

# R2-01-03 : TP 6

Développement orienté objets & Qualité de développement

POUR COMMENCER .....	1
EXERCICE 1 : TESTS UNITAIRES .....	1
EXERCICE BONUS .....	8

## Pour commencer

**Continuer** dans le projet créé pour les premiers TPs, pour se faire **créer** un package tp6.

**Continuer** à utiliser les outils mis à votre disposition vu en R1.01 et au début de ce module : les mécanismes automatiques dans l'IDE, le débogueur, le logger, etc.

---

## Exercice 1 : Tests unitaires

**Objectifs R2-03** : Aborder les tests unitaires

**Source** : wikipédia, [https://fr.wikipedia.org/wiki/Test\\_unitaire](https://fr.wikipedia.org/wiki/Test_unitaire)

Dans ce TP, vous allez mettre en pratique des tests unitaires sur une classe.

**Qu'est-ce qu'un test unitaire ?** Un test unitaire est une procédure permettant de vérifier le bon fonctionnement d'une partie précise d'un logiciel ou d'une portion d'un programme (appelée « unité » ou « module »). Dans notre cas, une classe.

**A quoi ça sert ?** On écrit un test pour confronter une réalisation à sa spécification. Le test définit un critère d'arrêt (état ou sorties à l'issue de l'exécution) et permet de statuer sur le succès ou sur l'échec d'une vérification. Grâce à la spécification, on est en mesure de faire correspondre un état d'entrée donné à un résultat ou à une sortie. Le test permet de vérifier que la relation d'entrée / sortie donnée par la spécification est bel et bien réalisée.

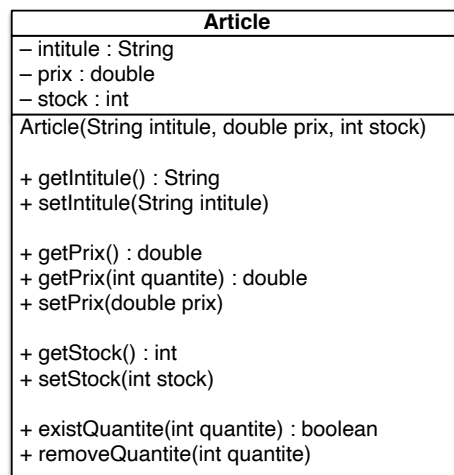
**Euh ?** En gros, on vérifie que ce que l'on donne en entrée permet d'avoir le bon résultat en sortie.

**Quelle librairie allons-nous utiliser ?** Dans le cadre des ressources R2.01 et R2.03, nous allons utiliser la librairie JUnit version 5.8. JUnit est un framework open source pour le développement et l'exécution de tests unitaires automatisables.

Notre exercice porte sur la création d'une classe **Article** dans un futur application d'e-Commerce. Cette classe va représenter un article par son libelle, son prix et la gestion du stock associé.

**Nous vous conseillons de lire l'ensemble de l'exercice avant de commencer à coder !** L'intérêt serait d'écrire chaque méthode de la classe **ET d'écrire les tests en même temps**, ou même avant la méthode à tester (Test Driven Development).

La classe `Article` est représentée par le diagramme UML ci-après.



### Spécifications d'un article :

- L'intitulé d'un article commence toujours par une majuscule et le reste est en minuscule.
- Un intitulé ne peut pas être null ou vide !
- Le prix est le prix unitaire d'un article. Mais l'achat de 100 articles et plus permet d'avoir une ristourne de 10%.
- Un prix ne peut pas valoir zéro ou être négatif.
- Le stock est géré par l'article. On ne peut pas acheter d'articles si le stock n'est pas suffisant.
- Le stock ne peut pas commencer à zéro ou être négatif.

