# Test Unitaires avec JUnit

## 1 Introduction

### 1.1 Les tests - Pourquoi, Comment, Quand

Tous les programmeurs savent qu'ils doivent écrire des tests, mais peu le font, ce qui donne un code moins stable. Pour tester un programme, il est possible d'utiliser des affichages à l'écran, d'utiliser un debugger, etc. mais ces solutions nécessitent un jugement humain et sont limitées.

Le test d'un programme sert principalement à gagner du temps car il limite la présence d'erreurs. Les tests de non-régression par exemple permettent de vérifier que les méthodes qui étaient fonctionnelles avant une modification le sont toujours après.

Pour cela on automatise les tests et on systématise leur exécution même après une modification mineure. Les tests sont primordiaux dans les méthodes AGILE (par exemple eXtreme Programming) où en TDD (Test Driven Development).

Le développement piloté par les tests est une technique de développement de logiciel qui préconise d'écrire les tests unitaires avant d'écrire le code source d'un logiciel. Il permet d'affiner l'analyse, les tests sont des use cases au sens d'UML. Ils vont permettre de modéliser les pré et les post conditions des algorithmes, d'éviter d'écrire du code inutile et sont une forme de documentation technique. De plus le test est un exemple d'utilisation de la classe qu'on vient d'écrire.

### 1.2 Les tests en Java : JUnit

**JUnit** est un outil Open Source servant à écrire des suites de tests de façon simple et systématique et d'en piloter la mise en œuvre. Pour plus de détails sur JUnit, vous pouvez consulter `www.junit.org`. La javadoc se trouve à `https://junit.org/junit5/docs/current/api/`. Les outils de test comme JUnit sont mis en œuvre pour tester des modules de programme de façon répétée.

Au départ, JUnit n'était qu'une série de classes mais a ensuite été intégré complètement divers environnements de développement. Les classes de test JUnit contiennent des méthodes de test sur une autre classe :

— Elles portent le nom testXXXX()
— Ou utilisent l'annotation @Test (à partir de jUnit 4.x)
— Elle mettent en œuvre les assertions

Contrairement aux versions précédentes de JUnit, JUnit 5 est composé de plusieurs modules différents provenant de trois sous-projets différents.

JUnit 5 = JUnit Platform + JUnit Jupiter + JUnit Vintage

— La plate-forme JUnit sert de base pour le lancement de cadres de test sur la JVM. Elle définit également l'API TestEngine pour le développement d'un cadre de test fonctionnant sur la plate-forme. En outre, la plate-forme fournit un programme pour lancer la plate-forme en ligne de commande et un Runner basé sur JUnit 4 pour

exécuter n'importe quel TestEngine sur la plate-forme dans un environnement basé sur JUnit 4.
— JUnit Jupiter est la combinaison du nouveau modèle de programmation et du modèle d'extension pour écrire des tests et des extensions dans JUnit 5. Le sous-projet Jupiter fournit un TestEngine pour exécuter des tests basés sur Jupiter sur la plate-forme.
— JUnit Vintage fournit un TestEngine pour exécuter des tests basés sur JUnit 3 et JUnit 4 sur la plate-forme.

Nous allons ici utiliser JUnit 5.

# 2 Utilisation de JUnit

JUnit permet d'effectuer des tests automatisés sans intervention humaine pour les interpréter.

## 2.1 Création des premiers tests

Télécharger les codes à tester sur le serveur pédagogique (classes `Vectors.java` et `Utils.java`) et mettez-les dans un package `sample` d'un nouveau projet Java (avec maven).
Pour créer une classe de test, utilisez `Tools>Create/Update Test` sur la classe `Vectors.java`.
Choisissez le framework `JUnit` puis OK.
JUnit génère les méthodes "initializer" et "finalizer" avec les annotations @BeforeAll et @AfterAll. Cela signifie que ce seront des méthodes lancées avant et après toutes les méthodes de tests de la classe. Elles peuvent être utiles pour mettre en place un environnement de test spécifique et revenir à un état initial. Par exemple, au lieu de créer une connexion de base de données dans un initialiseur de test et de créer une nouvelle connexion avant chaque méthode de test, vous pouvez utiliser un initialiseur de classe de test pour ouvrir une connexion avant d'exécuter les tests. Vous pouvez ensuite fermer la connexion avec le finaliseur de classe de test.
Lancez le test par le menu contextuel `Run File`.
Modifiez la méthode `testScalarMultiplication()` par [1] :

```
@Test
public void testScalarMultiplication() {
    System.out.println("scalarMultiplication");
    assertEquals(  0, Vectors.scalarMultiplication(new int[] { 0, 0}, new int[] { 0, 0}));
    assertEquals( 39, Vectors.scalarMultiplication(new int[] { 3, 4}, new int[] { 5, 6}));
    assertEquals(-39, Vectors.scalarMultiplication(new int[] {-3, 4}, new int[] { 5,-6}));
    assertEquals(  0, Vectors.scalarMultiplication(new int[] { 5, 9}, new int[] {-9, 5}));
    assertEquals(100, Vectors.scalarMultiplication(new int[] { 6, 8}, new int[] { 6, 8}));
}
```

puis la méthode `testEquals()` par :

```
@Test
public void testEquals() {
    System.out.println("equal");
    assertTrue(Vectors.equal(new int[] {}, new int[] {}));
    assertTrue(Vectors.equal(new int[] {0}, new int[] {0}));
    assertTrue(Vectors.equal(new int[] {0, 0}, new int[] {0, 0}));
```

