Les JUnits

Contenu

Les tests

Quand on écrit une application, il faut toujours la tester. Il existe plusieurs sortes de tests :

- les **tests unitaires** : ils permettent de tester si une méthode remplit bien son contrat ;
- les **tests d'intégration** : ils permettent de tester les interactions entre les objets, les services,… ;
- les **tests de fonctionnalité** : ils permettent de tester la réaction de l'application à une requête d'un utilisateur ;
- les **tests de performance** : cela regroupe entre autre les tests de charge (traiter un grand nombre de données ou de requêtes,…) et les tests de stress (retirer les ressources de l'application et effectuer des tests de récupération).
- les **tests d'acceptation** : ce sont des tests exécutés par les clients afin de vérifier que l'application répond bien à leurs besoins.

JUnit est un outil ayant pour but de faciliter l'écriture des tests unitaires. Dans ce chapitre, on va voir comment écrire des tests unitaires en utilisant les **Junit 5**. JUnit 5 est une nouvelle version de JUnit qui

- exploite les possibilités de Java 8 (expression lambda etc.) ;
- permet le paramétrage de tests ;
- se décompose en une architecte modulaire qui favorise ainsi leur intégration dans les outils de développement (IDE, build comme gradle ou maven).

Les JUnits reposent sur l'usage des **annotations** afin de marquer les classes et les méthodes de test.

Les tests unitaires

Les tests unitaires ont pour objectifs de tester si une méthode remplit bien son **contrat** c.-à-d. qu'elle fait ce qu'elle doit faire. Par exemple :

- vérifier qu'elle teste la validité des paramètres ;
- vérifier qu'elle envoie bien une exception dans tel ou tel cas ;
- vérifier qu'elle renvoie la valeur adéquate ;
- vérifier qu'elle met à jour les attributs ;
- etc.

On écrit une classe de tests par classe testée.

On peut y définir une méthode annotée **`@BeforeAll`** qui sera exécutée avant la première méthode de la suite de tests et également une méthode annotée **`@AfterAll`** qui sera exécutée après le dernier test de la suite. Ces méthodes doivent être statiques.

Chaque méthode de la classe testée peut être testée par plusieurs méthodes de la classe de tests. Chaque méthode de test va se concentrer sur un aspect de la méthode testée.

Outre divers imports expliqués dans le point qui suit, une classe de tests possède comme toute classe Java un état et un comportement. Dans l'état, on précise les objets qu'on va utiliser à plusieurs reprises dans les différents tests.

On **initialise** l'état avec une méthode annotée **`@BeforeEach`**. Cette méthode sera exécutée avant chaque méthode de tests.

Une méthode annotée **`@AfterEach`** permet de préciser un comportement qui sera exécuté **après** chaque méthode de tests.

Dans le comportement, on met toutes les méthodes de tests.

Il existe plusieurs annotations indiquant qu'il s'agit **d'une méthode de tests** : `@Test`, `@ParameterizedTest`, `@RepeatedTest(…)`, `@TestFactory`.

Exemple :

```
@Test
public void testGetClient(){ … }
```

Pour personnaliser l'affichage lors de l'exécution, on peut annoter les classes de tests et les méthodes de tests avec `@DisplayName("…")`.

Un premier exemple de tests JUnit 5

Partons d'un exemple simple afin d'expliquer comment écrire une classe de tests avec JUnit. Supposons qu'on veuille écrire une classe de tests pour la classe Commande dont le diagramme UML est donné ci-dessous :

Commande
- date : LocalDate - client : String …
+ Commande(date : LocalDate, client :String) + getDate() : LocalDate + getClient() : String …

