

# Exercices tests unitaires



## Table des matières

<b><i>Section 1 Exercices – Mode d’emploi.....</i></b>	<b><i>3</i></b>
<b><i>Section 2 Tester la classe Calculator.....</i></b>	<b><i>4</i></b>
<b>2.1 But.....</b>	<b>4</b>
<b>2.2 Exercice.....</b>	<b>4</b>
2.2.1 Créer le projet pour la classe Calculator.....	4
2.2.2 Eclipse : Créer le projet JUnit.....	5
2.2.3 NetBeans : Créer le projet Junit.....	7
2.2.4 Coder les méthodes de Calculator.....	7
2.2.5 Coder les tests pour add().....	8
2.2.6 Tester et coder subtract() et multiply().....	8
2.2.7 Coder les tests pour divide().....	8
<b>2.3 Livrables.....</b>	<b>9</b>
<b><i>Section 3 Utiliser un module de couverture de test.....</i></b>	<b><i>10</i></b>
<b>3.1 But.....</b>	<b>10</b>
<b>3.2 Exercice.....</b>	<b>10</b>
<b><i>Section 4 Réaliser un mock.....</i></b>	<b><i>12</i></b>
<b>4.1 But.....</b>	<b>12</b>
<b>4.2 Exercice.....</b>	<b>12</b>
4.2.1 Mock manuel.....	12
4.2.2 Utiliser EasyMock.....	13
<b><i>Section 5 Autre classe à créer et tester.....</i></b>	<b><i>14</i></b>
<b>5.1 But.....</b>	<b>14</b>
<b>5.2 Exercice.....</b>	<b>14</b>
5.2.1 Définir les tests unitaires de la classe Triangle.....	14
5.2.2 Coder la classe Triangle.....	14
<b><i>Section 6 Tester une application Spring.....</i></b>	<b><i>15</i></b>
<b>6.1 But.....</b>	<b>15</b>
<b>6.2 Enoncé.....</b>	<b>15</b>

## Section 1 Exercices – Mode d'emploi

---

Ce cours expose de multiples concepts, qui, sans pratique, ne sert à peut-être à rien.

Les exercices qui suivent sont la mise en pratique immédiate du cours.

Ils ne sont pas difficiles !

Encore faut-il les faire avec curiosité et ténacité pour que leurs objectifs soient atteints. Il s'agit d'un travail individuel sur le rendu, même s'il est conseillé de partager des idées avec vos collègues.

Tout exercice comportant le paragraphe Livrables signifie que le/les fichiers mentionnés doivent être zippés et envoyés à l'adresse [contact@pragma-tec.fr](mailto:contact@pragma-tec.fr).

Chaque nom de zip est de la forme :

Exo<n><votre nom>.zip ex Exo4Dupond.zip

Il est souhaitable que vous échangiez des idées entre collègues, par contre le résultat fourni doit être UNIQUE et ceci sera vérifié.

## Section 2 Tester la classe Calculator

---

### 2.1 But

- Exemple pour implémenter un test JUnit

### 2.2 Exercice

Nous allons implémenter les méthodes de l'interface ICalculator dans la classe Calculator :

Voici la définition de cette interface :

```
public interface ICalculator {

    int multiply(int a, int b);
    int divide(int a, int b);
    int add(int a, int b);
    int subtract(int a, int b);

}
```



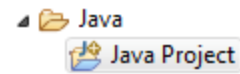
L'intérêt de créer une interface et de mettre à disposition aux développeurs et aux testeurs les définitions de ce qui est attendu. Chacun peut ensuite travailler en indépendance ...

Puis nous allons créer un module JUnit pour tester ces méthodes.

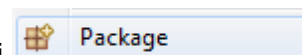
Mode opératoire proposé :

#### 2.2.1 Créer le projet pour la classe Calculator

Sous Eclipse, créer un nouveau projet de type Java project nommé appli.



Sous le nœud src de ce projet, créer le package appli



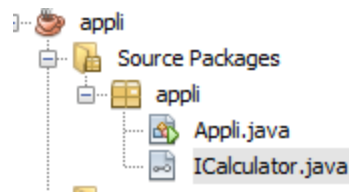
Dans ce package appli créer un fichier java Interface, le nommer Icalculator. Y copier coller le code ci-dessus.

Dans ce package appli créer une classe java Calculator qui implémente l'interface ICalculator. Eclipse crée automatiquement le squelette de la classe.

Nous reviendrons sur le code des méthodes après.

Sous NetBeans 8.2 créer un projet de type Java/Java Application de nom appli

Dans le package appli créer un fichier interface Icalculator.java




Y copier coller le code ci-dessus.

