**JUnit – Mockito**

**JUnit**

## **What is JUnit ?**

JUnit is a unit testing framework for the Java Programming Language. It is important in the test driven development, and is one of a family of unit testing frameworks collectively known as xUnit.

JUnit promotes the idea of "first testing then coding", which emphasis on setting up the test data for a piece of code which can be tested first and then can be implemented . This approach is like "test a little, code a little, test a little, code a little..." which increases programmer productivity and stability of program code that reduces programmer stress and the time spent on debugging.

## **Features**

* JUnit is an open source framework which is used for writing & running tests.
* Provides Annotation to identify the test methods.
* Provides Assertions for testing expected results.
* Provides Test runners for running tests.
* JUnit tests allow you to write code faster which increasing quality
* JUnit is elegantly simple. It is less complex & takes less time.
* JUnit tests can be run automatically and they check their own results and provide immediate feedback. There's no need to manually comb through a report of test results.
* JUnit tests can be organized into test suites containing test cases and even other test suites.
* Junit shows test progress in a bar that is green if test is going fine and it turns red when a test fails.

## **What is a Unit Test Case ?**

A Unit Test Case is a part of code which ensures that the another part of code (method) works as expected. To achieve those desired results quickly, test framework is required .JUnit is perfect unit test framework for java programming language.

A formal written unit test case is characterized by a known input and by an expected output, which is worked out before the test is executed. The known input should test a precondition and the expected output should test a postcondition.

There must be at least two unit test cases for each requirement: one positive test and one negative test. If a requirement has sub-requirements, each sub-requirement must have at least two test cases as positive and negative.

# **Basic Usage**

Now we'll show you a step by step process to get a kick start in Junit using a basic example.

## **Create a Class**

* Create a java class to be tested say MessageUtil.java in **C:\ > JUNIT\_WORKSPACE**

/\*

\* This class prints the given message on console.

\*/

public class MessageUtil {

private String message;

//Constructor

//@param message to be printed

public MessageUtil(String message){

this.message = message;

}

// prints the message

public String printMessage(){

System.out.println(message);

return message;

}

}

## **Create Test Case Class**

* Create a java test class say TestJunit.java.
* Add a test method testPrintMessage() to your test class.
* Add an Annotaion @Test to method testPrintMessage().
* Implement the test condition and check the condition using assertEquals API of Junit.

Create a java class file name TestJunit.java in **C:\ > JUNIT\_WORKSPACE**

import org.junit.Test;

import static org.junit.Assert.assertEquals;

public class TestJunit {

String message = "Hello World";

MessageUtil messageUtil = new MessageUtil(message);

@Test

public void testPrintMessage() {

assertEquals(message,messageUtil.printMessage());

}

}

## **Create Test Runner Class**

* Create a TestRunner java class.
* Use runClasses method of JUnitCore class of JUnit to run test case of above created test class
* Get the result of test cases run in Result Object
* Get failure(s) using getFailures() methods of Result object
* Get Success result using wasSuccessful() methods of Result object

Create a java class file name TestRunner.java in **C:\ > JUNIT\_WORKSPACE**to execute Test case(s)

import org.junit.runner.JUnitCore;

import org.junit.runner.Result;

import org.junit.runner.notification.Failure;

public class TestRunner {

public static void main(String[] args) {

Result result = JUnitCore.runClasses(TestJunit.class);

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

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

}

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

}

}

Compile the MessageUtil, Test case and Test Runner classes using javac

C:\JUNIT\_WORKSPACE>javac MessageUtil.java TestJunit.java TestRunner.java

Now run the Test Runner which will run test case defined in provided Test Case class.

C:\JUNIT\_WORKSPACE>java TestRunner

Verify the output.

Hello World

true

Now update TestJunit in **C:\ > JUNIT\_WORKSPACE** so that test fails. Change the message string.

import org.junit.Test;

import static org.junit.Assert.assertEquals;

public class TestJunit {

String message = "Hello World";

MessageUtil messageUtil = new MessageUtil(message);

@Test

public void testPrintMessage() {

message = "New Word";

assertEquals(message,messageUtil.printMessage());

}

}

Let's keep rest of the classes as is, and try to run same Test Runner

import org.junit.runner.JUnitCore;

import org.junit.runner.Result;

import org.junit.runner.notification.Failure;

public class TestRunner {

public static void main(String[] args) {

Result result = JUnitCore.runClasses(TestJunit.class);

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

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

}

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

}

}

Now run the Test Runner which will run test case defined in provided Test Case class.

