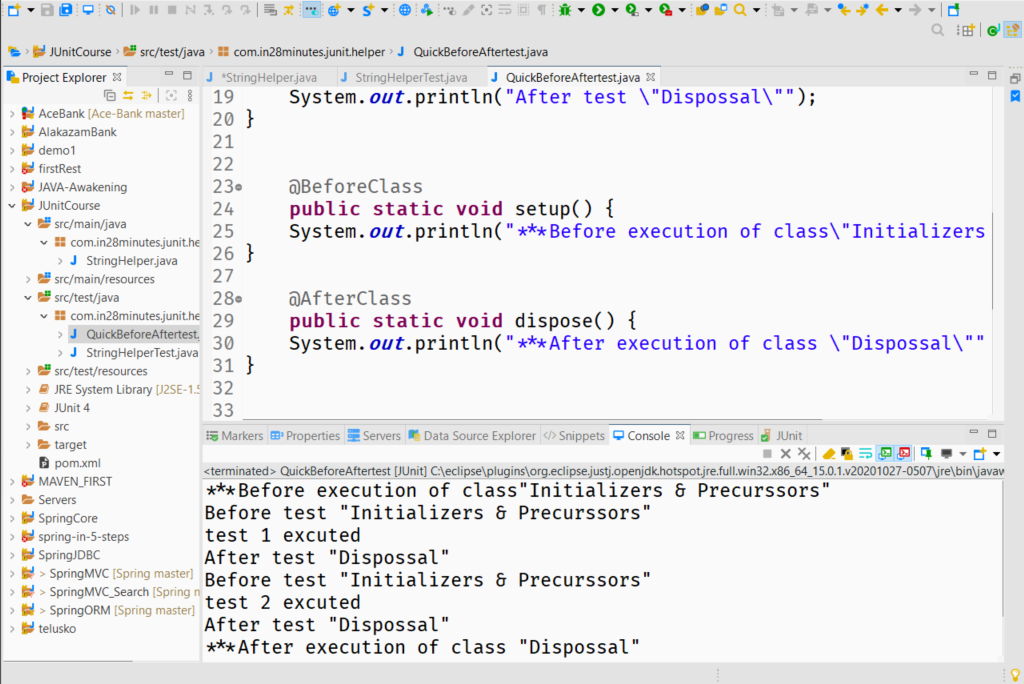
@Test

**public** **void** test2() {

System.***out***.println("test 2 excuted");

}

}

****

### assertEquals(expected,sample)

*Checks their reference* ***only***

### assertArrayEquals(expected,sample)

* Checks their values
* If failed , it will also tell at what position the test failed.
* if negative result, it will also provide like additional details of trace stack.

\*\*\* We can also get Exceptions in the Test Cases,

* So we can handle it by try-catch
* Or we can mention at @Test annotation, using expected attribute
* If exception thrown , the test will pass

@Test(expected=NullPointerException.**class**)

but if there is no exception its a failure

also a no failure == success

**package** com.in28minutes.junit.helper;

**import** **static** org.junit.Assert.*assertArrayEquals*;

**import** java.util.Arrays;

**import** org.junit.Test;

**public** **class** ArraysCompareTestJava {

@Test

**public** **void** testArray\_test1() {

**int** sample[]= {15,52,31,41,15,16,177};

**int** expected[]= {15,15,16,31,41,52,177};

Arrays.*sort*(sample);

// assertEquals(expected,sample);// checks the reference

*assertArrayEquals*(expected,sample);// checks the reference

}

@Test

**public** **void** testArray\_test2() {

**int** sample[]= {15,52,31,41,15,16,177};

**int** expected[]= {15,15,16,41,31,52,177};//wrong

/\* arrays first differed at element [3]; expected:<41> but was:<31> at org.junit.internal.ComparisonCriteria.arrayEquals(ComparisonCriteria.java:78)

\*/

Arrays.*sort*(sample);

*assertArrayEquals*(expected,sample);// checks the reference

}

@Test(expected=NullPointerException.**class**)

//if the mentioned exception will come then test pass

**public** **void** testArray\_test3() {

**int** sample[]= **null**;

**int** expected[]= {15,15,16,41,31,52,177};//wrong

Arrays.*sort*(sample);//NullPointerException

*assertArrayEquals*(expected,sample);// checks the reference

}

}

### Performance Testing

we can mention at @Test annotation, using timeout attribute.

package com.in28minutes.junit.helper;

import static org.junit.Assert.assertArrayEquals;

import java.util.Arrays;

import java.util.Iterator;

import org.junit.Test;

public class ArraysCompareTestJava {

@Test

public void testArray\_test1() {

int sample[]= {15,52,31,41,15,16,177};

int expected[]= {15,15,16,31,41,52,177};

Arrays.sort(sample);

// assertEquals(expected,sample);// checks the reference

assertArrayEquals(expected,sample);// checks the reference

}

@Test

public void testArray\_test2() {

int sample[]= {15,52,31,41,15,16,177};

int expected[]= {15,15,16,31,41,52,177};//wrong

Arrays.sort(sample);

assertArrayEquals(expected,sample);// checks the reference

}

@Test(expected=NullPointerException.class)

//if the mentioned exception will come then test pass

public void testArray\_test3() {

int sample[]= null;

int expected[]= {15,15,16,41,31,52,177};//wrong

Arrays.sort(sample);//NullPointerException

// assertArrayEquals(expected,sample);// checks the reference

}

**@Test(timeout=1000)//miliseconds**

//if the mentioned time limit will surpass , test will fail

public void testSort\_Performance() {

int sample[]= {15,15,16,41,31,52,177};//wrong

for (int i = 0; i < 1000000; i++) {

sample[0]=i;

Arrays.sort(sample);

}

}

}

### Parametrized Test

We can automate things to some extent by Parametrizing the test, Its very handy when we want to test a sigle method using numerous test cases. Here we will use String arrays en capsuled in a collection and using

1. on class

@RunWith(Parameterized.**class**)

**public** **class** StringHelperParameterized {

2. on methods containing parameters

@Parameters

**public** **static** Collection<String[]> testCondition() {

3. create a constructor of the testing fields

**public** StringHelperParameterized(String input, String expected) {

**super**();

**this**.input = input;

**this**.expected = expected;

}

* and use of field constructors
  + generate using *right click, > source > constructors using fields*
  + we can also use ctrl+C , ctrl+v on the classes in the packages.

**package** com.in28minutes.junit.helper;

**import** **static** org.junit.Assert.*assertEquals*;

**import** **static** org.junit.Assert.*assertFalse*;

**import** **static** org.junit.Assert.*assertTrue*;

**import** java.util.Arrays;

**import** java.util.Collection;

**import** org.junit.BeforeClass;

**import** org.junit.Test;

**import** org.junit.runner.RunWith;

**import** org.junit.runners.Parameterized;

**import** org.junit.runners.Parameterized.Parameters;

@RunWith(Parameterized.**class**)

**public** **class** StringHelperParameterized {

**static** StringHelper *helper*;

**private** String input;

**private** String expected;

**public** StringHelperParameterized(String input, String expected) {

**super**();

**this**.input = input;

**this**.expected = expected;

}

@BeforeClass

**public** **static** **void** before() {

*helper* = **new** StringHelper();

}

@Parameters

**public** **static** Collection<String[]> testCondition() {

String expectedOutputs [][]={{"AACD" ,"CD"},

{"ACD","CD"},{"CDEF","CDEF"},{"CDAA","CDAA"}};

**return** Arrays.*asList*(expectedOutputs);

/\*

\* all the String arrays contents will be passes to constructor and a new test

\* instance will be created

\*/

}

// AACD => CD, ACD => CD ,CDEF=> CDEF ,CDAA=> CDAA

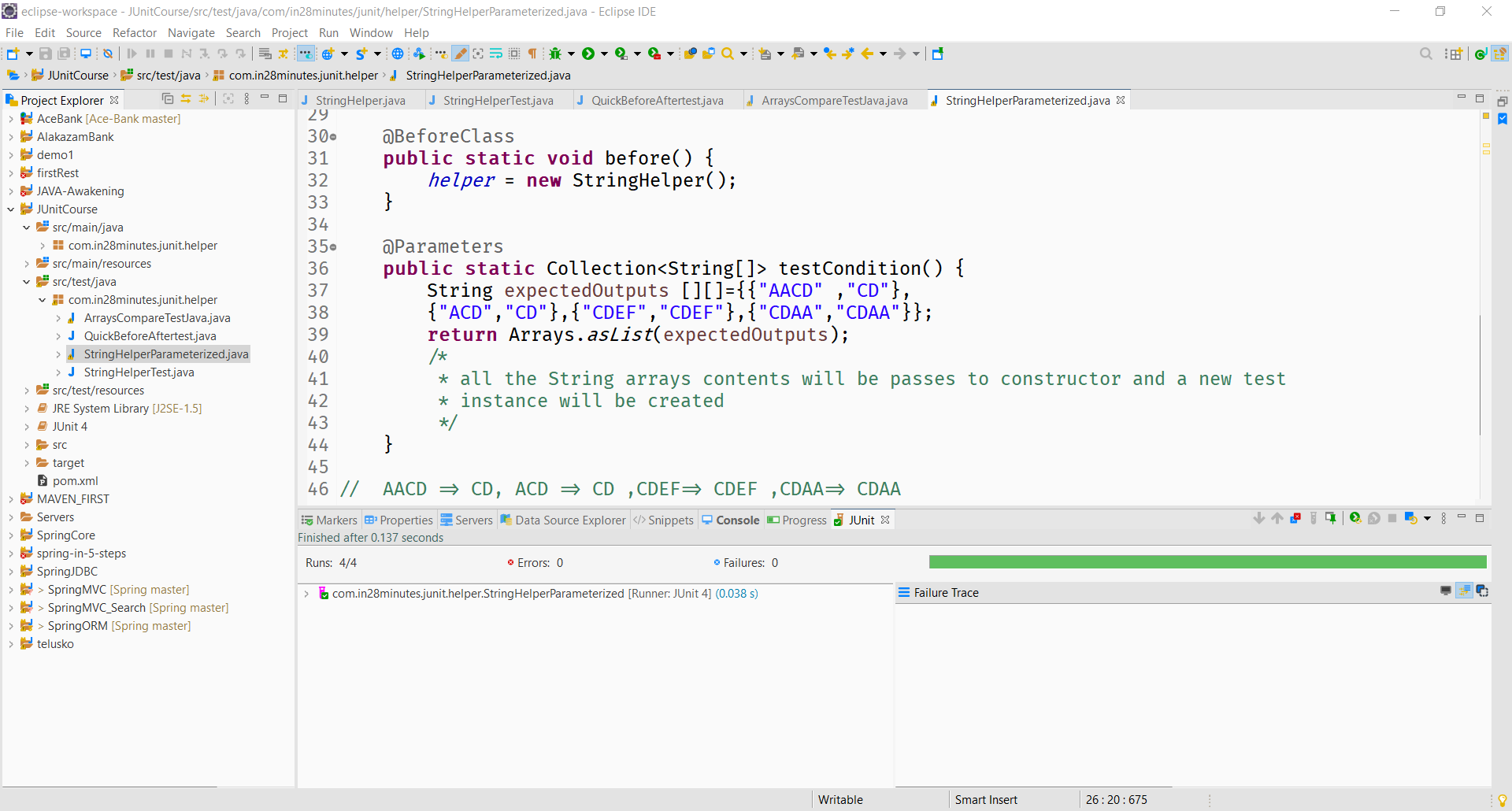
@Test

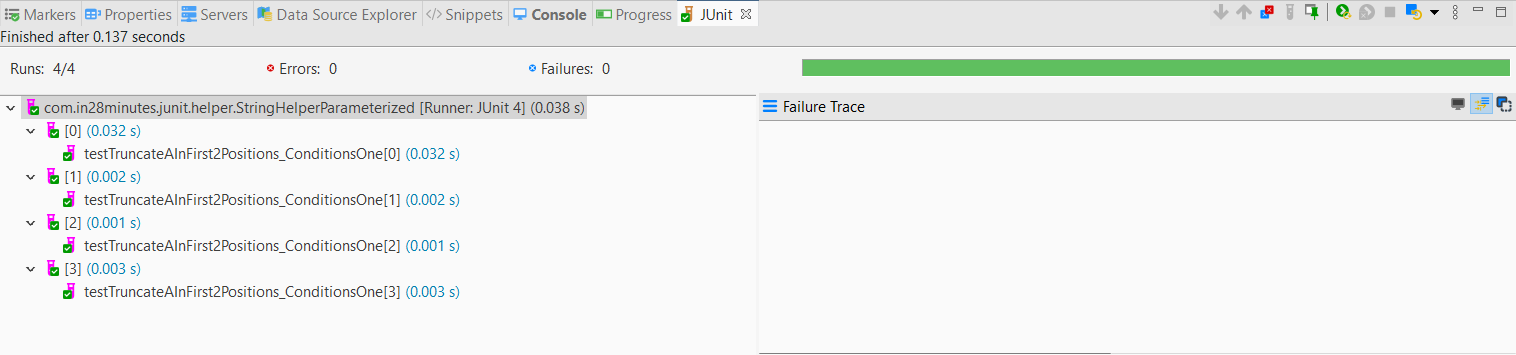
**public** **void** testTruncateAInFirst2Positions\_ConditionsOne() {

