JUnit

Jean-Marc Farinone

Maître de Conférences
Conservatoire National des Arts et Métiers
CNAM Paris (France)

Plan

- JUnit = ?, les tests
- Installer JUnit
- Une exemple d'illustration
- Syntaxe
- Exercice
- Bibliographie

Quand tester ?

- Construire les tests pendant (ou avant) le code (cf eXtreme Programming). Voir à `http://junit.sourceforge.net/doc/testinfected/testing.htm`
  - "During Development- When you need to add new functionality to the system, write the tests first. Then, you will be done developing when the test runs.
  - During Debugging- When someone discovers a defect in your code, first write a test that will succeed if the code is working. Then debug until the test succeeds."

JUnit : un framework

- Version 4.12 depuis le 4 décembre 2014. Voir à `http://search.maven.org/#search|gav|1|g:"junit" AND a:"junit"`
- JUnit est un framework pour écrire et exécuter des tests. Il fait partie de l'architecture de tests xUnit
- site de référence : `junit.org`
- JUnit propose :
  - des assertions qui vérifient les résultats à attendre du code développé
  - un environnement d'exécution de tests
- JUnit a été initialement écrit par Erich Gamma et Kent Beck
- Peut être utilisé pour faire des tests unitaires mais aussi pour des suites de tests
- JUnit version 4.x utilise les annotations (donc Java 1.5) ce qui n'est pas le cas avec les versions 3.x

Comment installer JUnit ?

- Télécharger `junit-4.XX.jar` et `hamcrest-core-1.3.jar` à partir du site `junit.org`
- Faire repérer `junit-4.XX.jar` (qui va amener les classes des paquetages `org.junit.*`) et `hamcrest-core-1.3.jar` (qui va amener les classes des paquetages `org.hamcrest.*`) par la variable `CLASSPATH` ou l'option `-cp (-classpath)` des commandes `java` et `javac`

Un exemple : `Calculator`, une classe à tester

```
package calc;

public class Calculator {
     private static int result; // Le "registre" de la calculette

    public void add(int n) { result = result + n; }

    public void substract(int n) {
        result = result - 1; // Bug : devrait être result = result - n
    }

    public void multiply(int n) {} // Non implémenté

    public void divide(int n) { result = result / n; }

    public void squareRoot(int n) {
        for (; ;) ; // Arg : une boucle infinie
    }

    public void square(int n) {
        result = n * n;
    }

    public void clear() { result = 0; }

    public int getResult() { return result; }
}
```

Une classe qui teste `Calculator` (1/2)

```
package junit4;

import calc.Calculator;
import org.junit.Before;
import org.junit.Ignore;
import org.junit.Test;
import static org.junit.Assert.*;

public class CalculatorTest {
  private static Calculator calculator =
new Calculator();

  @Before
  public void clearCalculator() {
    calculator.clear();
  }

  @Test
  public void add() {
    calculator.add(1);
    calculator.add(1);
    assertEquals(2, calculator.getResult());
  }
     // suite diapo suivante
```

- utilisation de la syntaxe Java 1.5
  - `import static`
  - annotation
- utilisation des classes et méthodes du framework JUnit
- utilisation du package `org.junit` (pour JUnit 4)

Une classe qui teste `Calculator` (2/2)

```
  @Test
  public void subtract() {
    calculator.add(10);
    calculator.substract(2);
    assertEquals(8, calculator.getResult());
  }

  @Test
  public void divide() {
    calculator.add(8);
    calculator.divide(2);
    assert calculator.getResult() == 5;
  }

  @Test(expected = ArithmeticException.class)
  public void divideByZero() {
    calculator.divide(0);
  }

  @Ignore("test à ignorer pour l'instant")
  @Test
  public void multiply() {
    calculator.add(10);
    calculator.multiply(10);
    assertEquals(100, calculator.getResult());
  }
}
```

- utilisation de `assertEquals(attendu, obtenu)` du framework JUnit
- Test des exceptions qui doivent être levées
- Remarque : on aurait pu (dû ?) commencer par écrire cette classe avant l'écriture complète de la classe `Calculator`

Méthodes `assertXXX()`

- Les diverses méthodes statiques `assertXXX()` de la classe `org.junit.Assert` du framework JUnit dont `assertEquals()` sont indiquées à : `http://junit.sourceforge.net/javadoc_40/org/junit/Assert.html`