C:\JUNIT\_WORKSPACE>java TestRunner

Verify the output.

Hello World

testPrintMessage(TestJunit): expected:<[New Wor]d> but was:<[Hello Worl]d>

false

# **JUnit – API**

## **Important API's of JUnit**

The most important package in JUnit is **junit.framework** which contain all the core classes. Some of the important class are

|  |  |  |
| --- | --- | --- |
| **Serial No** | **Class Name** | **Functionality** |
| 1 | Assert | A set of assert methods. |
| 2 | TestCase | A test case defines the fixture to run multiple tests. |
| 3 | TestResult | A TestResult collects the results of executing a test case. |
| 4 | TestSuite | A TestSuite is a Composite of Tests. |

## **Assert Class**

Following is the declaration for **org.junit.Assert** class:

public class Assert extends java.lang.Object

This class provides a set of assertion methods useful for writing tests. Only failed assertions are recorded. Some of the important methods of **Assert** class are:

|  |  |
| --- | --- |
| **S.N.** | **Methods & Description** |
| 1 | **void assertEquals(boolean expected, boolean actual)**  Check that two primitives/Objects are equal |
| 2 | **void assertFalse(boolean condition)**  Check that a condition is false |
| 3 | **void assertNotNull(Object object)**  Check that an object isn't null. |
| 4 | **void assertNull(Object object)**  Check that an object is null |
| 5 | **void assertTrue(boolean condition)**  Check that a condition is true. |
| 6 | **void fail()**  Fails a test with no message. |

Let's try to cover few of the above mentioned methods in an example. Create a java class file name TestJunit1.java in **C:\ > JUNIT\_WORKSPACE**

import org.junit.Test;

import static org.junit.Assert.\*;

public class TestJunit1 {

@Test

public void testAdd() {

//test data

int num= 5;

String temp= null;

String str= "Junit is working fine";

//check for equality

assertEquals("Junit is working fine", str);

//check for false condition

assertFalse(num > 6);

//check for not null value

assertNotNull(str);

}

}

Next, let's create a java class file name TestRunner1.java in **C:\ > JUNIT\_WORKSPACE** to execute Test case(s)

import org.junit.runner.JUnitCore;

import org.junit.runner.Result;

import org.junit.runner.notification.Failure;

public class TestRunner1 {

public static void main(String[] args) {

Result result = JUnitCore.runClasses(TestJunit1.class);

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

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

}

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

}

}

Compile the Test case and Test Runner classes using javac

C:\JUNIT\_WORKSPACE>javac TestJunit1.java TestRunner1.java

Now run the Test Runner which will run test case defined in provided Test Case class.

C:\JUNIT\_WORKSPACE>java TestRunner1

Verify the output.

true

## **TestCase Class**

Following is the declaration for **org.junit.TestCaset** class:

public abstract class TestCase extends Assert implements Test

A test case defines the fixture to run multiple tests. Some of the important methods of **TestCase** class are

|  |  |
| --- | --- |
| **S.N.** | **Methods & Description** |
| 1 | **int countTestCases()**  Counts the number of test cases executed by run(TestResult result). |
| 2 | **TestResult createResult()**  Creates a default TestResult object. |
| 3 | **String getName()**  Gets the name of a TestCase. |
| 4 | **TestResult run()**  A convenience method to run this test, collecting the results with a default TestResult object. |
| 5 | **void run(TestResult result)**  Runs the test case and collects the results in TestResult. |
| 6 | **void setName(String name)**  Sets the name of a TestCase. |
| 7 | **void setUp()**  Sets up the fixture, for example, open a network connection. |
| 8 | **void tearDown()**  Tears down the fixture, for example, close a network connection. |
| 9 | **String toString()**  Returns a string representation of the test case. |