*assertEquals*( expected,*helper*.truncateAInFirst2Positions(input));

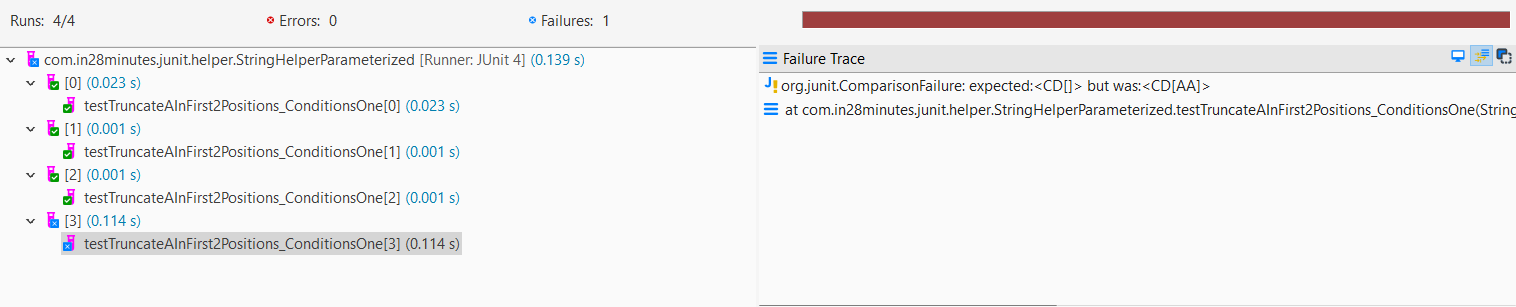
}

}





So 1 @Parameters method but **4 tests**

****

they are shown using indexes, here 3 passed and 1 failed

\*\*\* For another Parametrized test , use of a separate class is must

### Test Suites

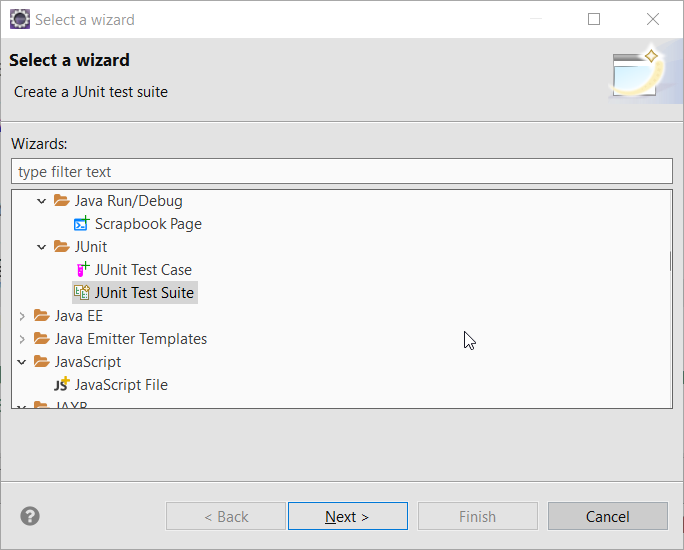
To organize our test we use test suite

If we do right click on a package and run as J Unit test, it will execute all the test in the

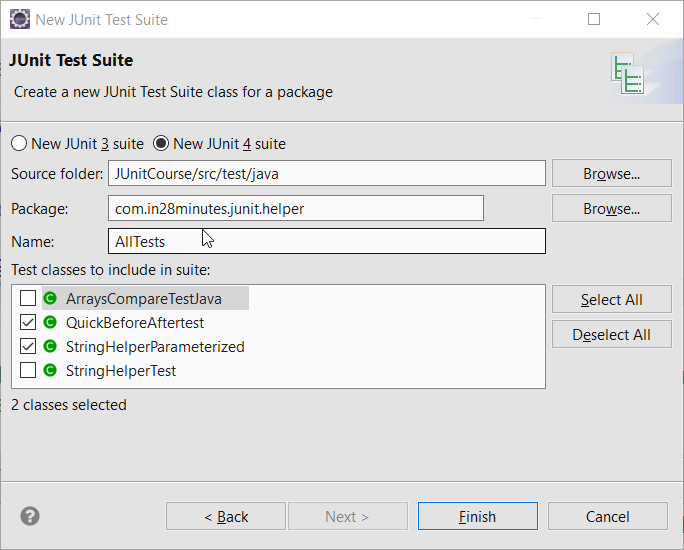
package, to get control over it we will use test suite.

**Steps:**

* Right click on the test package >>> J Unit test suite



* Select the desired test cases of the package



package com.in28minutes.junit.helper;

import org.junit.runner.RunWith;

import org.junit.runners.Suite;

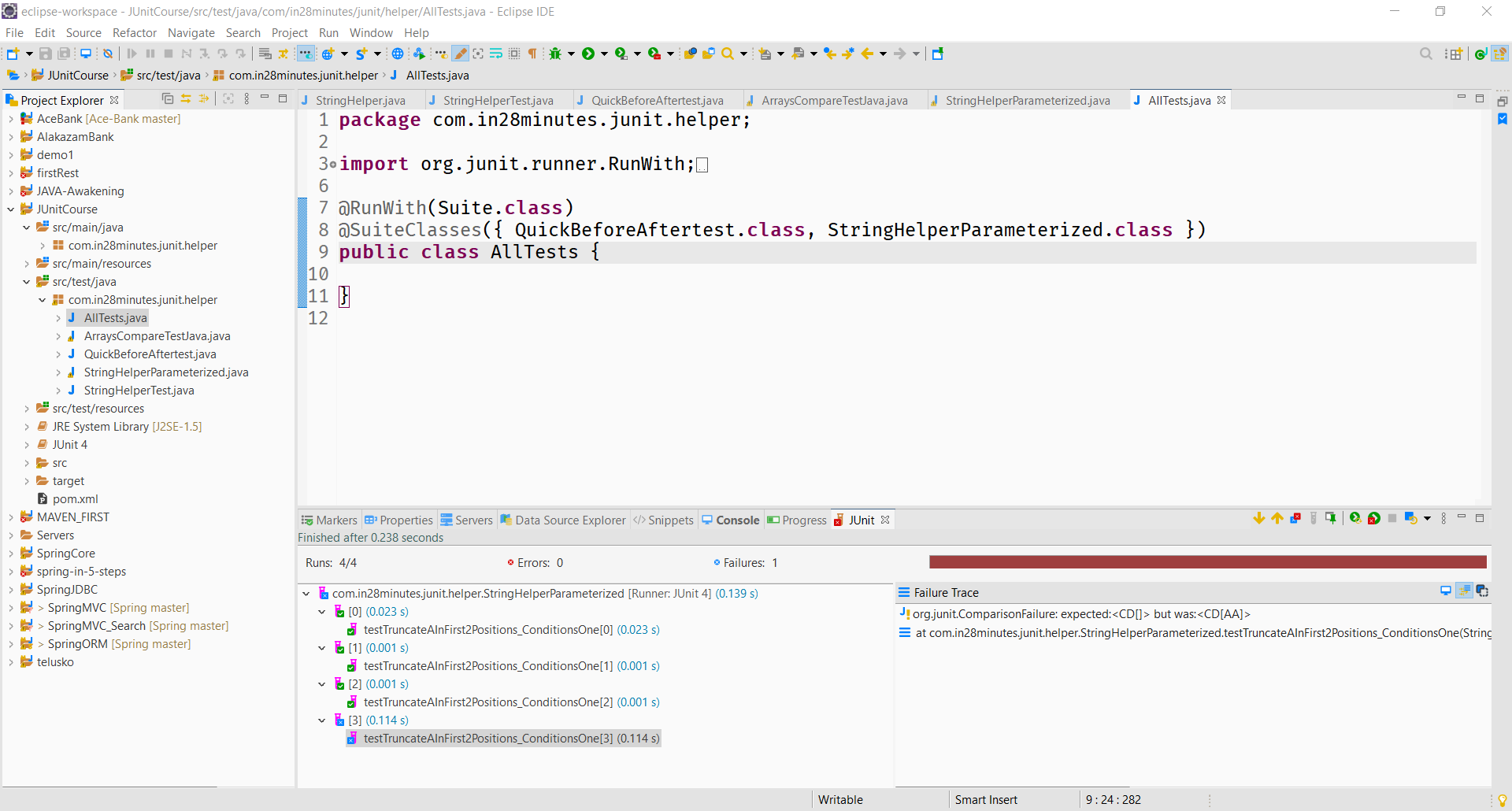
import org.junit.runners.Suite.SuiteClasses;

**@RunWith**(Suite.class)

**@SuiteClasses**({ QuickBeforeAftertest.class, StringHelperParameterized.class })

public class AllTests {

}

****

(Suite.class) **@SuiteClasses ->** Array of different classes

*Suites gives flexibility to select our desired test classes and increase the efficiency*

**MOCKITO**

*https://github.com/in28minutes/MockitoTutorialForBeginners*

## *Gain expertise on the most popular java mocking framework*

* [Installing Eclipse, Maven and Java](https://github.com/in28minutes/MockitoTutorialForBeginners" \l "installing-tools)
* [Running Examples](https://github.com/in28minutes/MockitoTutorialForBeginners" \l "running-examples)
* [Course Overview](https://github.com/in28minutes/MockitoTutorialForBeginners" \l "course-overview)
  + [Course Steps](https://github.com/in28minutes/MockitoTutorialForBeginners" \l "step-list)
  + [Expectations](https://github.com/in28minutes/MockitoTutorialForBeginners" \l "expectations)
* [About in28Minutes](https://github.com/in28minutes/MockitoTutorialForBeginners" \l "about-in28minutes)
  + [Our Beliefs](https://github.com/in28minutes/MockitoTutorialForBeginners" \l "our-beliefs)
  + [Our Approach](https://github.com/in28minutes/MockitoTutorialForBeginners" \l "our-approach)
  + [Find Us](https://github.com/in28minutes/MockitoTutorialForBeginners" \l "useful-links)
  + [Other Courses](https://github.com/in28minutes/MockitoTutorialForBeginners" \l "other-courses)

## Installing Tools

* PDF : <https://github.com/in28minutes/SpringIn28Minutes/blob/master/InstallationGuide-JavaEclipseAndMaven_v2.pdf>
* Video : <https://www.youtube.com/watch?v=eqRF4xHoGck>

## Course Overview

#### Mockito

* Step 01 : Set up an Eclipse Project with JUnit and Mockito frameworks. First Green Bar.
* Step 02 : Example to start understanding why we need mocks.
* Step 03 : What is a stub? Create an unit test using Stub? Disadvantages of Stubs.
* Step 04 : Your first Mockito code! Hurrah!!! Lets use Mockito to mock TodoService.
* Step 05 : Stubbing variations with Mockito. A few mockito examples mocking List class : Multiple return values, Argument Matchers and throwing exceptions.
* Some Theory : Mockito vs EasyMock <https://github.com/mockito/mockito/wiki/Mockito-vs-EasyMock>
* Step 06 : Introduction to BDD. Given When Then. BDD Mockito Syntax.
* Step 07 : How to verify calls on a mock? Verify how many times a method is called. We will add deleteTodo method to the TodoService.
* Step 08 : How to capture an argument which is passed to a mock?
* Step 09 : Hamcrest Matchers.
* Step 10 : Let's simplify things with Mockito Annotations. @Mock, @InjectMocks, @RunWith(MockitoJUnitRunner.class), @Captor
* Step 11 : JUnit Rules. Using MockitoJUnit.rule() instead of @RunWith(MockitoJUnitRunner.class).
* Step 12 : Real world Example with Spring
* Step 13 : What is a spy? How to spy with Mockito?
* Step 14 : Some Theory : Why does Mockito not allow stubbing final and private methods?
* Step 15 : Using PowerMock and Mockito to mock a Static Method.
* Step 16 : Using PowerMock and Mockito to invoke a private Method.
* Step 17 : Using PowerMock and Mockito to mock a constructor.
* Step 18 : Good Unit Tests.