Compilation, exécution et démonstration

- démonstration dans `...\Calculette`
- `compileAll.bat`

```
javac -cp cheminAbsoluOuRelatifQuiMeneA\junit-4.12.jar -d ../classes
calc/*.java junit4/CalculatorTest.java
```

- `executeTests.bat`

```
java -cp .;cheminQuiMeneA\junit-4.12.jar; cheminQuiMeneA\hamcrest-core-
1.3.jar -ea org.junit.runner.JUnitCore junit4.CalculatorTest
```

- L'exécution est lancée dans "l'environnement" `org.junit.runner.JUnitCore`

Résultat du test

- Démo dans `CalculetteJunit`, lancer les `.bat`
- La sortie est :

```
JUnit version 4.4
..E.E.I
Time: 0,015
There were 2 failures:
1) subtract(junit4.CalculatorTest)
java.lang.AssertionError: expected:<8> but was:<9>
at org.junit.Assert.fail(Assert.java:74)
...
at junit4.CalculatorTest.subtract(CalculatorTest.java:29)
...
2) divide(junit4.CalculatorTest)
java.lang.AssertionError
        at junit4.CalculatorTest.divide(CalculatorTest.java:36)
...

FAILURES!!!
Tests run: 4,  Failures: 2
```

- . = premier test lancé et réussi, .E = second test lancé et erreur, .E= troisième test lancé et erreur, . = quatrième test lancé et réussi, I = cinquième test ignoré (et donc pas lancé)

Les méthodes pour JUnit 4.x

- Avec JUnit4.x, on utilise les annotations sur les méthodes (à la place des conventions de noms des méthodes) :
  - `@Test` indique une méthode de test (au lieu de préfixer les méthodes de test par `test` : JUnit 3.x)
  - `@Before` indique une méthode d'initialisation (au lieu d'une méthode `setUp()` JUnit 3.x)
  - `@After` indique une méthode à exécuter après une méthode de test (au lieu d'une méthode `tearDown()` JUnit 3.x)
- On importe le paquetage `org.junit`
- Plus besoin d'hériter de la classe `TestCase` (JUnit 3.x)
- Plus besoin de créer une classe `TestSuite` (JUnit 3.x)

Rappel Java 1.4 :
les assertions

Les assertions (1/2)

- Source :
  `http://docs.oracle.com/javase/1.5.0/docs/guide/language/assert.html`
- De la forme :

```
assert Expression1 : Expression2 ;
```

où `Expression1` est un `boolean` ou un `Boolean`, `Expression2` est éventuellement absente

- Si `Expression1` est `true`, le reste de l'assertion est ignoré

Les assertions (2/2)

- Si `Expression1` est `false`,
- 1er cas : `Expression2` est absente
  - Une instance `AssertionError` est créée et cette exception `AssertionError` est levée
- 2ieme cas : `Expression2` est présente
  - `Expression2` est évaluée, convertie en `String`, et constitue le message de l'exception `AssertionError` qui est construit et levée

Utilisation des assertions

- Il faut lancer le code par :

  `java -ea ClassePrinc`

  pour utiliser les assertions sinon elles sont ignorées

Fin du rappel Java 1.4 : les assertions

Syntaxe JUnit 4 (1/2)

- Les méthodes qui testent sont annotées par `@Test`
- Les méthodes annotées par `@Before` sont lancées avant tout test (i.e. méthode annotée par `@Test`)
- Les méthodes annotées par `@After` sont lancées après tout test (i.e. méthode annotée par `@Test`)
- Une classe de test doit avoir au moins une méthode annotée `@Test`
- On peut utiliser les assertions auquel cas, exécuter le programme avec l'option `-ea` de `java`

Syntaxe JUnit 4 (2/2)

- L'annotation `@Test` peut avoir des paramêtres indiquant l'exception qui doit être levée. Si cette exception n'est pas levée ou qu'une autre exception est levée, le test échoue (cf. test `divideByZero()`)
- La méthode `multiply(int n)` de la classe `Calculator` n'est pas encore implémentée mais lorsqu'elle le sera on voudra la tester. En indiquant `@Ignore` avant (ou après) `@Test`, on précise que pour l'instant le test est ignoré. Lors de l'exécution des tests, les tests ignorés sont indiqués (ainsi on ne les oublie pas)

Fixture

- Les méthodes de test manipulent un (ou plusieurs) objets de la classe à tester. Deux méthodes de test utilisent des objets de la classe à tester (et d'autres classes) distincts
- Une fixture est un ensemble d'objets propre à l'exécution d'une méthode de test
- Ces objets sont, en général, construit dans les méthodes annotées `@Before`
- Les désallocations faites par une fixture sont en général faites dans les méthodes annotées `@After`