Let's try to cover few of the above mentioned methods in an example. Create a java class file name TestJunit2.java in **C:\ > JUNIT\_WORKSPACE**

import junit.framework.TestCase;

import org.junit.Before;

import org.junit.Test;

public class TestJunit2 extends TestCase {

protected double fValue1;

protected double fValue2;

@Before

public void setUp() {

fValue1= 2.0;

fValue2= 3.0;

}

@Test

public void testAdd() {

//count the number of test cases

System.out.println("No of Test Case = "+ this.countTestCases());

//test getName

String name= this.getName();

System.out.println("Test Case Name = "+ name);

//test setName

this.setName("testNewAdd");

String newName= this.getName();

System.out.println("Updated Test Case Name = "+ newName);

}

//tearDown used to close the connection or clean up activities

public void tearDown( ) {

}

}

Next, let's create a java class file name TestRunner2.java in **C:\ > JUNIT\_WORKSPACE** to execute Test case(s)

import org.junit.runner.JUnitCore;

import org.junit.runner.Result;

import org.junit.runner.notification.Failure;

public class TestRunner2 {

public static void main(String[] args) {

Result result = JUnitCore.runClasses(TestJunit2.class);

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

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

}

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

}

}

Compile the Test case and Test Runner classes using javac

C:\JUNIT\_WORKSPACE>javac TestJunit2.java TestRunner2.java

Now run the Test Runner which will run test case defined in provided Test Case class.

C:\JUNIT\_WORKSPACE>java TestRunner2

Verify the output.

No of Test Case = 1

Test Case Name = testAdd

Updated Test Case Name = testNewAdd

true

## **TestResult Class**

Following is the declaration for **org.junit.TestResult** class:

public class TestResult extends Object

A TestResult collects the results of executing a test case. It is an instance of the Collecting Parameter pattern. The test framework distinguishes between failures and errors. A failure is anticipated and checked for with assertions. Errors are unanticipated problems like an ArrayIndexOutOfBoundsException. Some of the important methods of **TestResult** class are

|  |  |
| --- | --- |
| **S.N.** | **Methods & Description** |
| 1 | **void addError(Test test, Throwable t)**  Adds an error to the list of errors. |
| 2 | **void addFailure(Test test, AssertionFailedError t)**  Adds a failure to the list of failures. |
| 3 | **void endTest(Test test)**  Informs the result that a test was completed. |
| 4 | **int errorCount()**  Gets the number of detected errors. |
| 5 | **Enumeration<TestFailure> errors()**  Returns an Enumeration for the errors. |
| 6 | **int failureCount()**  Gets the number of detected failures. |
| 7 | **void run(TestCase test)**  Runs a TestCase. |
| 8 | **int int runCount()**  Gets the number of run tests. |
| 9 | **void startTest(Test test)**  Informs the result that a test will be started. |
| 10 | **void stop()**  Marks that the test run should stop. |

Create a java class file name TestJunit3.java in **C:\ > JUNIT\_WORKSPACE**

import org.junit.Test;

import junit.framework.AssertionFailedError;

import junit.framework.TestResult;

public class TestJunit3 extends TestResult {

// add the error

public synchronized void addError(Test test, Throwable t) {

super.addError((junit.framework.Test) test, t);

}

// add the failure

public synchronized void addFailure(Test test, AssertionFailedError t) {

super.addFailure((junit.framework.Test) test, t);

}

@Test

public void testAdd() {

// add any test

}

// Marks that the test run should stop.

public synchronized void stop() {

//stop the test here

}

}

Next, let's create a java class file name TestRunner3.java in **C:\ > JUNIT\_WORKSPACE** to execute Test case(s)

import org.junit.runner.JUnitCore;

import org.junit.runner.Result;

import org.junit.runner.notification.Failure;

public class TestRunner3 {

public static void main(String[] args) {

Result result = JUnitCore.runClasses(TestJunit3.class);

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

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

}

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

}

}

Compile the Test case and Test Runner classes using javac

C:\JUNIT\_WORKSPACE>javac TestJunit3.java TestRunner3.java

Now run the Test Runner which will run test case defined in provided Test Case class.

