|  |  |
| --- | --- |
| |  | | --- | |  | |
| C34-VM Programmation Orientée Objet 1  **MODULE 03 - B**  **JUnit**  Automne 2023 |

Table des matières

[1. Tests Logiciels 3](#_Toc148107956)

[1.1 Les tests unitaires 3](#_Toc148107957)

[1.2 JUnit 3](#_Toc148107958)

[1.3 Intégrer et utiliser JUnit dans un projet 4](#_Toc148107959)

[1.4 Projet pour démo : D03A 4](#_Toc148107960)

[1.5 Étape 1 – Ajouter les librairies (dépendances JUnit) 4](#_Toc148107961)

[1.6 Étape 2 – Créer un répertoire racine de sources de test 4](#_Toc148107962)

[1.7 Étape 3 – créer une classe de tests 5](#_Toc148107963)

[1.8 Étape 4 – Importer les classes requises de la librairie JUnit 5](#_Toc148107964)

[1.9 Étape 5 – Ajouter des méthodes de test 6](#_Toc148107965)

[1.10 Étape 6 – Exécuter et interpréter les résultats 6](#_Toc148107966)

[1.11 D’autres tests 6](#_Toc148107967)

[Test en échec 6](#_Toc148107968)

[Test avec plusieurs conditions 7](#_Toc148107969)

[Tester une méthode sans valeur de retours 8](#_Toc148107970)

[Tester des champs ou méthodes private 8](#_Toc148107971)

[1.12 Définir des paramètres de tests avant et après les tests 9](#_Toc148107972)

[Démo 9](#_Toc148107973)

[1.13 Autres fonctionnalités de JUnit 10](#_Toc148107974)

[1.14 Choix des tests 11](#_Toc148107975)

[L03F\_Tests unitaires avec JUnit 11](#_Toc148107976)

# Tests Logiciels

Les tests sont une phase importante du développement logiciel.

Ils permettent de :

* S’assurer que l’application adopte le comportement attendu
* Détecter des comportements indésirables, non perçues au codage
* Avant de publier une application.

Les méthodologies de tests avancées (comme le [Extreme Programming](https://asana.com/fr/resources/extreme-programming-xp) ) préconisent même de préparer tous les tests AVANT de commencer à code.

## Les tests unitaires

Un **test unitaire** consiste à tester une petite portion du code, une seule fonctionnalité ou partie d’une fonctionnalité.

En **Java**, c’est souvent la vérification d’une seule **méthode**.

On peut faire des tests :

* De façon **manuelle** : on essaie les méthodes avec différentes valeurs et on vérifie si les résultats correspondent aux attentes.
* De façon **automatisée** : on utilise des outils spécialisés qui effectueront des tests automatiquement.

La méthode manuelle consomme plus de temps et les tests ne sont pas réutilisables.

La méthode automatique permet d’effectuer toute une série de tests à la volée, à mesure qu’on développe et/ou qu’on modifie le code.

Une suite de tests bien conçue pourra aussi être réutilisée tout au cours du développement (code-test-code-test-etc.), puis lorsqu’on apporte des modifications au code, après son déploiement (maintenance).

## JUnit

**JUnit** est un framework open-source, conçu spécifiquement pour faire des tests unitaires automatisés sur du code **Java**.

JUnit a été initialement développé par Erich Gamma et Kent Beck et est maintenu par le [**Apache Maven Project**](https://maven.apache.org/index.html)

## Intégrer et utiliser JUnit dans un projet

Les étapes suivantes permettent de mettre en place des tests JUnit :

1. Ajouter les librairies (dépendances) JUnit au projet
2. Créer un répertoire racine de tests
3. Créer une classe de test
4. Importer les classes requises de la librairie JUnit
5. Ajouter des méthodes de tests
6. Exécuter et interpréter les résultats

## Projet pour démo : D03A

Pour démontrer JUnit on utilise le projet déjà créé D03A.

1. Téléchargez, décompressez et ouvrez le projet D03A

## Étape 1 – Ajouter les librairies (dépendances JUnit)

Dans le cours on utilisera la version 5 (5.8.1) de JUnit, avec l’API de test jupiter, qui contient plusieurs améliorations et différences avec JUnit 4.

