Junit examples

<http://examples.javacodegeeks.com/category/core-java/junit/page/2/>

# JUnit Tutorial for Beginners

Posted by: [Alvin Reyes](http://examples.javacodegeeks.com/author/alvin-reyes/) in [junit](http://examples.javacodegeeks.com/category/core-java/junit/) October 14th, 2015

## 1. Introduction

In this post, we will discus the basics of setting up your JUnit Test cases. We’ll go step by step in creating test cases as we go along with creating our application. Before we dive into it though, why do we even need to create test cases? Isn’t it enough to just create the implementation since it’s what we are delivering anyway?

Although the actual implementation is part of the package, the JUnit Test case is a **bullet proof evidence** that what we wrote is what the actual requirements or functions will do. It is the concrete basis of a specific unit/function which does what it needs to do.

**Knowing the impact in the stability of the application.** The JUnit Test cases defines the stability of an application even after several extensions to it. If done correctly, it guarantees that the extension made to the system will not break the entire system as a whole. How does it prevent it? If the developers write clean unit and integration tests, it will report any side effects via the reporting plugins that the application uses.

**Regression and Integration Testing.** The effort of testing is relative to the applications size and changes done. By creating JUnit Test cases, regression and integration tests can be automated and can definitely save time and effort.

Overall, creating JUnit Test cases are definitely a must do by all developers, sadly there are still who don’t uses it’s power to it’s full extent and some just doesn’t do it. It’s sometimes a shame to think that one of the purest way of developing bullet proof code is not done by the developers. It can be because of the lack of training, experience or just pressure of not delivering the actual value (which is a problem within itself since although not part of the implementation, it’s a most valuable component of your code) but thats not an excuse especially that software are now globally taking over most of the major systems (medical, auto, planes, buildings) in the world. The stability of these systems rely on the stability of the unit test cases.

So as a precursor to being a skilled full blown developer that loves to do unit test, let’s dive into some of the beginners guide into doing it.

## 2. Tools

For this example, I’ll be using Java as the platform, Eclipse as the IDE, and Maven as the project management tool. If you’re not yet familiar with these tools, please visit the Java, Eclipse IDE and Maven site.

## 3. Step by Step Guide

### 3.1 Create your project

Let’s create a project first.

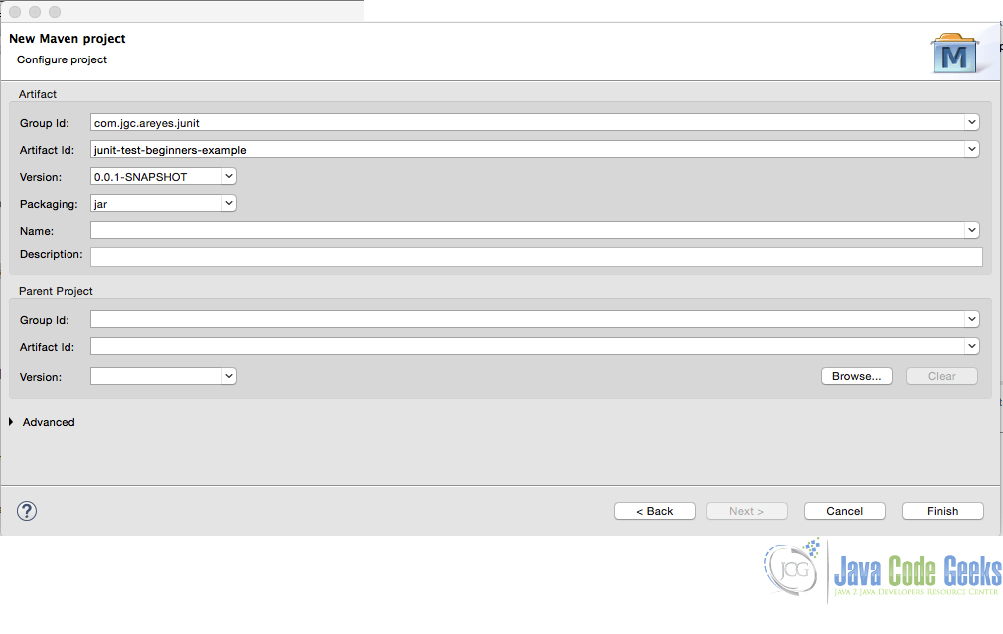
[](http://a5e2fba00d8bcb729d89839f.javacodegeeks.netdna-cdn.com/wp-content/uploads/2015/10/new_maven_junit_tst.jpg)

Figure 1.0 New Maven Project

After creating the project, you’ll be shown a project like the one below:

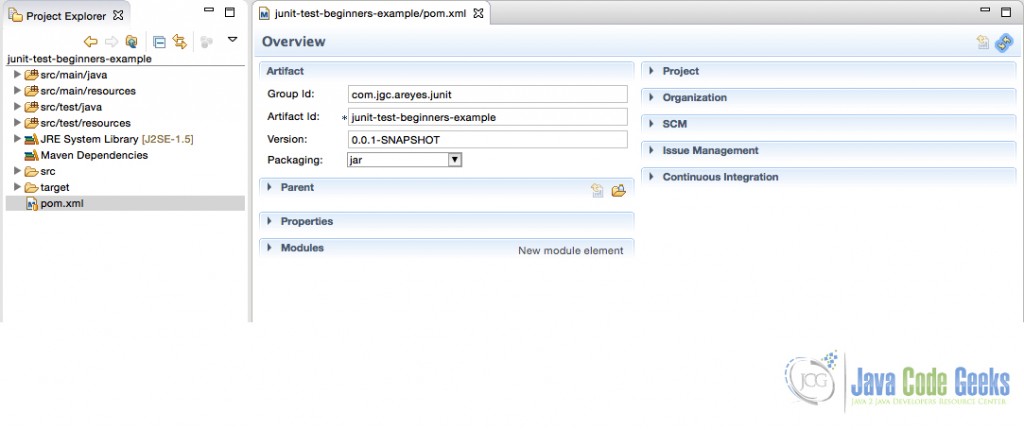
[](http://a5e2fba00d8bcb729d89839f.javacodegeeks.netdna-cdn.com/wp-content/uploads/2015/10/new_project_maven.jpg)

Figure 2.0 New Maven Project .xml

Make sure you include the Junit Library to your dependency list.

pom.xml

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>com.jgc.areyes.junit</groupId>

<artifactId>junit-test-beginners-example</artifactId>

<version>0.0.1-SNAPSHOT</version>

<dependencies>

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.12</version>

<scope>test</scope>

</dependency>

</dependencies>

</project>

### 3.2 Create the Service Class

Majority of the developers I know starts first with creating the implementation rather than the JUnit Test case, it isn’t a bad practice at all but wouldn’t it be more concrete if we create the JUnit Test cases first based on the design then create the implementation to pass all JUnit Test case? This is the case in the TDD, one of the most successful schemes of actual software development.

Assuming we are creating a service to manage an account. We need to introduce the following service methods:

* Create a new Account
* Update an Account
* Remove an Account
* List All Account Transactions

We have an OOP design that will handle this service and so we introduce the following classes (Class Diagram).

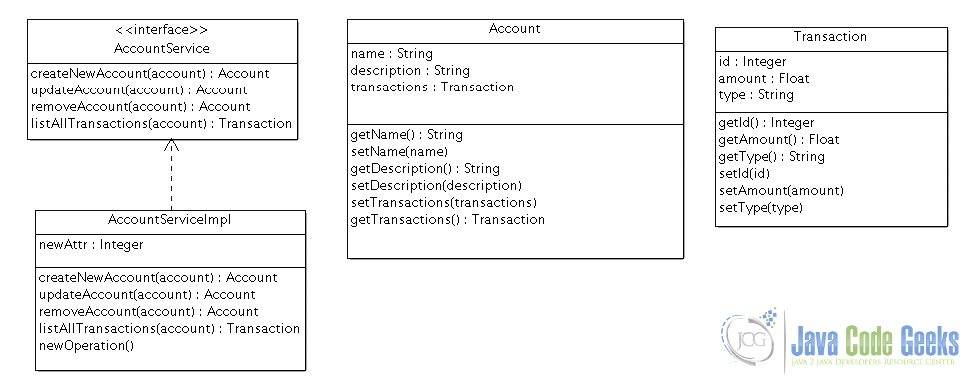
[](http://a5e2fba00d8bcb729d89839f.javacodegeeks.netdna-cdn.com/wp-content/uploads/2015/10/cdiag1.jpg)

Figure 3.0 Class Diagram

Here is the actual class that doesn’t have any implementation yet. We will create the implementation after creating the test cases for this class.

AccountServiceImpl.java

package com.areyes1.jgc.svc;

import java.util.List;

import com.areyes1.jgc.intf.AccountService;

import com.areyes1.jgc.obj.Account;

import com.areyes1.jgc.obj.Transaction;

public class AccountServiceImpl implements AccountService {

public Account createNewAccount(Account account) {

// TODO Auto-generated method stub

return null;

}

public Account updateAccount(Account account) {

// TODO Auto-generated method stub

return null;

}

public Account removeAccount(Account account) {

// TODO Auto-generated method stub

return null;

}

public List listAllTransactions(Account account) {

// TODO Auto-generated method stub

return null;

}

}

### 3.3 Create JUnit Test Cases

Now that we have the service placeholder, let’s create the Junit Test case for the AccountServiceImpl class. The Test cases will be the basis of your class functional aspect so you as a developer should write a solid and good test case (and not just fake it to pass).

When you create a test case, it will initially look like this:

AccountServiceImplTest.java

/\*\*

\*

\*/

package com.areyes1.jgc.svc;

import static org.junit.Assert.\*;

import org.junit.After;

import org.junit.Before;

import org.junit.Test;

/\*\*

\* @author alvinreyes

\*

\*/

public class AccountServiceImplTest {

/\*\*

\* Test method for {@link com.areyes1.jgc.svc.AccountServiceImpl#createNewAccount(com.areyes1.jgc.obj.Account)}.

\*/

@Test

public void testCreateNewAccount() {

fail("Not yet implemented");

}

/\*\*

\* Test method for {@link com.areyes1.jgc.svc.AccountServiceImpl#updateAccount(com.areyes1.jgc.obj.Account)}.

\*/

@Test

public void testUpdateAccount() {

fail("Not yet implemented");

}

/\*\*

\* Test method for {@link com.areyes1.jgc.svc.AccountServiceImpl#removeAccount(com.areyes1.jgc.obj.Account)}.

\*/

@Test

public void testRemoveAccount() {

fail("Not yet implemented");

}

/\*\*

\* Test method for {@link com.areyes1.jgc.svc.AccountServiceImpl#listAllTransactions(com.areyes1.jgc.obj.Account)}.

\*/

@Test

public void testListAllTransactions() {

fail("Not yet implemented");

}

}

As it can be seen above, it will fail the test case if it’s not yet implemented. From here on now, it’s a matter of discipline from the developer to create concrete and solid test cases.

Observing the JUnit Test cases.

* Using @Test to define a test method
* Put them all on the Test package (src/test/main/)
* Class always has a suffix of Test (AccountServiceImplTest)
* Methods always starts with “test”.

Here’s is the modified version of the JUnit Test cases. This will now be the basis of our implementation code.

*AccountServiceImplTest.java*

/\*\*

\*

\*/

package com.areyes1.jgc.svc;

import static org.junit.Assert.\*;

import org.junit.After;

import org.junit.Before;

import org.junit.Test;

import static org.hamcrest.CoreMatchers.\*;

import com.areyes1.jgc.obj.Account;

/\*\*

\* @author alvinreyes

\*

\*/

public class AccountServiceImplTest {

AccountServiceImpl accountService = new AccountServiceImpl();

/\*\*

\* Test method for {@link com.areyes1.jgc.svc.AccountServiceImpl#createNewAccount(com.areyes1.jgc.obj.Account)}.

\*/

@Test

public void testCreateNewAccount() {

Account newAccount = new Account();

newAccount.setName("Alvin Reyes");

newAccount.setDescription("This is the description");

Account newAcccountInserted = accountService.createNewAccount(newAccount);

// Check if the account has the same composition.

assertThat(newAccount, isA(Account.class));

assertEquals(newAccount.getName(), newAcccountInserted.getName());

}

/\*\*

\* Test method for {@link com.areyes1.jgc.svc.AccountServiceImpl#updateAccount(com.areyes1.jgc.obj.Account)}.

\*/

@Test

public void testUpdateAccount() {

// The old account (assumed that this came from a database or mock)

Account oldAccount = new Account();

oldAccount.setName("Alvin Reyes");

oldAccount.setDescription("This is the description");

String name = oldAccount.getName();

// Check if the account is still the same. it is expected to be different since we updated it.

Account expectedAccountObj = new Account();

expectedAccountObj = accountService.updateAccount(oldAccount);

assertThat(expectedAccountObj, isA(Account.class));

assertNotEquals(name, expectedAccountObj.getName());

}

/\*\*

\* Test method for {@link com.areyes1.jgc.svc.AccountServiceImpl#removeAccount(com.areyes1.jgc.obj.Account)}.

\*/

@Test

public void testRemoveAccount() {

// Set up the account to be removed.

Account toBeRemovedAccount = new Account();

toBeRemovedAccount.setName("Alvin Reyes");

toBeRemovedAccount.setDescription("This is the description");

// Removed the account.

assertTrue(accountService.removeAccount(toBeRemovedAccount));

}

/\*\*

\* Test method for {@link com.areyes1.jgc.svc.AccountServiceImpl#listAllTransactions(com.areyes1.jgc.obj.Account)}.

\*/

@Test

public void testListAllTransactions() {

// Dummy Transactions (can be mocked via mockito)

Account account = new Account();

account.setName("Alvin Reyes");

// Service gets all transaction

accountService.listAllTransactions(account);

// Check if there are transactions.

assertTrue(accountService.listAllTransactions(account).size() > 1);

}

}

Running the test case will show the following result.

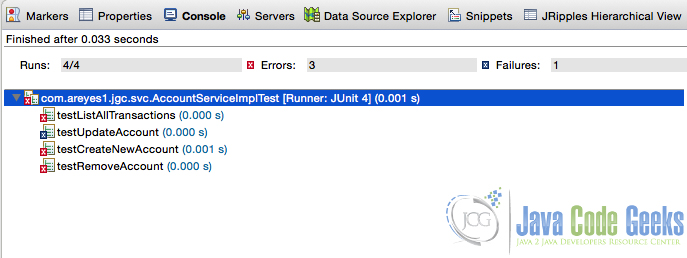
[](http://a5e2fba00d8bcb729d89839f.javacodegeeks.netdna-cdn.com/wp-content/uploads/2015/10/failed_test_case.jpg)

Figure 4.0 Failed Test cases

It failed cause we still have to code our implementation. Now in the implementation of our logic, our goal is make sure that these test cases succeeds!

### 3.4 Code the implementation

Now that the test cases are setup, we can now code our implementation. The test cases we created above will be the basis of how we will create the implementation. The goal is to pass the test cases!

package com.areyes1.jgc.svc;

import java.util.ArrayList;

import java.util.List;

import com.areyes1.jgc.intf.AccountService;

import com.areyes1.jgc.obj.Account;

import com.areyes1.jgc.obj.Transaction;

public class AccountServiceImpl implements AccountService {

public Account createNewAccount(Account account) {

// Dummy Dao! Database insert here.

// accountDao.insert(account);

// Ultimately return the account with the modification.

return account;

}

public Account updateAccount(Account account) {

// Dummy Dao! Database insert here.

// accountDao.update(account);

// Ultimately return the account with the modification.

account.setName("Alvin Reyes: New Name");

return account;

}

public boolean removeAccount(Account account) {

// Dummy Dao! Database insert here.

// accountDao.delete(account);

// Ultimately return the account with the modification.

// if exception occurs, return false.

return true;

}

public List listAllTransactions(Account account) {

// accountDao.loadAllTransactions(account);

List listOfAllTransactions = new ArrayList();

listOfAllTransactions.add(new Transaction());

listOfAllTransactions.add(new Transaction());

listOfAllTransactions.add(new Transaction());

account.setTransactions(listOfAllTransactions);

return listOfAllTransactions;

}

}

Running the test case will show the following result.