### Expectations

* You should know Java.
* You are NOT expected to have any experience with Eclipse or Maven.
* We will help you install Eclipse and get up and running with Maven.

### Running Examples

* Download the zip or clone the Git repository.
* Unzip the zip file (if you downloaded one)
* Open Command Prompt and Change directory (cd) to folder containing pom.xml
* Open Eclipse
  + File -> Import -> Existing Maven Project -> Navigate to the folder where you unzipped the zip
  + Select the right project
* Choose the Spring Boot Application file (search for @SpringBootApplication)
* Right Click on the file and Run as Java Application
* You are all Set

**COURSE HIGHLIGHTS**

You take 12 steps with JUnit and 18 steps with Mockito into unit testing proficiency.

**Mocking with Mockito**

* Step 01 : Set up an Eclipse Project with JUnit and Mockito frameworks. First Green Bar.
* Step 02 : Example to start understanding why we need mocks.
* Step 03 : What is a stub? Create an unit test using Stub? Disadvantages of Stubs.
* Step 04 : Your first Mockito code! Hurrah!!! Lets use Mockito to mock TodoService.
* Step 05 : Stubbing variations with Mockito. A few mockito examples mocking List class : Multiple return values, Argument Matchers and throwing exceptions.
* Step 06 : Introduction to BDD. Given When Then. BDD Mockito Syntax.
* Step 07 : How to verify calls on a mock? Verify how many times a method is called. We will add deleteTodo method to the TodoService.
* Step 08 : How to capture an argument which is passed to a mock?
* Step 09 : Hamcrest Matchers.
* Step 10 : Let's simplify things with Mockito Annotations. @Mock, @InjectMocks, @RunWith(MockitoJUnitRunner.class), @Captor
* Step 11 : JUnit Rules. Using MockitoJUnit.rule() instead of @RunWith(MockitoJUnitRunner.class).
* Step 12 : Real world Example with Spring
* Step 13 : What is a spy? How to spy with Mockito?
* Step 14 : Some Theory : Why does Mockito not allow stubbing final and private methods?
* Step 15 : Using PowerMock and Mockito to mock a Static Method.
* Step 16 : Using PowerMock and Mockito to invoke a private Method.
* Step 17 : Using PowerMock and Mockito to mock a constructor.
* Step 18 : Good Unit Tests.

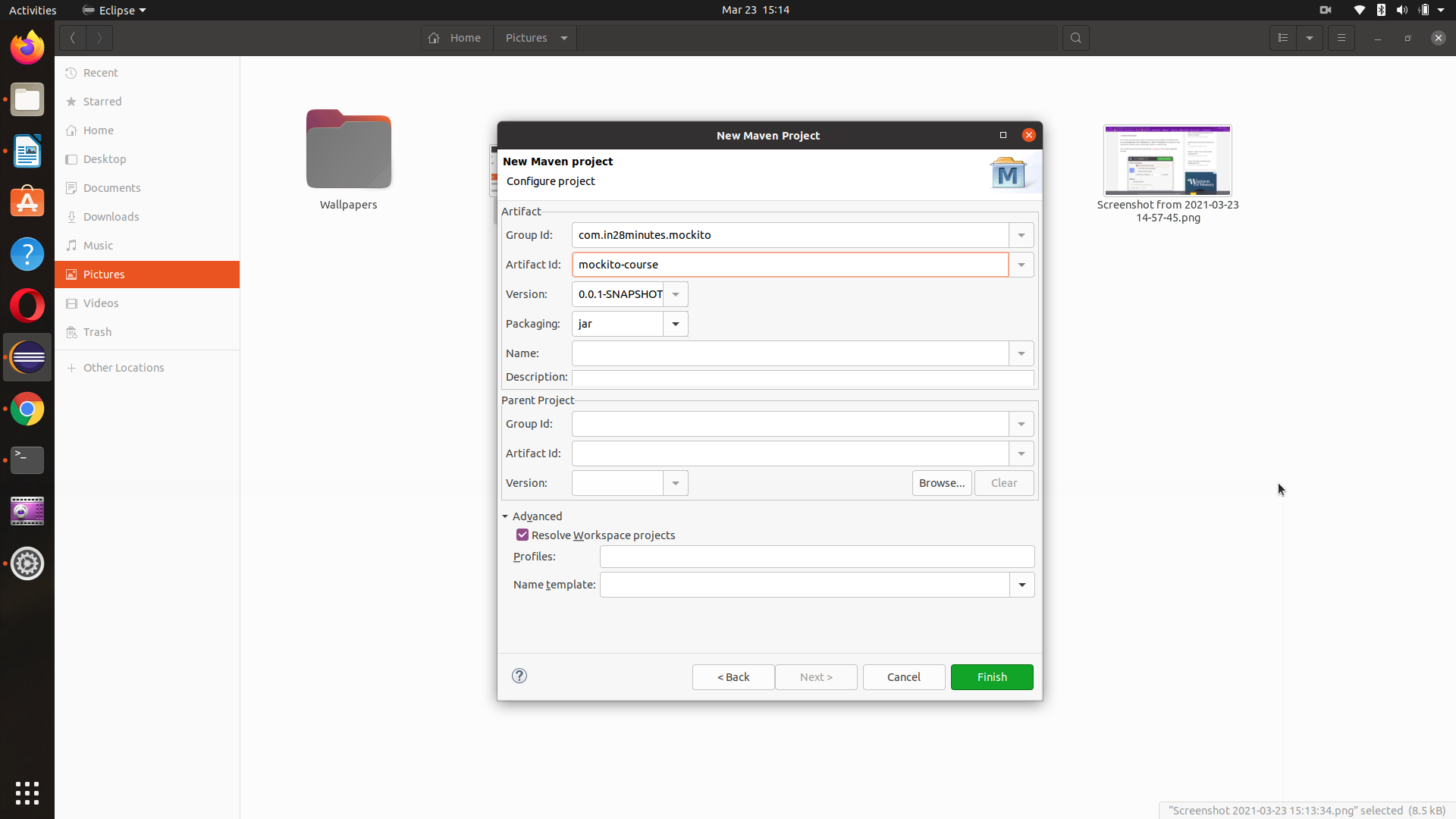
**JUnit Framework**

* Step 01 : Need for Unit Testing
* Step 02 : Setting up your First JUnit
* Step 03 : First Successful JUnit. Green Bar and assertEquals
* Step 04 : Refactoring Your First JUnit Test
* Step 05 : Second JUnit Example assertTrue and assertFalse
* Step 06 : @Before @After
* Step 07 : @BeforeClass @AfterClass
* Step 08 : Comparing Arrays in JUnit Tests
* Step 09 : Testing Exceptions in JUnit Tests
* Step 10 : Testing Performance in JUnit Tests
* Step 11 : Parameterized Tests
* Step 12 : Organize JUnits into Suites

https://github.com/in28minutes/MockitoTutorialForBeginners

Setting up the project

Create a simple Maven prokect (skip archtype)



<dependencies>

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.12</version>

<scope>test</scope>

</dependency>

<dependency>

<groupId>org.mockito</groupId>

<artifactId>mockito-all</artifactId>

<version>1.10.19</version>

<scope>test</scope>

</dependency>

</dependencies>

### test run

package com.in28minutes.mockito;

import static org.junit.Assert.assertTrue;

import org.junit.Test;

public class FirstMockitoTest {

@Test

public void test() {

assertTrue(true);

}

}

## Why we need Mockito ?

Stubs and mocks

Interface ( ToDOService.java )

// ToDOService will interact with wunderlist

//public List<String> reteriveToDo(String user)

We dont have implementation right now

//ToDoBuisnessimpl.java

//public List<String> reteriveToDo(String user)

**src/main/java**

**TodoService.java**

**package** com.in28minutes.data.api;

**import** java.util.List;

// External Service - Lets say this comes from WunderList

**public** **interface** TodoService {

**public** List<String> retrieveTodos(String user);

}

//for now we don’t have implementation for TodoService.java

**TodoBusinessImpl.java**

**package** com.in28minutes.business;

**import** java.util.ArrayList;

**import** java.util.List;

**import** com.in28minutes.data.api.TodoService;

// TodoBusinessImpl SUT (System under test)

**public** **class** TodoBusinessImpl {

// TodoService dependency

**private** TodoService todoService;

TodoBusinessImpl(TodoService todoService) {

// A kind of connection service

**this**.todoService = todoService;

}

**public** List<String> retrieveTodosRelatedToSpring(String user) {

List<String> filteredTodos = **new** ArrayList<String>();

//passing the user String to retrieve his ToDo List

List<String> allTodos = todoService.retrieveTodos(user);

**for** (String todo : allTodos) {

**if** (todo.contains("Spring")) {

filteredTodos.add(todo);

}

}

**return** filteredTodos;

}

}

**src/test/java**

**TodoServiceStub.java**

// a simple stub for the implementation of TodoService

**package** com.in28minutes.data.api;

**import** java.util.Arrays;

**import** java.util.List;

//STUB a sample implementation of the main service

**public** **class** TodoServiceStub **implements** TodoService{

**public** List<String> retrieveTodos(String user) {

// dummy to do list return

**return** Arrays.*asList*("Learn Spring MVC", "Learn Spring",

"Learn to Dance");

}

}

**TodoBusinessImplStubTest.java**

//StubTest

**package** com.in28minutes.business;

**import** **static** org.junit.Assert.*assertEquals*;

**import** java.util.List;

**import** org.junit.Test;

**import** com.in28minutes.data.api.TodoService;

**import** com.in28minutes.data.api.TodoServiceStub;

//import com.in28minutes.data.api.TodoServiceStub;

**public** **class** TodoBusinessImplStubTest {

@Test

**public** **void** testretrieveTodosRelatedToSpring\_usingAStub() {

TodoService todoService = **new** TodoServiceStub();

TodoBusinessImpl todoBusinessImpl = **new** TodoBusinessImpl(todoService);

List<String> filteredToDos = todoBusinessImpl

.retrieveTodosRelatedToSpring("Ranga");

*assertEquals*(2, filteredToDos.size());

// if we want to test several classes StubClass will start becoming more complex

}

}

We will create a stub (Sampl Implementation) of the Interface which ha no imp-le,mentation for now.