Rappel de l'exécution (1/2)

- Dans le code ci contre, un ordre des appels peut être est :
  `setUp`,
  `testEmptyCollection`,
  `setUp`,
  `testOneItemCollection`
- L'ordre de l'exécution des deux méthodes de tests peut changer

```
import org.junit.*;
import static org.junit.Assert.*;
import java.util.*;

public class SimpleTest {

    private Collection<Object> collection;

    @Before
    public void setUp() {
        collection = new ArrayList<Object>();
    }

    @Test
    public void testEmptyCollection() {
        assertTrue(collection.isEmpty());
    }

    @Test
    public void testOneItemCollection() {
        collection.add("itemA");
        assertEquals(1, collection.size());
    }
}
```

Rappel de l'exécution (2/2)

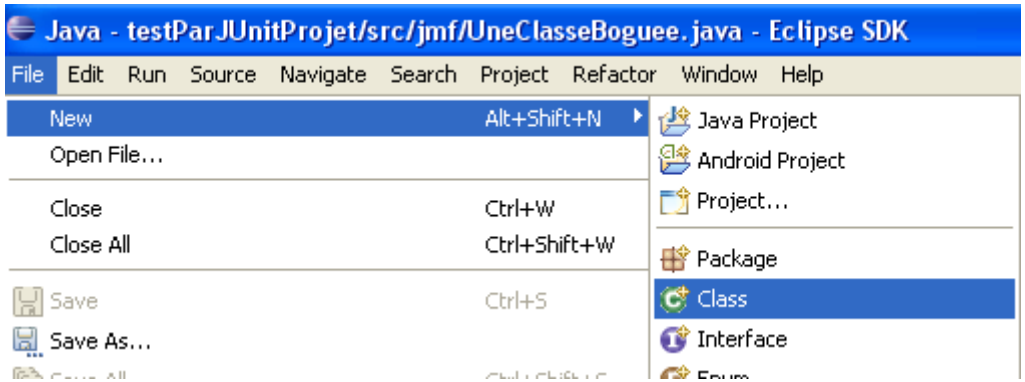
- "JUnit assumes that all test methods can be performed in an arbitrary order. Therefore tests should not depend other tests"
- source : `http://www.vogella.com/articles/JUnit/article.html#juniteclipse`

JUnit et Eclipse

- Les versions récentes d'Eclipse proposent d'ajouter automatiquement les `.jar` qui manquent

Créer une classe à tester

- Construire un projet puis une classe dans Eclipse (File | New | Class)

