

# SOMMAIRE

1. Le test Logiciel
2. Le Test automatique
3. Principes fondamentaux du Développement dirigé par les tests : *TDD Test Driven Development*
4. Framework Junit
5. Framework de tests automatisés



# FRAMEWORK JUNIT

Dr. Soukeina Ben Chikha



# LES @NNOTATIONS

# STRUCTURATION DES TESTS : ANNOTATIONS JUNIT

- Une annotation Java, c'est un mot clé précédé du symbole @, placée au-dessus d'un élément en Java : un nom de classe, de méthode, .... Elle permet de donner une information précise pour décrire cet élément.
- Cette information peut être utilisée pour modifier la manière dont le code va être exécuté.
- Les annotation définissent comment Junit va exécuter les tests.
- Ceci facilite le travail du développeur, qui ajoute les annotations, et JUnit les interprète
- Les annotations permettent d'économiser en terme de ligne de code.

# STRUCTURATION DES TESTS : ANNOTATIONS JUNIT

JUnit offre de nombreuses annotations utiles

```
import org.junit.jupiter.api.Test;
```

@sassertion	Description
@Test	<pre>public void unTest() {     // Arrange     ...      // Act     ...      // Assert     ... }</pre>

# STRUCTURATION DES TESTS : ANNOTATIONS JUNIT

```
import org.junit.jupiter.api.*;
```

@ssection	Description
@BeforeEach	<pre>public void methodeAppeleeAvantChaqueTest() {     //... }</pre>
@AfterEach	<pre>public void methodeAppeleeApresChaqueTest() {     //... }</pre>
@BeforeAll	<pre>public static void methodeAppeleeAvantTousLesTests() {     //... }</pre>
@AfterAll	<pre>public static void methodeAppeleeApresTousLesTests() {     //... }</pre>

# STRUCTURATION DES TESTS : ANNOTATIONS JUNIT

- `import org.junit.jupiter.params.ParameterizedTest;`
- `import org.junit.jupiter.params.provider.CsvSource;`
- `import org.junit.jupiter.params.provider.ValueSource;`

@sassertion	Description
@ParametrizedTest	Vous souhaitez réutiliser le même test avec plusieurs entrants (@ValueSource) voire plusieurs entrants/sortants (@CsvSource).
@Timeout	Si vous testez une méthode qui ne doit pas être trop lente, vous pouvez la forcer à échouer le test.
....	

# STRUCTURATION DES TESTS : ANNOTATIONS JUNIT

- Soit le code de la calculatrice
- On observe que les lignes 14, 26 et 37 (non visible) sont identiques

```
Calculatrice.java  CalculatriceTest.java ×
8  class CalculatriceTest {
9      @Test
10     void testAddDeuxNbrPos() {
11         //1-Arrange
12         int a=2;
13         int b=3;
14         Calculatrice cal= new Calculatrice();
15         //2-Act
16         int s = cal.add(a,b);
17         //3-Assert
18         //assertEquals(reultat_attendu, resultat_obtenu)
19         assertEquals(5,s);
20     }
21     @Test
22     void testMulDeuxNbrPos() {
23         //1-Arrange
24         int a=2;
25         int b=3;
26         Calculatrice cal= new Calculatrice();
27         //2-Act
28         int s = cal.mul(a,b);
29         //3-Assert
30         assertEquals(6,s);
31     }
32     @Test
33     void testDivDeuxNbrPos() {
```