Avec le projet ouvert dans IntelliJ :

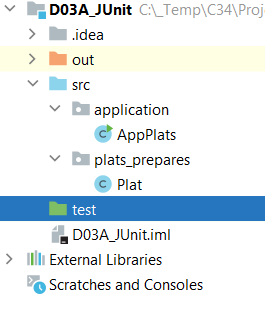
1. **File**
2. **Project structure**
3. Sélectionner **Librairies**
4. Utilisez le **+** pour ajouter
5. Sélectionner **From Maven**
6. Dans le champs entrer **org.junit.jupiter:junit-jupiter:5.8.1**
7. Puis **OK**-**OK**-**OK**

Alternative : dans un projet, si vous entrez import org.junit et laissez le curseur sur **junit** (ou ALT+ENTER), vous pouvez sélectionner **more actions** et ajouter JUnit 5.8.1

## Étape 2 – Créer un répertoire racine de sources de test

Les meilleures pratiques dictent de garder les tests dans un répertoire séparé du reste du code source.

1. Typiquement situé au même niveau que **scr**
2. Ajoutez un répertoire **test** dans le projet, au même niveau que src (clic droit sur le projet -> New -> Directory)
3. Marquez ce répertoire comme répertoire source de tests (clic droit sur test -> Mark directory as -> Test sources root).
4. Le répertoire devient en vert, marquer comme source est requis pour créer des classes.



## Étape 3 – créer une classe de tests

On regroupe les tests à l’intérieur d’une **classe** de tests.

Avant JUnit 5, on demandait d’ajouter le préfixe « **Test** » au nom des classes de test. Ce n’est plus une exigence, mais demeure une bonne idée.

Le nom de la classe devrait décrire l’objectif des tests qu’elle contient. Par exemple si on test toutes les méthodes d’une classe **NomDeClasse**, un bon nom de classe de test serait **NomDeClasseTest**.

1. Dans le répertoire **test**, créer une classe **PlatTest**.

## Étape 4 – Importer les classes requises de la librairie JUnit

La librairie JUnit inclus plusieurs packages, classes et méthodes.

1. Ici, on utilise les méthodes **Test** et **assert**, les imports suivants sont donc requis, ajoutez-les au début de votre classe de test :

* import org.junit.jupiter.api.Test;
* import static org.junit.jupiter.api.Assertions.\*;

Pour la classe Assertions, importer en static évite d’avoir à nommer la classe avant d’appeler ses méthodes.

1. On a aussi besoin d’avoir accès aux méthodes à tester, importez-les de la classe **Plat** :

* import plats\_prepares.Plat;

## Étape 5 – Ajouter des méthodes de test

* Les tests JUnit sont des méthodes dans la classe de test.
* Ces méthodes DOIVENT être précédées de l’annotation **@Test**.
* Avant JUnit 5, les méthodes devaient être précédées su préfixe **test**. Ce n’est plus requis, mais demeure une bonne idée.
* À l’intérieur de la méthode de test, on place le code de test.

En premier, on choisit de tester si la méthode isNomValide() fonctionne correctement.

1. Dans la classe **PlatTest**, ajoutez l’annotation **@Test**
2. Directement sous l’annotation, ajoutez une méthode : **public void testNomEstValide(){}**
3. On a besoin d’un objet Plat, dans la nouvelle méthode ajoutez : **Plat platTest = new Plat();**
4. On utilise la méthode assertEquals() qui retourne vrai si les 2 paramètres sont égaux. Dans la méthode de test ajoutez : **assertEquals(true, platTest.isNomValide("Lasagne"));**

## Étape 6 – Exécuter et interpréter les résultats

Exécutez directement la classe de test avec le "run " habituel.

Vous pouvez voir les résultats dans le panneau de résultats

Un test réussi a un crochet vert.

## D’autres tests

### Test en échec

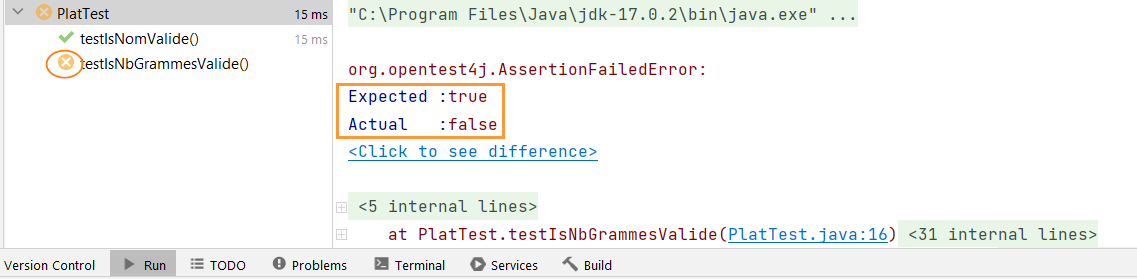
On teste isNbGrammesValide() avec une valeur de 99 grammes (poids trop petit, erreur de saisie présumée).