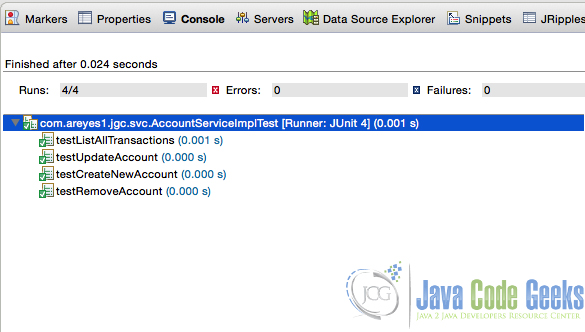
[](http://a5e2fba00d8bcb729d89839f.javacodegeeks.netdna-cdn.com/wp-content/uploads/2015/10/passed_test_case.jpg)

Figure 5.0 Passed Test case

### 3.5 Run your maven

Run your maven to see results.

-------------------------------------------------------

T E S T S

-------------------------------------------------------

Running com.areyes1.jgc.svc.AccountServiceImplTest

Tests run: 4, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.051 sec

Results :

Tests run: 4, Failures: 0, Errors: 0, Skipped: 0

[INFO]

[INFO] --- maven-jar-plugin:2.4:jar (default-jar) @ junit-test-beginners-example ---

[INFO] Building jar: /Users/alvinreyes/EclipseProjects/Java/junit-test-beginners-example/target/junit-test-beginners-example-0.0.1-SNAPSHOT.jar

[INFO]

[INFO] --- maven-install-plugin:2.4:install (default-install) @ junit-test-beginners-example ---

[INFO] Installing /Users/alvinreyes/EclipseProjects/Java/junit-test-beginners-example/target/junit-test-beginners-example-0.0.1-SNAPSHOT.jar to /Users/alvinreyes/.m2/repository/com/jgc/areyes/junit/junit-test-beginners-example/0.0.1-SNAPSHOT/junit-test-beginners-example-0.0.1-SNAPSHOT.jar

[INFO] Installing /Users/alvinreyes/EclipseProjects/Java/junit-test-beginners-example/pom.xml to /Users/alvinreyes/.m2/repository/com/jgc/areyes/junit/junit-test-beginners-example/0.0.1-SNAPSHOT/junit-test-beginners-example-0.0.1-SNAPSHOT.pom

[INFO]

[INFO] --- maven-resources-plugin:2.6:resources (default-resources) @ junit-test-beginners-example ---

[WARNING] Using platform encoding (UTF-8 actually) to copy filtered resources, i.e. build is platform dependent!

[INFO] Copying 0 resource

[INFO]

[INFO] --- maven-compiler-plugin:2.5.1:compile (default-compile) @ junit-test-beginners-example ---

[INFO] Nothing to compile - all classes are up to date

[INFO]

[INFO] --- maven-resources-plugin:2.6:testResources (default-testResources) @ junit-test-beginners-example ---

[WARNING] Using platform encoding (UTF-8 actually) to copy filtered resources, i.e. build is platform dependent!

[INFO] Copying 0 resource

[INFO]

[INFO] --- maven-compiler-plugin:2.5.1:testCompile (default-testCompile) @ junit-test-beginners-example ---

[INFO] Nothing to compile - all classes are up to date

[INFO]

[INFO] --- maven-surefire-plugin:2.12.4:test (default-test) @ junit-test-beginners-example ---

[INFO] Skipping execution of surefire because it has already been run for this configuration

[INFO] ------------------------------------------------------------------------

[INFO] BUILD SUCCESS

[INFO] ------------------------------------------------------------------------

[INFO] Total time: 9.456 s

[INFO] Finished at: 2015-10-12T16:28:01-05:00

[INFO] Final Memory: 11M/28M

[INFO] ------------------------------------------------------------------------

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

<http://examples.javacodegeeks.com/category/core-java/junit/>

# JUnit assertTrue example

Posted by: [Alvin Reyes](http://examples.javacodegeeks.com/author/alvin-reyes/) in [Core Java](http://examples.javacodegeeks.com/category/core-java/), [junit](http://examples.javacodegeeks.com/category/core-java/junit/) July 6th, 2015

## 1. Introduction

Every developer on every platform should always have a solid JUnit test case to secure the changes they made. This will minimise the potential risk of negativity to an existing system. There are a lot of ways to do JUnit Testing and a lot of functions available to do them. One of the most popular Unit Testing framework is the JUnit Library.

In this post, I’ll be discussing the usage of JUnit [assertTrue](http://junit.sourceforge.net/javadoc/org/junit/Assert.html#assertTrue%28boolean%29) and how can a developer use this to evaluate the test cases of their source using the [assertTrue](http://junit.sourceforge.net/javadoc/org/junit/Assert.html#assertTrue%28boolean%29) method from the Assert class object.

## 2. The Source

The source code below shows how can we use the assertTrue function.

JUnitTestAssertions.java

package com.areyes1.junitassertrue.sample;

package com.areyes1.junitassertrue.sample;

import static org.junit.Assert.assertTrue;

import org.junit.Before;

import org.junit.Test;

public class JUnitTestAssertions {

int totalNumberOfApplicants = 0;

int totalNumberOfAcceptableApplicants = 10;

@Before

public void setData(){

this.totalNumberOfApplicants = 9;

}

@Test

public void testAssertions() {

assertTrue((this.totalNumberOfApplicants != this.totalNumberOfAcceptableApplicants));

}

@Test

public void testAssertFalse() {

assertTrue((this.totalNumberOfApplicants == this.totalNumberOfAcceptableApplicants));

}

@Test

public void testAssertTrueWithMessage(){

assertTrue("Is total number of applicants acceptable?",(this.totalNumberOfApplicants != this.totalNumberOfAcceptableApplicants));

}

}

The [assertTrue](http://junit.sourceforge.net/javadoc/org/junit/Assert.html#assertTrue%28boolean%29) is a function from the [Assert](http://junit.sourceforge.net/javadoc/org/junit/Assert.html) object of the JUnit Library. It can be used to evaluate a specific condition that that runs on your application. This can be a simple boolean output if the user exist or not, an account has enough cash for a transaction or, in this case, a check if the number of applicants is within the acceptable range.

Running this example will give you an output in Eclipse.

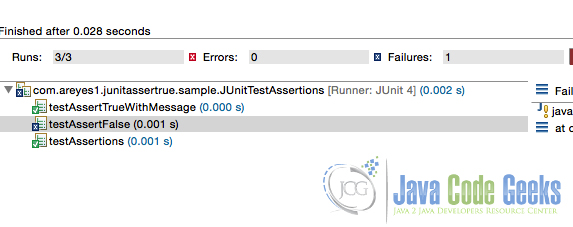
[](http://a5e2fba00d8bcb729d89839f.javacodegeeks.netdna-cdn.com/wp-content/uploads/2015/07/Screenshot-at-Jul-04-13-22-10.jpg)

Figure 1.0 JUnit Test result ran in Eclipse

## 3. Download the Eclipse project

This was an example of JUnit assertTrue.

**Download**  
You can download the full source code of this example here : [**junit-asserttrue-example**](http://a5e2fba00d8bcb729d89839f.javacodegeeks.netdna-cdn.com/wp-content/uploads/2015/07/junit-asserttrue-example.zip)

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

# JUnit assertFalse example

Posted by: [Alvin Reyes](http://examples.javacodegeeks.com/author/alvin-reyes/) in [junit](http://examples.javacodegeeks.com/category/core-java/junit/) July 14th, 2015

## 1. Introduction

To follow through with my previous post about [assertTrue](http://junit.sourceforge.net/javadoc/org/junit/Assert.html#assertTrue%28boolean%29), this will tackle about the exact opposite of that function. The [assertFalse](http://junit.sourceforge.net/javadoc/org/junit/Assert.html#assertFalse%28java.lang.String,%20boolean%29). The assertFalse is basically a function that can be used to check if a specific logic or process will return a false statement.

This can be in any conditional or structural logic that will return a boolean true or false. See the code example below.

## 2. The Source

JUnitAssertFalseExample.java

package com.areyes1.jgc.junit.assertfalse;

import org.junit.Before;

import org.junit.Test;

import static org.junit.Assert.assertFalse;

public class JUnitAssertFalseExample {

int totalNumberOfApplicants = 0;

int totalNumberOfAcceptableApplicants = 10;

@Before

public void setData() {

this.totalNumberOfApplicants = 9;

}

@Test

public void testAssertFalseFalse() {

assertFalse((this.totalNumberOfApplicants != this.totalNumberOfAcceptableApplicants));

}

@Test

public void testAssertFalse() {

assertFalse((this.totalNumberOfApplicants == this.totalNumberOfAcceptableApplicants));

}

@Test

public void testAssertFalseWithMessage() {

assertFalse(

"Is total number of applicants acceptable?",

(this.totalNumberOfApplicants != this.totalNumberOfAcceptableApplicants));

}

}

The [assertFalse](http://junit.sourceforge.net/javadoc/org/junit/Assert.html#assertFalse%28java.lang.String,%20boolean%29) is basically a function that can take any conditional statement that can either return a boolean (true or false). Developers can use it to test negative impacts and black box testing schemes.

Running this example will give you an output in Eclipse.

[](http://a5e2fba00d8bcb729d89839f.javacodegeeks.netdna-cdn.com/wp-content/uploads/2015/07/Screenshot-at-Jul-10-20-01-04.jpg)

Figure 1.0 Example of assertFalse

# JUnit assertEquals example

Posted by: [Alvin Reyes](http://examples.javacodegeeks.com/author/alvin-reyes/) in [junit](http://examples.javacodegeeks.com/category/core-java/junit/) July 15th, 2015

## 1. Introduction

To follow through with my previous post about [assertTrue](http://junit.sourceforge.net/javadoc/org/junit/Assert.html#assertTrue%28boolean%29) and [assertFalse](http://junit.sourceforge.net/javadoc/org/junit/Assert.html#assertEquals%28java.lang.Object[],%20java.lang.Object[]%29), this post will tackle on checking for an equality of a conditional statement on the test cases.

There is a method called [assertEquals](http://junit.sourceforge.net/javadoc/org/junit/Assert.html#assertEquals%28java.lang.Object[],%20java.lang.Object[]%29) in the JUnit library that can be used to check if two objects is equally defined or not. It can be used to check if a specific instance of an object is expected on a method called by the test, or if na object passed through a method was “polymorphed” correctly. This is to ensure that an object, decorated or not will have the same base properties as it is expected.

See the code example below.

## 2. The Source

JUnitAssertEqualExample.java

package com.areyes1.jgc.junit.assertequals;

import org.junit.Before;

import org.junit.Test;

import static org.junit.Assert.assertEquals;

public class JUnitAssertEqualsExample {

private JUnitAssertEqualsServiceExample junitAssertEqualsServiceSample;

private ServiceObject serviceObject;

@Before

public void setData() {

serviceObject = new ServiceObject();

junitAssertEqualsServiceSample = new JUnitAssertEqualsServiceExample();

junitAssertEqualsServiceSample.initiateMetaData(serviceObject);

}

@Test

public void testAssertEqualsFalse() {

// processed the item

ServiceObject newServiceObject = new ServiceObject();

junitAssertEqualsServiceSample.initiateMetaData(newServiceObject);

junitAssertEqualsServiceSample.processObject(serviceObject);

assertEquals(serviceObject,newServiceObject);

}

@Test

public void testAssertEquals() {

junitAssertEqualsServiceSample.processObject(serviceObject);

assertEquals(serviceObject,this.serviceObject);

}

@Test

public void testAssertEqualsWithMessage() {

junitAssertEqualsServiceSample.processObject(serviceObject);

assertEquals(

"Same Object",

serviceObject,serviceObject);

}

@Test

public void testAssertEqualsFalseWithMessage() {

ServiceObject newServiceObject = new ServiceObject();

junitAssertEqualsServiceSample.postProcessing(serviceObject);

assertEquals(

"Not the Same Object",

newServiceObject,serviceObject);

}

}

[Home](http://examples.javacodegeeks.com) » [Core Java](http://examples.javacodegeeks.com/category/core-java/) » [junit](http://examples.javacodegeeks.com/category/core-java/junit/) » JUnit assertEquals example

### About Alvin Reyes

Alvin Reyes

Alvin has an Information Technology Degree from Mapua Institute of Technology. During his studies, he was already heavily involved in a number of small to large projects where he primarily contributes by doing programming, analysis design. After graduating, he continued to do side projects on Mobile, Desktop and Web Applications. He founded TechnoWebHub Software Labs, a software development company that focuses on Web and Mobile Development.

[http://a5e2fba00d8bcb729d89839f.javacodegeeks.netdna-cdn.com/wp-content/themes/jarida/images/author_site.png](http://www.alvinjayreyes.com/)[http://a5e2fba00d8bcb729d89839f.javacodegeeks.netdna-cdn.com/wp-content/themes/jarida/images/author_twitter.png](http://twitter.com/alvinjayreyes)[http://a5e2fba00d8bcb729d89839f.javacodegeeks.netdna-cdn.com/wp-content/themes/jarida/images/author_facebook.png](https://www.facebook.com/alvinjayreyes)[http://a5e2fba00d8bcb729d89839f.javacodegeeks.netdna-cdn.com/wp-content/themes/jarida/images/author_google.png](https://plus.google.com/+AlvinReyes02142015)[http://a5e2fba00d8bcb729d89839f.javacodegeeks.netdna-cdn.com/wp-content/themes/jarida/images/author_linkedin.png](http://ph.linkedin.com/in/alvinpreyes)

# JUnit assertEquals example

Posted by: [Alvin Reyes](http://examples.javacodegeeks.com/author/alvin-reyes/) in [junit](http://examples.javacodegeeks.com/category/core-java/junit/) July 15th, 2015

## 1. Introduction

To follow through with my previous post about [assertTrue](http://junit.sourceforge.net/javadoc/org/junit/Assert.html#assertTrue%28boolean%29) and [assertFalse](http://junit.sourceforge.net/javadoc/org/junit/Assert.html#assertEquals%28java.lang.Object[],%20java.lang.Object[]%29), this post will tackle on checking for an equality of a conditional statement on the test cases.

There is a method called [assertEquals](http://junit.sourceforge.net/javadoc/org/junit/Assert.html#assertEquals%28java.lang.Object[],%20java.lang.Object[]%29) in the JUnit library that can be used to check if two objects is equally defined or not. It can be used to check if a specific instance of an object is expected on a method called by the test, or if na object passed through a method was “polymorphed” correctly. This is to ensure that an object, decorated or not will have the same base properties as it is expected.

See the code example below.

## 2. The Source

JUnitAssertEqualExample.java

|  |  |  |
| --- | --- | --- |
| 01 | package com.areyes1.jgc.junit.assertequals; | |
| 02 |  |

|  |  |  |
| --- | --- | --- |
| 03 | import org.junit.Before; | |
| 04 | import org.junit.Test; |

|  |  |
| --- | --- |
| 05 |  |
| 06 | import static org.junit.Assert.assertEquals; | |

|  |  |
| --- | --- |
| 07 |  |
| 08 | public class JUnitAssertEqualsExample { | |

|  |  |
| --- | --- |
| 09 |  |
| 10 | private JUnitAssertEqualsServiceExample junitAssertEqualsServiceSample; | |

|  |  |  |
| --- | --- | --- |
| 11 | private ServiceObject serviceObject; | |
| 12 | @Before |

|  |  |
| --- | --- |
| 13 | public void setData() { |
| 14 | serviceObject = new ServiceObject(); | |

|  |  |  |
| --- | --- | --- |
| 15 | junitAssertEqualsServiceSample = new JUnitAssertEqualsServiceExample(); | |
| 16 | junitAssertEqualsServiceSample.initiateMetaData(serviceObject); |

|  |  |  |
| --- | --- | --- |
| 17 | } | |
| 18 |  |

|  |  |
| --- | --- |
| 19 | @Test |
| 20 | public void testAssertEqualsFalse() { | |

|  |  |
| --- | --- |
| 21 | //  processed the item |
| 22 | ServiceObject newServiceObject = new ServiceObject(); | |

|  |  |  |
| --- | --- | --- |
| 23 | junitAssertEqualsServiceSample.initiateMetaData(newServiceObject); | |
| 24 | junitAssertEqualsServiceSample.processObject(serviceObject); |

|  |  |  |
| --- | --- | --- |
| 25 | assertEquals(serviceObject,newServiceObject); | |
| 26 | } |

|  |  |
| --- | --- |
| 27 |  |
| 28 | @Test | |

|  |  |
| --- | --- |
| 29 | public void testAssertEquals() { |
| 30 | junitAssertEqualsServiceSample.processObject(serviceObject); | |

|  |  |  |
| --- | --- | --- |
| 31 | assertEquals(serviceObject,this.serviceObject); | |
| 32 | } |

|  |  |
| --- | --- |
| 33 |  |
| 34 | @Test | |

|  |  |
| --- | --- |
| 35 | public void testAssertEqualsWithMessage() { |
| 36 | junitAssertEqualsServiceSample.processObject(serviceObject); | |

|  |  |
| --- | --- |
| 37 | assertEquals( |
| 38 | "Same Object", | |

|  |  |  |
| --- | --- | --- |
| 39 | serviceObject,serviceObject); | |
| 40 | } |

|  |  |
| --- | --- |
| 41 | @Test |
| 42 | public void testAssertEqualsFalseWithMessage() { | |

|  |  |
| --- | --- |
| 43 | ServiceObject newServiceObject = new ServiceObject(); |
| 44 | junitAssertEqualsServiceSample.postProcessing(serviceObject); | |

|  |  |
| --- | --- |
| 45 | assertEquals( |
| 46 | "Not the Same Object", | |

|  |  |  |
| --- | --- | --- |
| 47 | newServiceObject,serviceObject); | |
| 48 | } |

|  |  |
| --- | --- |
| 49 |  |
| 50 | } | |

The [assertEquals](http://junit.sourceforge.net/javadoc/org/junit/Assert.html#assertEquals%28Object,%20Object%29) is basically a function that takes two objects and see if they have the same instance object being used. The Example as shown above has 4 set of test that regresses the assertEquals. It checks for the same object that was processed and see if it’s still the same object in term of it’s instance, as it was passed before. A test case if the object used has a different one, and samples with messages in it.