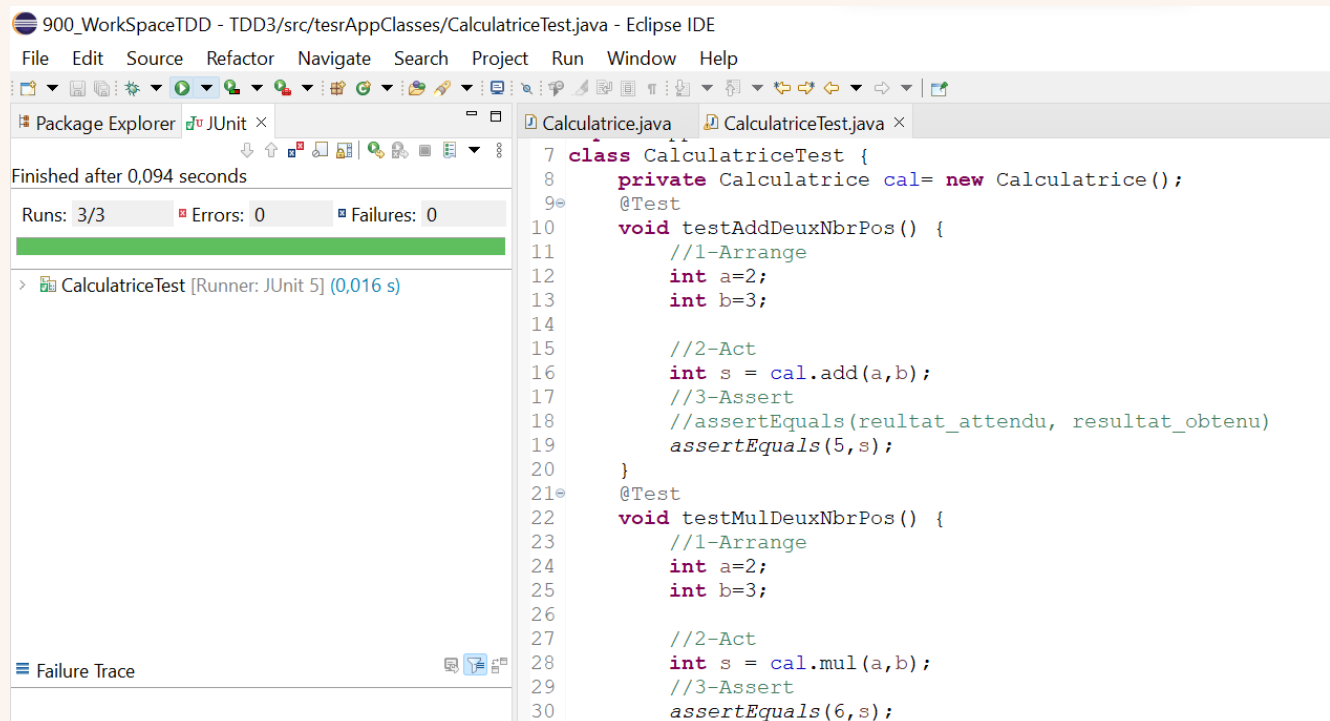
## STRUCTURATION DES TESTS

- Soit le code de la calculatrice
- On observe que les lignes 14, 26 et 37 (non visible) sont identiques
- SOLUTION 1 :
- Création attribut cal dans la classe CalculatriceTest
- Suppression des lignes 14, 26 et 37

```
Calculatrice.java  *CalculatriceTest.java x
7 class CalculatriceTest {
8     private Calculatrice cal= new Calculatrice();
9     @Test
10    void testAddDeuxNbrPos() {
11        //1-Arrange
12        int a=2;
13        int b=3;
14
15        //2-Act
16        int s = cal.add(a,b);
17        //3-Assert
18        //assertEquals(reultat_attendu, resultat_obtenu)
19        assertEquals(5,s);
20    }
21    @Test
22    void testMulDeuxNbrPos() {
23        //1-Arrange
24        int a=2;
25        int b=3;
26
27        //2-Act
28        int s = cal.mul(a,b);
29        //3-Assert
30        assertEquals(6,s);
31    }
32    @Test
```

# STRUCTURATION DES TESTS : ANNOTATIONS JUNIT

- Le test réussi mais attention il s'agit du même objet cal.
- Dans cet exemple il s'agit de création d'objet, tout autre code serait impossible à mettre sans méthode ou méthode statique qu'il faut appeler plus tard.



