



# LES TESTS.

Dans le développement logiciel et en Java

---

Concepteur Développeur d'Applications - 2025 - JBO\_v2



A decorative graphic on the left side of the slide. It consists of a large light green rounded square, a smaller dark green circle, and a purple semi-circle. The text is positioned to the right of these shapes.

# Les différents types de tests

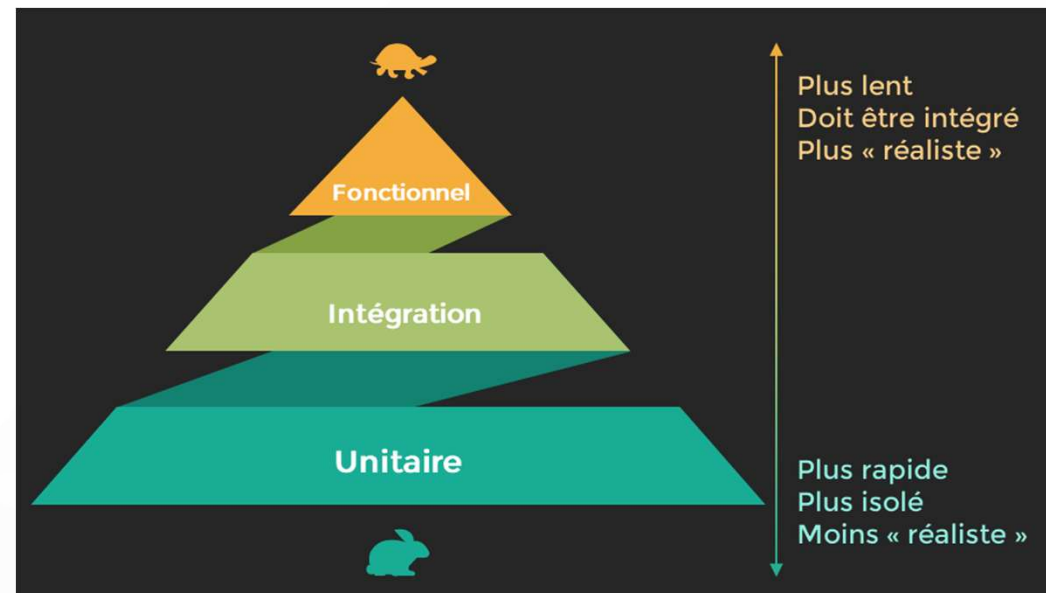
Comparez différents types de tests logiciels, tels que les tests unitaires, les tests d'intégration, les tests fonctionnels, les tests d'acceptation, et plus encore !

## Les différents types de tests

Les 3 types de tests automatisés les plus courants sont les tests unitaires, d'intégration et fonctionnels.

- Mike Cohn, l'un des fondateurs du mouvement Agile, présente ces types de tests sous forme de pyramide des tests, en particulier dans le livre [Succeeding with Agile](#) (en anglais).

En utilisant cette métaphore, l'auteur préconise la répartition de ces types de tests dans un projet de développement agile selon la pyramide ci-contre.



# Les différents types de tests

L'illustration indique qu'il y a davantage de tests rapides et autonomes à la base de la pyramide, puis de moins en moins à mesure que l'on grimpe vers le sommet.

Plus on approche du sommet, plus les tests sont longs et complexes à exécuter, tout en simulant des scénarios de plus en plus réalistes.

Définissons chacun de ces tests !

- **Tests unitaires** : La plus grosse section, à la base de la pyramide, est constituée des tests unitaires qui testent de "petites" unités de code. Plus précisément, ils testent que chaque fonctionnalité extraite de manière isolée se comporte comme attendu.
  - Ils sont très rapides et faciles à exécuter.
  - Si vous avez cassé quelque chose, vous pouvez le découvrir vite et tôt.
  - De bons tests unitaires sont **stables**, c'est-à-dire que le code de ces tests n'a pas besoin d'être modifié, même si le code de l'application change pour des raisons purement techniques.
  - Ils deviennent donc **rentables**, car ils ont été écrits une seule fois, mais exécutés de nombreuses fois.

# Les différents types de tests

Suite...

- **Tests d'intégration** : Au milieu de la pyramide se trouvent les tests d'intégration. Ils vérifient si vos unités de code fonctionnent ensemble comme prévu – **en présupposant que vos tests unitaires soient passés** ! Comme les tests d'intégration vérifient les interactions entre les unités, vous avez plus de certitude concernant le bon fonctionnement de l'application finale.
  - Ils peuvent nécessiter l'exécution de composants extérieurs (base de données, services web externe, etc.).
  - Le lancement de ces composants et l'interaction entre vos unités de code développées rendent ces types de test plus lents et potentiellement moins stables.
  - Mais vous simulez des scénarios plus proches de l'utilisation finale de l'application.

# Les différents types de tests

...

- Tests fonctionnels : (appelés **end-to-end** en anglais), visent à simuler le comportement d'un utilisateur final sur l'application, depuis l'interface utilisateur.
- L'ensemble du code développé est pris comme une boîte noire à tester, sans connaissance des briques et des unités de code qui la composent.
- Les simulations obtenues sont donc les plus proches de l'application utilisée dans des conditions réelles.
  - Ces tests nécessitent toute l'infrastructure nécessaire à l'application. Ces types de tests sont les plus lents à exécuter, et testent une partie beaucoup plus grande de votre code développé.
  - Cela crée une plus forte dépendance qui rend vos tests moins stables, donc moins rentables. Potentiellement, une modification simple de l'interface utilisateur (la couleur d'un bouton) pourrait nécessiter de recoder le test fonctionnel associé.



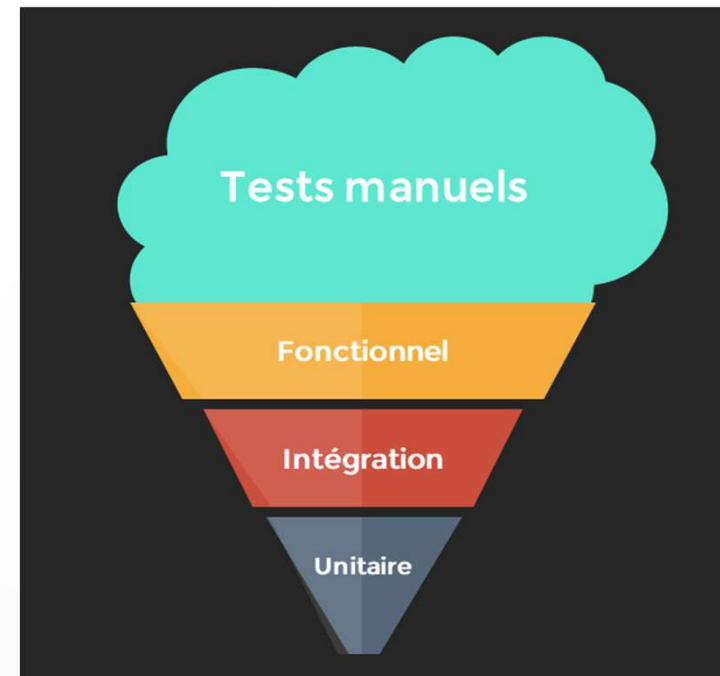
# Mauvaises pratiques

Malheureusement, ces préconisations agiles, datant de plus de dix ans, ne sont pas toujours suivies.