C:\JUNIT\_WORKSPACE>java TestRunner3

Verify the output.

true

## **TestSuite Class**

Following is the declaration for **org.junit.TestSuite** class:

public class TestSuite extends Object implements Test

A TestSuite is a Composite of Tests. It runs a collection of test cases. Some of the important methods of **TestSuite** class are

|  |  |
| --- | --- |
| **S.N.** | **Methods & Description** |
| 1 | **void addTest(Test test)**  Adds a test to the suite. |
| 2 | **void addTestSuite(Class<? extends TestCase> testClass)**  Adds the tests from the given class to the suite. |
| 3 | **int countTestCases()**  Counts the number of test cases that will be run by this test. |
| 4 | **String getName()**  Returns the name of the suite. |
| 5 | **void run(TestResult result)**  Runs the tests and collects their result in a TestResult. |
| 6 | **void setName(String name)**  Sets the name of the suite. |
| 7 | **Test testAt(int index)**  Returns the test at the given index. |
| 8 | **int testCount()**  Returns the number of tests in this suite. |
| 9 | **static Test warning(String message)**  Returns a test which will fail and log a warning message. |

Create a java class file name JunitTestSuite.java in **C:\ > JUNIT\_WORKSPACE** to create Test suite

import junit.framework.\*;

public class JunitTestSuite {

public static void main(String[] a) {

// add the test's in the suite

TestSuite suite = new TestSuite(TestJunit1.class, TestJunit2.class, TestJunit3.class );

TestResult result = new TestResult();

suite.run(result);

System.out.println("Number of test cases = " + result.runCount());

}

}

Compile the Test suite classes using javac

C:\JUNIT\_WORKSPACE>javac JunitTestSuite.java

Now run the Test Suite.

C:\JUNIT\_WORKSPACE>java JunitTestSuite

Verify the output.

No of Test Case = 1

Test Case Name = testAdd

Updated Test Case Name = testNewAdd

Number of test cases = 3

# **JUnit 4 Tutorial 1 – Basic usage**

This tutorial introduces the basic annotation supported in Junit 4.

Java

import org.junit.\*;

import static org.junit.Assert.\*;

import java.util.\*;

public class JunitTest1 {

private Collection collection;

@BeforeClass

public static void oneTimeSetUp() {

// one-time initialization code

System.out.println("@BeforeClass - oneTimeSetUp");

}

@AfterClass

public static void oneTimeTearDown() {

// one-time cleanup code

System.out.println("@AfterClass - oneTimeTearDown");

}

@Before

public void setUp() {

collection = new ArrayList();

System.out.println("@Before - setUp");

}

@After

public void tearDown() {

collection.clear();

System.out.println("@After - tearDown");

}

@Test

public void testEmptyCollection() {

assertTrue(collection.isEmpty());

System.out.println("@Test - testEmptyCollection");

}

@Test

public void testOneItemCollection() {

collection.add("itemA");

assertEquals(1, collection.size());

System.out.println("@Test - testOneItemCollection");

}

}

Result

Bash

@BeforeClass - oneTimeSetUp

@Before - setUp

@Test - testEmptyCollection

@After - tearDown

@Before - setUp

@Test - testOneItemCollection

@After - tearDown

@AfterClass - oneTimeTearDown

# **JUnit 4 Tutorial 2 – Expected Exception Test**

It’s used to test the exception throw by the method.

Java

import org.junit.\*;

public class JunitTest2 {

@Test(expected = ArithmeticException.class)

public void divisionWithException() {

int i = 1/0;

}

}

In above example, the **divisionWithException()** method will throw an **ArithmeticException** Exception, since this is an expected exception, so the unit test will pass.

# **JUnit 4 Tutorial 3 – Ignore Test**

This “Ignored” means the method is not ready to test, the JUnit engine will just bypass this method.

Java

import org.junit.\*;

public class JunitTest3 {

@Ignore("Not Ready to Run")

@Test

public void divisionWithException() {

System.out.println("Method is not ready yet");

}

}

In above example, JUnit will not test the **divisionWithException()** method.