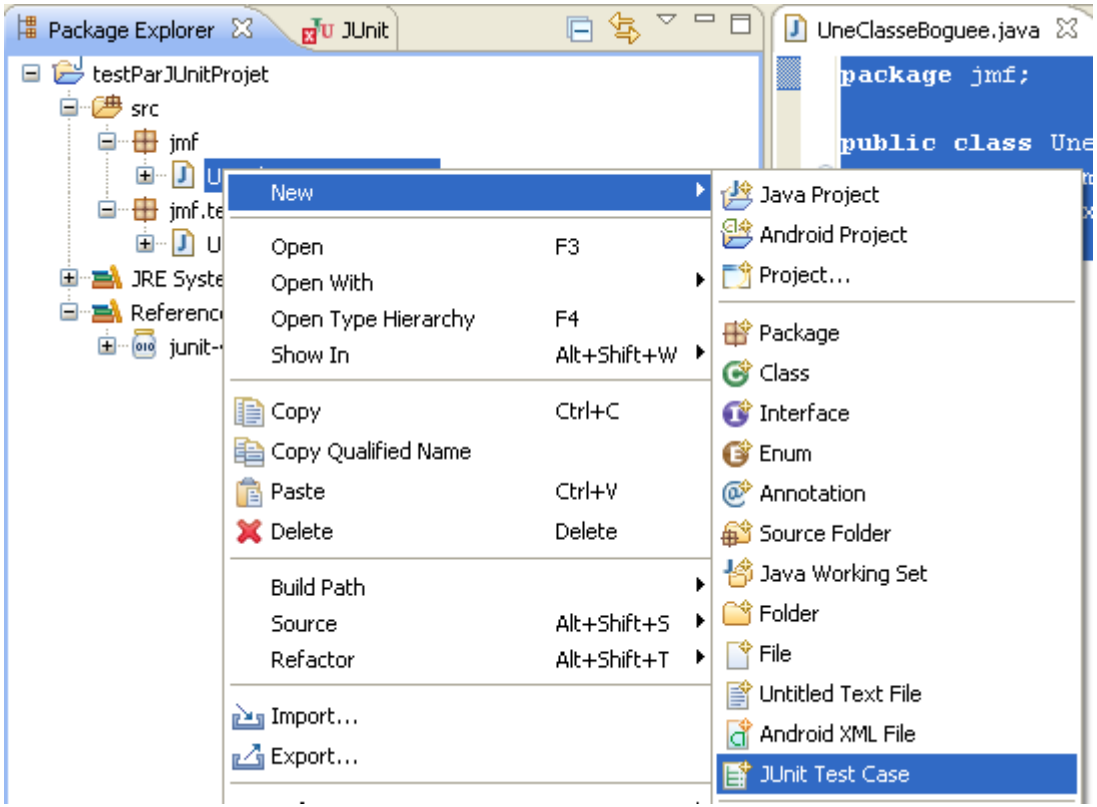


- Ecrire cette classe : par exemple :

```
package jmf;

public class UneClasseBoguee {
    public int multiply(int x, int y) {
        return x / y;
    }
}
```

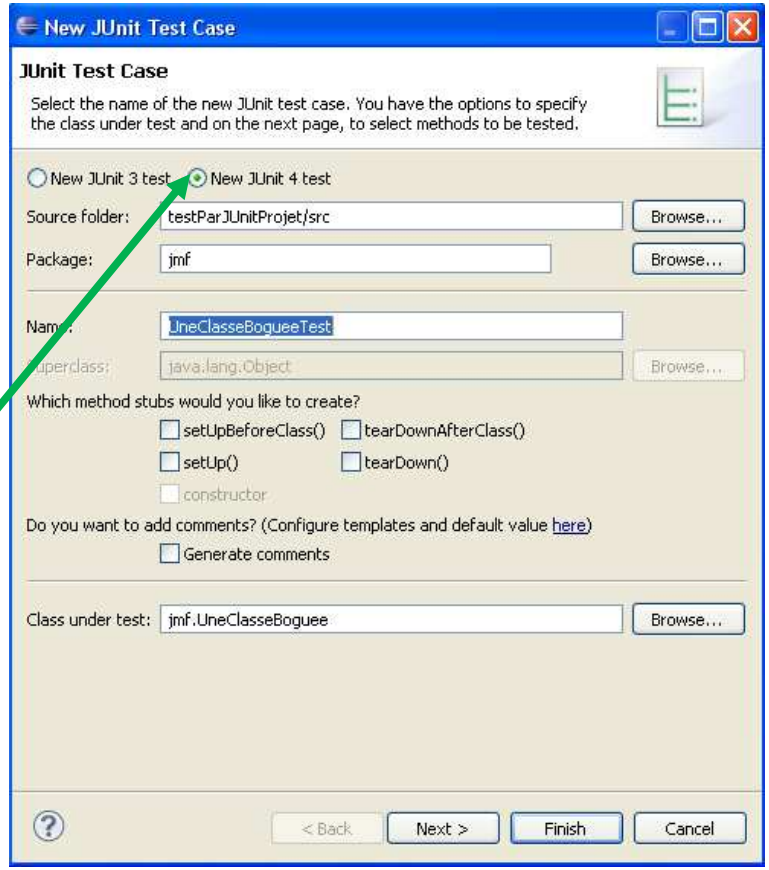
Créer une classe de test (1/5)

- Sélectionner la classe à tester
- Cliquez droit et sélectionner New | JUnit Test Case



