

# Tests unitaires

## - Tuto -

### Objectifs

- Ecrire des tests unitaires
- Utiliser l'environnement IntelliJ pour créer et exécuter les tests unitaires
- Aborder les annotations et les lambda expressions nécessaires dans l'écriture des tests unitaires

### Objectifs des tests unitaires

Un test unitaire a pour objectif de tester un composant logiciel individuel, habituellement un composant qui reçoit peu d'entrées et qui retourne un seul output.

JUnit permet d'effectuer des tests unitaires en Java, en général sur des méthodes. Plusieurs tests unitaires peuvent être créés sur une même méthode.

A noter que les tests unitaires ne permettent pas d'affirmer qu'un programme est correct si tous les tests unitaires réussissent. Ils permettent seulement, quand une méthode ne fournit pas un résultat attendu, de détecter des erreurs dans le code de cette méthode.

### Pattern AAA

Les tests unitaires respectent le pattern AAA (pour Arrange – Act – Assert).

#### Étape 1 : **Arrange**

Setup de tout ce qui est nécessaire pour exécuter le code à tester : initialisation de dépendances, mocks, data nécessaires pour le test...

#### Étape 2 : **Act**

Exécution du code à tester : invocation de la méthode à tester en y injectant les données préparées à l'étape 1.

### Étape 3 : **Assert**

Vérification des valeurs attendues : concordance des valeurs retournées par l'étape 2 avec celles attendues spécifiées dans les critères du test.

Le test unitaire réussit s'il y a concordance ; il échoue dans le cas contraire.

---

### Classe contenant les méthodes à tester

---

Soit la classe *Calculator* qui contient 4 méthodes sur lesquelles on souhaite créer des tests unitaires.

```
public class Calculator {  
  
    public Double add(Double number1, Double number2) {  
        return number1 + number2;  
    }  
  
    public Double subtract(Double number1, Double number2) {  
        return number1 + number2;  
    }  
  
    public Double multiply(Double number1, Double number2) {  
        return number1 * number2;  
    }  
  
    public Double divide(Double number1, Double number2) throws DivisionException {  
        if (number2 == 0)  
            throw new DivisionException(number2);  
        return number1 / number2;  
    }  
}
```

On peut créer un ou plusieurs tests unitaires sur chacune de ces méthodes.

---

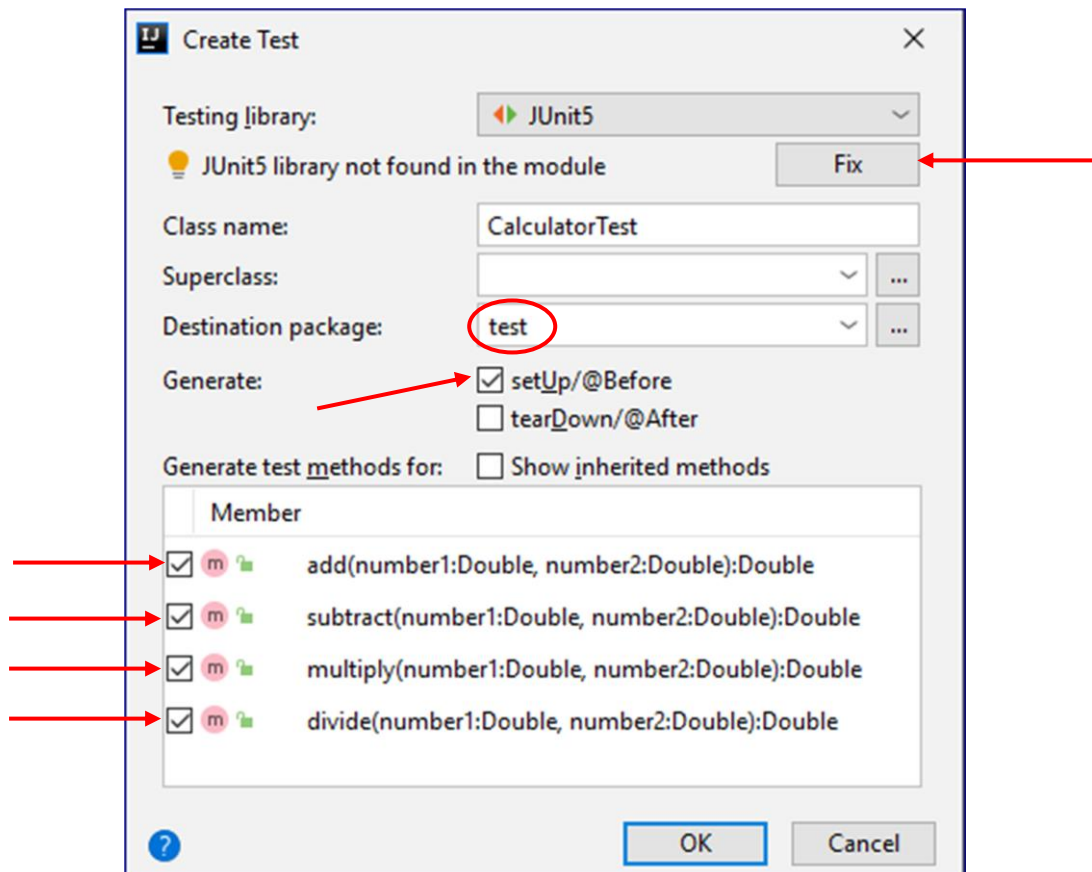
### Créer des tests unitaires dans IntelliJ