# **JUnit 4 Tutorial 4 – Time Test**

The “Time Test” means if an unit test takes longer than the specified number of milliseconds to run, the test will terminated and mark as failed.

import org.junit.\*;

public class JunitTest4 {

@Test(timeout = 1000)

public void infinity() {

while (true);

}

}

In above example, the **infinity()** method will not return, so the JUnit engine will mark it as failed and throw an exception

Bash

java.lang.Exception: test timed out after 1000 milliseconds

# **JUnit 4 Tutorial 5 – Suite Test**

By [mkyong](http://www.mkyong.com/author/mkyong/) | May 21, 2009 | Updated : August 30, 2012 | Viewed : 127,863 times

The “Suite Test” means bundle a few unit test cases and run it together. In Junit, both **@RunWith** and **@Suite** annotation are used to run the suite test.

The below example means both unit test **JunitTest1** and **JunitTest2** will run together after JunitTest5 is executed.

Java

import org.junit.runner.RunWith;

import org.junit.runners.Suite;

@RunWith(Suite.class)

@Suite.SuiteClasses({

JunitTest1.class,

JunitTest2.class

})

public class JunitTest5 {

}

Result

Bash

@BeforeClass - oneTimeSetUp

@Before - setUp

@Test - testEmptyCollection

@After - tearDown

@Before - setUp

@Test - testOneItemCollection

@After - tearDown

@AfterClass - oneTimeTearDown

P.S Result is from JunitTest1 and JunitTest2 unit test

**Mock (Mockito)**

## **Les doublures d'objets et les objets de type mock**

En POO, il existe plusieurs types d'objets, généralement appelés doublures, permettant de simuler le comportement d'un autre objet :

* dummy (fantôme, bouffon) : objets "vides" qui n'ont pas de fonctionnalités implémentées.
* stub (bouchon) : classes qui renvoient une valeur codée en dur à l'invocation d'une méthode
* fake (substitut, simulateur) : classes qui sont une implémentation partielle et qui, par exemple, renvoient toujours les mêmes réponses quels que soient les paramètres fournis
* spy (espion) : classe qui vérifie l'utilisation qui en est faite après l'exécution
* mock (simulacre) : classes qui agissent comme un stub et un spy

Le vocabulaire lié à ces types d'objets est assez confus dans la langue anglaise donc il l'est d'autant plus dans la langue française où l'on tente de le traduire. Ce chapitre va se concentrer essentiellement sur les objets de type mock.

Un objet de type doublure permet donc de simuler le comportement d'un autre objet concret de façon maitrisée.

L'emploi de doublures est largement utilisé pour les tests unitaires mais il peut aussi être mis en oeuvre lors des développements pour par exemple remplacer un objet qui n'est pas encore écrit.

L'utilisation des doublures permet aux tests unitaires de se concentrer sur les tests du code de la méthode qui correspond au System Under Test (SUT) sans avoir à se préoccuper des dépendances.

Les doublures ont pour rôle de simuler le comportement d'un objet permettant ainsi de réaliser les tests de l'objet de façon isolée et répétable.

Un objet de type mock permet de simuler le comportement d'un autre objet concret de façon maitrisée et de vérifier les invocations qui sont faites de cet objet.

Cette double fonctionnalité permet dans un test unitaire de faire des tests sur l'état (state test) et des tests sur le comportement (behavior test).

Les types d'objets mock

Il existe deux grands types d'objets mock :

* statique : ce sont des classes Java écrites ou générées avec un outil par le développeur
* dynamique : ils sont mis en oeuvre par un framework

Les objets mock peuvent être codés manuellement ou utiliser un framework qui va permettre de les générer dynamiquement. L'avantage des mocks dynamiques c'est qu'aucune classe implicite n'a besoin d'être écrite.