Running this example will give you an output in Eclipse.

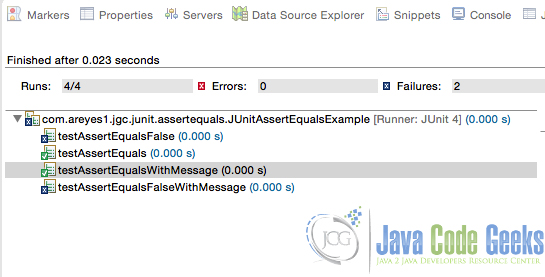
[](http://a5e2fba00d8bcb729d89839f.javacodegeeks.netdna-cdn.com/wp-content/uploads/2015/07/Screenshot-at-Jul-14-04-46-19.jpg)

Figure 1.0 Example of assertEquals

# JUnit SetUp / TearDown Example

Posted by: [Alvin Reyes](http://examples.javacodegeeks.com/author/alvin-reyes/) in [junit](http://examples.javacodegeeks.com/category/core-java/junit/) July 16th, 2015

## 1. Introduction

When we create JUnit test cases, we would normally setup our own configuration and data objects that can be used on our test cases. We needed them to be readily available when we create each of the method test cases and mock what was actually being used by the system at runtime.

We can prepare this within the test method but what a good alternative is override the [setup](http://junit.sourceforge.net/junit3.8.1/javadoc/junit/framework/TestCase.html#setUp%28%29) and [tearDown](http://junit.sourceforge.net/junit3.8.1/javadoc/junit/framework/TestCase.html#tearDown%28%29) method. These methods will be called for each test case method calls. This will allow the test case to do a prepration and post clean up process for each of the JUnit method test call.

## 2. The Source Code(s)

JUnitTestCaseWOAnnotation.java

package com.jgc.areyes1.junit;

import com.jgc.areyes1.junit.obj.Account;

import junit.framework.TestCase;

public class JUnitTestCaseWOAnnotation extends TestCase {

private AccountService accountService = new AccountService();

private Account dummyAccount;

@Override

protected void setUp() throws Exception {

System.out.println("Setting it up!");

dummyAccount = accountService.getAccountDetails();

}

public void testDummyAccount() {

System.out.println("Running: testDummyAccount");

assertNotNull(dummyAccount.getAccountCode());

}

public void testDummyAccountTransactions() {

System.out.println("Running: testDummyAccountTransactions");

assertEquals(dummyAccount.getAccountTransactions().size(),3);

}

@Override

protected void tearDown() throws Exception {

System.out.println("Running: tearDown");

dummyAccount = null;

assertNull(dummyAccount);

}

}

First things first is, we need to override the TestCase object from the JUnit Test class. This will allow the compiler to tag the class as a JUnit Test case class and have a new set of overridable methods for us to modify the behaviour of our specific class. We override the [setup](http://junit.sourceforge.net/junit3.8.1/javadoc/junit/framework/TestCase.html#setUp%28%29) and [tearDown](http://junit.sourceforge.net/junit3.8.1/javadoc/junit/framework/TestCase.html#tearDown%28%29) method so that we can do the preparation as well as the clean up process for each test method available.

Here’s the output:

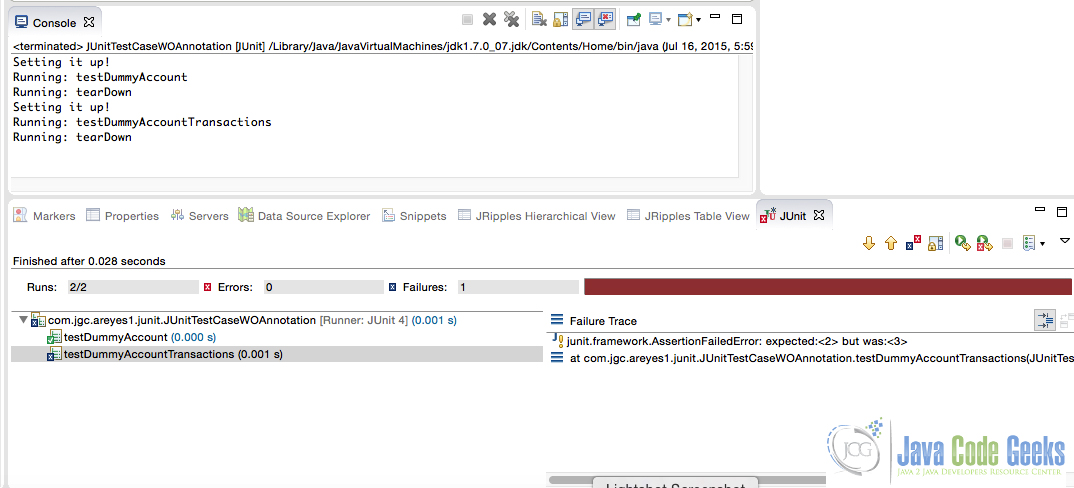
[](http://a5e2fba00d8bcb729d89839f.javacodegeeks.netdna-cdn.com/wp-content/uploads/2015/07/1_wo.jpg)

Figure 1.0 JUnit Test case setup/tearDown – non-annotation based approach

The example above is actually the old way of doing test cases, the new more flexible way is by using annotations to tag the class as a JUnit Test case. We then use the [@Before](http://junit.sourceforge.net/javadoc/org/junit/Before.html) ([setup](http://junit.sourceforge.net/junit3.8.1/javadoc/junit/framework/TestCase.html#setUp%28%29)) and [@After](http://junit.sourceforge.net/javadoc/org/junit/After.html) ([tearDown](http://junit.sourceforge.net/junit3.8.1/javadoc/junit/framework/TestCase.html#tearDown%28%29)) for our preparation and clean up. Here’s an example of the annotation based junit test case method.

*JUnitTestCaseWAnnotation.java*

package com.jgc.areyes1.junit;

import org.junit.After;

import org.junit.Before;

import org.junit.Test;

import com.jgc.areyes1.junit.obj.Account;

import static org.junit.Assert.\*;

public class JUnitTestCaseWAnnotation {

private AccountService accountService = new AccountService();

private Account dummyAccount;

@Before // setup()

public void before() throws Exception {

System.out.println("Setting it up!");

dummyAccount = accountService.getAccountDetails();

}

@Test

public void testDummyAccount() {

System.out.println("Running: testDummyAccount");

assertNotNull(dummyAccount.getAccountCode());

}

@Test

public void testDummyAccountTransactions() {

System.out.println("Running: testDummyAccountTransactions");

assertEquals(dummyAccount.getAccountTransactions().size(),3);

}

@After // tearDown()

public void after() throws Exception {

System.out.println("Running: tearDown");

dummyAccount = null;

assertNull(dummyAccount);

}

}

It utilises the [@Before](http://junit.sourceforge.net/javadoc/org/junit/Before.html) and [@After](http://junit.sourceforge.net/javadoc/org/junit/After.html) annotation for [setup](http://junit.sourceforge.net/junit3.8.1/javadoc/junit/framework/TestCase.html#setUp%28%29) and [tearDown](http://junit.sourceforge.net/junit3.8.1/javadoc/junit/framework/TestCase.html#tearDown%28%29) method calls respectively.

Here’s the output:

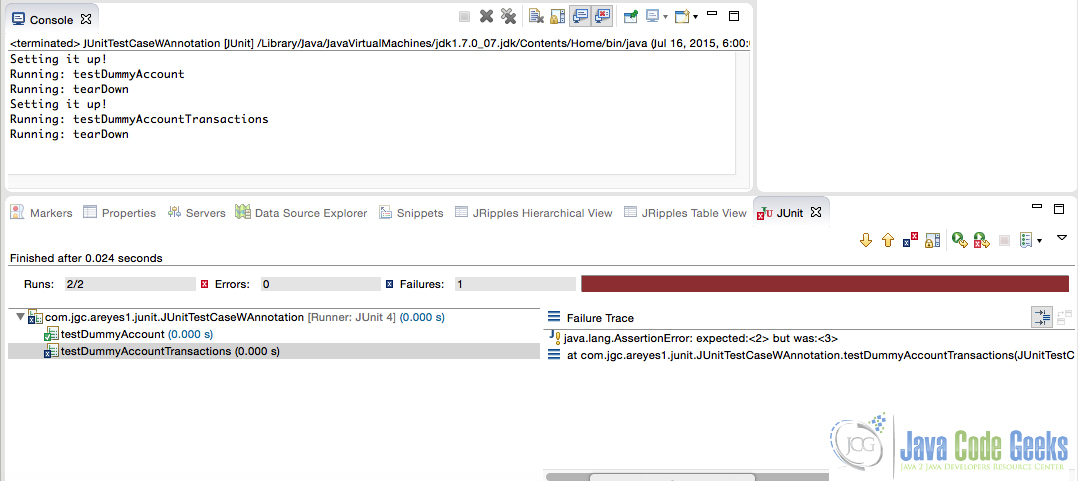
[](http://a5e2fba00d8bcb729d89839f.javacodegeeks.netdna-cdn.com/wp-content/uploads/2015/07/2_w.jpg)

Figure 1.0 JUnit Test case setup/tearDown – annotation based approach

# JUnit AssertThat Example

Posted by: [Alvin Reyes](http://examples.javacodegeeks.com/author/alvin-reyes/) in [junit](http://examples.javacodegeeks.com/category/core-java/junit/) July 17th, 2015

## 1. Introduction

The [assertThat](http://junit.sourceforge.net/javadoc/org/junit/Assert.html#assertThat%28T,%20org.hamcrest.Matcher%29) is one of the JUnit methods from the Assert object that can be used to check if a specific value match to an expected one.

It primarily accepts 2 parameters. First one if the actual value and the second is a matcher object. It will then try to compare this two and returns a boolean result if its a match or not. Example of it’s usage as per below.

## 2. The Source Code

JUnitTestAssertThatAssertions.java

package com.areyes1.junitassertrue.sample;

import static org.junit.Assert.\*;

import static org.hamcrest.CoreMatchers.\*;

import org.junit.Before;

import org.junit.Test;

public class JUnitTestAssertThatAssertions {

int totalNumberOfApplicants = 0;

@Before

public void setData(){

this.totalNumberOfApplicants = 9;

}

@Test

public void testAssertThatEqual() {

assertThat("123",is("123"));

}

@Test

public void testAssertThatNotEqual() {

assertThat(totalNumberOfApplicants,is(123));

}

@Test

public void testAssertThatObject() {

assertThat("123",isA(String.class));

}

@Test

public void testAssertThatWMessage(){

assertThat("They are not equal!","123",is("1234"));

}

}

The example shown above uses [is](http://hamcrest.org/JavaHamcrest/javadoc/1.3/) and [isA](http://hamcrest.org/JavaHamcrest/javadoc/1.3/) method from the hamcrest core to return a matcher object given the value. This is then used by the [assertThat](http://junit.sourceforge.net/javadoc/org/junit/Assert.html#assertThat%28T,%20org.hamcrest.Matcher%29) method that will then return a boolean result of the comparison.

Here’s the output:

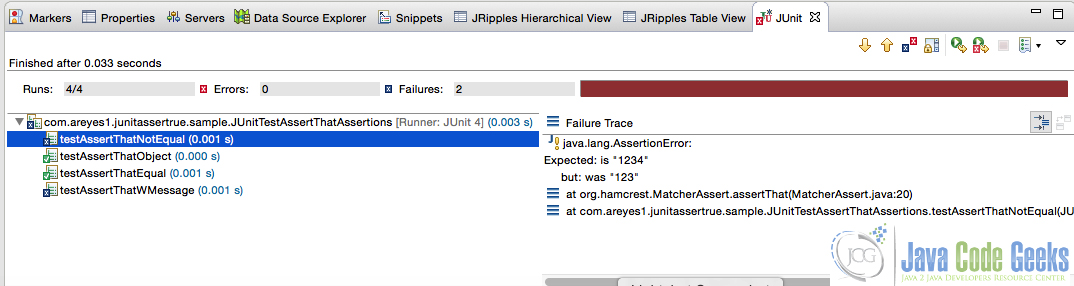
[](http://a5e2fba00d8bcb729d89839f.javacodegeeks.netdna-cdn.com/wp-content/uploads/2015/07/assertthat.jpg)

Figure 1.0 AssertThat output

Aside from the example above, there are far more methods that can be used to rigorously test one’s code. JUnits own hamcrest api has incorporate core, logical and object methods for this purpose.

### 2.1 Core Matchers

Before you start implementing your own Matcher’s, you should look at the core matchers that come with JUnit already. Here is a list of the matcher methods:

#### Core

* [any()](http://hamcrest.org/JavaHamcrest/javadoc/1.3/) Matches any object passed to it.
* [is()](http://hamcrest.org/JavaHamcrest/javadoc/1.3/) A matcher that checks if the given objects are equal.
* [describedAs()](http://hamcrest.org/JavaHamcrest/javadoc/1.3/) adds a description to the matcher

#### Logical

* [allOf()](http://hamcrest.org/JavaHamcrest/javadoc/1.3/) Takes an array of matchers and must all match the expected object.
* [anyOf()](http://hamcrest.org/JavaHamcrest/javadoc/1.3/) Takes an array of matcher and must match at least one of the matchers must report that it matches the target object.
* [not()](http://hamcrest.org/JavaHamcrest/javadoc/1.3/) Check if the object negates what was passed.

#### Object

* [equalTo()](http://hamcrest.org/JavaHamcrest/javadoc/1.3/) Equality check.
* [instanceOf()](http://hamcrest.org/JavaHamcrest/javadoc/1.3/) Check if an object is an instance of a given/expected object.
* [notNullValue()](http://hamcrest.org/JavaHamcrest/javadoc/1.3/) Check if the passed value is not null
* [nullValue()](http://hamcrest.org/JavaHamcrest/javadoc/1.3/) Tests whether the given object is null or not null.
* [sameInstance()](http://hamcrest.org/JavaHamcrest/javadoc/1.3/) Tests if the given object is the exact same instance as another.

The list above can all be used on [assertThat](http://junit.sourceforge.net/javadoc/org/junit/Assert.html#assertThat%28T,%20org.hamcrest.Matcher%29) method. It gives a wide range of possible scenarios to fully regress the extend of your algorithm, logic or processes of your application.

## 3. Bonus: Custom Matchers for your assertThat

You can actually create our own matcher and simply using that on your junit test case. See example of the custom matcher below.