---

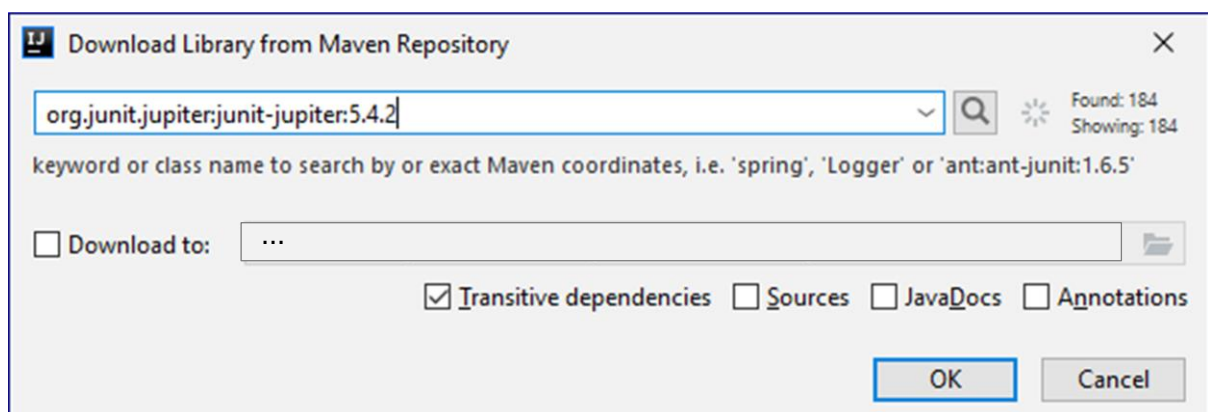
Créez un package *test*, par exemple dans le répertoire *src* du projet.

Pour créer des tests unitaires sur une classe, on peut utiliser un raccourci fournit par IntelliJ :

Éditez la classe ⇒ sélectionnez la classe dans la fenêtre d'édition  
⇒ Show Context Actions ⇒ Create Test



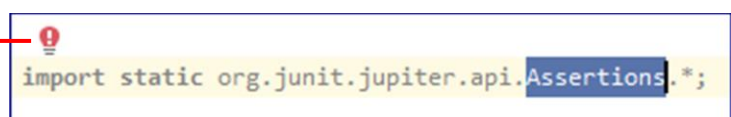
Downloadez si nécessaire la librairie JUnit :



Dans les classes de test, la classe *Assertions* sera utilisée.

Son import nécessite peut-être un ajout dans le classpath du projet.

Add library JUnit5 to classpath



L'écriture des tests unitaires en JUnit5 fait appel aux annotations.

---

## Introduction aux annotations en Java

---

Une annotation est un meta-tag ajouté au code.