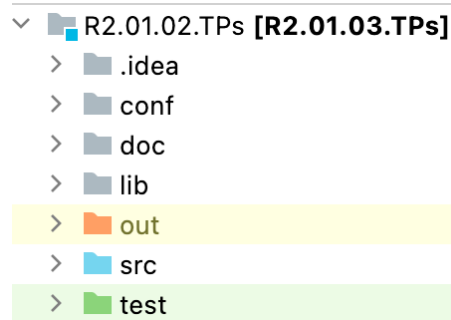
### Explication de certaines méthodes de la classe `Article` :

- `setIntitule(...)` enregistre l'intitulé de l'article en faisant respecter les spécifications (modifie la chaîne de caractères passée en paramètre) ; Provoque une exception de type `IntituleException` si la chaîne de caractères en paramètre est null ou vide.
- `getPrix(int quantite)` retourne le prix unitaire si la quantité demandée est inférieur à 100, retourne le prix unitaire avec une ristourne de 10% si la quantité demandée est supérieur ou égal à 100. Créer une constante `SEUIL` égale à 100 pour votre réalisation.
- `setPrix(double prix)` enregistre le prix unitaire de l'article ; Provoque une exception de type `PrixException` si le prix en paramètre est 0 ou inférieur à 0.
- `existQuantite(int quantite)` retourne vrai si la quantité d'articles demandée en paramètre est présente dans le stock, retourne faux sinon.
- `removeQuantite(int quantite)` enlève la quantité d'articles demandée en paramètre du stock. Normalement cette méthode ne doit être appelée que si la quantité existe dans le stock (en utilisant la méthode ci-avant). Pour plus de sécurité,

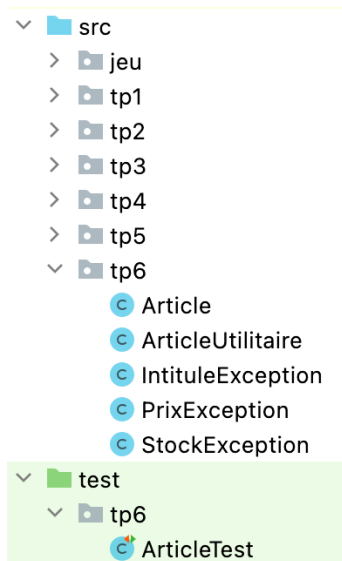
cette méthode provoque une exception de type **StockException** si la quantité en paramètre n'est pas présente dans le stock.

Dans la suite de ce document, nous allons présenter pas à pas comment intégrer l'intitulé d'un article et réaliser les tests unitaires associés.

Dans le paquetage **tp6**, créer la classe **Article**. Créer également un dossier **test** à la racine de votre projet au même niveau que **src**. Ensuite faire un clic droit sur le dossier **test**, puis **Mark Directory as ... > Test Sources Root**. A la fin, dans votre intellij, vous devez avoir votre dossier **test** en vert comme sur la figure ci-dessous.