Dans le même package appli créer un classe Calculator.

Indiquer que la classe implémente l'interface Icalculator

Un Alt Enter sur Calculator permet d'implémenter le squelette de la classe.

## 2.2.2 Eclipse : Créer le projet JUnit

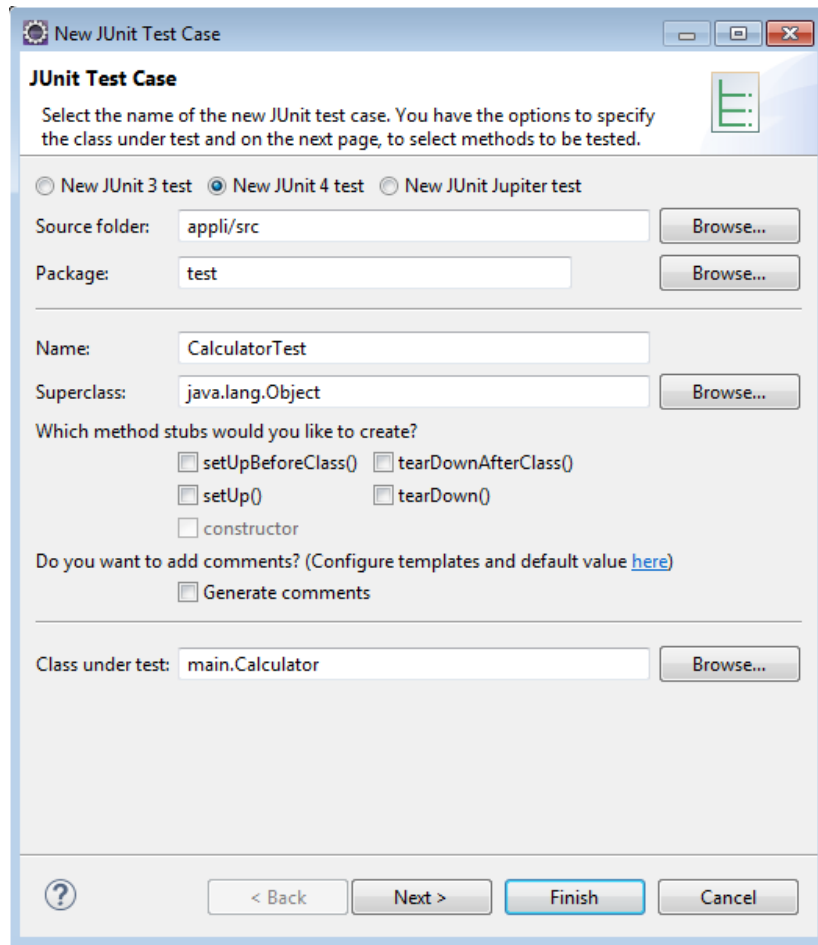
Sur le nœud src du projet appli, New ->  JUnit Test Case

Package :test

Name : CalculatorTest

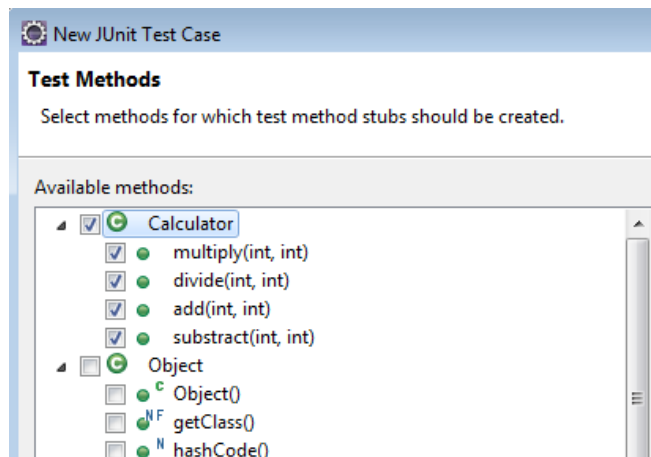
Class Under test : Calculator

Cocher New JUnit 4 test



Clic sur Next.