- En effet, de nombreuses équipes de développement, soumises à des contraintes fortes de délais et de coûts, ont développé de mauvaises pratiques de tests.

La plus connue est sûrement la mauvaise pratique (ou anti-pattern en anglais) du cône de glace de tests.

Elle vient souvent du fait qu'un projet ait été développé sans tests unitaires au début. Puis, pour rattraper la situation, les développeurs codent des tests fonctionnels et d'intégration, qui couvrent le plus de code possible en peu de temps de développement.




# Pourquoi tester ?

- Dans la vie d'un projet, pour répondre au besoin de maintenabilité, il faut pouvoir éviter toute **non-régression**. (dysfonctionnement d'une ancienne fonctionnalité développé à la suite de l'ajout d'une nouvelle fonctionnalité).
- Le seul moyen de garantir la **non-régression** passe par l'automatisation de l'exécution des procédures de tests. (**scénarios de tests**).
- Pour faciliter ce point, on opte pour l'utilisation de **Test Runner** (JUnit, PHPUnit etc..)
- **Testez pour faire face à l'inattendu**
  - Le risque zéro n'existe pas. Réduire le risque de combinaisons de scénarios de tests permet d'éviter trop de dépendances entre services
  - Plus les tests sont unitaires, moins le risques de combinaisons de scénarios alternatifs existent.
- **Testez pour faciliter la maintenance**
  - Ces tests vont grandement faciliter la correction et la maintenance du code.
- **Testez pour communiquer**
  - La lecture de vos tests doit permettre de comprendre non seulement ce que votre code est censé faire, mais aussi comment il fonctionne et à quel point il est opérationnel.





A decorative graphic on the left side of the slide. It consists of a large light green rounded rectangle, a smaller dark green circle, and a purple semi-circle. The text is positioned to the right of these shapes.

**ISTQB** Comité international de qualification du test logiciel

Être testeurs est un métier comme être développeur. Pour cela, des organisations mettent en place des bonnes pratiques et des certifications.

# ISTQB

L'ISTQB (en anglais : [International Software Testing Qualifications Board](#)) est le Comité international de qualification du test logiciel.

- Cette organisation délivre des certifications reconnues dans le monde entier.

Elle a été fondée en novembre 2002 à Édimbourg, comme association à but non lucratif, et est légalement enregistrée en Belgique.

<https://fr.wikipedia.org/wiki/ISTQB>

- L'ISTQB propose de valider le titre de testeur certifié ISTQB, une qualification standardisée pour le test logiciel.
- Les qualifications validées sont hiérarchisées et suivent des directives d'accréditation et d'examen.
- En janvier 2021, plus de 1 000 000 certifications ont été délivrées par l'ISTQB. Le comité de l'ISTQB est composé de 47 membres représentant plus de 71 pays.
- <https://www.istqb.org/>

## Le CFTL

Le [Comité français des tests logiciels](#) (CFTL) est un organisme français qui représente le métier du test logiciel et travaille sur les normes de ce métier.

- Il correspond à l'ISTQB anglais, qui édite des normes et propose des certifications.
- Le CFTL lui-même organise des passages de la certification ISTQB.
- Des consultants en qualification logicielle ou des entreprises peuvent être adhérents au CFTL.
- <https://cftl.fr/>



A decorative graphic on the left side of the slide. It consists of a large light green rounded rectangle, a smaller dark green circle, and a purple semi-circle. The text is positioned to the right of these shapes.

## Méthode de tests

Dans le développement logiciel, les patterns nous accompagnent pour coder efficacement. Dans les tests, les patterns permettent l'écriture de tests efficaces et lisibles

# Démarche du développement pilotés par les tests.

- Il s'agit de votre application. Plus précisément, on s'intéresse d'abord aux tests unitaires. Donc nous devons identifier la fonctionnalité à tester, qui sera implémentée en Java sous la forme d'une ou plusieurs classes.
- C'est le système à tester ou **SUT** (system under test).
- Le système à tester donne des résultats (ou plus généralement des sortants) à partir de données ou de paramètres de test (ou plus généralement des entrants).
- Puis, en vérifiant les résultats, le test est déclaré **en succès** ou **en échec**.

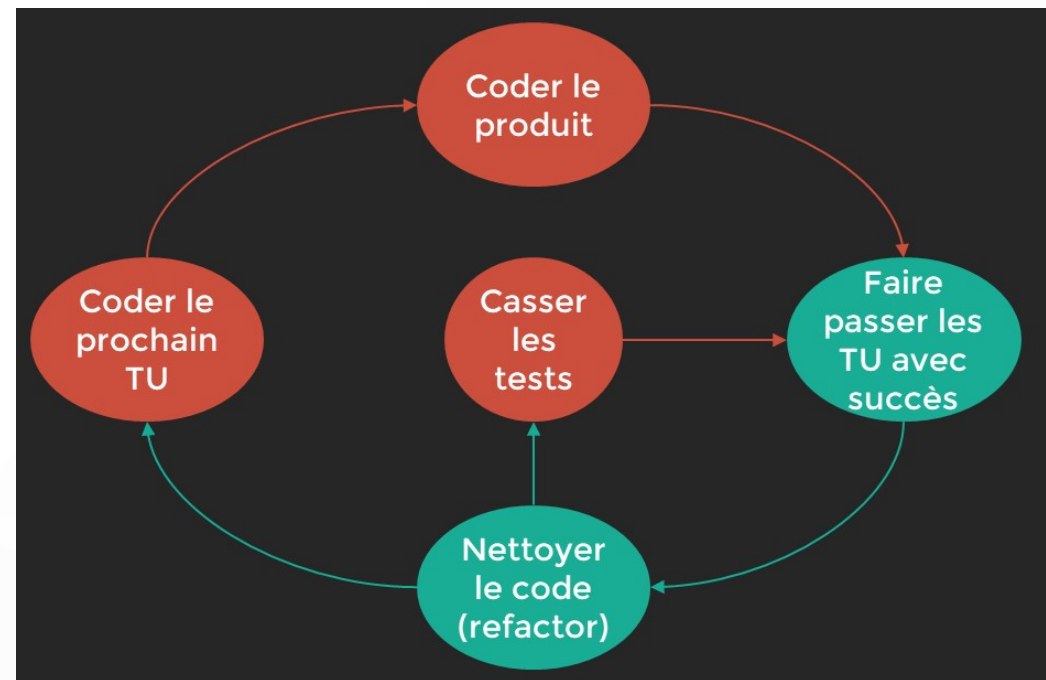


# Test Driven Development

Le TDD, pour **test-driven development** en anglais, ou développement piloté par les tests, consiste à ce que le code de votre application suive un plan fixé par les tests.

- *Souvent, on a plutôt tendance à faire l'inverse, coder l'application, puis la tester de manière manuelle ou automatique mais...*

Mais en codant d'abord le test, vous vous demandez directement quel objectif doit accomplir le code de votre application. Vous allez coder ce qui est nécessaire, pas plus, et ce code répondra au besoin exprimé clairement par le test.