1. ces lignes de code sont disponibles sur le serveur pédagogique

```
    assertTrue(Vectors.equal(new int[] {0, 0, 0}, new int[] {0, 0, 0}));
    assertTrue(Vectors.equal(new int[] {5, 6, 7}, new int[] {5, 6, 7}));

    assertFalse(Vectors.equal(new int[] {}, new int[] {0}));
    assertFalse(Vectors.equal(new int[] {0}, new int[] {0, 0}));
    assertFalse(Vectors.equal(new int[] {0, 0}, new int[] {0, 0, 0}));
    assertFalse(Vectors.equal(new int[] {0, 0, 0}, new int[] {0, 0}));
    assertFalse(Vectors.equal(new int[] {0, 0}, new int[] {0}));
    assertFalse(Vectors.equal(new int[] {0}, new int[] {}));

    assertFalse(Vectors.equal(new int[] {0, 0, 0}, new int[] {0, 0, 1}));
    assertFalse(Vectors.equal(new int[] {0, 0, 0}, new int[] {0, 1, 0}));
    assertFalse(Vectors.equal(new int[] {0, 0, 0}, new int[] {1, 0, 0}));
    assertFalse(Vectors.equal(new int[] {0, 0, 1}, new int[] {0, 0, 3}));
}
```

Exécutez ces tests.
Créez maintenant les tests pour la classe `Utils.java`.

## 2.2 "Initialiseur" et "Finaliseur" de test

**"Initialiseur" de test** L'annotation `@BeforeEach` marque une méthode comme une méthode d'initialisation de test. L'"initialiseur" de test est exécuté avant chaque cas de test dans la classe de test. Une méthode d'initialisation de test n'est pas forcément nécessaire pour exécuter des tests, mais si vous avez besoin d'initialiser certaines variables avant d'exécuter un test, ce type de méthode est approprié.

**"Finaliseur" de test** L'annotation `@AfterEach` marque une méthode comme une méthode de finalisation de test. Une méthode de finalisation de test est exécuté après chaque cas de test dans la classe de test. Une méthode de finalisation de test n'est pas nécessaire pour des tests, mais vous pouvez avoir besoin d'un "finaliseur" pour nettoyer toutes les données qui ont été nécessaires lors de l'exécution des cas de test.

## 2.3 Test sur des assertions simples

Modifiez la méthode `testConcatWords` par :

```
@Test
public void testConcatWords() {
System.out.println("concatWords");
    assertEquals("Hello, world!", Utils.concatWords(new String[]{"Hello", ", ", "world", "!"}));
}
```

## 2.4 Test sur un compteur de temps

L'annotation `@Timeout(value = 10, unit = TimeUnit.NANOSECONDS)` permet de tester que la méthode met moins de 10 nanosecondes à s'exécuter. `@Timeout(10)` testerait que la méthode met moins de 10 secondes à s'exécuter. Plus de détails sur les unités de temps ici :

https://junit.org/junit5/docs/5.5.1/api/org/junit/jupiter/api/Timeout.html et d'autres exemples https://junit.org/junit5/docs/current/user-guide/#writing-tests-declarative-timeouts

```
@Test
@Timeout(value = 10, unit = TimeUnit.NANOSECONDS)
    public void testComputeFactorial() {
        System.out.println("computeFactorial");
        final int factorialOf = 1 + (int) (30000 * Math.random());
        System.out.println("computing " + factorialOf + "!");
        System.out.println(factorialOf + "! = " + Utils.computeFactorial(factorialOf));
    }
```

## 2.5 Test de levée d'exception

La méthode `assertThrows` permet de vérifier que le code testé renvoie bien une instance de la classe `IllegalArgumentException` :

```
   @Test
public void checkExpectedException() {
System.out.println("checkExpectedException");
final int factorialOf = -5;
Exception exception = assertThrows(IllegalArgumentException.class, () ->
//code under test for throwing IllegalArgumentException
System.out.println(factorialOf + "! = " + Utils.computeFactorial(factorialOf)));
}
```

## 2.6 Désactiver un Test

Pour désactiver un test, il suffit d'ajouter l'annotation `@Disabled` au dessus de l'annotation `@Test` du test à désactiver.

## 2.7 Répéter un test

L'annotation `@RepeatedTest(10)` permet de répéter un test 10 fois.

## 2.8 Tests paramétrés

Les tests paramétrés permettent d'exécuter un test plusieurs fois avec des arguments différents. Ils sont déclarés comme les méthodes `@Test` ordinaires, mais utilisent l'annotation `@ParameterizedTest` à la place.

```
@ParameterizedTest
@ValueSource(ints = {1, 2, 3})
void testWithValueSource(int argument) {
assertTrue(argument > 0 && argument < 4);
}
```

D'autres "sources" d'arguments peuvent être utilisées https://junit.org/junit5/docs/current/user-guide/#writing-tests-parameterized-tests-sources.

### 2.9 Autres annotations

Vous trouverez un grand nombre d'annotations possibles dans JUnit 5 détaillées ici `https://junit.org/junit5/docs/current/user-guide/#writing-tests-annotations` et leurs usages sont explicités dans la documentation suivante `https://junit.org/junit5/docs/current/user-guide/#writing-tests`

## 3 Mise en œuvre

### 3.1 Traitement d'images

Terminez et tester vos classes du TP de traitement d'images.

### 3.2 Jeu de dames

Implémentez avec une **interface texte** (aucune interface graphique n'est demandée) un jeu de dames à deux joueurs en groupe de 4 personnes. Les règles du jeu de dames sont disponibles ici : `https://fr.wikipedia.org/wiki/Dames`.

Vous devrez en particulier :

— Permettre à chaque joueur de jouer alternativement
— Déplacer un pion
— Transformer un pion en dame
— Capturer un pion ou une dame
— Permettre la prise multiple pour une dame
— Enregistrer une partie en cours et ouvrir une partie enregistrée

Toutes les fonctions doivent être testées. Le projet sera versionné sur GitHub Classroom.