Il s'agit de programmation déclarative dans laquelle le programmeur signale ce qu'il veut faire et des outils génèrent le code correspondant.

Le compilateur ou la machine virtuelle extraient, à partir des annotations, des informations sur le comportement du programme.

Une annotation est appelée via @ suivi du nom de l'annotation, le tout placé devant le code à annoter.

Les parties de code qui peuvent être annotées sont :

- un package
- une classe, une interface, une annotation...
- une variable d'instance
- un constructeur
- une méthode
- une variable
- un argument

Les annotations peuvent recevoir ou non un ou plusieurs arguments (éléments).

En fonction du nombre d'arguments/éléments, on distingue trois types d'annotations :

- Marker : sans élément

*Exemple :*

```
@Override  
public String toString()
```

- Single-element annotation

*Exemple :*

```
@SuppressWarnings(value = "unchecked")  
public void methodX() { ... }
```

- Multi-value annotation

*Exemple :*

```
@JoinColumn(name="category", referencedColumnName = "id")  
private CategoryEntity category;
```

**Pour plus d'informations sur les annotations, référez-vous au module 11 sur les annotations disponible sur Moodle.**

---

## Annotations en JUnit

---

Quelques annotations de méthodes utiles en JUnit :

- **@BeforeEach**  
⇒ La méthode sera exécutée avant chaque méthode test de la classe
- **@AfterEach**  
⇒ La méthode sera exécutée après chaque méthode test de la classe
- **@BeforeAll**  
⇒ La méthode sera exécutée une seule fois avant toutes les méthodes test de la classe (la méthode doit être déclarée static)
- **@AfterAll**  
⇒ La méthode sera exécutée une seule fois après toutes les méthodes test de la classe (la méthode doit être déclarée static)
- **@Test**  
⇒ Méthode de test

---

## La classe Assertions

---

La classe *Assertions* contient des méthodes (déclarées *static*) qui permettent de comparer les valeurs attendues avec les valeurs réelles lors de l'exécution des méthodes à tester, ce qui permet de valider ou non les tests unitaires.

Exemples de méthodes de la classe *Assertions* :

- *assertEquals()*
- *assertNotEquals()*
- *assertArrayEquals()*
- *assertNull()*
- *assertNotNull()*
- *assertSame()*
- *assertNotSame()*
- *assertTrue()*
- *assertFalse()*
- *assertThrows()*
- *fail()*

---

## Classe de tests unitaires

---

Soit la classe *CalculatorTest* qui contient plusieurs tests unitaires sur des méthodes de la classe *Calculator*.

La méthode *setUp* est annotée *@BeforeEach*, ce qui implique qu'elle sera exécutée une fois avant chaque méthode de test. Dans cette méthode *setUp*, on instancie un nouvel objet de type *Calculator*.

Les méthodes *add()*, *subtract()* et *multiply()* sont des méthodes de tests unitaires car elles sont annotées *@Test*. Ces méthodes font appel à *assertEquals* qui permet de vérifier que le résultat de l'appel d'une méthode sur l'objet *calculator* correspond bien à la valeur attendue. Comme on manipule des variables de type *double*, on admet un delta.

```
import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.junit.jupiter.api.Assertions.assertThrows;

public class CalculatorTest {
    private Calculator calculator;

    @BeforeEach
    public void setUp() {
        calculator = new Calculator();
    }

    @Test
    public void add() {
        assertEquals( expected: 1000.0, calculator.add(900.0, 100.0), delta: 0.01);
    }

    @Test
    public void subtract() {
        assertEquals( expected: 800.0, calculator.subtract(900.0, 100.0), delta: 0.01);
    }

    @Test
    public void multiply() {
        assertEquals( expected: 200.0, calculator.multiply(100.0, 2.0), delta: 0.01);
    }
}
```

Comment créer des tests unitaires sur des méthodes susceptibles de lancer des exceptions ?

⇒ On propage l'exception.

Comment tester que les exceptions sont bien lancées en cas d'erreur ?

⇒ Il faut utiliser `assertThrows()`

N.B. `assertThrows()` utilise la notation Java des expressions lambda

---

## Introduction aux expressions lambda en Java

---

Une expression lambda est une fonction qui peut être créée sans appartenir à une classe. Elle ressemble à une déclaration de méthode sans nom.