We will do in the test/java folder

Stubs are dummy implementation of iterface

**Problem with stubs**

1. They are hard to handle dynami scenarios, Like if we want to vary our test
2. and if there are more method in buisnesss interface then
3. for all teh above 2 problems
4. we have to update same in the STUB so it will start growing complex. it will become complex

So we use mock using Mockito

# Mocking **Getting Started**

### Your first Mockito code! Hurrah!!!

### Lets use Mockito to mock TodoService

## Useful Snippets and References

Easier Static Imports

helps in suggesting imports while coding

Window > Preferences > Java > Editor > Content Assist > Favorites

org.junit.Assert

org.mockito.BDDMockito

org.mockito.Mockito

org.hamcrest.Matchers **not found**  >>> org.mockito.Matchers >>> org.hamcrest.Matcher

org.hamcrest.CoreMatchers

Mocks offer a lot more functionality than stubbing

/\*\*mock()

\* mock()

\* Creates mock object of given class or interface.

\* <p>

\* See examples in javadoc for {@link Mockito} class

\*

\* **@param** classToMock class or interface to mock

\* **@return** mock object

\*/

Nice mocks- > would return the default value by own if not specified by the user.

And we dont need a Stub

**TodoBusinessImplMockTest.java**

**package** com.in28minutes.business;

**import** **static** org.junit.Assert.*assertEquals*;

**import** **static** org.mockito.Mockito.*mock*;

**import** **static** org.mockito.Mockito.*when*;

**import** java.util.Arrays;

**import** java.util.List;

**import** org.junit.Test;

**import** com.in28minutes.data.api.TodoService;

//import com.in28minutes.data.api.TodoServiceStub;

**public** **class** TodoBusinessImplMockTest {

@Test

**public** **void** testretrieveTodosRelatedToSpring\_usingAMock1() {

// Mocking an interface for its implementation class

TodoService todoServiceMock = *mock*(TodoService.**class**);

List<String> todos=Arrays.*asList*("Learn Spring MVC", "Learn Spring","Learn to Dance");

*when*(todoServiceMock.retrieveTodos("Dummy")).thenReturn(todos);

TodoBusinessImpl todoBusinessImpl = **new** TodoBusinessImpl(todoServiceMock);

List<String> filteredToDos = todoBusinessImpl

.retrieveTodosRelatedToSpring("Dummy");

*assertEquals*(2, filteredToDos.size());

}

@Test

**public** **void** testretrieveTodosRelatedToSpring\_usingAMock2() {

TodoService todoServiceMock = *mock*(TodoService.**class**);

List<String> todos=Arrays.*asList*();

*when*(todoServiceMock.retrieveTodos("Dummy")).thenReturn(todos);

TodoBusinessImpl todoBusinessImpl = **new** TodoBusinessImpl(todoServiceMock);

List<String> filteredToDos = todoBusinessImpl

.retrieveTodosRelatedToSpring("Dummy");

*assertEquals*(0, filteredToDos.size());

// if we want to test several classes StubClass will start becoming more complex

}

@Test

**public** **void** testretrieveTodosRelatedToSpring\_usingAMock3() {

TodoService todoServiceMock = *mock*(TodoService.**class**);

TodoBusinessImpl todoBusinessImpl = **new** TodoBusinessImpl(todoServiceMock);

/\*

\* An example of nice mock , because its returning a empty list of String ehrn

\* no instruction is specified

\*/

List<String> filteredToDos = todoBusinessImpl.retrieveTodosRelatedToSpring("Dummy");

*assertEquals*(0, filteredToDos.size());

}

}

# Mocking Basics

Easy Mock didnt supported NiceMock, So the test cases were little brittle

Mockito supports NiceMock

**ListTest.java**

**package** com.in28minutes.business;

**import** **static** org.junit.Assert.*assertEquals*;

**import** **static** org.mockito.Matchers.*anyInt*;

**import** **static** org.mockito.Mockito.*mock*;

**import** **static** org.mockito.Mockito.*when*;

**import** java.util.List;

**import** org.junit.Test;

**public** **class** ListTest {

@Test

**public** **void** letsMockListSize() {

List listMock = *mock*(List.**class**);

*when*(listMock.size()).thenReturn(2);

*assertEquals*(2, listMock.size());

*assertEquals*(2, listMock.size());

}

@Test

**public** **void** letsMockListSizeFor2Scenario() {

List listMock = *mock*(List.**class**);

*when*(listMock.size()).thenReturn(2).thenReturn(3);

*assertEquals*(2, listMock.size());

*assertEquals*(3, listMock.size());

}

@Test(expected=RuntimeException.**class**)

**public** **void** letsMockListGet() {

List listMock = *mock*(List.**class**);

*when*(listMock.get(0)).thenReturn("BRAVO");

*assertEquals*("BRAVO", listMock.get(0));

// ArgumentMatcher -> anyInt(), anyString() , anyObject()

//when(listMock.get(anyInt())).thenReturn("BRAVO");

// ArgumentMatcher will override NiceMock

// EXCEPTION THROW

// when(listMock.get(anyInt())).thenThrow(new

// RuntimeException("Runtime Exception success"));

// NiceMock

*assertEquals*(**null**, listMock.get(2));

}

}

// ArgumentMatcher -> anyInt(), anyString() , anyObject()

*when*(listMock.get(*anyInt*())).thenReturn(2).thenReturn("BRAVO");

// ArgumentMatcher will override NiceMock

// EXCEPTION THROW

*when*(listMock.get(*anyInt*())).thenThrow(**new** RuntimeException("Runtime Exception success"));

// NiceMock

*assertEquals*(**null**, listMock.get(2));

**ListTest.java**

**package** com.in28minutes.business;

**import** **static** org.hamcrest.CoreMatchers.~~is~~;

**import** **static** org.junit.Assert.*assertEquals*;

**import** **static** org.junit.Assert.*assertThat*;

**import** **static** org.mockito.BDDMockito.*given*;

**import** **static** org.mockito.Matchers.*anyInt*;

**import** **static** org.mockito.Mockito.*mock*;

**import** **static** org.mockito.Mockito.*when*;

**import** java.util.List;

**import** org.junit.Test;

**public** **class** ListTest {

@Test

**public** **void** letsMockListSize() {

List listMock = *mock*(List.**class**);

*when*(listMock.size()).thenReturn(2);

*assertEquals*(2, listMock.size());

*assertEquals*(2, listMock.size());

}

@Test

**public** **void** letsMockListSizeFor2Scenario() {

List listMock = *mock*(List.**class**);

*when*(listMock.size()).thenReturn(2).thenReturn(3);

*assertEquals*(2, listMock.size());

*assertEquals*(3, listMock.size());

}

@Test

**public** **void** letsMockListGet() {

List listMock = *mock*(List.**class**);

*when*(listMock.get(0)).thenReturn("BRAVO");

*assertEquals*("BRAVO", listMock.get(0));

}

@Test

**public** **void** letsMockListGetNiceMock() {

List listMock = *mock*(List.**class**);

*when*(listMock.get(0)).thenReturn("BRAVO");

*assertEquals*(**null**, listMock.get(2));// NiceMock

}

@Test

**public** **void** letsMockListGetArgumenMatcher() {

List listMock = *mock*(List.**class**);

*when*(listMock.get(0)).thenReturn("BRAV0");

// ArgumentMatcher -> anyInt(), anyString() , anyObject()

*when*(listMock.get(*anyInt*())).thenReturn("BRAVO"); // Overrides previous condition

*assertEquals*("BRAVO", listMock.get(0));// Nice Mock

// ArgumentMatcher will override NiceMocks

}

@Test(expected = RuntimeException.**class**)

**public** **void** letsMockListGetWithException() {

List listMock = *mock*(List.**class**);

*when*(listMock.get(*anyInt*())).thenThrow(**new** RuntimeException("Runtime Exception success"));

// when(listMock.get(anyInt(),9)).thenThrow(new RuntimeException("Runtime Exception success"));

// The above statement is invalid

}

// BDD style

@Test(expected = RuntimeException.**class**)

**public** **void** letsMockListGet\_usingBDD() {

// given

List<String> listMock = *mock*(List.**class**);

*given*(listMock.get(*anyInt*())).willReturn("BDD");

// when

String element = listMock.get(0);

// then

*assertThat*(element, *is*("BDD"));

}

}

1. Argument matcheer maps similar tuype of inputs
2. It will overpower NiceMock
3. We can also throw Exception from our mocks
4. We cannot include hard code value and Argumentmatchers simulatanously.
   * *when*(listMock.get(**anyInt(),5**)).thenThrow(**new** RuntimeException("Runtime Exception success"));

## What You Will Learn during this Step:

### Introduction to BDD

### Given When Then

### BDD ([Behavior Driven Development](https://medium.com/javascript-scene/behavior-driven-development-bdd-and-functional-testing-62084ad7f1f2" \l ":~:text=Unit testing is a methodology,object%2C class%2C or module.))Mockito Syntax

Given -> Initial Setup

When -> Specific Action

Then -> Validation

|  |  |
| --- | --- |
| **Mockito Normal** | **Mockito BDD** |
| ***when***(listMock.get(*anyInt*())).**thenReturn**("BRAVO");  *assertEquals*("BRAVO", listMock.get(0)); | **given**(listMock.get(anyInt())).**willReturn**("BDD");  *assertThat*(element, *is*("BDD")); |
| “is” is a part of HamcrestMatchers |
| EA | AE |

**TodoBusinessImplMockitoTestGivenThen.java**

**package** com.in28minutes.business;

**import** **static** org.hamcrest.CoreMatchers.~~is~~;

**import** **static** org.junit.Assert.*assertThat*;

**import** **static** org.mockito.BDDMockito.*given*;

**import** **static** org.mockito.Mockito.*mock*;

**import** java.util.Arrays;

**import** java.util.List;

**import** org.junit.Test;

**import** com.in28minutes.data.api.TodoService;

**public** **class** TodoBusinessImplMockitoTestGivenThen {

@Test

**public** **void** testReterieveTodosRelatedToSpring\_usingBDD() {

// Given

// When

// Then

// GIVEN initial setup

TodoService todoService = *mock*(TodoService.**class**);

List<String> allTodos = Arrays.*asList*("Learn Spring MVC","Learn Spring", "Learn to Dance");

*given*(todoService.retrieveTodos("Ranga")).willReturn(allTodos);

TodoBusinessImpl todoBusinessImpl = **new** TodoBusinessImpl(todoService);

// WHEN

List<String> filteredTodos = todoBusinessImpl

.retrieveTodosRelatedToSpring("Ranga");

// THEN

*assertThat*(filteredTodos.size(), *is*(2));//AE

// is() => HAMCREST MAtchers

}

}

**ListTest.java**

**package** com.in28minutes.business;

**import** **static** org.hamcrest.CoreMatchers.~~is~~;

**import** **static** org.junit.Assert.*assertEquals*;

**import** **static** org.junit.Assert.*assertThat*;

**import** **static** org.mockito.BDDMockito.*given*;

**import** **static** org.mockito.Matchers.*anyInt*;

**import** **static** org.mockito.Mockito.*mock*;

**import** **static** org.mockito.Mockito.*when*;

**import** java.util.List;

**import** org.junit.Test;

**public** **class** ListTest {

@Test(expected = RuntimeException.**class**)

**public** **void** letsMockListGet\_usingBDD() {

//given

List<String> listMock = *mock*(List.**class**);

*given*(listMock.get(*anyInt*())).willReturn("BDD");

//when

String element = listMock.get(0);

//then

*assertThat*(element, *is*("BDD"));

}

}

## What You Will Learn during this Step:

### How to verify calls on a mock?

### Verify how many times a method is called.

### We will add deleteTodo method to the TodoService.

**TodoService.java**

**package** com.in28minutes.data.api;

**import** java.util.List;

// External Service - Lets say this comes from WunderList

**public** **interface** TodoService {

**public** List<String> retrieveTodos(String user);

**public** **void** deleteToDo(String todD);

}

**TodoBusinessImpl.java**

**package** com.in28minutes.business;

**import** java.util.ArrayList;

**import** java.util.List;

**import** com.in28minutes.data.api.TodoService;

// TodoBusinessImpl SUT (System under test)