# TDD : RED-GREEN-REFACTOR

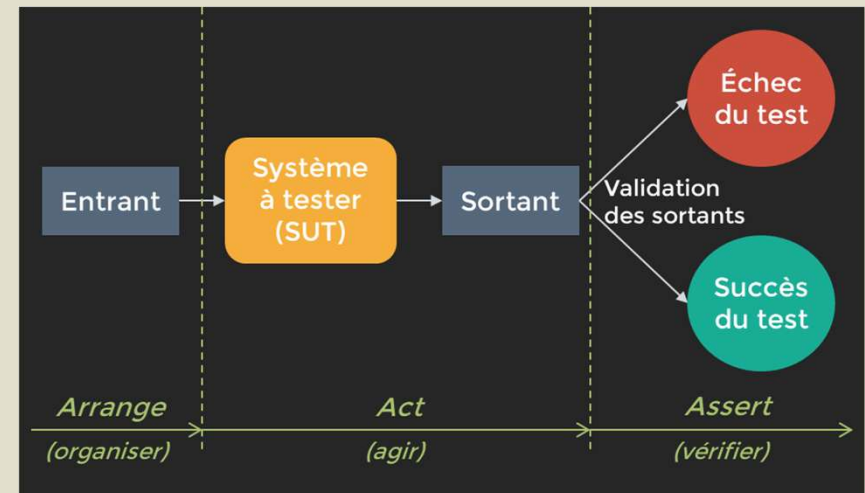
**Kent Beck**, l'inventeur du TDD, a fait découvrir à de nombreux développeurs le modèle suivant, appelé **red-green-refactor**

- Dans ce modèle, vous répétez cycliquement les étapes suivantes :
  - Écrivez un test unitaire qui échoue. ●
  - Écrivez le code qui permet de réussir le test. ✓
  - Nettoyez le code tout en gardant les tests en succès. ◆
  - Écrivez le prochain test et recommencez ! ↺
- Les tests qui échouent sont décrits comme rouges. Comme dans les feux de circulation, le rouge vous dit de vous arrêter et de faire fonctionner votre code. 🚦
- Quand le test est réussi, on passe au vert. Le vert vous dit de faire du refactoring. Cela signifie simplement que vous essayez de rendre votre code plus lisible et/ou plus élégant sans changer son comportement.
- Étant donné que le test est déjà en place, il vous dira immédiatement si vous cassez le moindre comportement, garantissant que vous êtes toujours concentré sur la fonctionnalité à tester en priorité.



# Pattern Arrange-Act-Assert

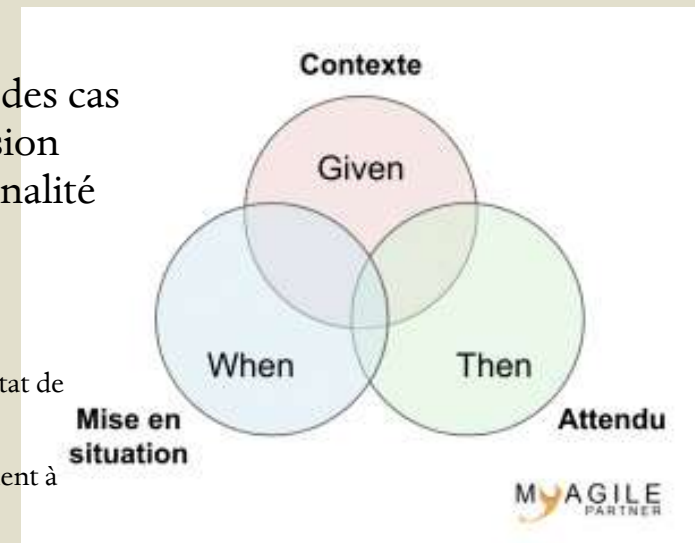
- **Arrange/Act/Assert (AAA)** est un modèle d'organisation des tests unitaires. Il décompose les tests en trois étapes claires et distinctes :
  - **Arrange** : Effectuez la configuration et l'initialisation requises pour le test.
  - **Act** : prendre les mesures requises pour le test.
  - **Assert** : Vérifiez-le(s) résultat(s) du test.
- Cela fait partie des bonnes pratiques pour l'écriture de tests en séparant clairement la configuration, les opérations et le résultat d'un test.
- Cela rend le code plus facile à lire et à comprendre.





# Pattern Given When Then

- **Given When Then** est un format structuré pour définir et organiser des cas de test. Il aide les testeurs et les développeurs à créer une compréhension commune du comportement attendu d'un système ou d'une fonctionnalité logicielle.
- Le cadre se compose de trois parties :
  - **Given** : cette section établit le contexte initial ou les préconditions du cas de test. Elle définit l'état de départ du système ou de la fonctionnalité testée.
  - **When** : dans cette partie, les actions ou événements spécifiques qui déclenchent le comportement à tester sont décrits. Elle définit les interactions de l'utilisateur ou les entrées du système.
  - **Then** : la section « **Then** » décrit les résultats attendus ou les résultats du cas de test. Elle spécifie ce qui devrait se produire en conséquence des actions prises dans la section « **When** ».



# Pattern Given When Then

- Le cadre GWT offre plusieurs avantages dans le contexte des tests logiciels :
  - **Clarté** : les cas de test **GWT** sont rédigés dans un langage simple et sont faciles à comprendre, même pour les parties prenantes non techniques. Cette clarté garantit que tous les intervenants impliqués dans le processus de test ont une compréhension commune de ce qui est testé.
  - **Consistance** : **GWT** encourage un format cohérent pour la conception des cas de test, ce qui facilite leur gestion et leur exécution au sein de différentes équipes et projets.
  - **Traçabilité** : chaque partie du format GWT fournit un lien de traçabilité clair. Les testeurs peuvent remonter des résultats de la section « **Then** » aux conditions et actions spécifiques des sections « **Given** » et « **When** ».
  - **Isolation** : Le format GWT encourage la division des cas de test en scénarios plus petits et plus ciblés. Cela permet une meilleure isolation des problèmes lorsque des défauts sont découverts.





# Qualité des tests

Écrire des tests, c'est aussi y mettre de la qualité dans son écriture comme notre code.

# Les principes F.I.R.S.T

Principes pour écrire des tests de qualités

- **F pour fast**
  - Qu'appelle-t-on rapide ? Vous devriez viser de nombreuses centaines ou milliers de tests par seconde. ⚡ Cela vous semble trop ? Pas si chaque test unitaire ne teste qu'une classe.
- **I pour isolé et indépendant**
  - Un problème ==> une cause du problème ==> une solution. Les principes AAA ou GIVEN/WHEN/THEN peuvent vous aider ici.
- **R pour répétable**
  - Si vous écrivez un test qui vous donne la confiance nécessaire en votre code, il doit vous dire la même chose, peu importe où, ou combien de fois, vous l'exécutez.



# Les principes F.I.R.S.T

Principes pour écrire des tests de qualités

- **S pour self-validating (autovalidation)**
  - Cela signifie en fait que l'exécution de vos tests ne laisse aucun doute sur leur succès ou leur échec. [JUnit](#) accomplit cela et échoue en rouge, ce qui vous laisse faire le rouge-vert-refactor.
- **T pour thorough**
  - Aujourd'hui, on considère souvent que le T signifie approfondi (« thorough »), c'est-à-dire que votre code est testé largement pour des cas négatifs et positifs. Étant donné que la meilleure manière d'écrire des tests approfondis est de s'assurer d'avoir écrit du code largement testable, ces deux aspects tendent vers le même résultat.