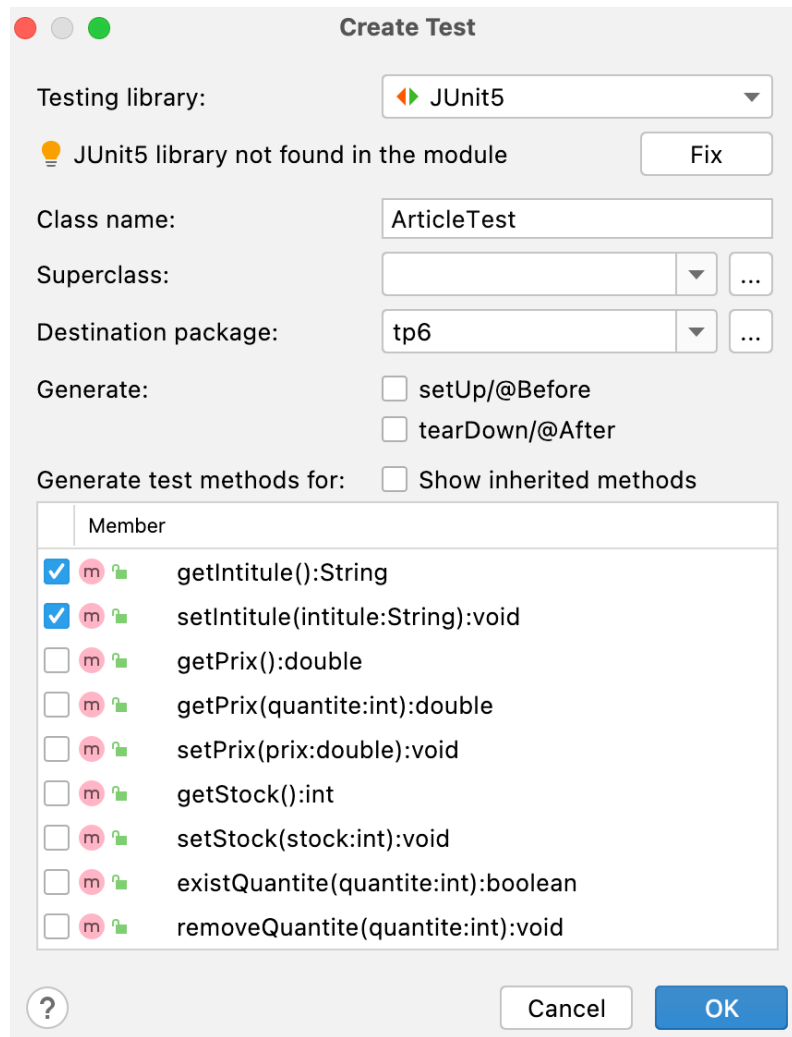


Le dossier **test** contiendra les tests unitaires de vos classes. A la fin de cet exercice, votre projet ressemblera à ça :



**Compléter** la classe **article** en respectant le diagramme UML et les spécifications associées ci-avant. Pour le moment, vous allez vous focaliser sur l'intitulé d'un article.

**Créer** le test unitaire associé à la classe **Article**. Pour ce faire, mettez votre curseur dans la classe **article** de votre IDE, puis **code > generate ... > Test ...** Une fenêtre apparaîtra pour vous proposer de créer votre test unitaire (une classe java nommée **ArticleTest**) dans un package **tp6** dans le dossier **test**. Voir figure ci-après.



**IMPORTANT :** à la première création de votre premier test l'IDE vous dira que la librairie JUnit n'est pas présente dans votre projet et vous proposera de résoudre le problème. Pour résoudre, cliquer sur **FIX**. L'IDE installera la librairie Junit (**org.junit.jupiter:junit-jupiter:5.8.1**).

A ce stade, vous avez deux classes : **Article** et **ArticleTest**.

#### La classe **Article**

```
public class Article {

    private String intitule;
    private double prix;
    private int stock;

    public Article(String intitule, double prix, int stock) throws IntituleException {

        setIntitule(intitule);
        setPrix(prix);
        setStock(stock);
    }

    public String getIntitule() {
```

```

        return intitule;
    }

    public void setIntitule(String intitule) throws IntituleException {
        if (intitule == null || intitule.isEmpty()) {
            throw new IntituleException("Un intitulé ne peut être null ou vide");
        }
        this.intitule = ArticleUtilitaire.capitalize(intitule);
    }
}

```

### La classe ArticleTest

```

import org.junit.jupiter.api.Test;

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

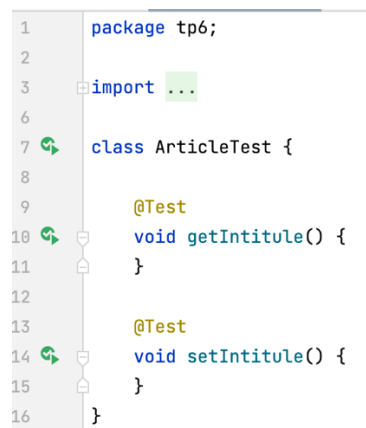
class ArticleTest {

    @Test
    void getIntitule() {
    }

    @Test
    void setIntitule() {
    }
}