**public** **class** TodoBusinessImpl {

// TodoService dependency

**private** TodoService todoService;

TodoBusinessImpl(TodoService todoService) {

// A kind of connection service

**this**.todoService = todoService;

}

**public** List<String> retrieveTodosRelatedToSpring(String user) {

List<String> filteredTodos = **new** ArrayList<String>();

//passing the user String to retrieve his ToDo List

List<String> allTodos = todoService.retrieveTodos(user);

**for** (String todo : allTodos) {

**if** (todo.contains("Spring")) {

filteredTodos.add(todo);

}

}

**return** filteredTodos;

}

**public** **void** deleteTodosNotRelatedToSpring(String user) {

List<String> filteredTodos = **new** ArrayList<String>();

//passing the user String to retrieve his ToDo List

List<String> allTodos = todoService.retrieveTodos(user);

**for** (String todo : allTodos) {

**if** (!todo.contains("Spring")) {

**this**.todoService.deleteToDo(todo);

}

}

}

}

**TodoBusinessImplMockitoTestGivenThen.java**

**package** com.in28minutes.business;

**import** **static** org.hamcrest.CoreMatchers.~~is~~;

**import** **static** org.junit.Assert.*assertThat*;

**import** **static** org.mockito.BDDMockito.*given*;

**import** **static** org.mockito.Mockito.*atLeast*;

**import** **static** org.mockito.Mockito.*mock*;

**import** **static** org.mockito.Mockito.*never*;

**import** **static** org.mockito.Mockito.*times*;

**import** **static** org.mockito.Mockito.*verify*;

**import** java.util.Arrays;

**import** java.util.List;

**import** org.junit.Test;

**import** com.in28minutes.data.api.TodoService;

**public** **class** TodoBusinessImplMockitoTestGivenThen {

@Test

**public** **void** testReterieveTodosRelatedToSpring\_usingBDD() {

// Given

// When

// Then

// GIVEN initial setup

TodoService todoService = *mock*(TodoService.**class**);

List<String> allTodos = Arrays.*asList*("Learn Spring MVC",

"Learn Spring", "Learn to Dance");

*given*(todoService.retrieveTodos("Ranga")).willReturn(allTodos);

TodoBusinessImpl todoBusinessImpl = **new** TodoBusinessImpl(todoService);

// WHEN

List<String> filteredTodos = todoBusinessImpl

.retrieveTodosRelatedToSpring("Ranga");

// THEN

*assertThat*(filteredTodos.size(), *is*(2));//AE

// is() => HAMCREST MAtchers

}

@Test

**public** **void** testDeleteTodosNotRelatedToSpring\_usingBDD() {

// GIVEN initial setup

TodoService todoServiceMock = *mock*(TodoService.**class**);

List<String> allTodos = Arrays.*asList*("Learn Spring MVC",

"Learn Spring", "Learn to Dance");

*given*(todoServiceMock.retrieveTodos("Ranga")).willReturn(allTodos);

TodoBusinessImpl todoBusinessImpl = **new** TodoBusinessImpl(todoServiceMock);

// WHEN

todoBusinessImpl.deleteTodosNotRelatedToSpring("Ranga");

// THEN

*verify*(todoServiceMock).deleteToDo("Learn To Dance");// if called

*then*(todoServiceMock).should().deleteToDo("Learn To Dance");// similar like above

*verify*(todoServiceMock,*never*()).deleteToDo("Learn To Dance");// if not called

*then*(todoServiceMock).should(*never*()).deleteToDo("Learn To Dance");// similar like above

*verify*(todoServiceMock,*times*(1)).deleteToDo("Learn To Dance");// called 1 times

*verify*(todoServiceMock,*atLeast*(5)).deleteToDo("Learn To Dance");// called at least 5 times

}}

## What You Will Learn during this Step:

### How to capture an argument which is passed to a mock?

**TodoBusinessImplMockitoTestGivenThen.java**

**package** com.in28minutes.business;

**import** **static** org.hamcrest.CoreMatchers.~~is~~;

**import** **static** org.junit.Assert.*assertThat*;

**import** **static** org.mockito.BDDMockito.*given*;

**import** **static** org.mockito.BDDMockito.*then*;

**import** **static** org.mockito.Mockito.*atLeast*;

**import** **static** org.mockito.Mockito.*mock*;

**import** **static** org.mockito.Mockito.*never*;

**import** **static** org.mockito.Mockito.*times*;

**import** **static** org.mockito.Mockito.*verify*;

**import** java.util.Arrays;

**import** java.util.List;

**import** org.junit.Test;

**import** org.mockito.ArgumentCaptor;

**import** com.in28minutes.data.api.TodoService;

**public** **class** TodoBusinessImplMockitoTestGivenThen {

@Test

**public** **void** testReterieveTodosRelatedToSpring\_usingBDD() {

/\*

\* Given -> When -> Then

\*/

// GIVEN-> initial setup

TodoService todoService = *mock*(TodoService.**class**);

List<String> allTodos = Arrays.*asList*("Learn Spring MVC", "Learn Spring", "Learn to Dance");

*given*(todoService.retrieveTodos("Ranga")).willReturn(allTodos);

TodoBusinessImpl todoBusinessImpl = **new** TodoBusinessImpl(todoService);

// WHEN -> Specific Action

List<String> filteredTodos = todoBusinessImpl.retrieveTodosRelatedToSpring("Ranga");

// THEN -> validation

*assertThat*(filteredTodos.size(), *is*(2));// AE

// is() => HAMCREST MAtchers

}

@Test

**public** **void** testDeleteTodosNotRelatedToSpring\_usingBDD() {

// GIVEN initial setup

TodoService todoServiceMock = *mock*(TodoService.**class**);

List<String> allTodos = Arrays.*asList*("Learn Spring MVC", "Learn Spring", "Learn to Dance");

*given*(todoServiceMock.retrieveTodos("Ranga")).willReturn(allTodos);

TodoBusinessImpl todoBusinessImpl = **new** TodoBusinessImpl(todoServiceMock);

// WHEN

todoBusinessImpl.deleteTodosNotRelatedToSpring("Ranga");

// THEN

*verify*(todoServiceMock).deleteToDo("Learn to Dance");// if called with "Learn To Dance"

*then*(todoServiceMock).should().deleteToDo("Learn to Dance");// similar like above

*verify*(todoServiceMock, *never*()).deleteToDo("Learn Spring MVC");// if not called with "Learn To Dance"

*then*(todoServiceMock).should(*never*()).deleteToDo("Learn Spring Season");// similar like above

*verify*(todoServiceMock, *times*(1)).deleteToDo("Learn to Dance");// called 1 times

*verify*(todoServiceMock, *atLeast*(1)).deleteToDo("Learn to Dance");// called at least 1 times

}

@Test

**public** **void** testDeleteTodosNotRelatedToSpring\_usingBDD\_argumentCapture() {

// declare argument captor

ArgumentCaptor<String> argumentCaptor = ArgumentCaptor.*forClass*(String.**class**);

// GIVEN initial setup

TodoService todoServiceMock = *mock*(TodoService.**class**);

List<String> allTodos = Arrays.*asList*("Learn Spring MVC", "Learn Spring", "Learn to Dance");

*given*(todoServiceMock.retrieveTodos("Ranga")).willReturn(allTodos);

TodoBusinessImpl todoBusinessImpl = **new** TodoBusinessImpl(todoServiceMock);

// WHEN

todoBusinessImpl.deleteTodosNotRelatedToSpring("Ranga");

// THEN

// capturing it on the specific method call

*then*(todoServiceMock).should().deleteToDo(argumentCaptor.capture());// similar like above

// validation

*assertThat*(argumentCaptor.getValue(), *is*("Learn to Dance"));

}

@Test

**public** **void** testDeleteTodosNotRelatedToSpring\_usingBDD\_multipleArgumentCapture() {

// declare argument captor [Here I want to capture Argument of type String]

ArgumentCaptor<String> argumentCaptor = ArgumentCaptor.*forClass*(String.**class**);

// GIVEN initial setup

TodoService todoServiceMock = *mock*(TodoService.**class**);

List<String> allTodos = Arrays.*asList*("Learn to rock and roll", "Learn Spring", "Learn to Dance");

*given*(todoServiceMock.retrieveTodos("Ranga")).willReturn(allTodos);

TodoBusinessImpl todoBusinessImpl = **new** TodoBusinessImpl(todoServiceMock);

// WHEN

todoBusinessImpl.deleteTodosNotRelatedToSpring("Ranga");

// THEN

// capturing it on the specific method call

*then*(todoServiceMock).should(*times*(2)).deleteToDo(argumentCaptor.capture());// similar like above

// checking and capturing Multiple calls

*assertThat*(argumentCaptor.getAllValues().size(), *is*(2));

}

}

\*\*\*If using capture() say multipe method call happens ,and we are using argumentCaptor.getValue(), then the tests will start failing

So if there is a chance of multiple calls , use argumentCaptor.getAllValues()

**MOCKITO ADVANCE**

## What You Will Learn during this Step:

* Hamcrest Matchers
* [https://github.com/i2n8minutes/MockitoTutorialForBeginners](https://github.com/in28minutes/MockitoTutorialForBeginners)

## Useful Snippets and References

* Set up static import for

import static org.hamcrest.CoreMatchers.hasItems;

**Hamcrest Matchers improve readiblity**

*assertThat(scores, hasSize(4));*

*assertThat(scores, hasItems(100, 101));*

*assertThat(scores, everyItem(greaterThan(90)));*

*assertThat(scores, everyItem(lessThan(200)));*

*// for String*

*assertThat("", isEmptyString());*

*assertThat(null, isEmptyOrNullString());*

Adding the depenency

<dependency>

<groupId>org.hamcrest</groupId>

<artifactId>hamcrest-library</artifactId>

<version>1.3</version>

<scope>test</scope>

</dependency>

**HamcrestMatchersTest.java**

**package** com.in28minutes.mockito;

//Matchers Library is more extendible than CoreMatchers

**import** **static** org.hamcrest.Matchers.*arrayContaining*;

**import** **static** org.hamcrest.Matchers.*arrayContainingInAnyOrder*;

**import** **static** org.hamcrest.Matchers.*arrayWithSize*;

//specific import

**import** **static** org.hamcrest.Matchers.*everyItem*;

**import** **static** org.hamcrest.Matchers.*greaterThan*;

**import** **static** org.hamcrest.Matchers.*hasItems*;

**import** **static** org.hamcrest.Matchers.*hasSize*;

**import** **static** org.hamcrest.Matchers.*isEmptyOrNullString*;

**import** **static** org.hamcrest.Matchers.*isEmptyString*;

**import** **static** org.junit.Assert.*assertThat*;

**import** java.util.Arrays;

**import** java.util.List;

**import** org.junit.Test;

**public** **class** HamcrestMatchersTest {

@Test

**public** **void** testHamcrestMatcher() {

List<Integer> list = Arrays.*asList*(55, 22, 11, 59);

// List

*assertThat*(list, *hasSize*(4));

*assertThat*(list, *hasItems*(11, 59));

*assertThat*(list, *everyItem*(*greaterThan*(10)));

// String

*assertThat*("", *isEmptyString*());

*assertThat*("", *isEmptyOrNullString*());

// Array

Integer[] marks = { 21, 12, 32, 52, 59, 22 };

*assertThat*(marks, *arrayWithSize*(6));

*assertThat*(marks, *arrayContaining*(21, 12, 32, 52, 59, 22));