# TESTS UNITAIRES

Quelle est la différence entre un bon et un mauvais codeur ?

# JUnit

## Pourquoi tester ?

– Rappelez-vous qu'un test :

1. Doit vérifier qu'une seule exigence à la fois - les autres exigences seront vérifiées par d'autres tests.
  1. On entend par exigence un aspect à garantir par le Logiciel. Il peut s'agir d'une exigence fonctionnelle ou d'une exigence technique.
2. Doit produire un résultat mesurable - l'objectif étant de vérifier la conformité des résultats aux attendus.
3. Doit être le plus rapide possible, à l'exception de certains tests techniques (tests de robustesse notamment).

# JUNIT

## Présentation

- Framework permettant l'écriture et l'exécution de tests automatisés. [JUnit 5](#), version publiée depuis 2017.
- Basiquement, il suffit d'écrire une classe contenant des méthodes annotées avec [@Test](#).
- JUnit 5 apporte son lot de nouvelles fonctionnalités :
  - Les tests imbriqués,
  - Les tests dynamiques,
  - Les tests paramétrés qui offrent différentes sources de données.
  - Un nouveau modèle d'extension
  - L'injection d'instances en paramètres des méthodes de tests.

## Architecture de JUnit5

- [JUnit 5](#) est livré sous la forme de différents modules :
  - L'API,
  - Le moteur d'exécution,
  - L'exécution et intégration.
- La version 5 est composé de 3 sous-projets :
  - [JUnit Platform](#) : API permettant aux outils de découvrir et exécuter des tests.
  - [JUnit Jupiter](#) : API reposant sur des annotations pour écrire des tests unitaires et un [TestEngine](#) pour les exécuter.
  - [JUnit Vintage](#) : pour l'exécution des tests des anciennes versions de JUnit 3 et 4.





# Les ANNOTATIONS

Annotation	Rôle
@Test	La méthode annotée est un cas de test.
@ParameterizedTest	La méthode annotée est un cas de test paramétré
@RepeatedTest	La méthode annotée est un cas de test répété
@TestFactory	La méthode annotée est une fabrique pour des tests dynamiques
@TestInstance	Configurer le cycle de vie des instances de tests
@TestTemplate	La méthode est un modèle pour des cas de tests à exécution multiple
@DisplayName	Définir un libellé pour la classe ou la méthode de test annotée
@BeforeEach	La méthode annotée sera invoquée avant l'exécution de chaque méthode de la classe annotée avec @Test, @RepeatedTest, @ParameterizedTest ou @Testfactory.
@AfterEach	La méthode annotée sera invoquée après l'exécution de chaque méthode de la classe annotée avec @Test, @RepeatedTest, @ParameterizedTest ou @Testfactory.



# LES ANNOTATIONS

Annotation	Rôle
@BeforeAll	La méthode annotée sera invoquée avant l'exécution de la première méthode de la classe annotée avec @Test, @RepeatedTest, @ParameterizedTest ou @Testfactory.
@AfterAll	La méthode annotée sera invoquée après l'exécution de toutes les méthodes de la classe annotées avec @Test, @RepeatedTest, @ParameterizedTest et @Testfactory.
@Nested	Indiquer que la classe annotée correspond à un test imbriqué
@Tag	Définir une balise sur une classe ou une méthode qui permettra de filtrer les tests exécutés.
@Disabled	Désactiver les tests de la classe ou la méthode annotée.
@ExtendWith	Enregistrer une extension



# Les annotations

Les méthodes annotées avec `@Test`, `@TestTemplate`, `@RepeatedTest`, `@BeforeAll`, `@AfterAll`, `@BeforeEach` ou `@AfterEach` ne doivent pas retourner de valeur.

Les annotations de JUnit Jupiter peuvent être utilisées comme méta-annotation : il est possible de définir des annotations, elles-mêmes annotées avec ces annotations pour qu'elles héritent de leurs caractéristiques.

```
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;
import org.junit.jupiter.api.Tag;
import org.junit.jupiter.api.Test;

@Target({ ElementType.TYPE, ElementType.METHOD })
@Retention(RetentionPolicy.RUNTIME)
@Tag("Mon_Tag")
@Test
public @interface TestAvecMonTag {
}
```

## L'écriture de tests standard

Basiquement, il faut écrire une classe contenant des méthodes annotées pour implémenter les cas de tests ou le cycle de vie des tests.

Les méthodes qui implémentent des cas de tests utilisent des assertions pour faire les vérifications requises.

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

public class MonTest {

    @Test
    public void simpleTest() {
        System.out.println("simpleTest");
        Assertions.assertTrue(true);
    }
}
```



# L'écriture de tests standard

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

@DisplayName("Ma classe de test JUnit5")
public class MonTest {

    @Test
    @DisplayName("Mon cas de test")
    void premierTest() {
        // ...
    }
}
```

## La définition d'un libellé

- Possibilité d'avoir un libellé qui sera affiché par les tests runners ou dans le rapport d'exécutions

## Le cycle de vie des tests

- Une classe de test JUnit peut avoir des méthodes annotées pour définir des actions exécutées durant le cycle de vie des tests.
- `@BeforeAll` : exécutée une seule fois avant l'exécution du premier test de la classe. [doit être static]
- `@BeforeEach` : exécutée avant chaque méthode de tests
- `@AfterEach` : exécutée après chaque méthode de tests
- `@AfterAll` : exécutée une seule fois après l'exécution de tous les tests de la classe [doit être static]



# Les assertions

Les assertions classiques permettent de faire des vérifications sur une instance ou une valeur ou effectuer des comparaisons.

Egalité	Nullité	Exceptions
assertEquals()	assertNull()	assertThrows()
assertNotEquals()	assertNotNull()	
assertTrue()		
assertFalse()		
assertSame()		
assertNotSame()		

```
import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.junit.jupiter.api.Assertions.assertNotSame;
import static org.junit.jupiter.api.Assertions.assertTrue;

import java.util.Arrays;
import java.util.LinkedList;
import java.util.List;
import org.junit.jupiter.api.Test;

public class PremiereClasseTest {

    @Test
    void monPremierTest() {
        assertTrue(true);
        assertTrue(this::isValide);
        assertTrue(true, () -> "Description " + "du cas " + "de test");
        List<String> attendu = Arrays.asList("e1", "e2", "e2");
        List<String> actual = new LinkedList<>(attendu);
        assertEquals(attendu, actual);
        assertEquals(attendu, actual, "Les listes ne sont pas égales");
        assertEquals(attendu, actual, () -> "Les listes " + "ne sont " + "pas égales");
        assertNotSame(attendu, actual, "Les instances sont les memes");
    }

    boolean isValide() {
        return true;
    }
}
```

# Les assertions

assertAll

- permet de regrouper plusieurs assertions qui seront toutes exécutées.

assertArrayEquals

- L'assertion `assertArrayEquals` vérifie que deux tableaux sont égaux.

assertIterableEquals

- permet de vérifier que deux Iterables sont égaux de manière profonde, ce qui implique plusieurs vérifications :
  - le nombre des éléments, l'ordre des éléments, égalité des éléments

assertLinesMatch

- vérifie que les éléments d'une `List<String>` sont en correspondance avec une autre `List<String>`. Cette assertion est un cas spécifique de comparaison de collections.


assertTimeout  
assertTimeoutPreemptively

- vérifie que les traitements fournis en paramètre s'exécutent avant le délai précisé. La différence entre les deux est que `assertTimeoutPreemptively` interrompt l'exécution des traitements si le délai est dépassé.

fail

- permet de faire échouer le test en levant une exception de type `AssertionFailedError`.