Écrivons maintenant la classe de tests :

```
package domaine;
import java.time.LocalDate;

/* pour pouvoir utiliser directement les méthodes de la classe Assertions fournie par
JUnit 5 */
import static org.junit.jupiter.api.Assertions.*;

/* nécessaire si on veut écrire une méthode qui s'exécute avant chaque méthode de
tests */
import org.junit.jupiter.api.BeforeEach;

/* nécessaire si on veut personnaliser les noms apparaissant lors de l'affichage du
résultat*/
import org.junit.jupiter.api.DisplayName;

/* nécessaire si on veut écrire une méthode qui s'exécute après chaque méthode de
tests */
import org.junit.jupiter.api.AfterEach;

/* nécessaire pour écrire les méthodes de tests annotées par @Test*/
import org.junit.jupiter.api.Test;

/* Annotation qui permet de personnaliser l'affichage du résultat*/
@DisplayName("Classe de tests de la classe Commande")
public class CommandeTest {

	/* Déclaration des champs qui seront utilisés dans le test*/
	private Commande commande;
	private LocalDate date;
```

```
	/* Annotation pour indiquer que la méthode qui suit sera exécutée avant chaque
	méthode de tests*/
	@BeforeEach
	void setUp() throws Exception{
		date = LocalDate.now();
		commande = new Commande(date, "Leconte");
	}

	/* Annotation pour indiquer que la méthode qui suit sera exécutée après chaque
	méthode de tests*/
	@AfterEach
	void tearDown() throws Exception{
		/* ... */
	}

	/* Annotation qui permet de personnaliser l'affichage du résultat*/
	@DisplayName("Vérification de la valeur renvoyée par getClient")

	/* Annotation pour dire qu'il s'agit d'une méthode de tests */
	@Test
	void testGetClient() {
		/*
		 * méthode qui vérifie que les deux paramètres sont égaux en utilisant
		 * la méthode equals. Il faut mettre en premier lieu ce qui est attendu
		 * et, en second, ce que renvoie la méthode.
		 */
		assertEquals("Leconte", commande.getClient());
	}
	@DisplayName("Test du constructeur avec en paramètre une date null")
	@Test
	void testCommande() {
		/*
		 * méthode qui permet de vérifier qu'une exception est bien lancée.
		 * Le premier paramètre est la classe de l'exception attendue
		 * Le deuxième est un exécutable qui permet d'indiquer le code devant
		 * lancer l'exception.
		 * Si une exception du type attendu est effectivement émise, celle-ci
		 * sera renvoyée par la méthode assertThrows.
		 */
		assertThrows(NullPointerException.class, () ->  new
			Commande(null,"Leconte"));
	}
}
```

Une classe de tests

En théorie, pour écrire une classe de tests avec JUnit 5 :

1. Chaque méthode de tests doit être précédée d'une annotation parmi les suivantes : **`@Test`, `@ParameterizedTest`, `@RepeatedTest(…)`, `@TestFactory`**. Pour pouvoir utiliser ces annotations, il faut faire l'import correspondant (`org.junit.jupiter.api.Test`, `org.junit.jupiter.params.ParameterizedTest`, `org.junit.jupiter.api.RepeatedTest`, `org.junit.jupiter.api.TestFactory`)
2. Afin de pouvoir utiliser les méthodes de la classe `Assertions` proposées par JUnit 5, il faut faire un `import static` de **`org.junit.jupiter.api.Assertions`**.
3. On peut écrire une méthode qui s'exécutera avant chaque méthode de tests. Pour cela, il faut mettre l'annotation **`@BeforeEach`** avant cette méthode et il faut importer

`org.junit.jupiter.api.BeforeEach`. Elle doit également être `void`. Cette méthode est en général utilisée pour initialiser des attributs de la classe de tests.

4. On peut écrire une méthode qui s'exécutera après chaque méthode de tests. Pour cela, il faut mettre l'annotation **`@AfterEach`** avant cette méthode et il faut importer `org.junit.jupiter.api.AfterEach`. De nouveau, elle doit être `void`.
5. Il est aussi possible d'écrire une méthode qui s'exécutera une seule fois avant la première méthode de tests (ou après la dernière méthode de tests) de la classe. Il faut mettre l'annotation **`@BeforeAll`** (`@AfterAll`) avant cette méthode et importer `org.junit.BeforeClass` (`org.junit.AfterClass`). Cette méthode doit être **`static`** et `void`.