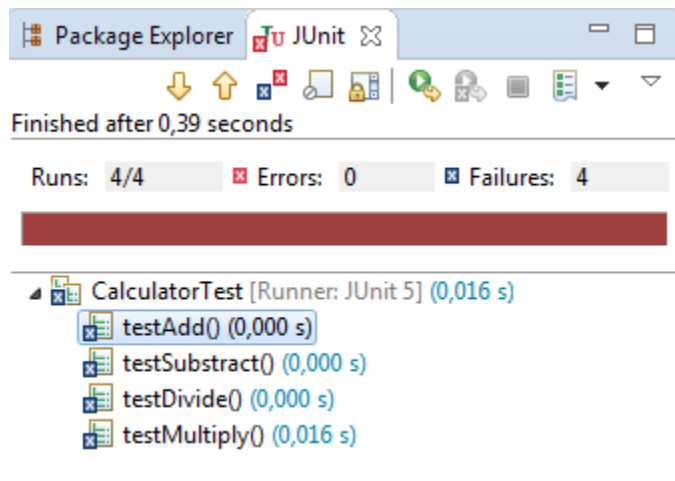
Sélectionner les 4 méthodes de la classe Calculator :



Tester la classe de test :

Désigner  CalculatorTest.java puis RunAs ->  2 JUnit Test

L'onglet JUnit montre le résultat suivant :



Editer le fichier CalculatorTest.java et remarquer :

- La présence de l'annotation `@Test`
- Chaque méthode n'accepte aucun paramètre et ne rend rien

### 2.2.3 NetBeans : Créer le projet Junit

Un nouveau Alt Enter sur Calculator permet de choisir un test en Junit. Laisser les choix par défaut dans la fenêtre qui s'ouvre.

### 2.2.4 Coder les méthodes de Calculator

Nous allons pratiquer le TDD : les tests avant le codage

Nous expliquons cependant comment sera fait le codage le moment venu :

Afin de créer des difficultés (et des bugs potentiels) il n'est pas permis d'utiliser les opérations arithmétiques courantes `+`, `-`, `*`, `/`.

Seul `++` ou `--` dans le code.

Ainsi pour faire une addition `add(int a, int b)` il faut au 1<sup>er</sup> paramètre a faire `a++` (ou `++a`) autant de fois qu'il y a de b si b est positif, `--` si b négatif ...

Pour faire une division `divide(int a, int b)` il faut compter combien de fois on peut retrancher b à a ... (et utiliser `subtract()`)

La tentative de division par 0 doit générer l'exception `ArithmeticException` :

```
if (b == 0) {
    throw new ArithmeticException();
}
```

### 2.2.5 Coder les tests pour add()

Commencer par implémenter `void testAdd()`

L'instruction `fail("chaîne")` est à utiliser sur un test qui échoue.

Exemple

```
void testAdd() {
    Calculator calc = new Calculator();
    int a, b, res;
    a = 5;
    b = 5;
    res = a + b;
    if (calc.add(a, b) != res) {
        fail("a et b positif");
    }
}
```

Tester :

- a et b positifs
- a négatif
- b négatif
- a et b négatifs
- autre ...

Le test

```
if (calc.add(a, b) != res) {
    fail("a et b positif");
}
```

Peut être avantageusement remplacé par ceci :

```
assertTrue("a et b positif", calc.add(a, b) == res);
```

Passer le test sans code dans Calculator. Constater que les tests ne passent pas.

Développer ensuite le code de `add` dans Calculator jusqu'à des tests valides.

Ajouter des tests si nécessaire.

Améliorer/clarifier ensuite le code et repasser les tests

### 2.2.6 Tester et coder subtract() et multiply()

Ecrire les tests puis coder comme ci-dessus.

### 2.2.7 Coder les tests pour divide()

Ecrire les tests puis le code de `divide()`

La nouveauté ici est de s'assurer que `divide()` génère bien une exception si division par 0.



Coder d'abord le contenu de `testDivide()` sans utiliser `b=0`

Puis pour le cas où `b = 0` (division par 0) on va écrire un test spécifique où on s'attend à recevoir une exception `ArithmeticException`. Voici son code :

```
@Test (expected = ArithmeticException.class)
public final void testDivideByZero() {
    Calculator calc = new Calculator();
    int a, b;
    a = 5;
    b = 0;
    calc.divide(a, b);
}
```

## 2.3 Livrables

Mettre sous un zip le projet Eclipse (ou Netbeans) appli en utilisant un explorateur de fichier (pas depuis eclipse)

## Section 3 Utiliser un module de couverture de test

---

### 3.1 But

- Utiliser EclEmma avec Eclipse
- Utiliser Jacoco avec NetBeans

### 3.2 Exercice

Il faut d'abord installer le module :

Sous Eclipse :

Aller dans Help >> Install New Software >> Add. Donner un nom (EclEmma) et une adresse (<http://update.eclEmma.org/>). Puis faites Ok et installer le plugin. Vous devrez accepter la licence puis redémarrer Eclipse.

A coté de l'icône debug, un nouvel icône est apparu : celui de EclEmma.