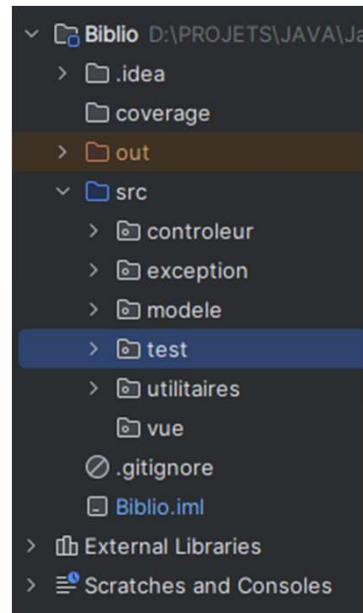
# Création d'une classe de test depuis l'IDE IntelliJ



## Créer une classe de test

Tout d'abord, avant tout, il faut créer un package `test` où sera regroupé toutes nos classes de tests.

Ce package se trouvera au même niveau que notre dossier `src`

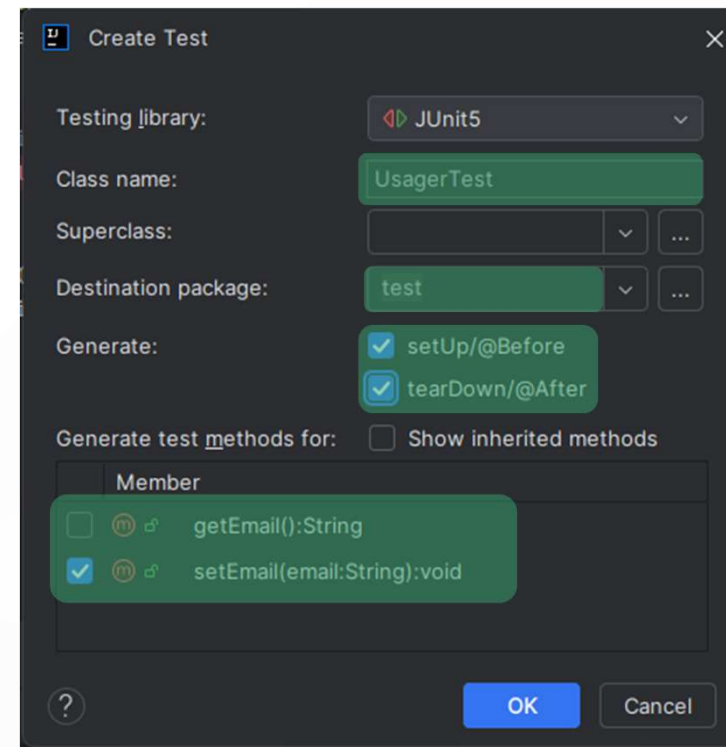
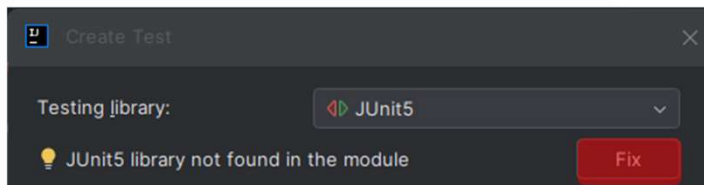


- Ensuite aller dans la classe dont vous voulez tester
- Sélectionner le nom de la classe
- Clic droit => [Generate...](#)
- Sélectionner [Test...](#)

## Créer une classe de test

Une nouvelle fenêtre apparaît pour vous permettre de créer votre classe de tests.

- A la 1ère création de la classe test, JUNIT5 ne sera pas intégré à votre projet. Il suffit de cliquer sur [fix](#) pour que l'IDE fasse l'intégration.



# Convention de nommage

Il y a dans Java une convention à appliquer dans l'écriture des tests.

Cela évite d'aller dans le code du test pour savoir quelle nature du test a échoué

- Il existe des conventions selon ce que vous appliquez comme pattern

- La classe de test se nomme
  - `NomDeClasseTest`
- Les méthodes de test prennent comme nom une description du test
  - `testSetNomWithNull()`
  - `testSetNomWithEmpty()`
- Avec le pattern Given-When-Then, les méthodes peuvent prendre cette convention :
  - `should_ComportementAttendu_when_EtatTesté`
  - `Should_return_success_when_addition_two_numbers`





# Tests paramétrés

Jouer avec les entrants et les sortants grâce aux tests paramétrés

# Quelques exemples

Voici quelques exemples d'annotations utiles pour l'écriture de tests.

- `@ParameterizedTest`
  - Permet de créer des tests paramétrés.
  - `@ValueSource` accepte tous les types primitifs de Java

```
@ParameterizedTest(name = "{0} x 0 doit être égal à 0")
@ValueSource(ints = { 1, 2, 42, 1011, 5089 })
public void test_Multiply_shouldReturnZero_ofZeroWithMultipleIntegers(int arg) {
    // Arrange -- Tout est prêt !

    // Act -- Multiplier par zéro
    int actualResult = caculatorUnderTest.multiply(arg, 0);

    // Assert -- ça vaut toujours zéro !
    assertEquals(0, actualResult);
}
```

CalculatorTest [Runner: JUnit 5] (0,047 s)

- testAddTwoPositiveNumbers() (0,005 s)
- test\_Multiply\_shouldReturnZero\_ofZeroWithMultipleIntegers(int) (0,014 s)
  - 1 x 0 doit être égal à 0 (0,014 s)
  - 2 x 0 doit être égal à 0 (0,001 s)
  - 42 x 0 doit être égal à 0 (0,000 s)
  - 1011 x 0 doit être égal à 0 (0,001 s)
  - 5089 x 0 doit être égal à 0 (0,001 s)
- testMultiplyTwoPositiveNumbers() (0,000 s)

## Quelques exemples

Voici quelques exemples d'annotations utiles pour l'écriture de tests.