The screenshot shows the Eclipse IDE interface. The top toolbar includes icons for File, Edit, Source, Refactor, Navigate, Search, Project, Run, Window, and Help. The Package Explorer on the left shows the project structure, including the 'JUnit' package. The main editor displays the source code of 'CalculatriceTest.java'. The code defines a class 'CalculatriceTest' with two test methods: 'testAddDeuxNbrPos()' and 'testMulDeuxNbrPos()'. Both methods follow the JUnit 5 pattern: they start with an '@Test' annotation, followed by an 'Arrange' phase (initializing variables 'a' and 'b'), an 'Act' phase (calling the method being tested), and an 'Assert' phase (using 'assertEquals' to verify the result). The 'testAddDeuxNbrPos()' method calls 'cal.add(a,b)' and asserts the result is 5. The 'testMulDeuxNbrPos()' method calls 'cal.mul(a,b)' and asserts the result is 6. The bottom status bar indicates the test run was finished after 0.094 seconds, with 3/3 runs, 0 errors, and 0 failures. The bottom left corner shows a 'Failure Trace' tab.

```
900_WorkSpaceTDD - TDD3/src/tesAppClasses/CalculatriceTest.java - Eclipse IDE
File Edit Source Refactor Navigate Search Project Run Window Help

Package Explorer JUnit x
Finished after 0,094 seconds
Runs: 3/3 Errors: 0 Failures: 0
> CalculatriceTest [Runner: JUnit 5] (0,016 s)

Failure Trace

7 class CalculatriceTest {
8     private Calculatrice cal= new Calculatrice();
9     @Test
10    void testAddDeuxNbrPos() {
11        //1-Arrange
12        int a=2;
13        int b=3;
14
15        //2-Act
16        int s = cal.add(a,b);
17        //3-Assert
18        //assertEquals(reultat_attendu, resultat_obtenu)
19        assertEquals(5,s);
20    }
21    @Test
22    void testMulDeuxNbrPos() {
23        //1-Arrange
24        int a=2;
25        int b=3;
26
27        //2-Act
28        int s = cal.mul(a,b);
29        //3-Assert
30        assertEquals(6,s);
31    }
32 }
```

# STRUCTURATION DES TESTS : ANNOTATIONS JUNIT

- Soit le code de la calculatrice
- On observe que les lignes 14, 26 et 37 (non visible) sont identiques
- **SOLUTION 2 :**
  - On dispose de l'annotation **@BeforeEach** qui permet de réaliser cela automatiquement.
  - Supprimer les lignes 14, 26 et 37 et ajouter le code

```
7 class CalculatriceTest {  
8     private Calculatrice cal;  
9     @BeforeEach  
10    void initCalc()  
11    {  
12        cal= new Calculatrice();  
13        System.out.println("InitialisationObjetCalc");  
14    }
```

- L'étape ARRANGE est allégée

# STRUCTURATION DES TESTS : ANNOTATIONS JUNIT

- Pour additionner deux nombres ou les multiplier nous avons utilisé 1 seul cas.
- Supposons que pour le cas de la multiplication on décide de tester pour plusieurs valeurs de a (5), si b=0 alors le résultat est toujours zéro.
- Ça serait trop lourd de faire cela avec 5 tests `@Test`
- L'annotation `@ParameterizedTest` offre une solution pour ce cas en utilisant `@ValueSource`
- `@ValueSource` accepte tous les types primitifs Java standard comme les valeurs ints, longs, strings, etc.
- `@ParameterizedTest` accepte un paramètre pour formater le nom du test en fonction du paramètre permettant de personnaliser l'affichage

# STRUCTURATION DES TESTS : ANNOTATIONS JUNIT

900\_WorkSpaceTDD - TDD3/src/tesAppClasses/CalculatriceTest.java - Eclipse IDE

File Edit Source Refactor Navigate Search Project Run Window Help

Package Explorer JUnit ×

Finished after 0,173 seconds

Runs: 13/13 Errors: 0 Failures: 0

CalculatriceTest [Runner: JUnit 5] (0,092 s)

- testMulDeuxNbrPos() (0,010 s)
- testDivDeuxNbrPos() (0,001 s)
- > multiplication\_plusieursParam(int, int, int) (0,022 s)
- ▼ multiplication\_avecZero\_plusieursParam(int) (0,001 s)
  - 1 \* 0 doit être égal à 0 (0,001 s)
  - 2 \* 0 doit être égal à 0 (0,001 s)
  - 42 \* 0 doit être égal à 0 (0,001 s)
  - 1011 \* 0 doit être égal à 0 (0,000 s)
  - 5089 \* 0 doit être égal à 0 (0,001 s)
- testAddDeuxNbrPos() (0,001 s)