Les frameworks de mocking peuvent utiliser plusieurs solutions pour mettre en oeuvre des mocks dynamiques :

* proxy : un proxy est un objet qui est utilisé à la place d'un autre objet. Il est alors nécessaire de fournir ce proxy à l'objet qui l'utilise en utilisant un constructeur ou un setter. Ceci nécessite donc qu'un mécanisme d'injection de dépendance soit mis en oeuvre dans la classe à tester (EasyMock, ...)
* instrumentation : un classloader spécifique est utilisé pour dynamiquement charger une classe à la place d'une autre notamment en utilisant la classe java.lang.Instrument de Java 1.5 (jmockit)
* AOP : permet d'invoquer la méthode d'un mock à la place de celle d'une implémentation concrète sans avoir à mettre en oeuvre une interface ni à requérir un mécanisme d'injection de dépendances. L'invocation de la méthode est interceptée et remplacée par l'invocation de la méthode du mock. Ceci ne doit cependant pas être une excuse pour ne pas écrire du code qui mette en oeuvre une bonne conception car en plus d'être testable cela rend le code plus compréhensible, plus maintenable et plus évolutif. (jeasytest, amock, ...)

Avec l'utilisation de proxies, il est indispensable d'avoir un mécanisme d'injection de dépendances permettant de fournir l'implémentation à utiliser. Ceci permet dans le cas des tests unitaires de fournir un objet de type mock qui sera utilisé lors de l'exécution des tests à la place d'une vraie instance de classe dépendante.

Ce mécanisme d'injection de dépendances peut être fourni par un framework (exemple : Spring) ou implémenté manuellement mais dans tous les cas le code à tester doit fournir une solution pour le réaliser.

Il existe plusieurs frameworks de mocking en Java qui permettent de créer dynamiquement des objets de type mock.

### Exemple d'utilisation dans les tests unitaires

Dans une application, les classes ont généralement des dépendances entre elles. Ceci est particulièrement vrai dans les applications développées en couches (présentation, service, métier, accès aux données (DAO), ...).

L'idée lors de l'exécution d'un test unitaire est de tester la plus petite unité de code possible, soit la méthode et uniquement le code de la méthode. Cependant les classes utilisées dans le code de cette méthode font généralement appel à un ou plusieurs autres objets. Le but n'est pas de tester ces objets qui feront eux-mêmes l'objet de tests unitaires mais de tester le code de la méthode : le test unitaire doit concerner uniquement la méthode et ne pas tester les dépendances.

Il faut donc une solution pour s'assurer que les objets dépendants fournissent les réponses désirées à leur invocation. Cette solution repose sur les objets de type simulacre.

Cela suppose que si le code de la méthode fonctionne comme voulu (validé par des tests unitaires) et que les dépendances fonctionnent de même (validées par leurs tests unitaires) alors ils fonctionneront normalement ensembles.

Les classes dépendantes ne doivent pas être testées dans les tests unitaires de la classe. Elles doivent être considérées comme testées, sachant que des tests unitaires qui leur sont dédiés doivent exister. Certaines classes doivent aussi être considérées comme testées : c'est notamment le cas des classes du JRE.

Il est très important que les tests unitaires ne concernent que le code de la méthode en cours de test. Autrement, il est difficile de trouver un bug qui peut être dans un objet dépendant de niveau -N.

Il est alors nécessaire de simuler le fonctionnement des classes dépendantes.

Le but d'un objet Mock est de remplacer un autre objet en proposant de forcer les valeurs de retour de ses méthodes selon certains paramètres.

Ainsi l'invocation d'un objet de type mock garantit d'avoir les valeurs attendues selon les paramètres fournis.

### La mise en oeuvre des objets de type mock

Un des avantages à utiliser des objets mock, notamment dans les tests unitaires, est qu'ils forcent le code à être écrit ou adapté par des refactoring pour qu'il respecte une conception permettant de le rendre testable.

Généralement, un objet de type mock est une implémentation d'une interface qui se limite le plus souvent à renvoyer des valeurs déterminées en fonction des paramètres reçus. L'interface est parfaitement adaptée puisque l'objet simulé et l'objet mock doivent avoir le même contrat.

Un objet de type mock possède donc la même interface que l'objet qu'il doit simuler, ce qui permet d'utiliser le mock ou une implémentation concrète de façon transparente pour l'objet qui l'invoque.

Les objets mock simulent le comportement d'autres objets mais ils sont aussi capables de vérifier les invocations qui sont faites sur le mock : nombres d'invocations, paramètres fournis, ordre d'invocations, ...