- Vous pouvez utiliser `@CsvSource` à la place de `@ValueSource`.

```
@ParameterizedTest(name = "{0} + {1} should equal to {2}")
@CsvSource({ "1,1,2", "2,3,5", "42,57,99" })
public void test_add_shouldReturnTheSum_ofMultipleIntegers(
    int arg1, int arg2, int expectedResult) {
    // Arrange -- Tout est prêt !

    // Act
    int actualResult = calculatorUnderTest.add(arg1, arg2);

    // Assert
    assertEquals(expectedResult, actualResult);
}
```

```
✓ CalculatorTest [Runner: JUnit 5] (0,065 s)
  ✓ testAddTwoPositiveNumbers() (0,000 s)
  > test_Multiply_shouldReturnZero_ofZeroWithMultipleIntegers(int) (0,013 s)
  ✓ test_add_shouldReturnTheSum_ofMultipleIntegers(int, int, int) (0,002 s)
    ✓ 1 + 1 should equal to 2 (0,002 s)
    ✓ 2 + 3 should equal to 5 (0,000 s)
    ✓ 42 + 57 should equal to 99 (0,001 s)
  ✓ testMultiplyTwoPositiveNumbers() (0,001 s)
```

# Quelques exemples

Utilisation des annotations :

- `@NullSource`
- `@EmptySource`
- `@ValueSource`

```
class BookTest {  
  
    private Book bookUnderTest;  
  
    @BeforeEach  
    public void AppelAvantChaqueTest() {  
        System.out.println("Appel avant chaque test");  
        // avant chaque tests : initialisation d'une instance  
        bookUnderTest = new Book();  
    }  
  
    @ParameterizedTest  
    @NullSource  
    void testSetTitleWithNullValue(String title) {  
        //fail("Not yet implemented");  
  
        Exception e = assertThrows(Exception.class, () -> {  
            bookUnderTest.setTitle(title);  
        });  
        assertEquals("Erreur : Null", e.getMessage());  
    }  
  
    @ParameterizedTest  
    @EmptySource  
    void testSetTitleWithEmptyValue(String title) {  
        //fail("Not yet implemented");  
  
        Exception e = assertThrows(Exception.class, () -> {  
            bookUnderTest.setTitle(title);  
        });  
        assertEquals("Erreur : Empty", e.getMessage());  
    }  
  
    @ParameterizedTest(name = "testSetTitleWithBlankValue + {0}")  
    @ValueSource(strings = { " ", " ", " " })  
    void testSetTitleWithBlankValue(String title) {  
        //fail("Not yet implemented");  
  
        Exception e = assertThrows(Exception.class, () -> {  
            bookUnderTest.setTitle(title);  
        });  
        assertEquals("Erreur : Blank", e.getMessage());  
    }  
}
```

# Annotations utiles

## Désactiver un test

- L'annotation `@Disabled` peut être utilisée sur une méthode ou sur une classe

```
@Test
@Disabled("A écrire plus tard")
void monTest() {
    fail("Non implémenté");
}
```

## Les tags

Les classes et les méthodes de tests peuvent être **tagguées** pour permettre d'utiliser ces tags ultérieurement pour déterminer les tests à exécuter.

Ils peuvent par exemple être utilisés pour créer différents scénarios de tests ou pour exécuter les tests uniquement sur des environnements dédiés.

```
@Tag("TestsOpérations")
```



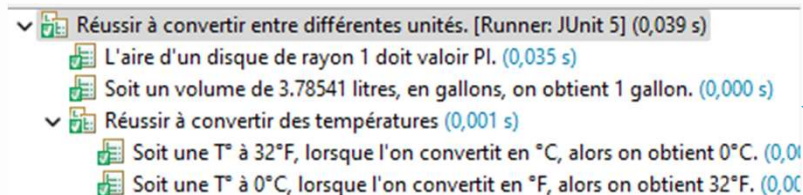


# Les tests imbriqués

Les tests imbriqués permettent de grouper des cas de test pour renforcer le lien qui existent entre-eux.

`@Nested` vous permet de grouper vos tests dans une classe interne.

Avec `@Nested`, si un seul test échoue, tout le groupe désigné par cette annotation échoue !



```
@Tag("ConversionTests")
@DisplayName("Réussir à convertir entre différentes unités.")
class ConversionCalculatorTest {

    private ConversionCalculator calculatorUnderTest = new ConversionCalculator();

    @Nested
    @Tag("TemperatureTests")
    @DisplayName("Réussir à convertir des températures")
    class TemperatureTests {

        @Test
        @DisplayName("Soit une T° à 0°C, lorsque l'on convertit en °F, alors on obtient 32°F.")
        public void celsiusToFahrenheit_returnsAFahrenheitTemperature_whenCelsiusIsZero() {
            Double actualFahrenheit = calculatorUnderTest.celsiusToFahrenheit(0.0);
            assertThat(actualFahrenheit).isCloseTo(32.0, withinPercentage(0.01));
        }

        @Test
        @DisplayName("Soit une T° à 32°F, lorsque l'on convertit en °C, alors on obtient 0°C.")
        public void fahrenheitToCelsius_returnsZeroCelsiusTemperature_whenThirtyTwo() {
            Double actualCelsius = calculatorUnderTest.fahrenheitToCelsius(32.0);
            assertThat(actualCelsius).isCloseTo(0.0, withinPercentage(0.01));
        }
    }

    @Test
    @DisplayName("Soit un volume de 3.78541 litres, en gallons, on obtient 1 gallon.")
    public void litresToGallons_returnsOneGallon_whenConvertingTheEquivalentLitres() {
        Double actualLitres = calculatorUnderTest.litresToGallons(3.78541);
        assertThat(actualLitres).isCloseTo(1.0, withinPercentage(0.01));
    }

    @Test
    @DisplayName("L'aire d'un disque de rayon 1 doit valoir PI.")
    public void radiusToAreaOfCircle_returnsPi_whenWeHaveARadiusOfOne() {
        Double actualArea = calculatorUnderTest.radiusToAreaOfCircle(1.0);
        assertThat(actualArea).isCloseTo(PI, withinPercentage(0.01));
    }
}
```



# Ecriture de cas de test

Pour nos classes Java, quelle est la démarche et réflexion pour écrire nos tests ?

# Comprendre les cas de test

L'écriture de cas de test est une pratique essentielle dans le développement logiciel pour garantir la qualité et la fiabilité du code.

- Les cas de test sont des scénarios définis pour vérifier le comportement d'une classe ou d'une méthode dans différentes conditions.
- Chaque cas de test doit être conçu pour tester une fonctionnalité spécifique de manière isolée, ce qui permet de détecter les erreurs et de valider le comportement attendu du code.



# Etapas pour écrire des cas de test

Pour écrire des cas de test, il suffit de suivre les étapes suivantes :

1. Identifier les scénarios de test
  - Avant d'écrire des cas de test, identifiez les différents scénarios à tester pour chaque méthode ou fonction de votre classe.
2. Écrire des méthodes de test
  - Créez des méthodes de test pour chaque scénario identifié. Utilisez les annotations `@Test` pour identifier les méthodes de test et ajoutez des assertions pour vérifier les résultats.
3. Configurer les conditions initiales
  - Utilisez les annotations `@Before` pour exécuter du code de configuration avant chaque méthode de test, par exemple, initialiser des objets nécessaires pour le test.
4. Nettoyer après les tests
  - Utilisez les annotations `@After` pour exécuter du code de nettoyage après chaque méthode de test, par exemple, libérer des ressources utilisées pour le test.

# Concentrez-vous sur le test des chemins critiques

Le concept de chemin critique en test unitaire fait référence à la séquence de chemins ou d'exécutions dans un programme qui sont essentiels au bon fonctionnement du logiciel.

- En termes simples, le chemin critique est le flux d'exécution dans votre code qui, s'il échoue, pourrait causer des dysfonctionnements majeurs ou empêcher l'application de fonctionner comme prévu.