```

Vous pouvez déjà lancer votre premier test en cliquant sur les flèches vertes à gauche de votre code :



Vu que vous n'avez pas encore complété vos tests unitaires, vous obtiendrez un résultat valide pour chaque test dans l'onglet run :

Test Results	35 ms
ArticleTest	35 ms
✓ setIntitule()	33 ms
✓ getIntitule()	2 ms

Vous allez maintenant créer vos premiers tests unitaires.

**Vérifier** la première spécification : « L'intitulé d'un article commence toujours par une majuscule et le reste est en minuscule ». Vous allez créer un article avec un intitulé en minuscule et vérifier que l'intitulé a bien été modifié par la classe comme attendu. Pour se faire, vous utilisez la méthode `assertEquals()` qui vérifie que les deux premiers paramètres

sont égaux, sinon le test sera faux. Dans notre cas que le résultat de la méthode `getIntitule()` et égale à la chaîne de caractères voulue. Voir ci-après le test.

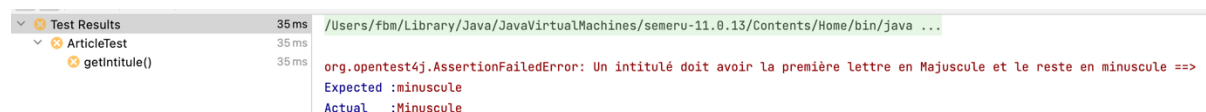
```
@Test
void getIntitule() throws IntituleException, PrixException {

    //
    Article article1 = new Article("minuscule", 2.0, 10);
    assertEquals(article1.getIntitule(), "Minuscule", "Un intitulé
doit avoir la première lettre en Majuscule et le reste en minuscule");
}
```

Réaliser un deuxième test, toujours dans `getIntitule()` de la classe `ArticleTest` avec un autre article avec un intitulé d'entrée en majuscule. Vous noterez que la méthode `assertEquals()` a un troisième paramètre. Ce paramètre est un message qui s'affichera sur le test n'est pas valide.

```
//
Article article2 = new Article("MAJUSCULE", 2.0, 10);
assertEquals(article2.getIntitule(), "Majuscule", "Un intitulé
doit avoir la première lettre en Majuscule et le reste en minuscule");
}
```

Modifier le code de la classe `Article` pour que le test ne soit pas valide et affiche le message. Ci-dessous un exemple de test non valide avec affichage du message associé :



```
Test Results 35 ms /Users/fbm/Library/Java/JavaVirtualMachines/semervu-11.0.13/Contents/Home/bin/java ...
ArticleTest 35 ms
getIntitule() 35 ms
org.opentest4j.AssertionFailedError: Un intitulé doit avoir la première lettre en Majuscule et le reste en minuscule ==>
Expected :minuscule
Actual :Minuscule
```

**IMPORTANT :** Plusieurs méthodes « assert » sont fournies avec la librairie JUnit, pour connaître toutes les méthodes qui vous seront utiles dans la suite :

<https://junit.org/junit5/docs/5.8.2/api/org.junit.jupiter.api/org/junit/jupiter/api/Assertions.html>

Vérifier la seconde spécification : « Un intitulé ne peut pas être null ou vide ». Pour vérifier cela, vous utiliserez la méthode `assertThrows(...)` qui ne valide pas le test si une exception n'est pas levée.

```
@Test
void setIntitule() throws IntituleException, PrixException {

    //
    assertThrows(IntituleException.class, () -> {
        new Article(null, 2.0, 10);
    }, "Un intitulé ne peut pas être null.");

    //
    assertThrows(IntituleException.class, () -> {
        new Article("", 2.0, 10);
    }, "Un intitulé ne peut pas être vide.");
}
```

Détaillons la méthode `assertThrows(...)` utilisée :

- Premier paramètre, la classe d'exception attendue, ici `IntituleException`
- Deuxième paramètre, un jeu d'instructions écrit sous la forme d'une lambda expression : `(paramètres) -> { instructions; }`. Ici, pas de paramètre et une seule instruction la création d'un article avec un intitulé null ou vide.