La mise en oeuvre d'un objet de type mock dans les tests unitaires suit généralement plusieurs étapes :

* définir le comportement du mock : méthodes invoquées, paramètres fournis, valeurs de retour ou exception ...
* exécuter le test en invoquant la méthode à tester
* vérifier des résultats du test
* vérifier les invocations du ou des objets de type mock : nombre d'invocations, ordre d'invocations, ...

## **108.2. L'utilité des objets de type mock**

Les objets de type mock peuvent être utilisés dans différentes circonstances :

* renvoyer des résultats déterminés notamment dans des tests unitaires automatisés
* obtenir un état difficilement reproductible (erreur d'accès réseau, ...)
* éviter d'invoquer des ressources longues à répondre (accès à une base de données, ...)
* invoquer un composant qui n'existe encore pas
* ...

Les objets de type mock sont donc très intéressants pour simuler le comportement de composants invoqués de façon distante (exemple : EJB, services web, RMI, ...) et particulièrement pour tester les cas d'erreurs (problème de communication, défaillance du composant ou du serveur qui gère leur cycle de vie, ...).

## **2. Testing with test doubles**

### 2.3. Mock object generation

You can create these mock objects manually (via code) or use a mock framework to simulate these classes. Mock frameworks allow you to create mock objects at runtime and define their behavior.

The classical example for a mock object is a data provider. In production a real database is used, but for testing a mock object simulates the database and ensures that the test conditions are always the same.

These mock objects can be provided to the class which is tested. Therefore, the class to be tested should avoid any hard dependency on external data.

Mocking or mock frameworks allows testing the expected interaction with the mock object, e.g., you test which methods have been called on the mock object.

## **3. Mocking frameworks**

Mocking frameworks make the creation of mock objects as simple as possible. Popular mock frameworks are EasyMock, jMock and Mockito.

The following lists the links to these frameworks.

# jMock

http://jmock.org/

# EasyMock

http://easymock.org/

# Mockito

https://github.com/mockito/mockito

## **4. Introduction to Mockito for mocking objects**

[**Mockito**](http://site.mockito.org/) is a popular mock framework which can be used in conjunction with JUnit. Mockito allows you to create and configure mock objects. Using Mockito simplifies the development of tests for classes with external dependencies significantly.

### 6.1. Creating and configuring mock objects

Mockito supports the creation of mock objects with the static mock(). If you add a static import fororg.mockito.Mockito.\*;, you can access Mockitos methods like mock() directly. Static imports allows you to call static members, i.e., methods and fields of a class directly without specifying the class.

To configure which values are returned at a method call Mockito defines a fluent API. You can use the verify()method to ensure that a method was called.

The when(....).thenReturn(....) call chain can be used to specify a condition and a return value for this condition. If you specify more than one value, they are returned in the order of specification until the last one is used. Afterwards the last specified value is returned. Mocks can also return different values depending on arguments passed into a method. You also use methods like anyString or anyInt to define that independent of the input value a certain return value should be returned.

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

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

*@Test*

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

// create mock

MyClass test = Mockito.mock(MyClass.**class**);

// define return value for method getUniqueId()

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

// use mock in test....

assertEquals(test.getUniqueId(), 43);

}

// Demonstrates the return of multiple values

*@Test*

**public** **void** testMoreThanOneReturnValue() {

Iterator i= mock(Iterator.**class**);

when(i.next()).thenReturn("Mockito").thenReturn("rocks");

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

//assert

assertEquals("Mockito rocks", result);

}

// this test demonstrates how to return values based on the input

*@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"));

}

// this test demonstrates how to return values independent of the input value

*@Test*

**public** **void** testReturnValueInDependentOnMethodParameter() {

Comparable c= mock(Comparable.**class**);

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

//assert

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

}

// return a value based on the type of the provide parameter

*@Test*

**public** **void** testReturnValueInDependentOnMethodParameter() {

Comparable c= mock(Comparable.**class**);

when(c.compareTo(isA(Todo.**class**))).thenReturn(0);

//assert

Todo todo = **new** Todo(5);

assertEquals(todo ,c.compareTo(**new** Todo(1)));

}

The doReturn(...).when(...).methodCall call chain works similar but is useful for void methods. The doThrowvariant can be used for methods which return void to throw an exception. This usage is demonstrated by the following code snippet.