CustomMatcher.java

package com.areyes1.junitassertthat.sample;

import org.hamcrest.BaseMatcher;

import org.hamcrest.Description;

import org.hamcrest.Matcher;

public class CustomMatcher {

public static Matcher matches(final Object expected){

return new BaseMatcher() {

protected Object expectedObject = expected;

public boolean matches(Object item) {

return expectedObject.equals(item);

}

public void describeTo(Description description) {

description.appendText(expectedObject.toString());

}

};

}

}

We can then use that matcher as part of our Junit Test source.

package com.areyes1.junitassertthat.sample;

import static org.junit.Assert.\*;

import static com.areyes1.junitassertthat.sample.CustomMatcher.\*;

import static org.hamcrest.CoreMatchers.\*;

import java.util.ArrayList;

import java.util.List;

import org.junit.Before;

import org.junit.Test;

public class JUnitTestAssertThatCustomMatcher {

ArrayList listOfValidStrings = new ArrayList();

private String inputValue = new String("Hello");

@Before

public void setData(){

listOfValidStrings.add("object\_1");

listOfValidStrings.add("object\_2");

listOfValidStrings.add("object\_3");

}

@Test

public void testLogic(){

assertThat(inputValue,matches("Hello"));

}

}

Here’s the output:

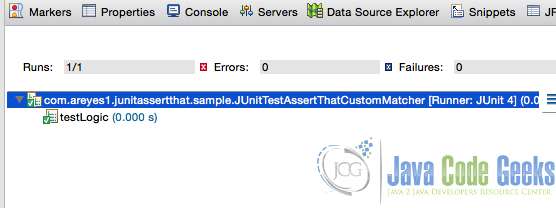
[](http://a5e2fba00d8bcb729d89839f.javacodegeeks.netdna-cdn.com/wp-content/uploads/2015/07/custom_matcher.jpg)

Figure 2.0 AssertThat output

# JUnit Report Example

Posted by: [Alvin Reyes](http://examples.javacodegeeks.com/author/alvin-reyes/) in [junit](http://examples.javacodegeeks.com/category/core-java/junit/) July 20th, 2015

## 1. Introduction

In all the test cases of an application, there is always a way to create a report of it one way or the other. JUnit is not an exception to this and there’s a ton of ways to create reports from it. It can even be incorporated to the maven site and have it part of the overall java tech documentations of a Java Application.

One of the most widely used JUnit reporting plugin is the surefire and in this post, I’ll be showcasing how it can be used on your JUnit test cases.

## 2. The Surefire reporting  plugin

The Surefire Report Plugin parses the generated TEST-\*.xml files under ${basedir}/target/surefire-reports and renders them using DOXIA, which creates the web interface version of the test results. It also allows generating reports in HTML format which will be the focus of our samples in this post. You can check the surefire plugin from [here](http://maven.apache.org/surefire/maven-surefire-report-plugin/).

To configure your project to have surefire plugin, just include the following reporting plugin on the pom.xml of your maven project.

pom.xml

<reporting>

<plugins>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-surefire-report-plugin</artifactId>

<version>2.18.1</version>

</plugin>

</plugins>

</reporting>

This will allow your project to call the surefire plugin goals to generate the HTML report.

## 3. Source

Let’s try creating a JUnit Test case and run the surefire report generation.

JUnitReportServiceExampleTest.java

package com.areyes1.junitreport.service;

import static org.junit.Assert.\*;

import org.junit.Before;

import org.junit.Test;

public class JUnitReportServiceExampleTest {

private JUnitReportServiceExample junitAssertEqualsServiceSample;

private ServiceObject serviceObject;

@Before

public void setData() {

serviceObject = new ServiceObject();

junitAssertEqualsServiceSample = new JUnitReportServiceExample();

junitAssertEqualsServiceSample.initiateMetaData(serviceObject);

}

@Test

public void testAssertEqualsFalse() {

// processed the item

ServiceObject newServiceObject = new ServiceObject();

junitAssertEqualsServiceSample.initiateMetaData(newServiceObject);

junitAssertEqualsServiceSample.processObject(serviceObject);

assertEquals(serviceObject,newServiceObject);

}

@Test

public void testAssertEquals() {

junitAssertEqualsServiceSample.processObject(serviceObject);

assertEquals(serviceObject,this.serviceObject);

}

@Test

public void testAssertEqualsWithMessage() {

junitAssertEqualsServiceSample.processObject(serviceObject);

assertEquals(

"Same Object",

serviceObject,serviceObject);

}

@Test

public void testAssertEqualsFalseWithMessage() {

ServiceObject newServiceObject = new ServiceObject();

junitAssertEqualsServiceSample.postProcessing(serviceObject);

assertEquals(

"Not the Same Object",

newServiceObject,serviceObject);

}

}

We create several test cases here that will succeed and fail. We then run the following command to generate the report.

mvn clean install test surefire-report:report

The report will be generated under the target folder of your maven project.

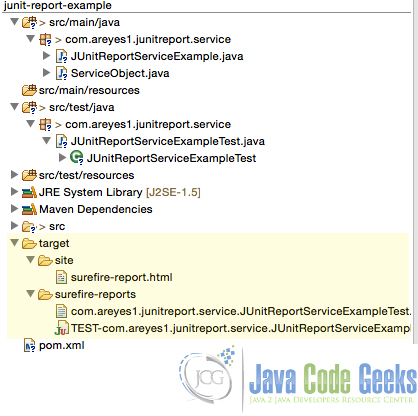
[](http://a5e2fba00d8bcb729d89839f.javacodegeeks.netdna-cdn.com/wp-content/uploads/2015/07/junitsurefire_target.jpg)

Figure 1.0 Surefire HTML and XML under the target folder

## 4. Reports

Here is an example of a report generated by the plugin.

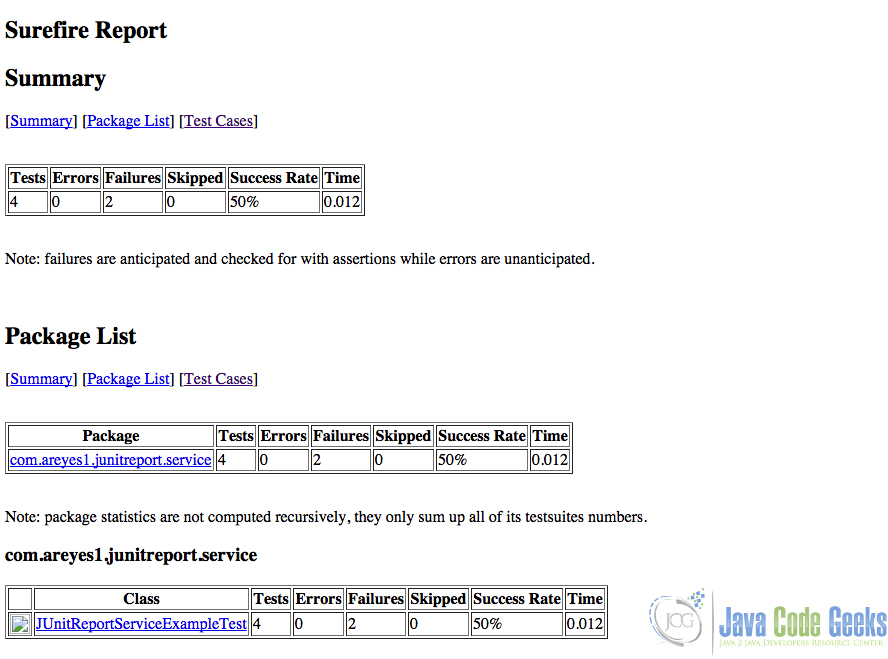
[](http://a5e2fba00d8bcb729d89839f.javacodegeeks.netdna-cdn.com/wp-content/uploads/2015/07/junitreport.jpg)

Figure 2.0 JUnit HTML report

It gives you the number of succeeded and failed test cases as well as the percentage covered (coverage) of your test cases. It’s a pretty neat reporting for projects that are keen to their quality of testing per build cycles. I see this as an extremely great feature for any Java application as the success of the test cases is directly proportional to the quality of the functionality of the application.

# Spring JUnit Test Example

Posted by: [Alvin Reyes](http://examples.javacodegeeks.com/author/alvin-reyes/) in [junit](http://examples.javacodegeeks.com/category/core-java/junit/) July 23rd, 2015

## 1. Introduction

Spring has it’s own testing framework for all objects created under it’s IOC container (that we all know is just the JVM instance). Spring has created it’s own paradigm of object creation and destruction and it made sense for it to have it’s own testing framework. The great thing about this is (as being Java) is it can be coupled with the core JUnit Testing framework, making it really easy to create JUnit Tests for its sources.

In this example, I’ll be showcasing how to do a Unit Tests for Spring beans using only annotations.

## 2. Maven POM

We need to include the following dependencies on our project. This will allow us to use Spring Test framework as well as the context package to allow us to inject our objects into our test cases.

pom.xml

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>com.jgc.areyes.junit</groupId>

<artifactId>junit-spring-example</artifactId>

<version>0.0.1-SNAPSHOT</version>

<dependencies>

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.12</version>

<scope>test</scope>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-test</artifactId>

<version>4.1.7.RELEASE</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context</artifactId>

<version>4.1.7.RELEASE</version>

</dependency>

</dependencies>

</project>

## 3. The main class file

The following source is the main junit test case code of our example. As it can be seen here, we used annotation based context configuration. This approach allow developers to wire up their beans via Java classes instead of XML files.

JUnitSpringExample.java

package com.areyes1.jgc.junit.spring;

import org.junit.After;

import org.junit.AfterClass;

import org.junit.Before;

import org.junit.BeforeClass;

import org.junit.Test;

import org.junit.runner.RunWith;

import static org.junit.Assert.\*;

import static org.hamcrest.CoreMatchers.\*;

import static org.hamcrest.BaseMatcher.\*;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.test.context.ContextConfiguration;

import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;

import org.springframework.test.context.support.AnnotationConfigContextLoader;

import com.areyes1.jgc.junit.spring.main.AppConfig;

import com.areyes1.jgc.junit.spring.service.SampleService;

import com.areyes1.jgc.unit.obj.Order;

@RunWith(SpringJUnit4ClassRunner.class)

@ContextConfiguration(classes = AppConfig.class, loader = AnnotationConfigContextLoader.class)

public class JUnitSpringExample {

@Autowired

private SampleService sampleService;

@Test

public void testSampleService() {

assertEquals(

"class com.areyes1.jgc.junit.spring.service.SampleServiceImpl",

this.sampleService.getClass().toString());

}

@Test

public void testSampleServiceGetAccountDescription() {

// Check if the return description has a Description: string.

assertTrue(sampleService.getOrderDescription().contains("Description:"));

}

@Test

public void testSampleServiceGetAccountCode() {

// Check if the return description has a Code: string.

assertTrue(sampleService.getOrderStringCode().contains("Code:"));

}

@Test

public void testSampleServiceCreateNewOrder() {

Order newOrder = new Order();

newOrder.setSecurityCode("XYZ");

newOrder.setDescription("Description");

if (newOrder != null) {

assertThat(sampleService.createOrder(newOrder),

instanceOf(Order.class));

assertNotNull("Security isn't null", newOrder.getSecurityCode());

assertNotNull("Description isn't not null",

newOrder.getDescription());

}

assertNotNull("New Order is not null", newOrder);

}

@Test

public void testSampleServiceGetOrder() {

Order existingOrder = sampleService.getOrder(0);

if (existingOrder != null) {

assertThat(sampleService.getOrder(0), instanceOf(Order.class));

assertNotNull("Security isn't null",

existingOrder.getSecurityCode());

assertNotNull("Description isn't null",

existingOrder.getDescription());

}

assertNotNull("Object is not null", existingOrder);

}

}

@RunWith – is an annotation to tag a class that this should Run with a specific runner class. There are a lot of alternative runners for JUnit but since we are using Spring test, we use: SpringJUnit4ClassRunner.class

@ContextConfiguration – Similar to calling the ClassPathContextConfiguration, this one is a pretty neat feature in Spring, especially in Unit test cases. It can be used to what type of ContextLoader for our class and the Java class files that we need to run on the ioc container (basically JVM) that are injectable on this class. For this case, we use AnnotationConfigContextLoader.class as our ContextLoader since we are creating a fully using annotation as our context metadata information and passed our AppConfig.class as our configuration stating point.

*AppConfig.java*

package com.areyes1.jgc.junit.spring.main;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

import com.areyes1.jgc.junit.spring.service.SampleService;

import com.areyes1.jgc.junit.spring.service.SampleServiceImpl;

@Configuration

public class AppConfig {

@Bean

public SampleService getSampleService() {

return new SampleServiceImpl();

}

}

After implementing these 2 annotations, passing the correct context loader and configuration classes, we can now use Spring based annotation on our JUnit Test case classes. You can see this in action by looking at the class above when we injected the SampleService via wiring ([@Autowired](http://docs.spring.io/autorepo/docs/spring-framework/3.0.x/api/org/springframework/beans/factory/annotation/Autowired.html)).

We then proceed with creating our JUnit Test cases for all specific method that the SampleService has (We need to include as much Junit Test case for each method as possible as this will greatly contribute to the coverage).

## 4. Output

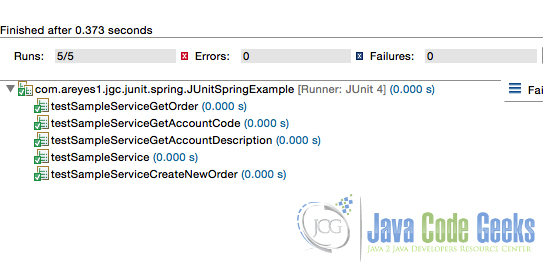
[](http://a5e2fba00d8bcb729d89839f.javacodegeeks.netdna-cdn.com/wp-content/uploads/2015/07/springjunit.jpg)

Figure 1.0 Spring JUnit Exampl

# JUnit Exception Handling Example

Posted by: [Alvin Reyes](http://examples.javacodegeeks.com/author/alvin-reyes/) in [junit](http://examples.javacodegeeks.com/category/core-java/junit/) August 3rd, 2015

## 1. Introduction

There are popular ways to test exception in JUnit. A developer can use the traditional try-catch statement, the [@Rule](http://junit.org/apidocs/org/junit/Rule.html) or the annotation based. In this post, I’ll be discussing these 3 simple and easy to implement ways to make your test case and functional scenarios bulletproof of incoming exceptions.

## 2. Source(s)

In this example, we use the traditional way catching an exception. We use the try-catch clause to catch and throw an assertion condition that returns the test case result.

JUnitTryCatchExample.java

package com.areyes1.junit.exceptions;

import static org.junit.Assert.\*;

import org.junit.Test;

public class JUnitTryCatchExample {

double value = 0.0d;

/\*\*

\* We are catching the exception using the traditional try-catch clause and return

\* an assertion.

\*/

@Test

public void testJUnitTryCatch() {

try {

if(value < 0.1d) {

fail("Value given is not as expected");

}

} catch (Exception e) {

assertTrue((value != 0.0d));

}

}

}

Let’s go through the code.

1. We introduce a double variable equating to 0.0. This will be our basis of the test case
2. We created a method and tag it as [@Test](http://junit.org/apidocs/org/junit/Test.html) as this is a test case.
3. We introduce a try catch and wrap the process of our test case. In our example, we want to check if the value is less the 0.1. If it does, it manually fail and throws the exception
4. The exception then evaluate an assertion and throws the overall result of the test case.

This is a basic example of how can we use the traditional try-catch clause on our test case, now let’s look at on the annotation based approach.

JUnitRuleAnnotation.java

package com.areyes1.junit.exceptions;

import org.junit.Rule;

import org.junit.Test;

import org.junit.rules.ExpectedException;

public class JUnitRuleAnnotation {

// We introduce this to create an expected exception object

@Rule

public ExpectedException expectedThrown = ExpectedException.none();

/\*\*

\* The test is expecting throw the expected exception (which in this case, the NumberFormatException)

\*/

@Test

public void testExpectedRuleException() {

expectedThrown.expect(NumberFormatException.class);

String numberStr = "abc";

Integer.valueOf(numberStr);

}

}

Unlike the try-catch statement clause usage, we use the [@Rule](http://junit.org/apidocs/org/junit/Rule.html) annotation to create an expected exception object. This is used to create an expected object exception for the test case to make sure that it throws the expected exception.

Let’s go deep into our code.

1. We introduce a [@Rule](http://junit.org/apidocs/org/junit/Rule.html) annotation. This is a specialized annotation to tag an object that we are imposing a rule for our test case.
2. We introduce the test method using [@Test](http://junit.org/apidocs/org/junit/Test.html)
3. We use the expectedThrown method and pass over the expected exception. This will then be used on the evaluation of the test case method.
4. Finally we introduce the actual code that we wanted to test. In this case, we want to test the integer boxing of a string. We are expecting that it throws a NumberFormatException.