Calculatrice.java

CalculatriceTest.java ×

```
47
48
49
50
51
52
53
54 @ParameterizedTest(name = " {0} * 0 doit être égal à 0")
55 @ValueSource(ints = { 1, 2, 42, 1011, 5089 })
56 public void multiplication_avecZero_plusieursParam(int arg) {
57     // Arrange -- Tout est prêt !
58
59     // Act -- Multiplier par zéro
60     int res = cal.mul(arg, 0);
61
62     // Assert -- ça vaut toujours zéro !
63     assertEquals(0, res);
64 }
65
66
```

# STRUCTURATION DES TESTS : ANNOTATIONS JUNIT

- Supposons que pour le cas de la multiplication on décide de tester pour :
  - Deux nombre positifs
  - Deux nombres négatifs
  - Un nombre négatif et un nombre positif
  - Un nombre positif et zéro
  - Un nombre négatif et zéro
- En d'autres termes c'est un test avec 3 arguments ( a, b et res)

# STRUCTURATION DES TESTS : ANNOTATIONS JUNIT

900\_WorkSpaceTDD - TDD3/src/tesrAppClasses/CalculatriceTest.java - Eclipse IDE

File Edit Source Refactor Navigate Search Project Run Window Help

Package Explorer JUnit x

Finished after 0,188 seconds

Runs: 13/13 Errors: 0 Failures: 0

CalculatriceTest [Runner: JUnit 5] (0,093 s)

- testMulDeuxNbrPos() (0,010 s)
- testDivDeuxNbrPos() (0,001 s)
- multiplication\_plusieursParam(int, int, int) (0,024 s)
  - 1 + 1 doit être égal à -1 (0,024 s)
  - 2 + -3 doit être égal à -6 (0,002 s)
  - 4 + -5 doit être égal à 20 (0,001 s)
  - 9 + 0 doit être égal à 0 (0,001 s)
  - 6 + 0 doit être égal à 0 (0,000 s)
- testAddDeuxNbrPos() (0,002 s)
- multiply\_shouldReturnZero\_ofZeroWithMultipleInte

```
61
62
63 @ParameterizedTest(name = "{0} + {1} doit être égal à {2}")
64 @CsvSource({ "-1,1,-1", "2,-3,-6", "-4,-5,20", "-9,0,0", "6,0,0"})
65 public void multiplication_plusieursParam(int arg1, int arg2, int ResAttendu)
66 {
67     // Arrange -- Tout est prêt !
68
69     // Act
70     int ResObtenu = cal.mul(arg1, arg2);
71
72     // Assert
73     assertEquals(ResAttendu, ResObtenu);
74 }
75
76
77
78
79
80
```

## STRUCTURATION DES TESTS : ANNOTATIONS JUNIT

- Soit un test qui vérifie d'une méthode ne prend pas trop de temps. Il faut exprimer trop par une valeur : pas plus de 3 secondes.
- On ajoute dans la classe calculatrice une méthode qui simule une exécution longues 4sec :

```
public void longCalcul() {  
    try {  
        // Attendre 4 secondes  
        Thread.sleep(4000);  
    } catch (InterruptedException e) {  
        e.printStackTrace();  
    }  
}
```



# STRUCTURATION DES TESTS : ANNOTATIONS JUNIT

900\_WorkSpaceTDD - TDD3/src/tesAppClasses/CalculatriceTest.java - Eclipse IDE

File Edit Source Refactor Navigate Search Project Run Window Help

Package Explorer JUnit ×

Finished after 1,195 seconds

Runs: 14/14 Errors: 1 Failures: 0

CalculatriceTest [Runner: JUnit 5] (1,115 s)

- testMulDeuxNbrPos() (0,009 s)
- testDivDeuxNbrPos() (0,000 s)
- > multiplication\_plusieursParam(int, int, int) (0,023 s)
- > multiplication\_avecZero\_plusieursParam(int) (0,002 s)
- testAddDeuxNbrPos() (0,001 s)
- longCalcul\_neDoitpasDepasser1sec() (1,019 s)

Calculatrice.java CalculatriceTest.java ×

```
126
127
128 @Timeout(1)
129 @Test
130 public void longCalcul_neDoitpasDepasser1sec() {
131     // Arrange
132
133     // Act
134     cal.longCalcul();
135
136     // Assert
137     // ...
138 }
139
140
```