- Troisième paramètre, un message si le test n'est pas valide

A ce stade, votre classe **ArticleTest** doit ressembler à ça :

```
class ArticleTest {

    @Test
    void getIntitule() throws IntituleException, PrixException {

        //
        Article article1 = new Article("minuscule", 2.0, 10);
        assertEquals(article1.getIntitule(), "Minuscule", "Un intitulé
doit avoir la première lettre en Majuscule et le reste en minuscule");

        //
        Article article2 = new Article("MAJUSCULE", 2.0, 10);
        assertEquals(article2.getIntitule(), "Majuscule", "Un intitulé
doit avoir la première lettre en Majuscule et le reste en minuscule");
    }

    @Test
    void setIntitule() throws IntituleException, PrixException {

        //
        assertThrows(IntituleException.class, () -> {
            new Article(null, 2.0, 10);
        }, "Un intitulé ne peut pas être null.");

        //
        assertThrows(IntituleException.class, () -> {
            new Article("", 2.0, 10);
        }, "Un intitulé ne peut pas être vide.");
    }
}
```

## A vous de jouer pour la suite des spécifications !

**Compléter** les classes **article** et **articleTest** en respectant le diagramme UML et les spécifications associées ci-avant et **créer** les classes **PrixException** et **StockException**.

```
class ArticleTest {

    @Test
    void getIntitule() throws IntituleException, PrixException {
        ...
    }

    @Test
    void setIntitule() throws IntituleException, PrixException {
        ...
    }

    @Test
    void getPrix() throws PrixException, IntituleException {
        ...
    }
}
```

```

@Test
void setPrix() {
    ...
}

@Test
void existQuantite() throws PrixException, IntituleException {
    ...
}

@Test
void removeQuantite() throws PrixException, IntituleException, StockException {
    ...
}
}

```

Les « assert » dont vous aurez besoin :

- **assertTrue(boolean condition)** : vérifie que la condition soit vraie
- **assertFalse(boolean condition)** : vérifie que la condition soit fausse
- **assertDoesNotThrow(...)** : vérifie que le jeu d'instruction ne renvoie pas d'exception

Pour plus d'informations, <https://junit.org/junit5/docs/5.8.2/api/org.junit.jupiter.api/org/junit/jupiter/api/Assertions.html>

#### **PETIT PLUS, les avantages des tests unitaires :**

**Trouver les erreurs rapidement** : Les tests sont exécutés durant tout le développement, permettant de visualiser si le code fraîchement écrit correspond au besoin.

**Sécuriser la maintenance** : Lors d'une modification d'un programme, les tests unitaires signalent les éventuelles régressions. En effet, certains tests peuvent échouer à la suite d'une modification, il faut donc soit réécrire le test pour le faire correspondre aux nouvelles attentes, soit corriger l'erreur se situant dans le code.

**Documenter le code** : Les tests unitaires peuvent servir de complément à la javadoc, il est très utile de lire les tests pour comprendre comment s'utilise une méthode. De plus, il est possible que la documentation ne soit plus à jour, mais les tests eux correspondent à la réalité de l'application.

## Exercice bonus

Faire des tests unitaires sur la table des opérations vu dans le TP5. Tester les classes Addition, Soustraction et Multiplication. Vérifier les méthodes **calculResultat()** et **isReponseJuste()**. Créer et tester une nouvelle classe **Division** héritant d'opération.

Faire des tests unitaires sur le jeu Faërun. Tester dans un premier dans les classes filles de la classe **Guerrier**, puis la classe **Chateau**, etc.