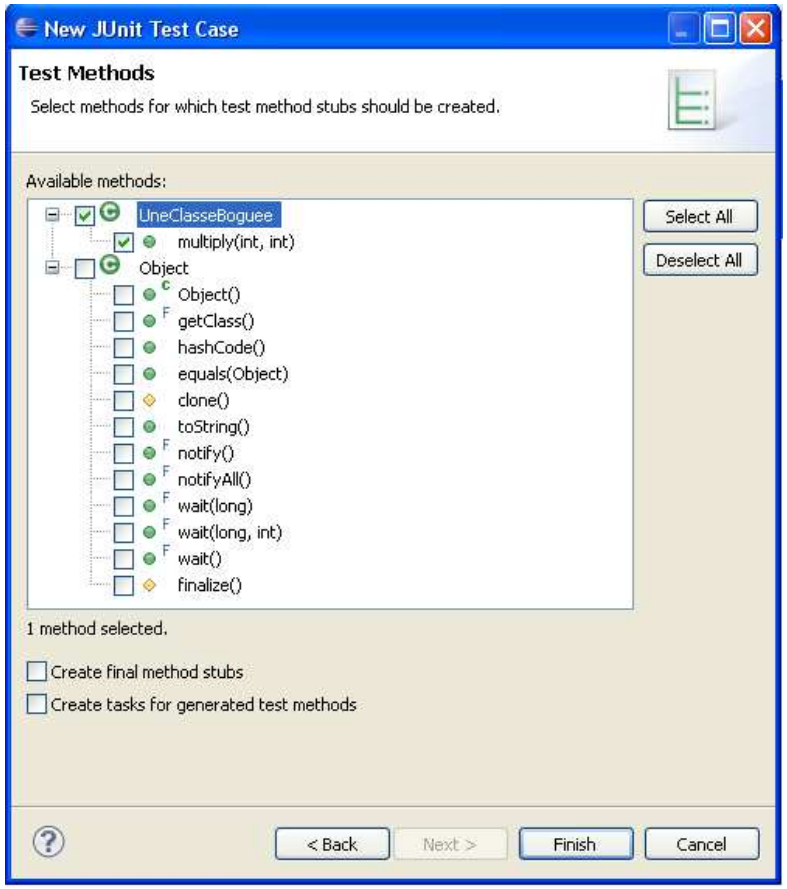
Créer une classe de test (2/5)

- Dans la fenêtre "New JUnit Test Case", indiquer un nom de paquetage et un nom de classe de test
- Indiquer bien la version JUnit que vous voulez utiliser à l'aide d'un des boutons radio. Par exemple pour JUnit 4, sélectionner le bouton radio "New JUnit4 test"
- Cliquez "Next >"



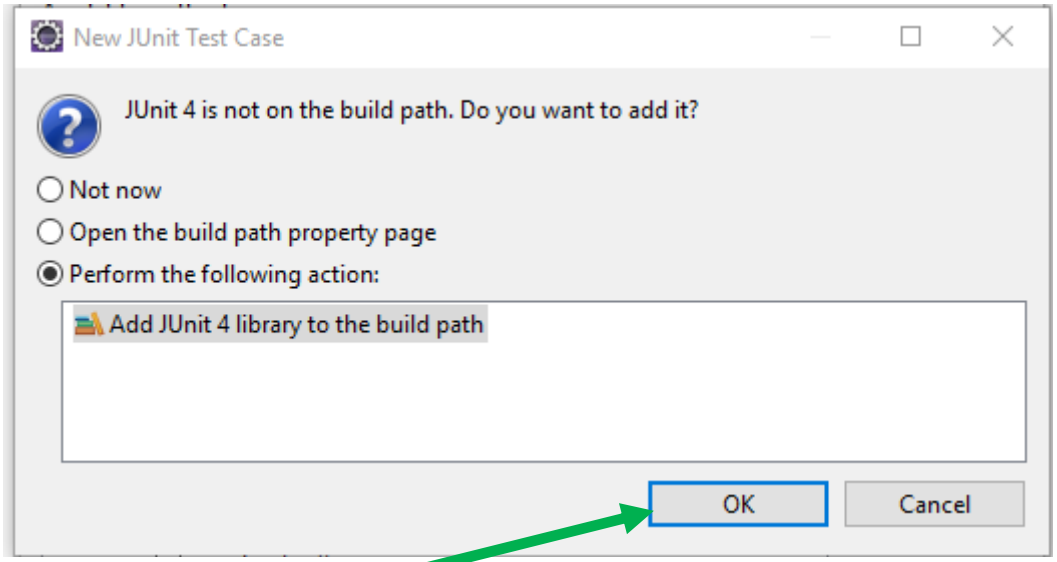
Créer une classe de test (3/5)

- Dans la fenêtre de sous titre Test Methods, indiquez (en cochant) les méthodes à tester. Cliquez Finish



Créer une classe de test (4/5)

- Les versions récentes d'Eclipse proposent alors d'ajouter automatiquement les bibliothèques JUnit :