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

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

// this test demonstrates how use doThrow

*@Test(expected=IOException.class)*

**public** **void** testForIOException() {

// create an configure mock

OutputStream mockStream = mock(OutputStream.**class**);

doThrow(**new** IOException()).when(mockStream).close();

// use mock

OutputStreamWriter streamWriter= **new** OutputStreamWriter(mockStream);

streamWriter.close();

}

### 6.2. Verify the calls on the mock objects

Mockito keeps track of all the method calls and their parameters to the mock object. You can use the verify()method on the mock object to verify that the specified conditions are met, i.e., that a method has been called with certain parameters. This kind of testing is sometimes called behavior testing, because it does not check the result of a method call, but it checks that a method is called with the right parameters.

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

*@Test*

**public** **void** testVerify() {

// create and configure mock

MyClass test = Mockito.mock(MyClass.**class**);

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

// call method testing on the mock with parameter 12

test.testing(12);

test.getUniqueId();

test.getUniqueId();

// 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(mock, never()).someMethod("never called");

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

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

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

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

}

### 6.3. Spy

@Spy or the spy() method can be used to wrap a real object. Every call, unless specified otherwise, is delegated to the object.

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

// Lets mock a LinkedList

List list = **new** LinkedList();

List spy = spy(list);

//You have to use doReturn() for stubbing

doReturn("foo").when(spy).get(0);

// this would not work

// real method is called so spy.get(0)

// throws IndexOutOfBoundsException (list is still empty)

when(spy.get(0)).thenReturn("foo");

The verifyNoMoreInteractions() allows you to check that no other method was called.

### 6.4. Using @InjectMocks for dependency injection via Mockito

You also have the @InjectMocks annotation which tries to do constructor, method or field dependency injection based on the type. The following code is a slightly modified example from the Javadoc.

// Mockito can construct this class via constructor

**public** **class** ArticleManager {

ArticleManager(ArticleCalculator calculator, ArticleDatabase database) {

}

}

// Mockito can also perform method injection

**public** **class** ArticleManager {

ArticleManager() { }

**void** setDatabase(ArticleDatabase database) { }

**void** setCalculator(ArticleCalculator calculator) { }

}

// Mockito can also perform field injection

**public** **class** ArticleManager {

**private** ArticleDatabase database;

**private** ArticleCalculator calculator;

}

*@RunWith(MockitoJUnitRunner.class)*

**public** **class** ArticleManagerTest {

*@Mock* **private** ArticleCalculator calculator;

*@Mock* **private** ArticleDatabase database;

*@Spy* **private** UserProvider userProvider = **new** ConsumerUserProvider();

// creates instance of ArticleManager

// and performs constructor injection on it

*@InjectMocks* **private** ArticleManager manager;

*@Test* **public** **void** shouldDoSomething() {

// assume that ArticleManager has a method called initialize which calls a method

// addListener with an instance of ArticleListener

manager.initialize();

// validate that addListener was called

verify(database).addListener(any(ArticleListener.**class**));

}

}

For more information see the [**Javadoc of InjectMock**](http://docs.mockito.googlecode.com/hg/1.9.5/org/mockito/InjectMocks.html).

### 6.5. Creating mock objects based on annotations

Mockito also supports the creation of mock objects based on the @Mock

If you use annotations, you must initialize this mock objects. The MockitoRule allows this, it invokes the static method MockitoAnnotations.initMocks(this) to populate the annotated fields. Alternatively you can use@RunWith(MockitoJUnitRunner.class).

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

**public** **class** MockitoTest {

// assume there is a class MyDatabase

*@Mock*

MyDatabase databaseMock;

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

*@Test*

**public** **void** testQuery() {

// assume there is a class called ClassToTest

// which could be tested

ClassToTest t = **new** ClassToTest(databaseMock);

// call a method

**boolean** check = t.query("\* from t");

// test the return type

assertTrue(check);

// test that the query() method on the

// mock object was called

verify(databaseMock).query("\* from t");

}

}

### 6.6. Limitations

Mockito has certain limitations. It can not test the following constructs:

* final classes
* anonymous classes
* primitive types