1. Identification des chemins critiques :
  - **Composants centraux du système** : Identifiez les parties du code qui sont cruciales pour les fonctionnalités principales de votre application.
  - *Par exemple, dans une application bancaire, les opérations de transfert de fonds ou de calcul de soldes seraient des chemins critiques.*
  - **Détection des points de défaillance potentiels** : Si certaines parties du code ont plus de chances de contenir des bugs (en raison de leur complexité ou de leur importance), elles sont aussi considérées comme critiques.
  - **Flux d'exécution incontournable** : Toute séquence d'instructions qui doit obligatoirement être exécutée pour que l'application atteigne un certain état ou accomplisse une tâche spécifique.
2. Pourquoi tester les chemins critiques ?
  - **Minimiser les risques** : Tester ces chemins en priorité permet de réduire le risque de défauts dans les fonctionnalités essentielles.
  - **Maximiser la couverture avec un effort limité** : En se concentrant sur les chemins critiques, vous pouvez garantir que les parties les plus importantes de votre application sont fiables, même si vous ne couvrez pas tous les scénarios possibles.
  - **Performance et efficacité** : Les tests sur les chemins critiques aident à identifier les problèmes potentiels plus rapidement et à optimiser les performances dans les zones où cela compte le plus.

# Les chemins critiques ?

## Exemple concret

- Une application Calculatrice
  - La division par zéro par exemple
- Une application Bibliothèque
  - La validation dans nos setters permettant la cohérence de nos objets métiers.
- Une API de gestion de compte utilisateur
  - Vérification des identifiants de l'utilisateur.
  - Génération d'un token d'authentification.

## Stratégie de test

- **Décomposition du code** : Découpez votre code en différentes parties pour identifier les chemins critiques.
- **Création de cas de tests** : Créez des cas de test qui ciblent spécifiquement ces chemins critiques. Assurez-vous que chaque partie critique du code est testée pour différentes entrées, conditions, et états du système.



```
public class Calculator { no usages new *  
    public int add(int a, int b) { new *  
        return a + b;  
    }  
}
```

```
class CalculatorTest {  neojero  
  
    private Calculator calculatorUnderTest; 4 usages  
  
    @BeforeEach  neojero  
    void setUp() {  
        calculatorUnderTest = new Calculator();  
    }  
  
    @AfterEach  neojero  
    void tearDown() {  
        calculatorUnderTest = null;  
    }  
  
    @Test  neojero  
    @Tag("testAddWithTwoNumbers")  
    void testAddWithTwoNumbers() {  
        int result = calculatorUnderTest.add(a: 1, b: 2);  
        assertEquals(expected: 3, result);  
    }  
  
    @Test  neojero  
    void testSubtractWithTwoNumbers() {  
        int result = calculatorUnderTest.sub(a: 5, b: 3);  
        assertEquals(expected: 2, result);  
    }  
}
```

# Exemple pratique

Considérons la classe simple Calculator avec une méthode add pour effectuer une addition.

```
class CalculatorTest {  neojero *

    private Calculator calculatorUnderTest; 6 usages

    @BeforeEach  neojero
    void setUp() {
        calculatorUnderTest = new Calculator();
    }

    @AfterEach  neojero
    void tearDown() {
        calculatorUnderTest = null;
    }

    @Test  neojero
    @Tag("testAddWithTwoNumbers")
    void testAddWithTwoNumbers() {
        int result = calculatorUnderTest.add(a: 1, b: 2);
        assertEquals(expected: 3, result);
    }

    @Test  neojero
    void testSubtractWithTwoNumbers() {
        int result = calculatorUnderTest.sub(a: 5, b: 3);
        assertEquals(expected: 2, result);
    }
}
```

```
@ParameterizedTest(name = "{0} + {1} should equal to {2}") new *
@CsvSource({ "1,1,2", "2,3,5", "42,57,99" })
@Tag("TestWithMethodsAAA")
public void testAddShouldReturnTheSumOfMultipleIntegers(
    int arg1, int arg2, int expectedResult) {
    // Arrange -- Tout est prêt !

    // Act
    int actualResult = calculatorUnderTest.add(arg1, arg2);

    // Assert ou utilisation de AssertJ
    //assertEquals(expectedResult, actualResult);
    // utilisation de AssertJ
    assertThat(actualResult).isEqualTo(expectedResult);
}
```

```
@ParameterizedTest(name = "{index} => {0} : {1} divide by {2} is impossible") new *
@CsvSource(delimiter = '|', textBlock = """
    positive number | 2 | 0
    negatif number | -2 | 0
""")
public void testDivideWithZero(String description, int a, int b) {
    // ARRANGE : fait avec @Parameterized

    // ACT
    Exception exception = assertThrows(CalculException.class, () -> {
        calculatorUnderTest.divide(a,b);
    });

    // ASSERT
    assertEquals(exception.getMessage(), actual: "Division par zéro interdite");
}
```

```
✓ CalculatorTest (test)
  ✓ testSubtractWithTwoNumbers()
  ✓ testAddWithTwoNumbers()
  ✓ testDivideWithZero(String, int, int)
    ✓ 1 => positive number : 2 divide by 0 is impossible
    ✓ 2 => negatif number : -2 divide by 0 is impossible
  ✓ testAddShouldReturnTheSumOfMultipleIntegers(int, int, int)
    ✓ 1 + 1 should equal to 2
    ✓ 2 + 3 should equal to 5
```

# La classe CalculatorTest





## Autre exemple pratique

Considérons la classe simple [Book](#), un attribut isbn de type String et ses méthodes accesseur et mutateur.

Dans le setter pour l'attribut, nous avons 3 conditions de contrôles sur la donnée.

Mettons en place les différents tests pour cette méthode.

```
public String getIsbn() { no usages
    return isbn;
}

public void setIsbn(String isbn) { 4 usages
    // test valeur null
    if (isbn == null) {
        throw new NullPointerException("Isbn is null");
    }

    // test valeur vide
    if (isbn.isBlank() || isbn.isEmpty()) {
        throw new IllegalArgumentException("Isbn is empty or blank");
    }

    if (isbn.length() < 5) {
        throw new IllegalArgumentException("Isbn is too short");
    }

    this.isbn = isbn;
}
```

# La classe BookTest

```
class BookTest { new *  
  
    private Book bookUnderTest; 5 usages  
  
    @BeforeEach new *  
    void setUp() {  
        bookUnderTest = new Book( nom: "test", titre: "test", dispo: true, isbn: "02ab5");  
    }  
  
    @AfterEach new *  
    void tearDown() {  
        bookUnderTest = null;  
    }  
  
    @ParameterizedTest new *  
    @NullSource  
    void testSetIsbnWithNull(String isbn) {  
        Exception exception = assertThrows(NullPointerException.class, () -> {  
            bookUnderTest.setIsbn(isbn);  
        });  
        assertEquals(exception.getMessage(), actual: "Isbn is null");  
    }  
}
```

```
✓ BookTest (test)  
  > ✓ testSetIsbnWithNull(String)  
  > ✓ testSetIsbnWithEmpty(String)  
  ✓ testSetIsbnWithShortLength(String)  
    ✓ a is too short  
    ✓ a2 is too short  
    ✓ a20 is too short  
    ✓ a20s is too short
```

```
@ParameterizedTest new *  
@EmptySource  
void testSetIsbnWithEmpty(String isbn) {  
    Exception exception = assertThrows(IllegalArgumentException.class, () -> {  
        bookUnderTest.setIsbn(isbn);  
    });  
    assertEquals(exception.getMessage(), actual: "Isbn is empty or blank");  
}  
  
@ParameterizedTest(name = "{0} is too short") new *  
@ValueSource(strings = {"a", "a2", "a20", "a20s"})  
void testSetIsbnWithShortLength(String isbn) {  
    Exception exception = assertThrows(IllegalArgumentException.class, () -> {  
        bookUnderTest.setIsbn(isbn);  
    });  
    assertEquals(exception.getMessage(), actual: "Isbn is too short");  
}  
}
```





# Utilisation de bibliothèques

Certaines librairies vous permettent d'améliorer et donner du sens à vos tests.