- Cliquez le bouton OK

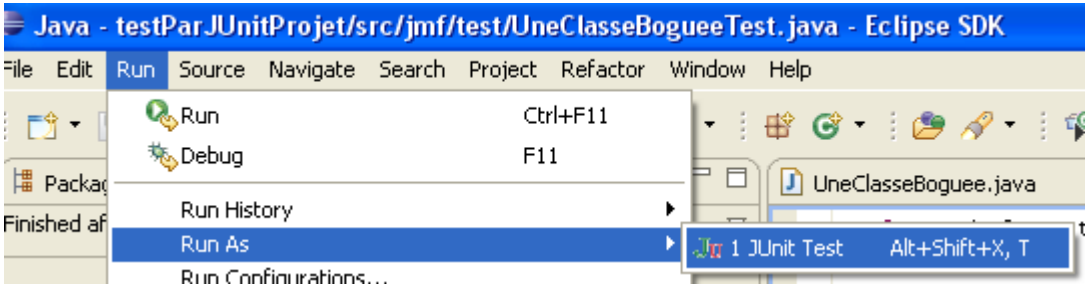
Créer une classe de test (5/5)

- La classe de test apparaît. Compléter la par du code approprié :

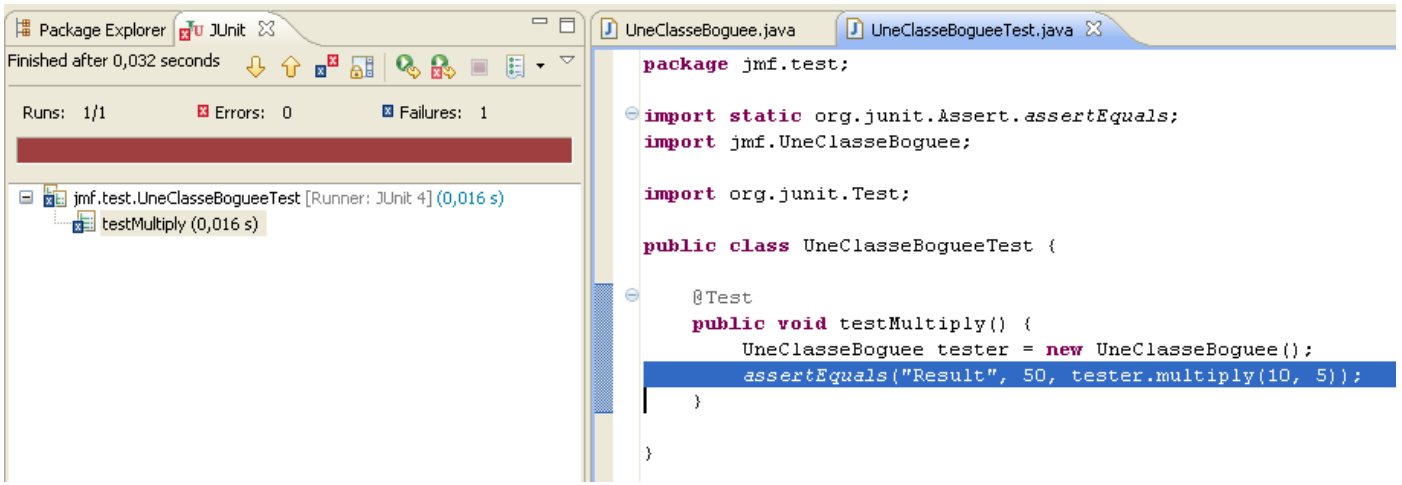
```
package jmf.test;

import static org.junit.Assert.assertEquals;
import jmf.UneClasseBoguee;

import org.junit.Test;

public class UneClasseBogueeTest {

    @Test
    public void testMultiply() {
        UneClasseBoguee tester = new UneClasseBoguee();
        assertEquals("Result", 50, tester.multiply(10, 5));
    }
}
```

Exécuter le test

- Lancer l'exécution du test par Run | Run As | JUnit Test



- Le résultat du test apparaît dans l'onglet JUnit



Exercice sur JUnit 4

- Ecrire des tests pour des classes modélisant des sommes d'argent et un porte feuille

Bibliographie

- Le site originel de JUnit : `http://junit.org/`
- L'API JUnit `http://junit.sourceforge.net/javadoc_40/index.html`
- La FAQ pour JUnit `http://junit.sourceforge.net/doc/faq/faq.htm`
- Un tutorial pour JUnit version 4 (donc avec les annotations) : `http://www.devx.com/Java/Article/31983` de Antonio Goncalves (qui compare JUnit 3 et 4). Cet article a beaucoup inspiré ce support de cours
- Cours de Pascal Graffion (merci Pascal)

Fin