# STRUCTURATION DES TESTS : ANNOTATIONS JUNIT

900\_WorkSpaceTDD - TDD3/src/tesrAppClasses/CalculatriceTest.java - Eclipse IDE

File Edit Source Refactor Navigate Search Project Run Window Help

Package Explorer JUnit x

Finished after 1,179 seconds

Runs: 14/14 Errors: 0 Failures: 0

CalculatriceTest [Runner: JUnit 5] (1,104 s)

- testMulDeuxNbrPos() (0,009 s)
- testDivDeuxNbrPos() (0,000 s)
- > multiplication\_plusieursParam(int, int, int) (0,021 s)
- > multiplication\_avecZero\_plusieursParam(int) (0,002 s)
- testAddDeuxNbrPos() (0,001 s)
- longCalcul\_neDoitpasDepasser2sec() (1,017 s)

Calculatrice.java

CalculatriceTest.java x

```
126
127
128 @Timeout(2)
129 @Test
130 public void longCalcul_neDoitpasDepasser2sec() {
131     // Arrange
132
133     // Act
134     cal.longCalcul();
135
136     // Assert
137     // ...
138 }
139
140
```

```
public void longCalcul() {
    try {
        // Attendre 1 secondes
        Thread.sleep(1000);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
```

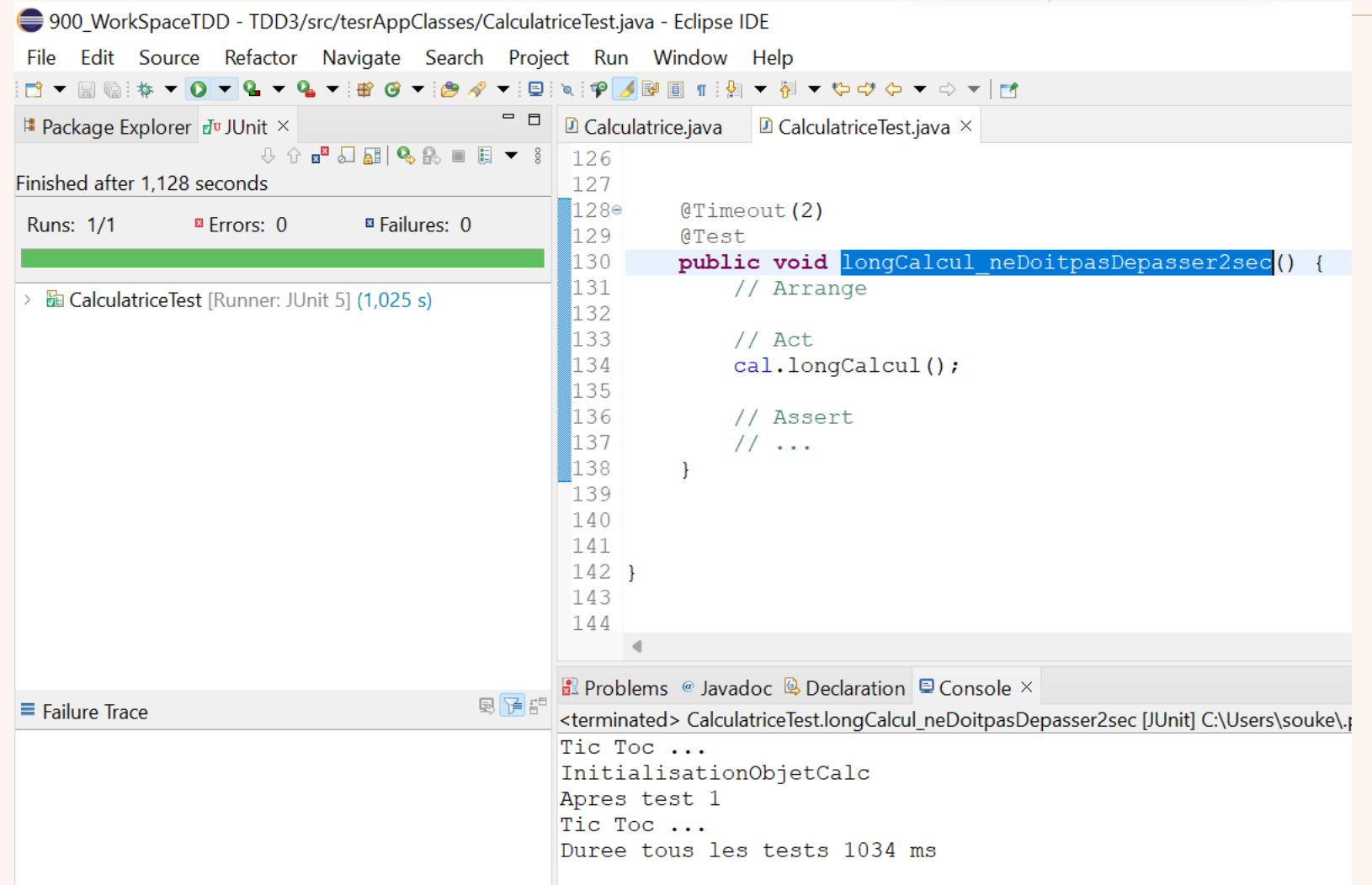
# STRUCTURATION DES TESTS : ANNOTATIONS JUNIT