Cliquer sur ce bouton et regarder le source :

- Une ligne verte indique un code exécuté
- Une ligne rouge, code non exécuté
- Ligne jaune code partiellement exécuté

La fenêtre Coverage donne le pourcentage de couverture;

Sous NetBeans 8.2 :

Tools > Plugins choisir l'onglet Available Plugins

Saisir coverage dans la recherche . TikiOne JavaCoverage doit apparaître.

L'installer

Re démarrer Netbeans

La fonctionnalité est appellable depuis le nœud du projet, menu contextuel

Test with JaCoCoverage

Regarder la couverture du code.

Modifier ou ajouter des tests si la couverture de code est faible.

Eventuellement enlever du code mort (code inutile) dans la classe Calculator.

## Section 4 Réaliser un mock

---

### 4.1 But

- Ecrire une classe qui va dépendre de la classe Calculator
- Réaliser un mock de Calculator : un mock manuel et mock dynamique

### 4.2 Exercice

Créer dans le package appli la classe SpecificCalculation.

Cette classe contient la méthode

```
public int addsub(int a, int b, int c)
```

qui rend un entier résultat de l'opération  $a + b - c$  en utilisant obligatoirement les opérations de Calculation.

Ecrire un seul test pour l'instant dans un nouveau fichier de test.

Passer le test.

Ecrire le code de addsub() et passez le test jusqu'à l'obtention de test OK.

Le défaut est que l'on utilise pour le test unitaire de SpecificCalculation le code de Calculator.

#### 4.2.1 Mock manuel

Nous allons faire un mock manuel :

Dans SpecificCalculation ajouter la méthode :

```
protected ICalculator creerCalculator() {
    return new Calculator();
}
```

Modifier addsub() pour qu'elle utilise `creerCalculator()`

Cette étape est nécessaire pour que le fichier de test implémente son propre objet Icalculator.

Dans le fichier de test SpecificCalculationTest, copier ce code :

```
@Test
public void test1AvecMock() {
    SpecificCalculation calc = new SpecificCalculation() {
        // ecriture classe dérivée de façon anonyme
    };
}
```

```

        protected ICalculator creerCalculator(){
            return new MockCalculator();
        }
    };

    int a=-5, b=10, c=10, res;
    res = a + b - c;
    assertTrue("a négatif, b positif, c positif", calc.addsub(a, b, c)
    == res);
}

```

Analyser ce code et adaptez le à votre besoin.

La classe MockCalculator est manquante : la créer dans le même package que SpecificCalculationTest.java

Cette classe utilise l'interface Icalculator.

Coder les méthodes, cette fois-ci en utilisant les opérateurs + - \* / (c'est un bouchon).


Ecrire l'ensemble des tests que vous jugez nécessaires dans SpecificCalculationTest.java

Passer les tests.

#### 4.2.2 Utiliser EasyMock

Le site <https://www.jmdoudoux.fr/java/dej/chap-objets-mock.htm> peut être utile.

Télécharger l'archive depuis le site <http://easymock.org/> et la décompresser dans un répertoire du système. Il suffit alors d'ajouter le fichier easymock.jar dans le classpath.

Avec NetBeans, sur le nœud  Test Libraries

Add JAR/Folder...

appeler l'item de menu

Recopier ensuite le fichier SpecificCalculationEasyMocktest.java dans Test Package. Regarder le code et exécuter le test.

## Section 5 Autre classe à créer et tester

---

### 5.1 But

- Ecrire une classe
- Ecrire les tests de cette classe
- travail en autonomie

### 5.2 Exercice

Une application nécessite une classe Triangle.

Les longueurs des côtés de chaque triangle s'exprime avec une valeur décimale (un float).

Le constructeur Triangle(float max) permet de définir un triangle de côtés définis au hasard (random) ne devant pas dépasser la longueur max.

Une méthode doit exister pour indiquer (retourner à l'appelant de la méthode) si le triangle est quelconque (on dit aussi scalène), isocèle ou équilatéral.

Il existe un/des « geteurs » pour lire la longueur des côtés.

#### 5.2.1 Définir les tests unitaires de la classe Triangle

Dans le même projet Eclipse ou NetBeans que celui des exercices précédents, définir les tests de cette classe.

Définir éventuellement un mock pour obtenir des résultats prédictifs ...

#### 5.2.2 Coder la classe Triangle

Coder la classe puis passer les tests jusqu'à tout OK.



## Section 6 Tester une application Spring

---

### 6.1 But

Tester une API écrite avec Spring Boot.

### 6.2 Enoncé

Dézipper le fichier APIMS2Employees.zip

Utiliser le fichier sql fourni.

Utiliser netBeans ou équivalent pour ouvrir le projet api.

Compiler le projet.

Réaliser les tests unitaires de l'API.