The @Rule annotation allows us to specify a specific Exception class that the test case expects. This is a very powerful Junit feature as it will allow developers to fail proof their methods.

Aside from the @Rule, we can actually pass an expected exception class on the [@Test](http://junit.org/apidocs/org/junit/Test.html) annotation. See example below.

JUnitAnnotationExample.java

package com.areyes1.junit.exceptions;

import org.junit.Test;

public class JUnitAnnotationExample {

/\*\*

\* This means that the method is expecting a number format exception.

\*/

@Test(expected=NumberFormatException.class)

public void testAnnotationExample() {

String numberStr = "abc";

Integer.valueOf(numberStr);

}

}

My personal favourite, using annotations only. The [@Test](http://junit.org/apidocs/org/junit/Test.html) annotation accepts an expected and message parameter, to tag the test method that it will only return the expected exception and the message it will give you if ever it didn’t.

Let’s go over the method

1. We create a typical annotation based test case using [@Test](http://junit.org/apidocs/org/junit/Test.html)
2. Instead of creating an [@Rule](http://junit.org/apidocs/org/junit/Rule.html), we use the expected attribute of the [@Test](http://junit.org/apidocs/org/junit/Test.html) to pass the expected exception.
3. We test our code with it.

This is somehow a more cleaner and straightforward approach. Instead of creating an [@Rule](http://junit.org/apidocs/org/junit/Rule.html) explicitly, we use the expected attribute for our test case.

## 3. Add-on: Custom Annotations

It is possible to create a custom annotation class to be used for test cases. This will allow developers to fully customise the behaviour of the test case once an assertion is evaluated and an exception is thrown.

StringCalculatorTest.java

@RunWith(ExpectsExceptionRunner.class)

public class StringCalculatorTest {

@Test

@ExpectsException(type = IllegalArgumentException.class, message = "negatives not allowed: [-1]")

public void throwsExceptionWhenNegativeNumbersAreGiven() throws Exception {

// act

calculator.add("-1,-2,3");

}

}

**the annotation**  
*ExpectsException.java*

@Retention(RetentionPolicy.RUNTIME)

@Target({ElementType.METHOD})

public @interface ExpectsException {

Class type();

String message() default "";

}

**unner class (copy pasted code)**  
ExpectsExceptionRunner.javapublic class ExpectsExceptionRunner extends BlockJUnit4ClassRunner {

public ExpectsExceptionRunner(Class klass) throws InitializationError {

super(klass);

}

@Override

protected Statement possiblyExpectingExceptions(FrameworkMethod method, Object test, Statement next) {

ExpectsException annotation = method.getAnnotation(ExpectsException.class);

if (annotation == null) {

return next;

}

return new ExpectExceptionWithMessage(next, annotation.type(), annotation.message());

}

class ExpectExceptionWithMessage extends Statement {

private final Statement next;

private final Class expected;

private final String expectedMessage;

public ExpectExceptionWithMessage(Statement next, Class expected, String expectedMessage) {

this.next = next;

this.expected = expected;

this.expectedMessage = expectedMessage;

}

@Override

public void evaluate() throws Exception {

boolean complete = false;

try {

next.evaluate();

complete = true;

} catch (AssumptionViolatedException e) {

throw e;

} catch (Throwable e) {

if (!expected.isAssignableFrom(e.getClass())) {

String message = "Unexpected exception, expected but was ";

throw new Exception(message, e);

}

if (isNotNull(expectedMessage) && !expectedMessage.equals(e.getMessage())) {

String message = "Unexpected exception message, expected but was";

throw new Exception(message, e);

}

}

if (complete) {

throw new AssertionError("Expected exception: "

+ expected.getName());

}

}

private boolean isNotNull(String s) {

return s != null && !s.isEmpty();

}

}

}

# JUnit Test Suite Example

Posted by: [Alvin Reyes](http://examples.javacodegeeks.com/author/alvin-reyes/) in [junit](http://examples.javacodegeeks.com/category/core-java/junit/) August 7th, 2015

## 1. Introduction

A JUnit Test suite is used to bundle multiple test cases together in a single run. This is usually used if you want to integrated several JUnit Test cases that makes up a specific functionality from the integration. The developer uses the [@RunWith](http://junit.sourceforge.net/javadoc/org/junit/runner/RunWith.html) and [@SuiteClasses](http://junit.sourceforge.net/javadoc/org/junit/runners/Suite.html) to run Suite tests for this purposes.

In this example, I’ll be showing a simple approach on how you’ll create a Test Suite that will call 3 test cases.  
 

## 2. Sources

We first prepare our test cases. In this example, we created 3 different test case classes.

JUnitTest1Suite1.java

package com.areyes1.jgc.test.suite;

import static org.junit.Assert.assertTrue;

import org.junit.Before;

import org.junit.Test;

public class JUnitTest1Suite1 {

int totalNumberOfApplicants = 0;

int totalNumberOfAcceptableApplicants = 10;

@Before

public void setData(){

this.totalNumberOfApplicants = 9;

}

@Test

public void testAssertions() {

assertTrue((this.totalNumberOfApplicants != this.totalNumberOfAcceptableApplicants));

}

@Test

public void testAssertFalse() {

assertTrue((this.totalNumberOfApplicants == this.totalNumberOfAcceptableApplicants));

}

@Test

public void testAssertTrueWithMessage(){

assertTrue("Is total number of applicants acceptable?",(this.totalNumberOfApplicants != this.totalNumberOfAcceptableApplicants));

}

}

JUnitTest2Suite1.java

package com.areyes1.jgc.test.suite;

import static org.hamcrest.CoreMatchers.is;

import static org.hamcrest.CoreMatchers.isA;

import static org.junit.Assert.assertThat;

import java.util.ArrayList;

import org.junit.Before;

import org.junit.Test;

public class JUnitTest2Suite1 {

int totalNumberOfApplicants = 0;

int totalNumberOfAcceptableApplicants = 10;

ArrayList listOfValidStrings = new ArrayList();

@Before

public void setData(){

this.totalNumberOfApplicants = 9;

listOfValidStrings.add("object\_1");

listOfValidStrings.add("object\_2");

listOfValidStrings.add("object\_3");

}

@Test

public void testAssertThatEqual() {

assertThat("123",is("123"));

}

@Test

public void testAssertThatNotEqual() {

assertThat(totalNumberOfApplicants,is(123));

}

@Test

public void testAssertThatObject() {

assertThat("123",isA(String.class));

}

@Test

public void testAssertThatWMessage(){

assertThat("They are not equal!","123",is("1234"));

}

}

JUnitTest3Suite1

package com.areyes1.jgc.test.suite;

import static org.junit.Assert.assertEquals;

import static org.junit.Assert.assertNotNull;

import static org.junit.Assert.assertNull;

import org.junit.After;

import org.junit.Before;

import org.junit.Test;

import com.jgc.areyes1.junit.AccountService;

import com.jgc.areyes1.junit.obj.Account;

public class JUnitTest3Suite1 {

private AccountService accountService = new AccountService();

private Account dummyAccount;

@Before // setup()

public void before() throws Exception {

System.out.println("Setting it up!");

dummyAccount = accountService.getAccountDetails();

}

@Test

public void testDummyAccount() {

System.out.println("Running: testDummyAccount");

assertNotNull(dummyAccount.getAccountCode());

}

@Test

public void testDummyAccountTransactions() {

System.out.println("Running: testDummyAccountTransactions");

assertEquals(dummyAccount.getAccountTransactions().size(),3);

}

@After // tearDown()

public void after() throws Exception {

System.out.println("Running: tearDown");

dummyAccount = null;

assertNull(dummyAccount);

}

}

After creating or identifying our JUnit test cases that we want to include on this suite, we then create our JUnit Test case suite that will call the 3 tests case classes. Take note of the usage of [@RunWith](http://junit.sourceforge.net/javadoc/org/junit/runner/RunWith.html) and[@SuiteClasses](http://junit.sourceforge.net/javadoc/org/junit/runners/Suite.html) as per below.

*JUnitSuite1.java*

package com.areyes1.jgc.test.suite;

import org.junit.runner.RunWith;

import org.junit.runners.Suite;

import org.junit.runners.Suite.SuiteClasses;

@RunWith(Suite.class)

@SuiteClasses({

JUnitTest1Suite1.class,

JUnitTest2Suite1.class,

JUnitTest3Suite1.class

})

public class JUnitSuite1 {

}

The [@RunWith](http://junit.sourceforge.net/javadoc/org/junit/runner/RunWith.html) annotation is used to tag a class that it should run with a specific class. In this example, we are running this class with a Suite.class to tell our runner that we required that class for this run. The [@SuiteClasses](http://junit.sourceforge.net/javadoc/org/junit/runners/Suite.html) is the annotation that is used to specify the JUnit Test cases that this suite will be composed of. As specified, we called the 3 unit test case class we created.

These are all test cases that we want to run in this suite. As we can see, we have different test case scenarios for each. The purpose of us creating a test suite for them is so that we can consolidate their execution and results into one.

## 3. Running the JUnit Test Suite

You can run the test case by:

1. Eclipse JUnit Test case run
2. Create a Runner class from your main entry point

### 3.1 Eclipse JUnit Test case run

The eclipse way is straight forward, we right click on the JUnitSuite1.java > Run As > JUnit Test.

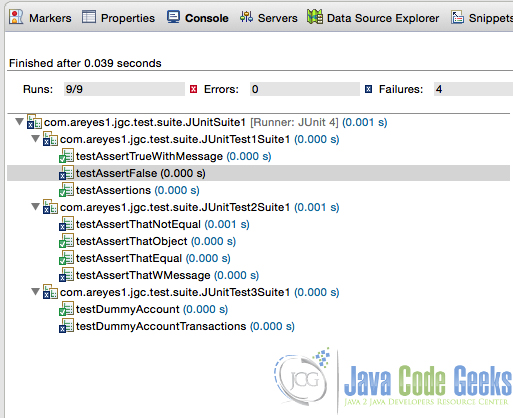
[](http://a5e2fba00d8bcb729d89839f.javacodegeeks.netdna-cdn.com/wp-content/uploads/2015/08/Screenshot-at-Aug-07-16-43-47.jpg)

Figure 1.0 JUnit Test Suite Run via Eclipse result

### 3.2 Create a Runner class from your main entry point

We can also run the test case using a Runner class. Code of the runner below

TestRunnerForTestSuite.java

package com.areyes1.jgc.test.suite.runner;

import org.junit.runner.JUnitCore;

import org.junit.runner.notification.Failure;

import com.areyes1.jgc.test.suite.JUnitSuite1;

public class TestRunnerForTestSuite {

public static void main(String[] args) {

org.junit.runner.Result result = JUnitCore.runClasses(JUnitSuite1.class);

for (Failure failure : result.getFailures()) {

System.out.println(failure.toString());

}

System.out.println(result.wasSuccessful());

}

}

Result of running the runner below

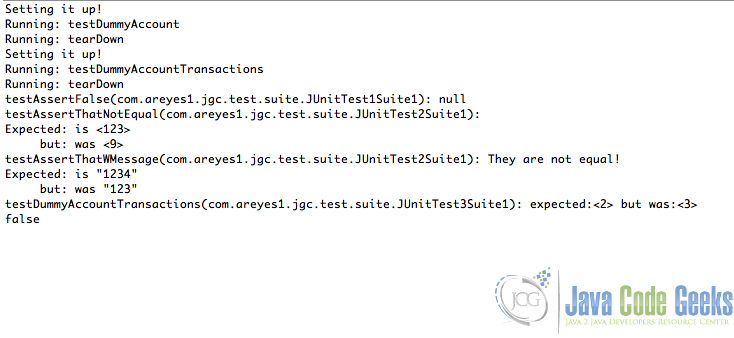
[](http://a5e2fba00d8bcb729d89839f.javacodegeeks.netdna-cdn.com/wp-content/uploads/2015/08/Screenshot-at-Aug-07-16-46-39.jpg)

Figure 2.0 Running the runner class result

As you might have observe, the test suite runs all the test case class that’s associated to it. It’s grouping all of these test case to come up with a unified results of specific scenarios of the system. This is extremely a valuable feature especially if we need to test different scenarios of integrated services for a common functional requirement. A great example would be if there is a separate module/service for account opening and account cash transfer. With this feature, we can combine the test cases of these two functional requirements and come up with a unified test case of an end to end account opening to cash transfer process.

# JUnit Best Practices

Posted by: [Alvin Reyes](http://examples.javacodegeeks.com/author/alvin-reyes/) in [junit](http://examples.javacodegeeks.com/category/core-java/junit/) August 20th, 2015

## 1. Introduction

There are a lot of ways on how a developer can do [JUnit](http://junit.org/) test cases, but overall it really depends on the strategy of how a specific functional code can be tested to it’s limits. It’s the developers responsibility to introduce thorough test cases to make sure that the code is stable and ultimately does what it has to do.

In this blog post, I’ll share some of the best practices I have learned and discuss my experience on them.

## 2. Best Practices

The following best practices are a compilation of research and personal experience.

### 2.1 TDD approach. Create your test case before anything else.

In TDD, one usually starts with the test cases before coding the actual implementation. The test cases are expected to be created with respected and tightly coupled to how the system or the module was designed. This will then be the basis of the code implementation.

### 2.2 Avoid writing test cases that can cause impact

Test cases that can cause impact exhibit the following problems:

* Test cases that manipulates data
* Test cases that require manual intervention

Developers should avoid creating test cases that manipulate real data on any environment. This can cause inconsistencies of data in test environments or in the worst case scenario, manipulate real business case data into stale data. We should avoid this at all costs.

Test cases should be automatically be re-runnable in the exact same way, something that cannot be achieved if manual intervention is required by the developer. The idea behind running a test case is that the scenario should be executed as if it would be if called from a business case. We don’t want to do anything outside the scenarioas this will compromise the integrity of the test case results.

### 2.3 Never skip tests

In builds, developers can always include the test case runs. In Maven, test cases are executed by default when the install, prepare and perform goals are called. It’s always a great thing to have the test cases executed, as this will ensure that the functional aspects of the system are working as expected. Some of the major test cases can be incorporated into this scheme and that is something that developers should not overlook.

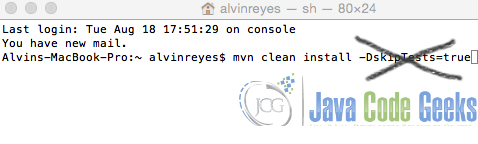
[](http://a5e2fba00d8bcb729d89839f.javacodegeeks.netdna-cdn.com/wp-content/uploads/2015/08/no_skip_test.jpg)

Figure 1.0 Don’t skip your test cases

### 2.4 Sensible Test Case names!

All developers would agree on this. The test cases names should be valid and meaningful.

#### 2.4.1 Create Order Implementation

createOrder()

|  |  |
| --- | --- |
| 1 | public Order createOrder(Order order) { |
| 2 | Order newOrder = new Order(); |
| 3 | newOrder.setOrderId(new Random().nextInt()); |
| 4 | newOrder.setSecurityCode("XYZ"); |
| 5 | newOrder.setOrderStatus("INITIATED"); |
| 6 | newOrder.setOrderDate(new Date()); |
| 7 | orderDao.createOrder(newOrder); |
| 8 | return newOrder; |
| 9 | } |

#### 2.4.2 Create Order Test Case

testSampleServiceCreateOrder()

|  |  |
| --- | --- |
| 01 |  |
| 02 | @Test |
| 03 | public void testSampleServiceCreateOrder() { |
| 04 | Order newOrder = new Order(); |
| 05 | newOrder.setSecurityCode("XYZ"); |
| 06 | newOrder.setDescription("Description"); |
| 07 | if (newOrder != null) { |
| 08 | assertThat(sampleService.createOrder(newOrder),instanceOf(Order.class)); |
| 09 | assertNotNull("Security isn't null", newOrder.getSecurityCode()); |
| 10 | assertNotNull("Description isn't not null",newOrder.getDescription()); |
| 11 | } |
| 12 | } |

### 2.5 Always aim to do one assertion for each test method

One assertion = One test method. That’s the rule of thumb. If that rule is not followed try to break down the redundant scenarios.

testAssertEqualsFalseWithMessage()

|  |  |
| --- | --- |
| 1 | @Test |
| 2 | public void testAssertEqualsFalseWithMessage() { | |
| 3 | ServiceObject newServiceObject = new ServiceObject(); | | |
| 4 | junitAssertEqualsServiceSample.postProcessing(serviceObject); | | | |
| 5 | assertEquals("Not the Same Object",newServiceObject,serviceObject); | | | | | |
| 6 | } | | | | |

### 2.6 Assertions, maximize it!

The JUnit Assert package has a lot of methods that can be used to do test cases. Combine this with a hamcrest and you get an extremely powerful api that uses pattern matching utilities.

import JUnit utilities that can be used

|  |  |  |
| --- | --- | --- |
| 1 | import static org.junit.Assert.\*; | |
| 2 | import static org.hamcrest.CoreMatchers.\*; | |
| 3 | import static org.hamcrest.BaseMatcher.\*; |

### 2.7 Annotations

Java 5 has introduced us to the concept of Annotations. JUnit 3 started to compliment this new JDK something that worked perfectly. The following are the basic and mostly used annotations.