On place ceci dans une autre méthode.

1. Dans la classe PlatTest, ajoutez le code ci-dessous :

|  |
| --- |
| @Test  public void testIsNbGrammesValide(){  Plat platTest = new Plat();  assertEquals(true, platTest.isNbGrammesValide(99));  } |

Exécutez et observez le résultat d’un test en échec.



Modifions ce test pour confirmer que 99 ne devrait pas être valide.

1. Changez le **true** du assertEquals pour **false**, la vraie valeur attendue.

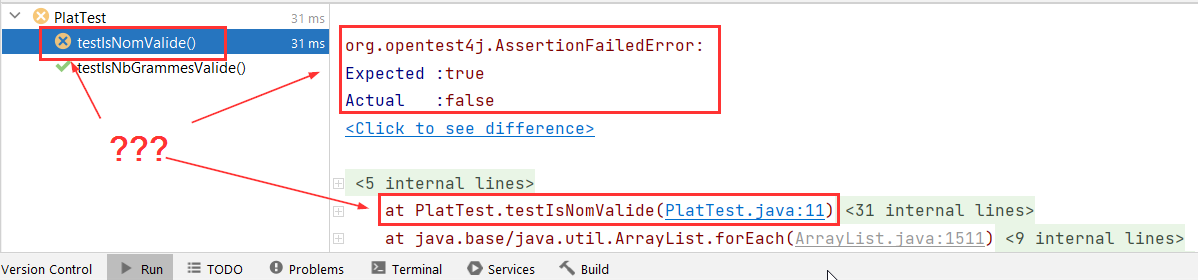
|  |
| --- |
| On utilise la méthode assetEquals(). Vous pouvez aussi explorer les méthodes :  assertTrue()  assertFalse()  assertNull()  assertNotNull()  et des plus spécialisées comme :  assertArrayEquals()  et plus… |

### Test avec plusieurs conditions

On peut mettre plusieurs conditions dans la même méthode de test.

Ajoutons une condition erronée dans la méthode testIsNomValide()

1. Dans la méthodes testIsNomValide, à la suite du code déjà écrit, ajoutez : **assertEquals(true, platTest.isNomValide("Un nom vraiment trop long de plat pour entrer sur l’emballage"))**;
2. Exécutez la classe de test.



On voit bien qu’il y a eu échec, mais quelle condition l’a causé? Pas évident dans le panneau de résultats. Dans un tel cas, chaque condition devrait être dans sa propre méthode de test.

### Tester une méthode sans valeur de retours

Dans la classe **Plat**, la méthode **setPrix()** ne retourne rien. Si on veut vérifier le résultat, on doit valider à partir d’une autre information. Ici, comme la méthode modifie un champ, on peut vérifier que la bonne valeur est assignée à ce champ.

1. Dans la classe **PlatTest**, ajoutez le code ci-dessous :

|  |
| --- |
| @Test public void testSetPrix(){  Plat platTest = new Plat("Lasagne", 430, 350, 40, 7.92, 9.99);  platTest.setPrix(11.99);  assertEquals(11.99, platTest.getPrix()); } |

1. Exécutez les tests.

On constate que le test est réussi.

### Tester des champs ou méthodes private

On ne peut pas directement, c’est tout l’objectif de modificateur **private**.

Il n’est PAS recommandé de changer le modificateur d’un champ pour les tests, on désire plutôt tester les méthodes qui l’utilisent et vérifier que les résultats sont ceux attendus.

Au pire (en dernier recours), s’il n’y en a pas déjà une, on peut ajouter une méthode d’accès publique (un getter) avec des commentaires « POUR TESTS SEULEMENT » et s’assurer que ces méthodes ne seront pas publiées avec le code final.