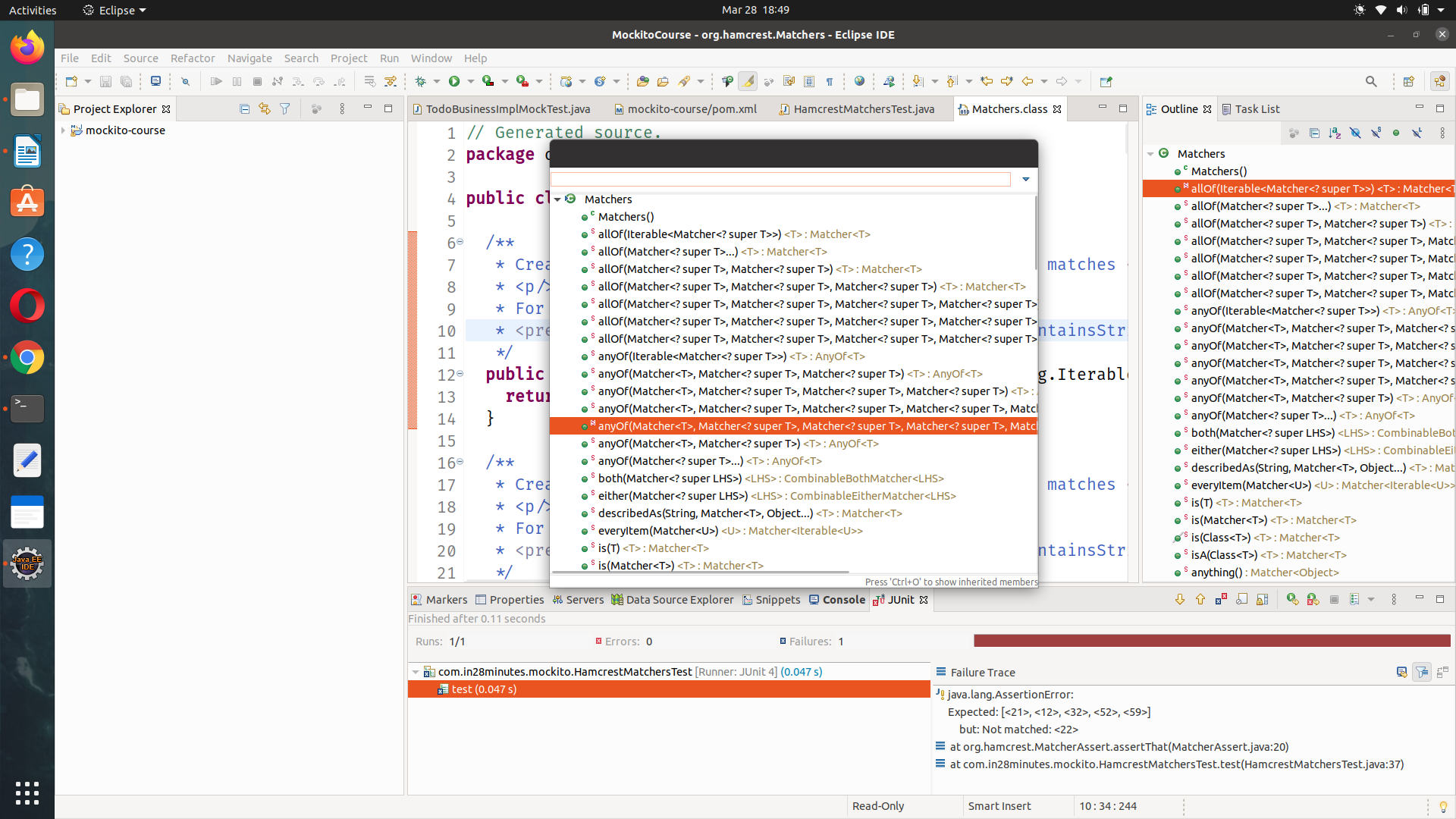
// order preservation is must

*assertThat*(marks, *arrayContainingInAnyOrder*(21, 32, 12, 52, 59, 22));

}

}

Open Matchers class and press **ctrl + o** for detailed info of the class

****

## What You Will Learn during this Step:

* Mockito Annotations
  + @Mock
  + @InjectMocks
  + @RunWith(MockitoJUnitRunner.class)
  + @Captor

**TodoBusinessImplMockitoInjectMockTestGivenThen.java**

**package** com.in28minutes.business;

**import** **static** org.hamcrest.CoreMatchers.~~is~~;

**import** **static** org.junit.Assert.*assertThat*;

**import** **static** org.mockito.BDDMockito.*given*;

**import** **static** org.mockito.BDDMockito.*then*;

**import** **static** org.mockito.Mockito.*atLeast*;

**import** **static** org.mockito.Mockito.*never*;

**import** **static** org.mockito.Mockito.*times*;

**import** **static** org.mockito.Mockito.*verify*;

**import** java.util.Arrays;

**import** java.util.List;

**import** org.junit.Rule;

**import** org.junit.Test;

**import** org.junit.runner.RunWith;

**import** org.mockito.ArgumentCaptor;

**import** org.mockito.Captor;

**import** org.mockito.InjectMocks;

**import** org.mockito.Mock;//important

**import** org.mockito.junit.MockitoJUnit;

**import** org.mockito.junit.MockitoRule;

**import** org.mockito.runners.MockitoJUnitRunner;

**import** com.in28minutes.data.api.TodoService;

//@RunWith(MockitoJUnitRunner.class)

**public** **class** TodoBusinessImplMockitoInjectMockTestGivenThen {

@Rule

**public** MockitoRule mockitoRule = MockitoJUnit.*rule*();

// mocking with annotation

@Mock

TodoService todoService;

/\*

\* OMIT: TodoService todoService = mock(TodoService.class);

\*/

// Automatically inject its dependencies if declared with @Mocks via ConstructorInjection

@InjectMocks

TodoBusinessImpl todoBusinessImpl;

/\*

\* OMIT: TodoBusinessImpl todoBusinessImpl = new TodoBusinessImpl(todoService);

\*/

// Creating Argument Captor

@Captor

ArgumentCaptor<String> argumentCaptor = ArgumentCaptor.*forClass*(String.**class**);

/\*

\* OMIT: ArgumentCaptor<String> argumentCaptor =

\* ArgumentCaptor.forClass(String.class);

\*/

@Test

**public** **void** testReterieveTodosRelatedToSpring\_usingBDD() {

/\*

\* Given -> When -> Then

\*

\*/

// TodoService todoServiceMock = mock(TodoService.class);

// GIVEN initial setup-> No need, because we are using @Mock & JUnitRunner

List<String> allTodos = Arrays.*asList*("Learn Spring MVC", "Learn Spring", "Learn to Dance");

*given*(todoService.retrieveTodos("Ranga")).willReturn(allTodos);

// TodoBusinessImpl todoBusinessImpl = new TodoBusinessImpl(todoService);

// WHEN

List<String> filteredTodos = todoBusinessImpl.retrieveTodosRelatedToSpring("Ranga");

// THEN

*assertThat*(filteredTodos.size(), *is*(2));// AE

}

@Test

**public** **void** testDeleteTodosNotRelatedToSpring\_usingBDD() {

List<String> allTodos = Arrays.*asList*("Learn Spring MVC", "Learn Spring", "Learn to Dance");

*given*(todoService.retrieveTodos("Ranga")).willReturn(allTodos);

// WHEN

todoBusinessImpl.deleteTodosNotRelatedToSpring("Ranga");

// THEN

*verify*(todoService).deleteToDo("Learn to Dance");// if called

*then*(todoService).should().deleteToDo("Learn to Dance");// similar like above

*verify*(todoService, *never*()).deleteToDo("Learn to Drink");// if not called

*then*(todoService).should(*never*()).deleteToDo("Learn to Drink");// similar like above

*verify*(todoService, *times*(1)).deleteToDo("Learn to Dance");// called 1 times

*verify*(todoService, *atLeast*(1)).deleteToDo("Learn to Dance");// called at least 5 times

}

@Test

**public** **void** testDeleteTodosNotRelatedToSpring\_usingBDD\_argumentCapture() {

List<String> allTodos = Arrays.*asList*("Learn Spring MVC", "Learn Spring", "Learn to Dance");

*given*(todoService.retrieveTodos("Ranga")).willReturn(allTodos);

// WHEN

todoBusinessImpl.deleteTodosNotRelatedToSpring("Ranga");

// THEN

*then*(todoService).should().deleteToDo("Learn to Dance");// similar like above

// capturing it on the specific method call

*then*(todoService).should().deleteToDo(argumentCaptor.capture());// similar like above

// checking

*assertThat*(argumentCaptor.getValue(), *is*("Learn to Dance"));

}

@Test

**public** **void** testDeleteTodosNotRelatedToSpring\_usingBDD\_MultipleArgumentCapture() {

List<String> allTodos = Arrays.*asList*("Learn Django MVC", "Learn Spring", "Learn to Dance");

*given*(todoService.retrieveTodos("Ranga")).willReturn(allTodos);

// WHEN

todoBusinessImpl.deleteTodosNotRelatedToSpring("Ranga");

// THEN

*then*(todoService).should().deleteToDo("Learn to Dance");// similar like above

// capturing it on the specific method call

*then*(todoService).should(*times*(2)).deleteToDo(argumentCaptor.capture());// similar like above

// checking and capturing Multiple calls

*assertThat*(argumentCaptor.getAllValues().size(), *is*(2));

}

}

\*\*\* While using @InjectMocks on a ref variable, it will check for the dependences of that class and verify if it matches with any of the @Mock references. If yes, it will initiate a constructor injection.

What You Will Learn during this Step:

* JUnit Rules. Using MockitoJUnit.rule() instead of @RunWith(MockitoJUnitRunner.class).

@RunWith(MockitoJUnitRunner.class) enable the creation of mocks with appropriate injections. But if we want to run two runner, like *Spring Test* runner its not possible . To fix this issue they came up with

**@Rule ,**

**its since Junit 4.7, So we will start using this @Rule instead of** ~~@RunWith(MockitoJUnitRunner.class)~~

**@Rule runs before and after the test**

**and unlike Runner .class (one runner possiblke) we can have as many as @Rule we want.**

**But @Rule should be public.**

**So @Rule provide a good flex.**

@Rule

public MockitoRule mockitoRule = MockitoJUnit.rule();

**TodoBusinessImplMockitoInjectMockTestGivenThen.java**

package com.in28minutes.business;

~~//@RunWith(MockitoJUnitRunner.class)~~

public class TodoBusinessImplMockitoInjectMockTestGivenThen {

@Rule

public MockitoRule mockitoRule = MockitoJUnit.rule();

/\* mocking with annotation \*/

@Mock

TodoService todoService;

## What You Will Learn during this Step:

* Real world Example with Spring.
* Not very different from previous example.