* [@RunWith](http://junit.sourceforge.net/javadoc/org/junit/runner/RunWith.html) – Use to specify a runner object that the test case will be called from.
* [@Test](http://junit.sourceforge.net/javadoc/org/junit/Test.html) – use to tag a method as a test case.
* [@Before](http://junit.sourceforge.net/javadoc/org/junit/Before.html) – use to tag a method to be called before a test case. This usually being used to initialse data for the test case.
* [@After](http://junit.sourceforge.net/javadoc/org/junit/After.html) – use to tag a method to be called after a test case. This is usually used to do clean up processes.

There’s more useful annotations available and developers should maximise these.

### 2.8 Packaging convention

The package naming convention should always be directly the same to the one of the package of your implementation. This will make it more maintainable as it generally forms a pattern for developers once they need to update an existing test case.

### 2.9 Test code is separated from Production / release code

This is self-explanatory. You should always separate your test code and production code. Maven has a great way of separating this by introducing a package convention for test cases and implementation source files. I extremely suggest using Maven or Gradle as it generally setups everything for you.

[](http://a5e2fba00d8bcb729d89839f.javacodegeeks.netdna-cdn.com/wp-content/uploads/2015/08/packaging_conv.jpg)

Figure 2.0 Package convention

### 2.10 Make use of @Before, @After – Never construct the test case class

Annotations again! [@Before](http://junit.sourceforge.net/javadoc/org/junit/Before.html) and [@After](http://junit.sourceforge.net/javadoc/org/junit/After.html) are annotations that can be use to tag a method if you want it to be called upon initialization and destruction of test object respectively.

|  |  |
| --- | --- |
| 01 |  |
| 02 | @Before |
| 03 | public void setData(){ |
| 04 | this.totalNumberOfApplicants = 9; |
| 05 | listOfValidStrings.add("object\_1"); |
| 06 | listOfValidStrings.add("object\_2"); |
| 07 | listOfValidStrings.add("object\_3"); |
| 08 | } |
| 09 | @After // tearDown() |
| 10 | public void after() throws Exception { |
| 11 | dummyAccount = null; |
| 12 | assertNull(dummyAccount); |

|  |  |
| --- | --- |
| 13 | } |

### 2.11 Don’t just pass the test just for the sake of passing it!

Of course, everyone can just write ‘assert(true)’ just for the sake of passing a test case, but then what’s the point of doing it in the first place? The whole strategy of doing Junit test case is the fact that we want to ensure that the system is stable and scalable against any new extensions and requirements. It doesn’t make sense to just treat it as a way to manipulate SQM results.

|  |  |
| --- | --- |
| 1 | @Test |
| 2 | public void testGenerateAccount() { | |
| 3 | assert(true); // wow. don't do this. just don't | | | |
| 4 | } | | |

### 2.12 Do not write your own catch blocks that exist only to pass a test

It is unnecessary to write your own catch blocks that exist only to pass a test because the JUnit framework takes care of the situation for you. For example, suppose you are writing unit tests for the following method:

|  |  |  |
| --- | --- | --- |
| 1 | final class Foo{ | |
| 2 | int foo(int i) throws IOException; | | |
| 3 | } |

Here we have a method that accepts an integer and returns an integer and throws an IOException if it encounters an error. Further suppose that we expect the method to throw an IOException if a parameter is passed with the value nine. Below you can find the wrong way to write a unit test that confirms that the method behaves that way:

|  |  |
| --- | --- |
| 01 | // Don't do this - it's not necessary to write the try/catch! |
| 02 | @Test |
| 03 | public void foo\_nine(){ |
| 04 | boolean wasExceptionThrown = false; |
| 05 | try{ |
| 06 | new Foo().foo(9); |
| 07 | } |
| 08 | catch (final IOException e){ |
| 09 | wasExceptionThrown = true; |
| 10 | } |
| 11 | assertTrue(wasExceptionThrown); |
| 12 | } |

Instead of manually catching the expected exception, use the expected attribute on JUnit’s @Test annotation.

|  |  |
| --- | --- |
| 1 | // Do this instead |
| 2 | @Test(expected = IOException.class) |
| 3 | public void foo\_nine() throws Exception { |
| 4 | new Foo().foo(9); |
| 5 | } |

We declare that the test method throws Exception rather than IOException – see below for the reason. The JUnit framework will make sure that this test passes if and only if the foo method throws an IOException – there’s no need to write your own exception handling.

### 2.13 Do not write your own catch blocks that exist only to print a stack trace

As we’ve already seen, it is a best practice for unit tests not to write anything. After all, unit tests are written to be consumed by the JUnit framework and JUnit doesn’t care or monitor what gets printed. For example, suppose you are writing unit tests for the following method:

|  |  |
| --- | --- |
| 1 | final class Foo { |
| 2 | int foo(int i) throws IOException; | |

|  |  |
| --- | --- |
| 3 | } |

Here we have a method that accepts an integer and returns an integer and throws an IOException if it encounters an error. Here is the wrong way to write a unit test that confirms that the method returns three when passed seven:

|  |  |
| --- | --- |
| 01 | // Don't do this - it's not necessary to write the try/catch! |
| 02 | @Test |
| 03 | public void foo\_seven() { |
| 04 | try{ |
| 05 | assertEquals(3, new Foo().foo(7)); |
| 06 | } |
| 07 | catch (final IOException e){ |
| 08 | e.printStackTrace(); |
| 09 | } |
| 10 | } |

The method under test specifies that it can throw IOException, which is a checked exception. Therefore, the unit test won’t compile unless you catch the exception or declare that the test method can propagate the exception. The second alternative is much preferred because it results in shorter and more focused tests:

|  |  |  |
| --- | --- | --- |
| 1 | // Do this instead | |
| 2 | @Test |

|  |  |  |
| --- | --- | --- |
| 3 | public void foo\_seven() throws Exception { | |
| 4 | assertEquals(3, new Foo().foo(7)); |

|  |  |
| --- | --- |
| 5 | } |

We declare that the test method throws Exception rather than throws IOException – see below for the reason. The JUnit framework will make sure that this test fails if any exception occurs during the invocation of the method under test – there’s no need to write your own exception handling.

### 2.14 Avoid the Threads sa much as possible

We don’t want to spawn multiple threads on your test case unless it’s really necessary and avoid pausing it (Thread.sleep) as this will greatly impact the build time and execution. When a unit test uses Thread.sleep it does not reliably indicate a problem in the production code. For example, such a test can fail because it is run on a machine that is slower than usual. Aim for unit tests that fail if and only if the production code is broken. Rather than using Thread.sleep in a unit test, refactor the production code to allow the injection of a mock object that can simulate the success or failure of the potentially long-running operation that must normally be waited for.

### 2.15 Loggers

Use loggers to create an info comment on your test cases. This will make it easier to view runtime exceptions that can happen when running your test cases. It is advisable to use log4j or any extensions and implementations it has such as [SL4J](http://www.slf4j.org/)

### 2.16 Maven or Gradle – Build tools to automate

My personal favourite is Maven. It has everything needed to build, compile and run test cases for your system. It has an extensive plugin repository backed up by some of the best open source developers. We have long gone past manual ant builds, we now have better and improved build tools and mechanism. We should use them at our disposal.

* [Maven](http://maven.apache.org/)
* [Gradle](https://gradle.org/)

### 2.17 Test Case Coverage and report using surefire plugin

A surefire plugin is one way of creating an html page report of the test case coverage run. Highly advisable for developers to use this so that they can have a clear and concise metrics report of their individual or all their specific test case.

* [surefire plugin](http://maven.apache.org/surefire/maven-surefire-report-plugin/)
* [Java Docs plugin](https://maven.apache.org/plugins/maven-javadoc-plugin/)

### 2.18 The 80% test coverage rule

Rule of thumb says, test coverage must be at least 80%! Projects need to hit this targets as much as possible! There are certain cases specifically for legacy systems that haven’t done any test case ever since they were started (again a no no!). For this, make sure that for all the extensions made by the project to the system, should at least increase the coverage proportionally to the actual change.

### 2.19 Mock your data!

There’s a ton of mocking api available to be used. Mocking is basically a way to create a shallow proxy object that can be used on your test cases.

* [EasyMock](http://easymock.org/)
* [Mockito](http://mockito.org/)

### 2.20 Be creative with your test case!

There will be a lot of scenarios possible for your test cases and in some instances, you might be dealing with a service that needs to generate specific HTML templates, velocity, jasper or wsdl files. Developers need to be creative to account for these specific scenarios. It’s not just about testing your Java Service code and check if it will give you no exception. It should do what it supposed to do.

## 3. The General Tip: Keep tests small and fast

Executing every test for the entire system shouldn’t take hours. Indeed, developers will more consistently run tests that execute quickly. Without regularly running the full set of tests, it will be difficult to validate the entire system when changes are made. Errors will start to creep back in, and the benefits of unit testing will be lost. This means stress tests and load tests for single classes or small frameworks of classes shouldn’t be run as part of the unit test suite; they should be executed separately.

There’s a lot of resources available that identify best practices in creating JUnit Test case. It all comes really down to the discipline. Developers need to have that mindset: it’s not just all about the implementation, it is also about the proof of how stable, scalable and functionally working the code is.

# JUnit Integration Test Example

Posted by: [Alvin Reyes](http://examples.javacodegeeks.com/author/alvin-reyes/) in [junit](http://examples.javacodegeeks.com/category/core-java/junit/) August 28th, 2015

## 1. Introduction

Integration tests are test cases that test highly coupled external services. A great example of this is services in a SOA environment. In this scheme we will create services (or micro services) that are usually deployed on a different container and only exposed specific implementation of it’s own, consumed by a more sophisticated system.

By rule of thumb, we always need to separate this kind of tests from the internal service of the application. This separation, in JUnit Test case perspective, is called “categorising”, a pretty straight forward definition, we categorise a specific test case by creating a mark on them using Java Interface class. In this example, I’ll be giving a simple example of how we can write and categorise test cases by using the [@Category](http://junit.org/javadoc/latest/org/junit/experimental/categories/Category.html) annotation.

## 2. Source

The following codes are the source codes made for this example. Link to this project can be found at the end of this article.

### 2.1 Java Test Interface

We first need to mark our test cases. This is for us to categorise the test case that it’s an Integration tests.

IntegrationTest.java

|  |  |  |
| --- | --- | --- |
| 1 | package com.areyes1.jgc.junit.integ; | |
| 2 | public interface IntegrationTest {} |

### 2.2 Test Case

After that, we will use the interface class to mark our test case as an Integration test.

JUnitAssertExampleTests.java

package com.areyes1.jgc.junit.integ;

import org.junit.Before;

import org.junit.Test;

import org.junit.experimental.categories.Category;

import com.areyes1.jgc.junit.integ.JUnitAssertEqualsServiceExample;

import com.areyes1.jgc.junit.integ.ServiceObject;

import static org.junit.Assert.assertEquals;

@Category(IntegrationTest.class)

public class IntegrationTestSampleTest {

private JUnitAssertEqualsServiceExample junitAssertEqualsServiceSample;

private ServiceObject serviceObject;

@Before

public void setData() {

serviceObject = new ServiceObject();

junitAssertEqualsServiceSample = new JUnitAssertEqualsServiceExample();

junitAssertEqualsServiceSample.initiateMetaData(serviceObject);

}

@Test

public void testAssertEqualsFalse() {

// processed the item

ServiceObject newServiceObject = new ServiceObject();

junitAssertEqualsServiceSample.initiateMetaData(newServiceObject);

junitAssertEqualsServiceSample.processObject(serviceObject);

assertEquals(serviceObject,newServiceObject);

}

@Test

public void testAssertEquals() {

junitAssertEqualsServiceSample.processObject(serviceObject);

assertEquals(serviceObject,this.serviceObject);

}

@Test

public void testAssertEqualsWithMessage() {

junitAssertEqualsServiceSample.processObject(serviceObject);

assertEquals(

"Same Object",

serviceObject,serviceObject);

}

@Test

public void testAssertEqualsFalseWithMessage() {

ServiceObject newServiceObject = new ServiceObject();

junitAssertEqualsServiceSample.postProcessing(serviceObject);

assertEquals(

"Not the Same Object",

newServiceObject,serviceObject);

}

}

### 2.3 Maven Config

To ensure that this class will only be executed upon running the integration-test, we need to add the following configuration on our pom.xm

<build>

<plugins>

<plugin>

<artifactId>maven-failsafe-plugin</artifactId>

<version>2.12</version>

<dependencies>

<dependency>

<groupId>org.apache.maven.surefire</groupId>

<artifactId>surefire-junit47</artifactId>

<version>2.12</version>

</dependency>

</dependencies>

<configuration>

<groups>com.areyes1.jgc.junit.integ.IntegrationTest</groups>

</configuration>

<executions>

<execution>

<goals>

<goal>integration-test</goal>

</goals>

<configuration>

<includes>

<include>\*\*/\*.class</include>

</includes>

</configuration>

</execution>

</executions>

</plugin>

</plugins>

</build>

We added the configuration that all test cases, which are categorised under Integrate test, will be part of the integration-test goal.

### 2.4 Maven command

Run the following: mvn integration-test

## 3. Output

Here’s the output of our log file:

-------------------------------------------------------

T E S T S

-------------------------------------------------------

Running com.areyes1.jgc.junit.integ.IntegrationTestSampleTest

Tests run: 4, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.073 sec

Results:

Tests run: 4, Failures: 0, Errors: 0, Skipped: 0

[INFO] ------------------------------------------------------------------------

[INFO] BUILD SUCCESS

[INFO] ------------------------------------------------------------------------

[INFO] Total time: 1.843 s

[INFO] Finished at: 2015-08-27T18:54:16+08:00

[INFO] Final Memory: 9M/107M

[INFO] ------------------------------------------------------------------------

# “No JUnit Tests Found” Error Solution

Posted by: [Alvin Reyes](http://examples.javacodegeeks.com/author/alvin-reyes/) in [junit](http://examples.javacodegeeks.com/category/core-java/junit/) September 3rd, 2015

## 1. Introduction

One of the first steps that developers take in order to create Unit Test case is to ensure that their environment is ready. This means that all required libraries are setup properly and that the JUnit Test cases are correctly annotated. There might be a couple of technical hiccups like proper source folder configuration in your specific IDE, but regardless, this issues shouldn’t be a roadblock to begin with.

## 2. Issue: No Junit Tests Found

Developers might encounter issues along the way of setting up their test case. I myself have encountered a lot of issues before I even get my maven project building successfully. For this particular post, I will discuss how to resolve the “No JUnit Test Found” error.

## 3. Solution

### 3.1 Library setup

We need to ensure that our project has the correct dependencies or libraries. Legacy projects that do not use maven dependencies often need to import their libraries manually. This manual way of doing it is prone to inconsistent configurations and might lead to more erroneous setup. Convert your project to maven. Invest on converting your project to a full blown maven configured project.

Using [**maven**](http://maven.apache.org/) will really get some of those setup in tact and you’ll never have to worry about missing any dependencies again.

pom.xml

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>com.areyes1.jgc.itest</groupId>

<artifactId>junit-no-test-found</artifactId>

<version>0.0.1-SNAPSHOT</version>

<dependencies>

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.12</version>

<scope>test</scope>

</dependency>

</dependencies>

</project>

In Eclipse, the libraries will be represented as a maven dependency as shown below.