Elle peut être passée comme un objet et être exécutée à la demande

### Syntaxe :

Liste d'arguments ou `()` -> expression ou `{ instructions }`

*Exemples d'expressions lambda en Java :*

`(a,b,c) -> (a+b)*c`

`(book1, book2) ->`

`{ if (book1.getPagesCount() > book2.getPagesCount()) return book1;  
 else return book2; }`

`p -> p.getGender() == Person.Sex.MALE && p.getAge() >= 18  
 && p.getAge() <= 25`

`email -> System.out.println(email)`

### Où utiliser des expressions lambda ?

Une expression lambda peut s'utiliser là où on attend le code d'une méthode.


Une interface qui ne comprend la déclaration que d'une seule méthode est appelée une **interface fonctionnelle**.

Par conséquent, une expression lambda peut être utilisée là où une interface fonctionnelle est déclarée comme argument.

Exemple :

```
public interface OperationInterface {  
    int operation (int a, int b);  
}
```

→ Interface fonctionnelle



```

public class Tool {
    public int calculate(int a, int b, OperationInterface operator) {
        return operator.operation(a, b);
    }
}

```

```

public static void main(String[] args) {
    Tool tool = new Tool();
    System.out.println("40 + 2 = " + tool.calculate ( a: 40, b: 2, (a, b) -> a + b ));
    System.out.println("20 - 10 = " + tool.calculate ( a: 20, b: 10, (a, b) -> a - b ));
}

```

**Les expressions lambda sont quelque peu détaillées dans le module 12 disponible sur Moodle.**

Exemples de test unitaire sur la méthode *divide* qui est susceptible de lever une exception :

```

@Test
public void divide() throws DivisionException {
    assertEquals( expected: 50.0, calculator.divide(100.0, 2.0), delta: 0.01);
}

@Test
public void divideException() {
    assertThrows(DivisionException.class, () -> {
        calculator.divide(900.0, 0.0);
    });
}

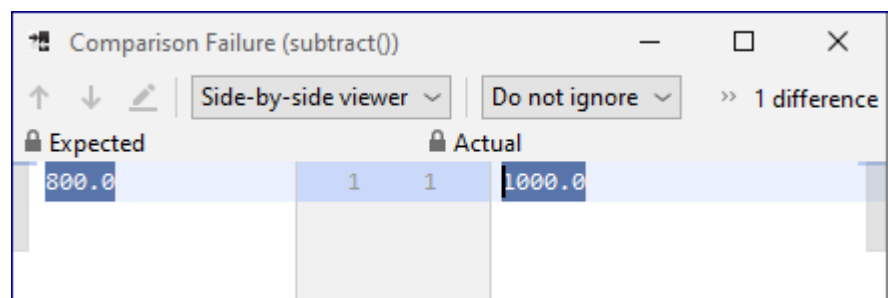
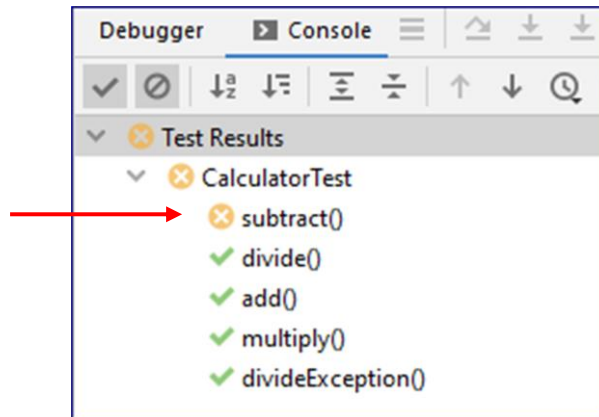
```



## Exécuter les tests unitaires

Exécutez la classe *CalculatorTest*.

Les tests unitaires qui réussissent sont précédés de V, ceux qui échouent sont précédés de X.



Le test unitaire *subtract* nous permet d'identifier une erreur dans le code. En effet, le programmeur a utilisé l'opérateur '+' au lieu de l'opérateur '-'.