# Coder des tests faciles à lire

## Assertions de bibliothèques tierces

Il est aussi possible d'utiliser d'autres bibliothèques d'assertions compatibles avec JUnit 5 telles que :

- [AssertJ](#)
- [Hamcrest](#)
- [Truth](#)

Cela permet d'étendre les possibilités d'assertions qu'offre JUnit 5, d'affiner vos assertions et rendre votre code de tests plus lisible.

## Les suppositions

- Les suppositions permettent de conditionner l'exécution de tout ou partie d'un cas de test.
- Elles peuvent interrompre un test (sans le faire échouer) si une condition est remplie ou peuvent conditionner l'exécution de certains traitements d'un test selon une condition.
  - `assumeTrue` permet d'exécuter la suite des traitements du test uniquement si la valeur booléenne fournie est true.
  - `assumeFalse` permet d'exécuter la suite des traitements du test uniquement si la valeur booléenne fournie est false.
  - `assumeThat` permet d'exécuter le traitement fourni uniquement si la valeur booléenne fournie est true.



## Pourquoi AssertJ ?

L'objectif est que vos tests expriment ce que doivent faire vos fonctionnalités.

Les tests sont des outils pour communiquer avec d'autres développeurs, mais aussi avec vous-même dans quelques jours ou quelques mois.

Si vous comprenez quel comportement est attendu, alors vous savez si ça fonctionne !

- La bibliothèque populaire AssertJ , utilisée par de nombreux projets Java, permet de rendre les assertions de tests un peu plus naturelles à lire que les assertions de JUnit.
- Elle permet de chaîner plusieurs vérifications pour en faire une plus complexe.

### Installation d'une librairie dans un projet IntelliJ

1. Depuis votre projet => Clic droit sur le nom du projet (ou ctrl+alt+shift+S)
2. Sélectionner Open Module Setting
3. Dans project Setting => Modules
4. Puis dans Dependencies
5. Cliquer sur + (ou alt+insert)
6. Sélectionner Jars and Directories et indiquer l'emplacement où se trouve votre fichier .jar

# JUnit vs AssertJ

Voici quelques exemples

Cas de test	Assertions JUnit	Assertions AssertJ
Un nom est compris entre 5 et 10 caractères.	<code>assertTrue(name.length &gt; 4 &amp;&amp; name.length &lt; 11);</code>	<code>assertThat(name).hasSizeGreaterThan(4).hasSizeLessThan(11);</code>
Un nom est situé dans la première moitié de l'alphabet	<code>assertTrue(name.compareTo("A") &gt;= 0 &amp;&amp; name.compareTo("M") &lt;= 0);</code>	<code>assertThat(name).isBetween("A", "M");</code>
Une date et heure locale se situent aujourd'hui ou dans le futur.	<code>assertTrue(dateTime.toLocalDate().isAfter(LocalDate.now())    dateTime.toLocalDate().isEqual(LocalDate.now()));</code>	<code>assertThat(dateTime.toLocalDate()).isAfterOrEqualTo(LocalDate.now());</code>

<https://assertj.github.io/doc/#assertj-core-assertions-guide>



# Utilisation

Pour utiliser AssertJ, il faut donc ajouter la dépendance dans votre projet, soit directement dans votre classpath du projet, soit avec Maven.

```
<dependencies>
  <dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter</artifactId>
    <version>5.5.1</version>
    <scope>test</scope>
  </dependency>
  <dependency>
    <groupId>org.assertj</groupId>
    <artifactId>assertj-core</artifactId>
    <version>3.13.2</version>
    <scope>test</scope>
  </dependency>
</dependencies>
```

Ceci est un exemple

```
@Test
void testMultiplyTwoPositiveNumbers() {
    //1er étape
    //fail("Not yet implemented");

    // 2eme etape écriture de mon test.
    // Arrange
    int a = 2;
    int b = 3;

    // Act
    //Calculator calculator = new Calculator();
    int calcul = calculatorUnderTest.multiply(a,b);

    // Assert
    //assertEquals(6, calcul);
    // utilisation de AssertJ
    assertThat(calcul).isEqualTo(6);
}
```



# Couverture de code

Contrôlez la couverture de vos tests et la qualité du code



# Couverture de code

Java propose de nombreux outils qui permettent de vous aider à effectuer ces types de vérification pour :

- améliorer la qualité du code avec le formatage, et la détection de code mal écrit, source de bugs potentiels.
- sécuriser votre produit en vérifiant les dépendances de votre code, notamment avec le projet OWASP.

- Un code est couvert par les tests s'il est exécuté par au moins un test.
- Avec la fonctionnalité [Coverage](#) de l'IDE, vous pouvez obtenir un état de la couverture de code de votre projet

Element	Coverage	Covered Instructions	Missed Instructions	Total Instructions
AppforTest	52,6 %	111	100	211
src	52,6 %	111	100	211
test	61,3 %	100	63	163
BookTest.java	0,0 %	0	63	63
CalculatorTest.java	100,0 %	100	0	100
principal	22,9 %	11	37	48
Book.java	0,0 %	0	33	33
App.java	0,0 %	0	4	4
Calculator.java	100,0 %	11	0	11

# Couverture de code

On peut identifier 4 façons principales de mesurer une quantité de code, en comptant

- Le nombre de lignes
- Le nombre d'instructions
- Le nombre de branches
- Le nombre de fonctions/méthodes.

Il n'y a pas de méthode de mesure parfaite. Chacune possède ses avantages et ses inconvénients. Parfois, il faudra prendre en compte plusieurs mesures.

- **le nombre de lignes** : c'est ce qui semble le plus intuitif. Mais est-ce le plus juste ? Évidemment, les outils de comptage vont éliminer les lignes inutiles, vides, ou de commentaires. Mais il est possible d'avoir des lignes plus complexes que d'autres ;
- **le nombre d'instructions** : pour prendre en compte la complexité d'une ligne, on peut compter toutes les instructions. Mais lorsqu'un développeur code des tests, il étudie les différents cas possibles, y compris les cas d'exception qui doivent être traités avec la même importance, même si les cas d'exception nécessitent souvent moins de code
- **le nombre de branches** : au lieu de compter linéairement le code, on va compter le nombre de zones de code par rapport à des conditions. Typiquement, un ensemble d'instructions if/else génère deux zones de code, que l'on appelle aussi branches. Cela permet de prendre en compte à parts égales les différents cas possibles ;
- **le nombre de fonctions/méthodes** : cette fois, on va compter juste le nombre de fonctions/méthodes de l'application. Dès que cette méthode est appelée au moins une fois, le code est indiqué comme couvert, peu importe le nombre de lignes ou de branches parcourues au sein de cette fonction.