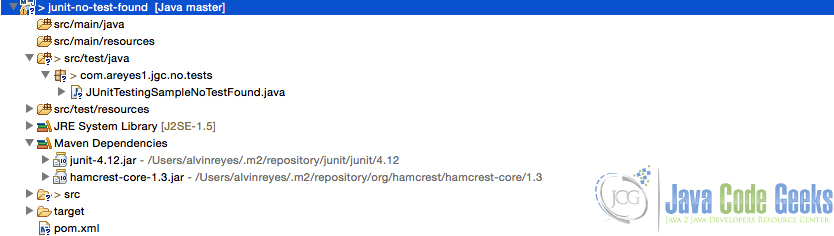
[](http://a5e2fba00d8bcb729d89839f.javacodegeeks.netdna-cdn.com/wp-content/uploads/2015/09/Screenshot-at-Sep-02-21-02-37.jpg)

Figure 1.0 JUnit Maven Dependency

### 3.2 [@RunWith](http://junit.sourceforge.net/javadoc/org/junit/runner/RunWith.html) annotation

By default this is implicitly called. When an annotated method ([@Test](http://junit.sourceforge.net/javadoc/org/junit/Test.html)) is detected, it automatically calls as default runner for that entire class. If the issue persists though, you can explicitly indicate it on your class, just like the one below:

JUnitTestingSampleNoTestFound.java

package com.areyes1.jgc.no.tests;

import static org.junit.Assert.assertFalse;

import org.junit.Before;

import org.junit.Test;

import org.junit.runner.RunWith;

import org.junit.runners.JUnit4;

@RunWith(JUnit4.class)

public class JUnitTestingSampleNoTestFound {

int totalNumberOfApplicants = 0;

int totalNumberOfAcceptableApplicants = 10;

@Test

public void testAssertFalseWithMessage() {

assertFalse(

"Is total number of applicants acceptable?",

(this.totalNumberOfApplicants != this.totalNumberOfAcceptableApplicants));

}

}

@Test

public void testInvoke() {

assertFalse(

"Is total number of applicants acceptable?",

(this.totalNumberOfApplicants != this.totalNumberOfAcceptableApplicants));

}

### 3.4 All else fail? Just clean up your workspace and make sure all configurations are set.

I’ve seen and heard many stories of how they encounter this error. Some suggests that the best solution is to restart your IDE. But I do think it’s just that. When you restart your IDE, the whole configuration will be refreshed automatically, and this might be the solution to that problem you’re encountering while running JUnit Test case!

# JUnit Test Order Example

Posted by: [Alvin Reyes](http://examples.javacodegeeks.com/author/alvin-reyes/) in [junit](http://examples.javacodegeeks.com/category/core-java/junit/) September 8th, 2015

## 1. Introduction

One of the rarely used features in JUnit is the usage of [@FixMethodOrder](http://junit.org/apidocs/index.html?org/junit/FixMethodOrder.html) annotation. This is primarily used to indicate an order of test method calls in a specific JUnit Test case class. This is actually not a recommended practice cause we want method calls to be independent and arbitrary in nature. Test cases method should not be dependent on each other except for integration test cases.

Of course, this feature wouldn’t be included on the latest JUnit Library if not for it’s positive usage. Here are some examples that this feature might come in handy.

* Test case to execute specific service method to satisfy a functional scenario. Bad practice but in the real world application, this can be the case.
* Order of test case will also reflect on the representation in the report. It would certainly make sense from a report perspective that all test cases are defined in a specific order.

## 2. The Source

Here is a bit of example I made to showcase how can we order a specific test case.

package com.areyes1.jgc.tests;

import org.junit.Assert;

import org.junit.FixMethodOrder;

import static org.hamcrest.CoreMatchers.\*;

import org.junit.Test;

import org.junit.runners.MethodSorters;

@FixMethodOrder(MethodSorters.DEFAULT)

public class JUnitOrderSampleServiceTests {

@Test

public void testC() {

System.out.println("C");

int numberCResult = 0;

for (int i=0;i<1000;i++) {

//complex loop

numberCResult++;

}

Assert.assertThat(numberCResult,isA(Integer.class));

}

@Test

public void testA() {

System.out.println("A");

int numberAResult = 0;

for (int i=0;i<10000;i++) {

//complex loop

numberAResult++;

}

Assert.assertThat(numberAResult,isA(Integer.class));

}

@Test

public void testD() {

System.out.println("D");

int numberDResult = 0;

for (int i=0;i<100000;i++) {

//complex loop

numberDResult++;

}

Assert.assertThat(numberDResult,isA(Integer.class));

}

@Test

public void testB() {

System.out.println("B");

int numberBResult = 0;

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

//complex loop

numberBResult++;

}

Assert.assertThat(numberBResult,isA(Integer.class));

}

}

It starts with the annotation [@FixMethodOrder](http://junit.org/apidocs/index.html?org/junit/FixMethodOrder.html). This annotation will marked the test case to run on a specific order given a sorter. There are 3 types of sorters provided by the JUnit Library, these are:

* [MethodSorters.DEFAULT](http://junit.org/apidocs/index.html?org/junit/FixMethodOrder.html) – By default, the test cases will run on the matter of it’s declaration in the class. We really don’t need to specify this since this will be the default sorter. According to the documentation, the default behaviour is deterministic – means that it may very well depend on a number of factors such as object creation or even placement of declaration. It is deterministic on the virtual machine level but never predictable.
* [MethodSorters.JVM](http://junit.org/apidocs/index.html?org/junit/FixMethodOrder.html) – This is arbitrary one, the order is not guaranteed as it will depend on the creation of the objects in the JVM.
* [MethodSorters.NAME\_ASCENDING](http://junit.org/apidocs/index.html?org/junit/FixMethodOrder.html) – This will look at the method names and sort it in ascending order (A-Z).

## 3. Results

Here are some of the results made from each of the orders as discussed.

### 3.1 Using [MethodSorters.DEFAULT](http://junit.org/apidocs/index.html?org/junit/FixMethodOrder.html)

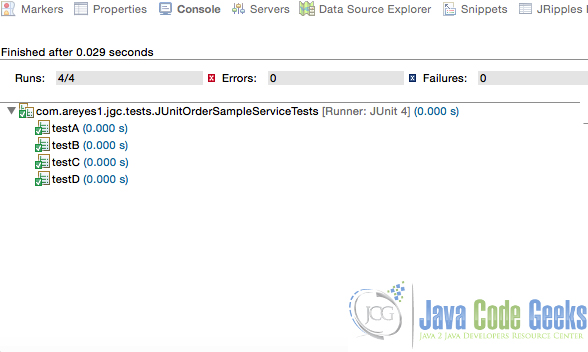
[](http://a5e2fba00d8bcb729d89839f.javacodegeeks.netdna-cdn.com/wp-content/uploads/2015/09/method_sort_default.jpg)

Figure 1.0 MethodSorters.DEFAULT result

In this run, it uses the names of the methods in an ascending order. We didn’t enforce the NAME\_ASCENDING sort in this method and it can vary in a few runs.

### 3.2 Using [MethodSorters.JVM](http://junit.org/apidocs/index.html?org/junit/FixMethodOrder.html)

[](http://a5e2fba00d8bcb729d89839f.javacodegeeks.netdna-cdn.com/wp-content/uploads/2015/09/method_sort_jvm.jpg)

Figure 2.0 MethodSorters.JVM result

In this run, it uses the sequence of it’s declaration on the Test case class. Indicating that the sequence was the actual creation of the methods in the stack.

### 3.3 Using [MethodSorters.NAME\_ASCENDING](http://junit.org/apidocs/index.html?org/junit/FixMethodOrder.html)

[](http://a5e2fba00d8bcb729d89839f.javacodegeeks.netdna-cdn.com/wp-content/uploads/2015/09/method_sort_name_asc.jpg)

Figure 3.0 MethodSorters.NAME\_ASCENDING result

In this run, its the same as the [MethodSorters.DEFAULT](http://junit.org/apidocs/index.html?org/junit/FixMethodOrder.html). In this however, we are enforcing this order. The default one is completely arbitrary in nature and may not be always the case.

# JUnit Mockito Example

Posted by: [Alvin Reyes](http://examples.javacodegeeks.com/author/alvin-reyes/) in [Mockito](http://examples.javacodegeeks.com/category/core-java/mockito/) September 14th, 2015

## 1. Introduction

Methods and ways to create Unit Test cases have evolve ever since it’s introduction. New tools and API´s are now available and they provide a more advanced scheme on creating and executing JUnit Test cases. Services have become more incubated, so creating integration test cases has been a daunting task for any developer. The introduction of this new development approach made revolutionary changes in Unit testing as well and a lot of testing framework came and rise the level at the playing field. With this, the need to create mocking objects to mimic Java objects in their runtime has never been more important, especially on critical enterprise software.

In this post, I’ll be showing one of the most widely used and popular JUnit Testing Mocking framework – [Mockito](http://mockito.org/).

## 2. Mockito Framework

Mockito is one of the widely used testing API for Java. Tons of examples are accepted by the massive Java community. Back in 2008, Dan North said that this was the future model of testing java applications. The popularity of Mockito and the overall Java projects in Github that use this API, clearly state that the prediction was true.

## 3. Eclipse Example

Let’s take a deep dive into an example. In this example we’ll create the following:

* Create a new Maven Project
* Define the dependencies we need. That is JUnit and Mockito
* Code some examples

### 3.1 Maven Project

Let’s first create a new Maven project. In your eclipse click on File > New Project > Maven Project. Tick on create a simple project fill up the group id, artefact id and hit Finish.

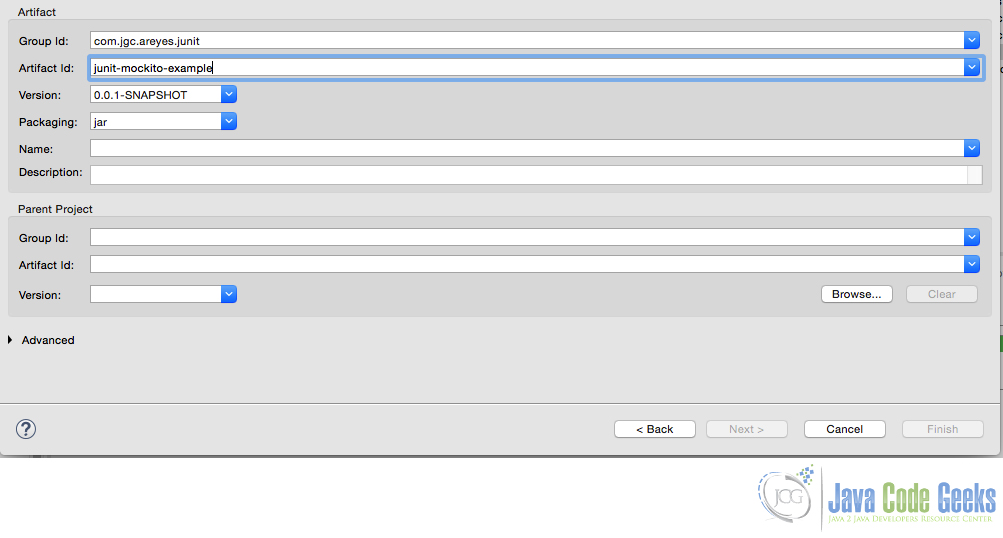
[](http://a5e2fba00d8bcb729d89839f.javacodegeeks.netdna-cdn.com/wp-content/uploads/2015/09/new_maven_project.jpg)

Figure 1.0 Maven project

### 3.2 pom.xml configuration

We then include the dependencies we need. This will download the libraries for our project.

pom.xml

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>com.jgc.areyes.junit</groupId>

<artifactId>junit-mockito-example</artifactId>

<version>0.0.1-SNAPSHOT</version>

<dependencies>

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.12</version>

</dependency>

<dependency>

<groupId>org.mockito</groupId>

<artifactId>mockito-all</artifactId>

<version>1.9.5</version>

</dependency>

</dependencies>

</project>

### 3.3 Test Case Example

JUnitServiceTestExample.java

package com.areyes1.jgc.mockito.e;

import static org.mockito.Mockito.\*;

import static org.junit.Assert.\*;

import java.util.Iterator;

import org.junit.Test;

import org.mockito.Matchers;

import org.mockito.Mockito;

/\*\*

\* The Class JUnitServiceTestExample.

\*/

public class JUnitServiceTestExample {

/\*\*

\* Test1.

\*/

@Test

public void testSimpleInt() {

// create mock

TestService test = Mockito.mock(TestService.class);

// define return value for method getUniqueId()

when(test.getUniqueId()).thenReturn(43);

// use mock in test....

assertEquals(test.getUniqueId(), 43);

}

/\*\*

\* Test more than one return value.

\*/

// Demonstrates the return of multiple values

@Test

public void testMoreThanOneReturnValue() {

Iterator i = mock(Iterator.class);

when(i.next()).thenReturn("Mockito").thenReturn("is neat!!");

String result = i.next() + " " + i.next();

assertEquals("Mockito is neat!!", result);

}

/\*\*

\* Test return value dependent on method parameter.

\*/

@Test

public void testReturnValueDependentOnMethodParameter() {

Comparable c = mock(Comparable.class);

when(c.compareTo("Mockito")).thenReturn(1);

when(c.compareTo("Eclipse")).thenReturn(2);

// assert

assertEquals(1, c.compareTo("Mockito"));

}

/\*\*

\* Test return value in dependent on method parameter.

\*/

@Test

public void testReturnValueInDependentOnMethodParameter() {

Comparable c = mock(Comparable.class);

when(c.compareTo(anyInt())).thenReturn(-1);

assertEquals(-1, c.compareTo(9));

}

@Test

public void testVerify() {

// create and configure mock

TestService test = Mockito.mock(TestService.class);

when(test.getUniqueId()).thenReturn(43);

// call method testing on the mock with parameter 12

test.testing(12);

test.getUniqueId();

test.getUniqueId();

test.someMethod("Hello World");

test.someMethod("called at least once");

test.someMethod("called at least twice");

test.someMethod("called five times");

test.someMethod("called at most 3 times");

// now check if method testing was called with the parameter 12

verify(test).testing(Matchers.eq(12));

// was the method called twice?

verify(test, times(2)).getUniqueId();

// other alternatives for verifiying the number of method calls for a

// method

verify(test, never()).someMethod("never called");

verify(test, atLeastOnce()).someMethod("called at least once");

// Will all fail because we didn't met the conditions.

verify(test, atLeast(2)).someMethod("called at least twice");

verify(test, times(5)).someMethod("called five times");

verify(test, atMost(3)).someMethod("called at most 3 times");

}

}

The example above showcases the different unique usage of [Mockito](http://mockito.org/). Aside from just mocking objects, it also perfectly compliments what we call “behaviour driven” test cases. This means that the test case is aimed at testing the behaviour or any method calls within the services aside from the output itself.

Let’s go over each method:

* testSimpleInt – the test case creates a mock class and calls the method. It enforces the method to use 43 as it’s return. This is then tested via an [assertEquals](http://junit.sourceforge.net/javadoc/org/junit/Assert.html) method as shown.
* testMoreThanOneReturnValue – the cases mocked an iterator class and sets a new value for the first record. The example shown concatenates two new strings on the 1st element. This record is then tested via an [assertEquals](http://junit.sourceforge.net/javadoc/org/junit/Assert.html) method.
* testReturnValueInDependentOnMethodParameter – The test case shows how we can dynamically use other results even in our comparison logic. In this example, we forced the comparisons to return values that are then tested via an [assertEquals](http://junit.sourceforge.net/javadoc/org/junit/Assert.html) method.
* testVerify – the test case showcases how we can test the behaviour of a method within the class. It tests how many calls were made to the method and if there are any changes to the return types. This is a powerful feature because not only allows developers to test results, but also the behaviour of a specific service can be tested.

Mockito has redefined the creation of test cases. Almost every project globally uses the API. It´s not just about mocking objects and classes but its also that it has created a venue for developers to develop more concrete, bulletproof test cases that ensure the stability of the software.

## 4 Output

Running the test case above will give the output below.