## Définir des paramètres de tests avant et après les tests

On peut ajouter du code qui sera exécuté avant ou après les tests. Ce code doit aussi être dans une méthode. Les annotations ci-dessous précèdent ce type de méthode :

* @BeforeAll – code exécuté avant tous les tests (avant qu’aucun test ne soit effectué
* @BeforeEach – code exécuté avant chaque test
* @AfterAll – code exécuté après que tous les tests soient terminés
* @AfterEach – code exécuté après chaque test.

Dans ce cours on utilisera principalement le **@BeforeEach**, pour lequel on montre un exemple.

Quelques notes à propos des autres méthodes cependant :

* Le @BeforeAll et @AfterAll sont utilisés pour préparer l’environnement à exécuter les tests. Une utilisation typique qu’on ne verra pas ici est la connexion à une base de données et la fermeture de cette connexion à la fin, des opérations qui n’ont souvent besoin d’être fait qu’une seule fois. D’autres exemples incluent l’ouverture/fermeture de fichiers, réservation de mémoire etc.
* Le @BeforeAll et @AfterAll doivent être déclaré comme **static**, ce qui limite l’utilisation de variables déclarées dans son code. On peut contourner ce comportement en ajoutant, au début du fichier de classe : **@TestInstance(Lifecycle.PER\_CLASS)**.

### Démo

1. Ajoutez le code ci-dessous pour imprimer du texte avant et après les tests et après chaque test. Exécutez les tests pour visualiser les résultats (chaque annotation doit être importée)

|  |
| --- |
| @BeforeAll  public static void setup(){  System.out.println("===================");  System.out.println("= DÉBUT DES TESTS =");  System.out.println("===================");  }  @AfterAll  public void finTests(){  System.out.println("=================");  System.out.println("= FIN DES TESTS =");  System.out.println("=================");  }  @AfterEach  public void apresChaqueTest(){  System.out.println("= Test complété =");  } |

Dans notre classe on remarque qu’on créé un objet **Plat** dans chaque méthode de test. On va déplacer cette opération dans une méthode @BeforeEach, une utilisation typique.

1. Tout d’abord, on aura besoin d’utiliser une variable de type Plat dans toutes nos méthodes. On créé un champ de type Plat à la classe de test. Dans la classe avant les méthodes, entrez **Plat platTest;**

Remarquez les détails suivants :

* La classe de test peut avoir des champs
* Un champ de classe peut être un objet
* On peut déclarer une instance de classe sans créer de nouvel objet (sans le new). La création de l’instance sera faite plus tard.
* Cette variable sera disponible dans toutes les méthodes de la classe, puisque c’est un champ.

1. On créé l’objet dans une méthode @BeforeEach, pour utiliser un nouvel objet à chaque test. Entrez le code suivant avant les méthodes de test :

|  |
| --- |
| @BeforeEach  public void avantChaqueTest(){  platTest = new Plat();  } |

1. Exécutez les tests pour voir les résultats

## Autres fonctionnalités de JUnit

Plusieurs autres outils et fonctionnalités sont disponibles avec JUnit. En voici quelques unes, présentées pêle-mêle, pour votre exploration :

* @DisplayName permet de renommer un test dans le panneau de résultats
* @Nested permet de regrouper des tests en sections à l’affichage des résultats
* On peut créer plusieurs classes de tests
* On peut regrouper une suite de tests composée de tests sélectionnés dans une ou plusieurs classes de test (@RunWith, @Suite…)
* On peut tester pour des exceptions
* On peut tester pour des timeouts
* Et plus…

Recherchez les informations sur JUnit à divers endroits sur internet dont :

[https://junit.org](https://junit.org/)

## Choix des tests

Quelles valeurs doit-on tester dans une suite de tests?

* Ça dépend de la fonctionnalité de la méthode.
* On devrait tester des valeurs qui fonctionne et vérifier qu’on obtient le résultat désiré
* On devrait tester des valeurs qui ne sont pas autorisées et vérifier que le programme gère ces valeurs tel qu’attendu
* On devrait tester les valeurs limites, de très grandes valeurs., de très petites valeurs, 0 etc.
* On devrait tester des valeurs absentes (null, 0, String vide etc.)
* On devrait tester le comportement en absence de données de base (fichiers, bases de données non disponibles)
* Etc.

### L03F\_Tests unitaires avec JUnit

Le laboratoire L03F permet de pratiquer l’utilisation de tests unitaires avec le framework JUnit.