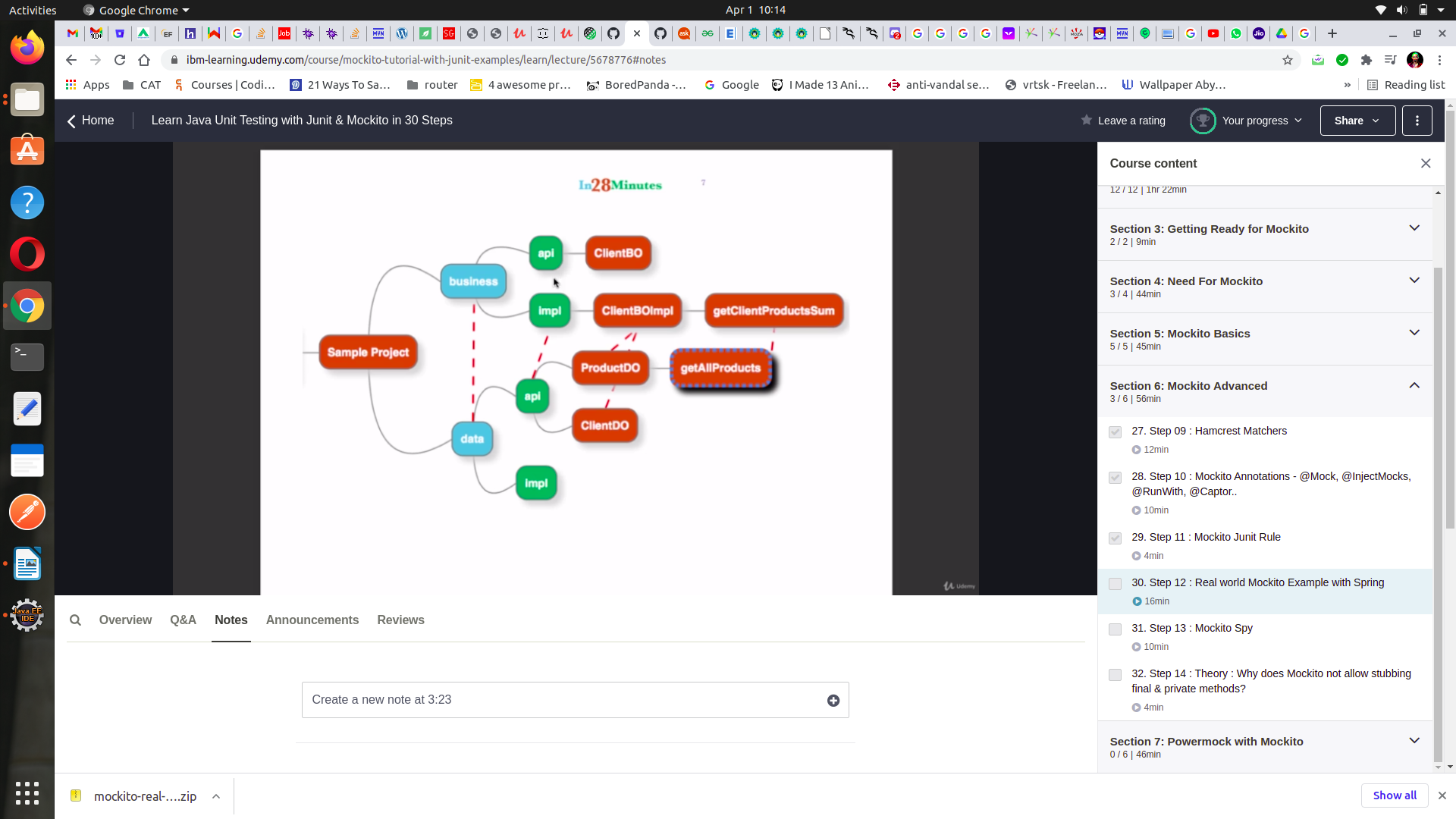
## Useful Snippets and References

* Download the zip mockito-real-world-example-with-spring.zip and setup this maven project

https://github.com/in28minutes/MockitoTutorialForBeginners/blob/master/mockito-real-world-example-with-spring.zip

api-> has all the interfaces

Impl -> contains all the implementation



1. So here the **buisness** layar depends on the **data** layar
2. data layar gets all the **products** from the database and buisness layar calculate the total **sum** out of it

So here we will kmock product dao and the client the dao, which are dependencies of **Buisness IMPL**

***https://github.com/in28minutes/MockitoTutorialForBeginners/blob/master/mockito-real-world-example-with-spring.zip***

## What You Will Learn during this Step:

* Understand what a Spy does?
* Creating a spy with Mockito?
* Overriding specific methods in a spy?

Adding items to a dummy mocked list will not mean reality , because its mocked, But it want to do so we can do with **Spy**

**Spy** also allows override certain methods, when stubbed it will overrid the specific functionality,

**Spy** allows you to keep track of certain object as well as override a specific functionality according to one needs. So a certain part of real logic is retained and certain part of logic can be overrided

**Spy** is not really used in real life, Its preferred for legacy system.

**MockitoSpySample.java**

**package** com.in28minutes.mockito;

**import** **static** org.junit.Assert.*assertEquals*;

**import** **static** org.mockito.Mockito.*mock*;

**import** **static** org.mockito.Mockito.*never*;

**import** **static** org.mockito.Mockito.*spy*;

**import** **static** org.mockito.Mockito.*stub*;

**import** **static** org.mockito.Mockito.*verify*;

**import** java.util.ArrayList;

**import** org.junit.Test;

**public** **class** MockitoSpySample {

@Test

**public** **void** testDummy() {

ArrayList arrayListMock = *mock*(ArrayList.**class**);// mock return default value ie 0

*assertEquals*(0, arrayListMock.size());

*stub*(arrayListMock.size()).toReturn(5);// stubbing will return 5 value

arrayListMock.add("Dummy");// this will not change from 5|>6

*assertEquals*(5, arrayListMock.size());

}

@Test

**public** **void** testSpy() {

ArrayList arrayListMock = *spy*(ArrayList.**class**);// mock return default value ie 0

*assertEquals*(0, arrayListMock.size());

arrayListMock.add("Dummy");// this will change from 0|>1

*assertEquals*(1, arrayListMock.size());

arrayListMock.remove("Dummy");// this will change from 1|>0

*assertEquals*(0, arrayListMock.size());

// Stubbing will override the normal functionality of size()

*stub*(arrayListMock.size()).toReturn(5);// stubbing will return 5 value

arrayListMock.add("Tummy");// this will not change from 5|>6 because

*assertEquals*(5, arrayListMock.size());

System.***out***.println(arrayListMock);// Tummy

// Verification also works fine in Spy

*verify*(arrayListMock).add("Tummy");

*verify*(arrayListMock, *never*()).clear();

}

}

## What You Will Learn during this Step:

* Some Theory : Why does Mockito not allow stubbing final and private methods?

## Useful Snippets and References

<https://github.com/mockito/mockito/wiki/Mockito-And-Private-Methods>

<https://github.com/mockito/mockito/wiki/FAQ>

# Mockito And Private Methods

*Mockito does not allow mockng/stubbing private/final methods. The reason goes that they want to promote good design principle and high testing standards.*

**Why Mockito doesn't mock private methods?**

Firstly, we are not dogmatic about mocking private methods. We just don't care about private methods because from the standpoint of testing, private methods don't exist. Here are a couple of reasons Mockito doesn't mock private methods:

1. It requires hacking of classloaders that is never bullet proof and it changes the API (you must use custom test runner, annotate the class, etc.).
2. It is very easy to work around - just change the visibility of method from private to package-protected (or protected).
3. It requires the team to spend time implementing & maintaining it. And it does not make sense given point (2) and a fact that it is already implemented in different tool (powermock).
4. Finally... Mocking private methods is a hint that there is something wrong with Object Oriented understanding. In OO you want objects (or roles) to collaborate, not methods. Forget about pascal & procedural code. Think in objects.

**FAQ**

**What is Mockito?**

Mockito is a mocking framework for Java. Mockito allows convenient creation of substitutes of real objects for testing purposes. Enjoy clean tests with mock objects, improved TDD experience and beautiful mocking API. See the main page or examples for more.

**Is it really a mocking framework?**

There is a bit of confusion around the vocabulary. Technically speaking, Mockito is a Test Spy framework. Usually developers use Mockito instead of a mocking framework. Test Spy framework allows to verify behaviour (like mocks) and stub methods (like good old hand-crafted stubs).

**Why is Mockito so simple?**

To promote simple test code that hopefully pushes the developer to write simple and clean application code. I wrote this paragraph long before version 1.5. Mockito is still quite lean but the number of features increased because many users found valid cases for them. It's OSS after all, isn't it?

**What are the limitations of Mockito**

Mockito 3.x specific limitations

• Requires Java 8+

• Cannot mock constructors

Mockito 2.x specific limitations

• Requires Java 6+

• Cannot mock constructors

• Cannot mock static methods

• Cannot mock equals(), hashCode(). Firstly, you should not mock those methods. Secondly, Mockito defines and depends upon a specific implementation of these methods. Redefining them might break Mockito.

• Mocking is only possible on VMs that are supported by Objenesis. Don't worry, most VMs should work just fine.

• Spying on real methods where real implementation references outer Class via OuterClass.this is impossible. Don't worry, this is extremely rare case.

​ Mockito 1.x Specific limitations

• Needs Java 5+

• Cannot mock final classes

• Cannot mock final methods - their real behavior is executed without any exception. Mockito cannot warn you about mocking final methods so be vigilant.

• Cannot mock static methods

• Cannot mock constructors

• Cannot mock equals(), hashCode(). Firstly, you should not mock those methods. Secondly, Mockito defines and depends upon a specific implementation of these methods. Redefining them might break Mockito.

• Mocking is only possible on VMs that are supported by Objenesis (Note Objenesis is in version 2.1). Don't worry, most VMs should work just fine.

• Spying on real methods where real implementation references outer Class via OuterClass.this is impossible. Don't worry, this is extremely rare case.

Do you mock classes & interfaces?

Yes, the api is the same for mocking classes or interfaces.

​ What values do mocks return by default?

In order to be transparent and unobtrusive all Mockito mocks by default return 'nice' values. For example: zeros, falseys, empty collections or nulls. Refer to javadocs about stubbing to see exactly what values are returned by default.

​ Can I have xxxx feature?

Please send us or a pull request an example where you need something more than Mockito can offer.

​ Can I mock private methods?

No. From the standpoint of testing... private methods don't exist. More about private methods here.

​ Is Mockito thread-safe?

For healthy scenarios Mockito plays nicely with threads. For instance, you can run tests in parallel to speed up the build. Also, you can let multiple threads call methods on a shared mock to test in concurrent conditions. Check out a timeout() feature for testing concurrency.

However Mockito is only thread-safe in healthy tests, that is tests without multiple threads stubbing/verifying a shared mock. Stubbing or verification of a shared mock from different threads is NOT the proper way of testing because it will always lead to intermittent behavior. In general, mutable state + assertions in multi-threaded environment lead to random results. If you do stub/verify a shared mock across threads you will face occasional exceptions like: WrongTypeOfReturnValue, etc.

​ Can I verify toString()?

No. You can stub it, though. Verification of toString() is not implemented mainly because:

• When debugging, IDE calls toString() on objects to print local variables and their content, etc. After debugging, the verification of toString() will most likely fail.

• toString() is used for logging or during string concatenation. Those invocations are usually irrelevant but they will change the outcome of verification.

Can I "reset" a mock?

Recently we decided to go on with this feature for tricky scenarios where mocks are created by the container (see issue 55). Before that, the lack of a reset method was deliberate to promote good testing habits and to make the API simpler.

Instead of reset() please consider writing simple, small and focused test methods over lengthy, over-specified tests. The discussion about this feature was here.

​ What are unfinished verification/stubbing errors?

Mockito validates if you use it correctly all the time. Examples of incorrect use:

//Oups, someone forgot thenReturn() part:

when(mock.get());

//Oups, someone put the verified method call inside verify() where it should be outside:

verify(mock.execute());

//Oups, someone has used EasyMock for too long and forgot to specify the method to verify:

verify(mock);

Mockito throws exceptions if you misuse it so that you will know if your tests are written correctly. The only problem is that Mockito does the validation next time you use the framework. Therefore sometimes the exception is thrown in the next test and you have to manually find the previous test that was not written correctly.

​ Why does Mockito keep ThreadLocal state?

Mockito uses ThreadLocal state to implement a gorgeous mocking syntax in a language full of constraints (yes, it's java). Fortunately, every time you interact with Mockito framework it validates the ThreadLocal state in case you misused the api.

​ Can I thenReturn() an inlined mock() ?

Unfortunately you cannot do this:

when(m.foo()).thenReturn(mock(Foo.class));

// ^

The reason is that detecting unfinished stubbing wouldn't work if we allow above construct. We consider this as a 'trade off' of framework validation (see also previous FAQ entry). However you can slightly change the code to make it working:

//extract local variable and start smiling:

Foo foo = mock(Foo.class);

when(m.foo()).thenReturn(foo);

​ Can I stub chained getters?

when(mock.getA().getB()).thenReturn(...);

This sort of stubbing, e.g. mock to return mock, to return mock, etc. should be used very sporadically, ideally never. It clearly points out violation of the Law of Demeter. You don't want to mess with Demeter. Since you have been warned check out Mockito deep stubs.