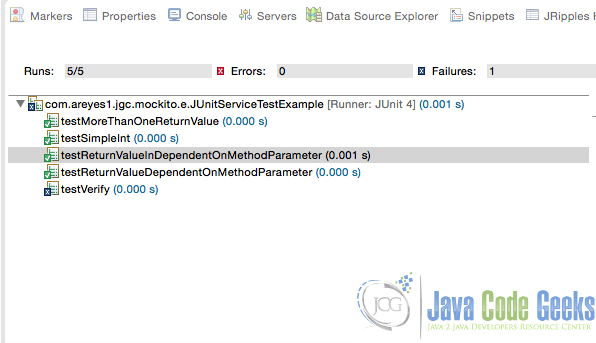
[](http://a5e2fba00d8bcb729d89839f.javacodegeeks.netdna-cdn.com/wp-content/uploads/2015/09/output_mockito.jpg)

Figure 2.0 Mockito test cases output

--------------------------

# JUnit Categories Example

Posted by: [Alvin Reyes](http://examples.javacodegeeks.com/author/alvin-reyes/) in [junit](http://examples.javacodegeeks.com/category/core-java/junit/) September 23rd, 2015

## 1. Introduction

JUnit has an awesome feature of organizing group of test cases called Categorizing. It can help developers differentiate test cases from one another. In this post, I’ll showcase how easy it is to categorize unit tests by [@Category](http://junit.org/apidocs/org/junit/experimental/categories/Categories.html).

## 2. Maven Project and Configuration

pom.xml

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>com.jgc.areyes.junit</groupId>

<artifactId>junit-categories-example</artifactId>

<version>0.0.1-SNAPSHOT</version>

<build>

<plugins>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-surefire-plugin</artifactId>

<version>2.17</version>

<dependencies>

<dependency>

<groupId>org.apache.maven.surefire</groupId>

<artifactId>surefire-junit47</artifactId>

<version>2.17</version>

</dependency>

</dependencies>

</plugin>

</plugins>

</build>

<dependencies>

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.12</version>

<scope>test</scope>

</dependency>

</dependencies>

</project>

## 3. Source code sample

Define the interfaces first. In order for us to group test cases, we need to create a unifier/union on them. We use interface class to tag a specific class or method to a group. Here are the interfaces that we will use in this example.

Interfaces

public interface FunctionalGroupTests1 {}

public interface FunctionalGroupTests2 {}

public interface IntegrationTests {}

public interface SanityTests {}

We then use those interfaces on our test cases. This will differentiate the test case for our own test purposes. In the example below, we tag tests method by category using the [@Category](http://junit.org/apidocs/org/junit/experimental/categories/Categories.html) annotation

*JUnitTestCategoryExample.java*

package com.areyes.junit.svc;

import static org.hamcrest.CoreMatchers.isA;

import static org.junit.Assert.\*;

import org.junit.Assert;

import org.junit.Test;

import org.junit.experimental.categories.Category;

import com.areyes.junit.cat.intf.FunctionalGroupTests1;

import com.areyes.junit.cat.intf.FunctionalGroupTests2;

import com.areyes.junit.cat.intf.IntegrationTests;

import com.areyes.junit.cat.intf.SanityTests;

public class JUnitTestCategoryExample {

@Test @Category(FunctionalGroupTests1.class)

public void testFunctionalTests1Test1() {

// You're test case here: Below is just an example.

int numberInLoop = 0;

for (int i=0;i<1000;i++) {

numberInLoop++;

}

System.out.println("FunctionalGroupTests1: testFunctionalTests1Test1");

Assert.assertThat(numberInLoop,isA(Integer.class));

}

@Test @Category(FunctionalGroupTests1.class)

public void testFunctionalTests1Test2() {

// You're test case here: Below is just an example.

int numberInLoop = 0;

for (int i=1000;i<4000;i++) {

numberInLoop++;

}

System.out.println("FunctionalGroupTests1: testFunctionalTests1Test2");

Assert.assertThat(numberInLoop,isA(Integer.class));

}

@Test @Category(FunctionalGroupTests2.class)

public void testFunctionalTests2Test1() {

// You're test case here: Below is just an example.

int numberInLoop = 0;

do{

numberInLoop++;

}while(numberInLoop != 1000);

System.out.println("FunctionalGroupTests2: testFunctionalTests2Test1");

Assert.assertThat(numberInLoop,isA(Integer.class));

}

@Test @Category(FunctionalGroupTests2.class)

public void testFunctionalTests2Test2() {

System.out.println("FunctionalGroupTests2: testFunctionalTests2Test2");

}

@Test @Category({IntegrationTests.class,FunctionalGroupTests1.class})

public void testIntegrationTestsTest1() {

System.out.println("IntegrationTests: testIntegrationTestsTest1");

}

@Test @Category(SanityTests.class)

public void testSanityTestsTest1() {

System.out.println("SanityTests: testSanityTestsTest1");

}

}

[Home](http://examples.javacodegeeks.com) » [Core Java](http://examples.javacodegeeks.com/category/core-java/) » [junit](http://examples.javacodegeeks.com/category/core-java/junit/) » JUnit Categories Example

### About Alvin Reyes

Alvin Reyes

Alvin has an Information Technology Degree from Mapua Institute of Technology. During his studies, he was already heavily involved in a number of small to large projects where he primarily contributes by doing programming, analysis design. After graduating, he continued to do side projects on Mobile, Desktop and Web Applications. He founded TechnoWebHub Software Labs, a software development company that focuses on Web and Mobile Development.

[http://a5e2fba00d8bcb729d89839f.javacodegeeks.netdna-cdn.com/wp-content/themes/jarida/images/author_site.png](http://www.alvinjayreyes.com/)[http://a5e2fba00d8bcb729d89839f.javacodegeeks.netdna-cdn.com/wp-content/themes/jarida/images/author_twitter.png](http://twitter.com/alvinjayreyes)[http://a5e2fba00d8bcb729d89839f.javacodegeeks.netdna-cdn.com/wp-content/themes/jarida/images/author_facebook.png](https://www.facebook.com/alvinjayreyes)[http://a5e2fba00d8bcb729d89839f.javacodegeeks.netdna-cdn.com/wp-content/themes/jarida/images/author_google.png](https://plus.google.com/+AlvinReyes02142015)[http://a5e2fba00d8bcb729d89839f.javacodegeeks.netdna-cdn.com/wp-content/themes/jarida/images/author_linkedin.png](http://ph.linkedin.com/in/alvinpreyes)

# JUnit Categories Example

Posted by: [Alvin Reyes](http://examples.javacodegeeks.com/author/alvin-reyes/) in [junit](http://examples.javacodegeeks.com/category/core-java/junit/) September 23rd, 2015

## 1. Introduction

JUnit has an awesome feature of organizing group of test cases called Categorizing. It can help developers differentiate test cases from one another. In this post, I’ll showcase how easy it is to categorize unit tests by [@Category](http://junit.org/apidocs/org/junit/experimental/categories/Categories.html).

## 2. Maven Project and Configuration

pom.xml

|  |  |
| --- | --- |
| 01 | <project xmlns="<http://maven.apache.org/POM/4.0.0>" xmlns:xsi="<http://www.w3.org/2001/XMLSchema-instance>" |
| 02 | xsi:schemaLocation="<http://maven.apache.org/POM/4.0.0> <http://maven.apache.org/xsd/maven-4.0.0.xsd>"> |

|  |  |
| --- | --- |
| 03 | <modelVersion>4.0.0</modelVersion> |
| 04 | <groupId>com.jgc.areyes.junit</groupId> | |

|  |  |  |
| --- | --- | --- |
| 05 | <artifactId>junit-categories-example</artifactId> | |
| 06 | <version>0.0.1-SNAPSHOT</version> |

|  |  |
| --- | --- |
| 07 | <build> |
| 08 | <plugins> | |

|  |  |
| --- | --- |
| 09 | <plugin> |
| 10 | <groupId>org.apache.maven.plugins</groupId> | |

|  |  |  |
| --- | --- | --- |
| 11 | <artifactId>maven-surefire-plugin</artifactId> | |
| 12 | <version>2.17</version> |

|  |  |
| --- | --- |
| 13 | <dependencies> |
| 14 | <dependency> | |

|  |  |  |
| --- | --- | --- |
| 15 | <groupId>org.apache.maven.surefire</groupId> | |
| 16 | <artifactId>surefire-junit47</artifactId> |

|  |  |  |
| --- | --- | --- |
| 17 | <version>2.17</version> | |
| 18 | </dependency> |

|  |  |  |
| --- | --- | --- |
| 19 | </dependencies> | |
| 20 | </plugin> |

|  |  |  |
| --- | --- | --- |
| 21 | </plugins> | |
| 22 | </build> |

|  |  |
| --- | --- |
| 23 | <dependencies> |
| 24 | <dependency> | |

|  |  |
| --- | --- |
| 25 | <groupId>junit</groupId> |
| 26 | <artifactId>junit</artifactId> | |

|  |  |  |
| --- | --- | --- |
| 27 | <version>4.12</version> | |
| 28 | <scope>test</scope> |

|  |  |  |
| --- | --- | --- |
| 29 | </dependency> | |
| 30 | </dependencies> |

|  |  |
| --- | --- |
| 31 | </project> |

## 3. Source code sample

Define the interfaces first. In order for us to group test cases, we need to create a unifier/union on them. We use interface class to tag a specific class or method to a group. Here are the interfaces that we will use in this example.

Interfaces

|  |  |
| --- | --- |
| 1 | public interface FunctionalGroupTests1 {} |
| 2 | public interface FunctionalGroupTests2 {} |

|  |  |  |
| --- | --- | --- |
| 3 | public interface IntegrationTests {} | |
| 4 | public interface SanityTests {} |

We then use those interfaces on our test cases. This will differentiate the test case for our own test purposes. In the example below, we tag tests method by category using the [@Category](http://junit.org/apidocs/org/junit/experimental/categories/Categories.html) annotation

JUnitTestCategoryExample.java

|  |  |  |
| --- | --- | --- |
| 01 | package com.areyes.junit.svc; | |
| 02 |  |

|  |  |  |
| --- | --- | --- |
| 03 | import static org.hamcrest.CoreMatchers.isA; | |
| 04 | import static org.junit.Assert.\*; |

|  |  |
| --- | --- |
| 05 |  |
| 06 | import org.junit.Assert; | |

|  |  |
| --- | --- |
| 07 | import org.junit.Test; |
| 08 | import org.junit.experimental.categories.Category; | |

|  |  |
| --- | --- |
| 09 |  |
| 10 | import com.areyes.junit.cat.intf.FunctionalGroupTests1; | |

|  |  |  |
| --- | --- | --- |
| 11 | import com.areyes.junit.cat.intf.FunctionalGroupTests2; | |
| 12 | import com.areyes.junit.cat.intf.IntegrationTests; |

|  |  |  |
| --- | --- | --- |
| 13 | import com.areyes.junit.cat.intf.SanityTests; | |
| 14 |  |

|  |  |  |
| --- | --- | --- |
| 15 | public class JUnitTestCategoryExample { | |
| 16 |  |

|  |  |  |
| --- | --- | --- |
| 17 | @Test @Category(FunctionalGroupTests1.class) | |
| 18 | public void testFunctionalTests1Test1() { |

|  |  |  |
| --- | --- | --- |
| 19 | //  You're test case here: Below is just an example. | |
| 20 | int numberInLoop = 0; |

|  |  |  |
| --- | --- | --- |
| 21 | for (int i=0;i<1000;i++) { | |
| 22 | numberInLoop++; |

|  |  |  |
| --- | --- | --- |
| 23 |  | |
| 24 | } |

|  |  |  |
| --- | --- | --- |
| 25 | System.out.println("FunctionalGroupTests1: testFunctionalTests1Test1"); | |
| 26 | Assert.assertThat(numberInLoop,isA(Integer.class)); |

|  |  |  |
| --- | --- | --- |
| 27 | } | |
| 28 |  |

|  |  |  |
| --- | --- | --- |
| 29 | @Test @Category(FunctionalGroupTests1.class) | |
| 30 | public void testFunctionalTests1Test2() { |

|  |  |  |
| --- | --- | --- |
| 31 | //  You're test case here: Below is just an example. | |
| 32 | int numberInLoop = 0; |

|  |  |  |
| --- | --- | --- |
| 33 | for (int i=1000;i<4000;i++) { | |
| 34 | numberInLoop++; |

|  |  |  |
| --- | --- | --- |
| 35 |  | |
| 36 | } |

|  |  |  |
| --- | --- | --- |
| 37 | System.out.println("FunctionalGroupTests1: testFunctionalTests1Test2"); | |
| 38 | Assert.assertThat(numberInLoop,isA(Integer.class)); |

|  |  |  |
| --- | --- | --- |
| 39 | } | |
| 40 |  |

|  |  |  |
| --- | --- | --- |
| 41 | @Test @Category(FunctionalGroupTests2.class) | |
| 42 | public void testFunctionalTests2Test1() { |

|  |  |  |
| --- | --- | --- |
| 43 | //  You're test case here: Below is just an example. | |
| 44 | int numberInLoop = 0; |

|  |  |
| --- | --- |
| 45 | do{ |
| 46 | numberInLoop++; | |

|  |  |
| --- | --- |
| 47 | }while(numberInLoop != 1000); |
| 48 | System.out.println("FunctionalGroupTests2: testFunctionalTests2Test1"); | |

|  |  |  |
| --- | --- | --- |
| 49 | Assert.assertThat(numberInLoop,isA(Integer.class)); | |
| 50 | } |

|  |  |
| --- | --- |
| 51 |  |
| 52 | @Test @Category(FunctionalGroupTests2.class) | |

|  |  |
| --- | --- |
| 53 | public void testFunctionalTests2Test2() { |
| 54 | System.out.println("FunctionalGroupTests2: testFunctionalTests2Test2"); | |

|  |  |  |
| --- | --- | --- |
| 55 | } | |
| 56 |  |

|  |  |  |
| --- | --- | --- |
| 57 | @Test @Category({IntegrationTests.class,FunctionalGroupTests1.class}) | |
| 58 | public void testIntegrationTestsTest1() { |

|  |  |  |
| --- | --- | --- |
| 59 | System.out.println("IntegrationTests: testIntegrationTestsTest1"); | |
| 60 | } |

|  |  |
| --- | --- |
| 61 |  |
| 62 | @Test @Category(SanityTests.class) | |

|  |  |
| --- | --- |
| 63 | public void testSanityTestsTest1() { |
| 64 | System.out.println("SanityTests: testSanityTestsTest1"); | |

|  |  |  |
| --- | --- | --- |
| 65 | } | |
| 66 | } |

## 4. Running our example

### 4.1 Running Tests by Category

We can run specific test case category by running the commands in Maven below. mvn test -Dgroups="com.areyes.junit.cat.intf.FunctionalGroupTests1, com.areyes.junit.cat.intf.FunctionalGroupTests2"  
mvn test -Dgroups="com.areyes.junit.cat.intf.IntegrationTests, com.areyes.junit.cat.intf.SanityTests"

### 4.2 Running tests by Category Profile

Alternatively, we can run tests by profile. We need to update our pom.xml and add a new profiles. We will then use these profiles and tag the categories we created to each as shown below.

pom.xml

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>com.jgc.areyes.junit</groupId>

<artifactId>junit-categories-example</artifactId>

<version>0.0.1-SNAPSHOT</version>

<build>

<plugins>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-surefire-plugin</artifactId>

<version>2.17</version>

<dependencies>

<dependency>

<groupId>org.apache.maven.surefire</groupId>

<artifactId>surefire-junit47</artifactId>

<version>2.17</version>

</dependency>

</dependencies>

<configuration>

<groups>${testcase.groups}</groups>

<excludes>

<exclude>${exclude.tests}</exclude>

</excludes>

<includes>

<include>${include.tests}</include>

</includes>

</configuration>

</plugin>

</plugins>

</build>

<profiles>

<profile>

<id>sanityTests</id>

<properties>

<testcase.groups>com.areyes.junit.svc.SanityTests</testcase.groups>

</properties>

</profile>

<profile>

<id>functionalGroupTests1</id>

<properties>

<testcase.groups>com.areyes.junit.svc.FunctionalGroupTests1</testcase.groups>

</properties>

</profile>

<profile>

<id>functionalGroupTests2</id>

<properties>

<testcase.groups>com.areyes.junit.svc.FunctionalGroupTests2</testcase.groups>

</properties>

</profile>

<profile>

<id>integrationTests</id>

<properties>

<testcase.groups>com.areyes.junit.svc.IntegrationTests</testcase.groups>

</properties>

</profile>

</profiles>

<dependencies>

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.12</version>

<scope>test</scope>

</dependency>

</dependencies>

</project>

Run them by using the following maven command: mvn test -pfunctionalGroupTests1

-----------------------------------------