## Remarque

- **Attention à l'annotation @Timeout.**
- Dans un travail d'équipe, tout le monde peut exécuter les tests, et vous ne maîtrisez pas la puissance de la machine qui va exécuter les tests.
- Possibilité d'obtenir des faux positifs à cause d'un ordinateur lent ou d'un serveur surchargé.

# STRUCTURATION DES TESTS : ANNOTATIONS JUNIT

- Dans Eclipse, si on sélectionne un test, il est le seul à être exécuté



900\_WorkSpaceTDD - TDD3/src/testAppClasses/CalculatriceTest.java - Eclipse IDE

File Edit Source Refactor Navigate Search Project Run Window Help

Package Explorer JUnit x

Finished after 1,128 seconds

Runs: 1/1 Errors: 0 Failures: 0

> CalculatriceTest [Runner: JUnit 5] (1,025 s)

Failure Trace

```
126
127
128 @Timeout(2)
129 @Test
130 public void longCalcul_neDoitpasDepasser2sec() {
131     // Arrange
132
133     // Act
134     cal.longCalcul();
135
136     // Assert
137     // ...
138 }
139
140
141
142 }
143
144
```

Problems Javadoc Declaration Console x

```
<terminated> CalculatriceTest.longCalcul_neDoitpasDepasser2sec [JUnit] C:\Users\souke\...
Tic Toc ...
InitialisationObjetCalc
Apres test 1
Tic Toc ...
Duree tous les tests 1034 ms
```

# RECAP

- Nous avons vu quelques @notation :
  - Pour faire une action avant ou après vos tests ;
  - Pour rendre paramétrables les tests ;
  - Pour la gestion du temps de traitement
- Les annotations JUnit aident à écrire des tests plus clairs sans répétitions inutile : structuration du code de test

## TP2

- Utiliser les assertions suivantes :
  - **@AferEach** avec la méthode razCalculatrice:  
Afficher le message « Après test » + numéro de test  
Mettre l'objet cal à null.

- Utiliser les assertions suivantes :
  - **@BeforeAll** et **@AfterAll** avec les méthodes `startTest` et `endTest` pour calculer la durée de tout les tests.

On aura besoin d'une méthode qui sera appelée avant tous les test pour récupérer l'heure de début et une méthode qui sera appelé après tous les test pour récupérer l'heure de fin et afficher le temps.

N.B. On ne peut utiliser la solution mettre la première méthode au début puis l'autre la dernière, vu qu'on n'est pas sûr de ne pas revenir au fichier test et ajouter du code.

Voir code fourni pour le calcul d'une durée

## TP2

```
import java.time.Instant;
import java.time.Duration;

Instant debTest;
Instant endTest;
debTest=Instant.now();
//....
endTest=Instant.now();
long dureeTests=Duration.between(endTest, debTest).toMillis();
System.out.println(dureeTests);
```



## TP2

- Utiliser les assertions suivantes :
  - **@ParameterizedTest** , **@ValueSource** et **@CsvSource** pour tester l'addition avec plusieurs valeurs.
- Pour les rapides
  - Ajouter une méthode pour la soustraction entre réels dans la classe calculatrice
  - Ne pas oublier Rouge, Vert, Refactoring
  - Utiliser **@ParameterizedTest** , **@ValueSource** et **@CsvSource**
  - Tester **@Timeout**