## **Powermock**

**Benefetial when using test with static methods**

## What You Will Learn during this Step:

* Add dependency on PowerMock.
* Using PowerMock and Mockito to mock a Static Method.
* PowerMockitoMockingStaticMethodTest

PowerMockitoMockingStaticMethodTest

## Useful Snippets and References

\*Maven can call up its depedencies as well as dependencies (transitive dependecies)

pom.xml

<dependency>

<groupId>org.powermock</groupId>

<artifactId>powermock-api-mockito</artifactId>

<version>1.6.4</version>

<scope>test</scope>

</dependency>

<dependency>

<groupId>org.powermock</groupId>

<artifactId>powermock-module-junit4</artifactId>

<version>1.6.4</version>

<scope>test</scope>

</dependency>

*Starter Code*

**SystemUnderTest.java**

**package** com.in28minutes.powermock;

**import** java.util.ArrayList;

**import** java.util.List;

**interface** Dependency {

// Not implemented -> So we will mock

List<Integer> retrieveAllStats();

}

**public** **class** SystemUnderTest {

**private** Dependency dependency;

**public** **int** methodUsingAnArrayListConstructor() {

ArrayList list = **new** ArrayList();

**return** list.size();

}

**public** **int** methodCallingAStaticMethod() {

// privateMethodUnderTest calls static method SomeClass.staticMethod

List<Integer> stats = dependency.retrieveAllStats();

**long** sum = 0;

**for** (**int** stat : stats)

sum += stat;

**return** UtilityClass.*staticMethod*(sum);

}

**private** **long** privateMethodUnderTest() {

List<Integer> stats = dependency.retrieveAllStats();

**long** sum = 0;

**for** (**int** stat : stats)

sum += stat;

**return** sum;

}

}

**UtilityClass.java**

**package** com.in28minutes.powermock;

**public** **class** UtilityClass {

**static** **int** staticMethod(**long** value) {

// Some complex logic is done here...

**throw** **new** RuntimeException(

"I dont want to be executed. I will anyway be mocked out.");

}

}

**package** com.in28minutes.powermock;

**import** **static** org.junit.Assert.*assertEquals*;

**import** **static** org.mockito.BDDMockito.*given*;

**import** **static** org.mockito.Mockito.*when*;

**import** java.util.Arrays;

**import** java.util.List;

**import** org.junit.Rule;

**import** org.junit.Test;

**import** org.junit.runner.RunWith;

**import** org.mockito.InjectMocks;

**import** org.mockito.Mock;//important

**import** org.mockito.junit.MockitoJUnit;

**import** org.mockito.junit.MockitoRule;

**import** org.powermock.api.mockito.PowerMockito;

**import** org.powermock.core.classloader.annotations.PrepareForTest;

**import** org.powermock.modules.junit4.PowerMockRunner;

//@RunWith(MockitoJUnitRunner.class)

@RunWith(PowerMockRunner.**class**)// Specific Runner for static methods

@PrepareForTest(UtilityClass.**class**)

**public** **class** MockingStaticMethodTest {

/\*

\* STEPS to mock a static method

\* => Specific Runner

\* => Initialize which class is having the static method

\*/

// mocking with annotation

@Mock

Dependency dependency;

// Automatically inject its dependencies if declared with @Mocks via ConstructorInjection

@InjectMocks

SystemUnderTest systemUnderTest;

@Test

**public** **void** testStaticMethod() {

/\*

\* Given -> When -> Then

\*

\*/

List<Integer> stats = Arrays.*asList*(1, 2, 3, 4);

*given*(dependency.retrieveAllStats()).willReturn(stats);

// WHEN

PowerMockito.*mockStatic*(UtilityClass.**class**);

*when*(UtilityClass.*staticMethod*(10)).thenReturn(100); //Static methods cannot be mocked normally

**int** result = systemUnderTest.methodCallingAStaticMethod();

// THEN

*assertEquals*(100,result);

PowerMockito.*verifyStatic*();

UtilityClass.*staticMethod*(10);}}

package com.in28minutes.powermock;

import static org.junit.Assert.assertEquals;

import static org.mockito.Matchers.anyLong;

import static org.mockito.Mockito.when;

import java.util.Arrays;

import org.junit.Test;

import org.junit.runner.RunWith;

import org.mockito.InjectMocks;

import org.mockito.Mock;

import org.powermock.api.mockito.PowerMockito;

import org.powermock.core.classloader.annotations.PrepareForTest;

import org.powermock.modules.junit4.PowerMockRunner;

@RunWith(PowerMockRunner.class)

@PrepareForTest({ UtilityClass.class})

public class PowerMockitoMockingStaticMethodTest {

@Mock

Dependency dependencyMock;

@InjectMocks

SystemUnderTest systemUnderTest;

@Test

public void powerMockito\_MockingAStaticMethodCall() {

when(dependencyMock.retrieveAllStats()).thenReturn(

Arrays.asList(1, 2, 3));

PowerMockito.mockStatic(UtilityClass.class);

when(UtilityClass.staticMethod(anyLong())).thenReturn(150);

assertEquals(150, systemUnderTest.methodCallingAStaticMethod());

//To verify a specific method call

//First : Call PowerMockito.verifyStatic()

//Second : Call the method to be verified

PowerMockito.verifyStatic();

UtilityClass.staticMethod(1 + 2 + 3);

// verify exact number of calls

//PowerMockito.verifyStatic(Mockito.times(1));

}

}

## What You Will Learn during this Step:

* Using PowerMock and Mockito to invoke a private Method.
* PowerMockitoTestingPrivateMethodTest

## Snippets

long value = (Long) Whitebox.invokeMethod(systemUnderTest,

"privateMethodUnderTest");

*invokeMethod() can be used to test pricvate/ innerclass methods*

**package** com.in28minutes.powermock;

**import** **static** org.junit.Assert.*assertEquals*;

**import** **static** org.mockito.BDDMockito.*given*;

**import** java.util.Arrays;

**import** java.util.List;

**import** org.junit.Rule;

**import** org.junit.Test;

**import** org.junit.runner.RunWith;

**import** org.mockito.InjectMocks;

**import** org.mockito.Mock;//important

**import** org.mockito.junit.MockitoJUnit;

**import** org.mockito.junit.MockitoRule;

**import** org.powermock.modules.junit4.PowerMockRunner;

**import** org.powermock.reflect.Whitebox;

//@RunWith(MockitoJUnitRunner.class)

@RunWith(PowerMockRunner.**class**)// Specific Runner for static methods

**public** **class** InvokingPrivateMethodTest {

// mocking with annotation

@Mock

Dependency dependency;

// Automatically inject its dependencies if declared with @Mocks via ConstructorInjection

@InjectMocks

SystemUnderTest systemUnderTest;

@Test

**public** **void** testStaticMethod() **throws** Exception {

/\*

\* Given -> When -> Then

\*

\*/

List<Integer> stats = Arrays.*asList*(1, 2, 3, 4);

*given*(dependency.retrieveAllStats()).willReturn(stats);

// WHEN

**long** result = Whitebox.*invokeMethod*(systemUnderTest, "privateMethodUnderTest");

// invokeMethod() can be used to test pricvate/ innerclass methods

// THEN

*assertEquals*(10,result);

}

}

package com.in28minutes.powermock;

import static org.junit.Assert.assertEquals;

import static org.mockito.Mockito.when;

import java.util.Arrays;

import org.junit.Test;

import org.junit.runner.RunWith;

import org.mockito.InjectMocks;

import org.mockito.Mock;

import org.powermock.modules.junit4.PowerMockRunner;

import org.powermock.reflect.Whitebox;

@RunWith(PowerMockRunner.class)

public class PowerMockitoTestingPrivateMethodTest {

@Mock

Dependency dependencyMock;

@InjectMocks

SystemUnderTest systemUnderTest;

@Test

public void powerMockito\_CallingAPrivateMethod() throws Exception {

when(dependencyMock.retrieveAllStats()).thenReturn(

Arrays.asList(1, 2, 3));

long value = (Long) Whitebox.invokeMethod(systemUnderTest,

"privateMethodUnderTest");

assertEquals(6, value);

}

}

# What You Will Learn during this Step:

* Using PowerMock and Mockito to mock a constructor
* PowerMockitoMockingConstructorTest.java -- PowerMockito.whenNew(ArrayList.class).withAnyArguments().thenReturn( mockList);

ArrayList list = **new** ArrayList();

**return** list.size();

We will mock this constructor, so it will npot give the value of zero.

### **/src/test/java/com/in28minutes/powermock/PowerMockitoMockingConstructorTest.java**

package com.in28minutes.powermock;

import static org.junit.Assert.assertEquals;

import static org.mockito.Mockito.mock;

import static org.mockito.Mockito.stub;

import java.util.ArrayList;

import org.junit.Test;

import org.junit.runner.RunWith;

import org.mockito.InjectMocks;

import org.mockito.Mock;

import org.powermock.api.mockito.PowerMockito;

import org.powermock.core.classloader.annotations.PrepareForTest;

import org.powermock.modules.junit4.PowerMockRunner;

@RunWith(PowerMockRunner.class)

@PrepareForTest({ SystemUnderTest.class /\*To be able to mock the Constructor, we need to add in the Class that creates the new object\*/})

public class PowerMockitoMockingConstructorTest {

private static final int SOME\_DUMMY\_SIZE = 100;

@Mock

Dependency dependencyMock;

@InjectMocks

SystemUnderTest systemUnderTest;

@Test

public void powerMockito\_MockingAConstructor() throws Exception {

ArrayList<String> mockList = mock(ArrayList.class);

stub(mockList.size()).toReturn(SOME\_DUMMY\_SIZE);

PowerMockito.whenNew(ArrayList.class).withAnyArguments().thenReturn(

mockList);

int size = systemUnderTest.methodUsingAnArrayListConstructor();

assertEquals(SOME\_DUMMY\_SIZE, size);

}

}

**package** com.in28minutes.powermock;

**import** **static** org.junit.Assert.*assertEquals*;

**import** **static** org.mockito.Mockito.*when*;

**import** java.util.ArrayList;

**import** java.util.Arrays;

**import** java.util.List;

**import** org.junit.Test;

**import** org.junit.runner.RunWith;

**import** org.mockito.InjectMocks;

**import** org.mockito.Mock;//important

**import** org.powermock.api.mockito.PowerMockito;

**import** org.powermock.core.classloader.annotations.PrepareForTest;

**import** org.powermock.modules.junit4.PowerMockRunner;

@RunWith(PowerMockRunner.**class**) // Specific Runner for static methods

@PrepareForTest(SystemUnderTest.**class**)

**public** **class** MockingConstructorTest {

/\*

\* Prepare the class for test => SystemUnderTest (class making use of the

\* constructor) not the class with the constructor like -> ArrayList Override

\* the constructor

\*/

@Mock

ArrayList mockList;

// Automatically inject its dependencies if declared with @Mocks via ConstructorInjection

@InjectMocks

SystemUnderTest systemUnderTest;

@Test

**public** **void** testStaticMethod() **throws** Exception {

/\*

\* Given -> When -> Then

\*

\*/

List<Integer> stats = Arrays.*asList*(1, 2, 3, 4);

// WHEN

*when*(mockList.size()).thenReturn(stats.size()); // Static methods cannot be mocked normally

PowerMockito.*whenNew*(ArrayList.**class**).withAnyArguments().thenReturn(mockList);

**int** size = systemUnderTest.methodUsingAnArrayListConstructor();

*assertEquals*(4, size);

}

}

## What You Will Learn during this Step:

* Good Unit Tests -- Basics --- Readable --- Fails only when there are real logic failures

\*\*\* testing/mocking private methods, static methods, constructors are considered a bad practice so one should avoid doing it unit its needed in legacy project.

\*\*\*

@Test

public void methodName\_scenario\_result() {

}

\*\*\* readable ~~>~~ BDD -> Give-When-Then

Fails only when there are real logic failures

should not depends on other inconsistency

ie -> database inconsistencey

## Exercises and References

* FIRST.

https://pragprog.com/magazines/2012-01/unit-tests-are-first

* Patterns -
* http://xunitpatterns.com
* <https://github.com/mockito/mockito/wiki/How-to-write-good-tests>