Une classe interne de tests

Lorsqu'on écrit une classe de tests, il arrive souvent qu'on veuille regrouper plusieurs méthodes de tests nécessitant une même configuration initiale. Comme vu précédemment, nous pouvons utiliser l'annotation @BeforeEach pour séparer la configuration du test et son exécution mais alors cette configuration s'exécutera pour toutes les méthodes de la classe de test. Afin de créer des groupes de tests (typiquement pour partager un @BeforeEach), il suffit de créer une classe interne dans la classe de test et de l'annoter @Nested. De cette manière, une méthode annotée @BeforeEach ne s'exécutera qu'avant les méthodes de ce groupe là.

Cette technique est très pratique pour grouper les tests qui concerne un même état de la classe testée , le nom de la classe interne s'appelle alors GivenXXXX afin de nommer l'état. Par exemple, un test pour la classe Array pourrait contenir les classes internes suivantes : GivenEmpty, GivenFull, GivenContainingHoles, GivenSorted, … et chacune contiendrait un groupe de méthode qui testerait l'insertion, la suppression, le parcours, etc.

Une suite de tests

Parfois, on veut exécuter plusieurs classes de tests en même temps. Si on dispose des classes `ClientTest` et `CommandeTest` (toutes deux dans le package `domaine`) et qu'on désire les exécuter en même temps, on peut écrire une autre classe comme suit

```
package tests;

import org.junit.platform.runner.JUnitPlatform;
import org.junit.platform.suite.api.SelectClasses;
import org.junit.runner.RunWith;

@RunWith(JUnitPlatform.class)
@SelectClasses({domaine.CommandeTest.class,domaine.ClientTest.class})
/*
 * On met toutes les classes de tests qu'on souhaite exécuter.
 */
public class AllTests {
	// ne rien écrire dans cette classe.
}
```

Une autre solution consiste à écrire une classe dans laquelle on précise les package où se situe les classes de tests à exécuter (cela exécutera aussi les classes de tests situées dans les sous-packages des packages indiqués).

```java
package tests;

import org.junit.platform.runner.JUnitPlatform;
import org.junit.platform.suite.api.SelectPackages;
import org.junit.runner.RunWith;

@RunWith(JUnitPlatform.class)
@SelectPackages({"domaine"})
/*
 * On met tous les packages contenant les classes de tests qu'on souhaite exécuter.
 */

public class AllTests {
	// ne rien écrire dans cette classe.
}
```

La façon de préciser les classes à exécuter ne se limite pas aux cas présentés ci-dessus. Pour plus d'infos, vous pouvez consulter https://howtodoinjava.com/junit5/junit5-test-suites-examples/.

Les méthodes de test : les assert

La classe `Assertions` de JUnit 5 offre toute une série de méthodes qu'on peut utiliser pour écrire des méthodes de tests. Voici la liste des principales méthodes existantes :

- **`assertEquals`**`(type attendu, type calculé);`

  où `type` peut être n'importe quoi.

  Si le type est un type primitif (`byte`, `char`, `int`, …), ces méthodes vérifient que la valeur calculée est bien égale à la valeur attendue. Sinon, ces méthodes utilisent la méthode `equals` pour vérifier qu'il s'agit du même « `Object` ». Pour `double` ou `float`, il vaut mieux utiliser la méthode suivante :

  `assertEquals(type attendu, type calculé, type delta);`

  où `type` peut être `float` ou `double`.

  Cette méthode vérifie que la valeur attendue est la même que la valeur calculée avec une tolérance de delta (delta devant être strictement positif).

- **`assertSame`**`(Object attendu, Object calculé);`
  Cette méthode vérifie que l'objet attendu et l'objet calculé ont la même adresse (==) .
- **`assertNotSame`**`(Object attendu, Object calculé);`
  Cette méthode vérifie que l'objet attendu et l'objet calculé n'ont pas la même adresse (!=) .
- **`assertNull`**`(Object o);`
- **`assertNotNull`**`(Object o);`
- **`assertTrue`**`(boolean condition);`
- **`assertFalse`**`(boolean condition);`
- **`fail`**`();`
  L'exécution d'une de cette méthode provoque l'échec du test.

Pour toutes les méthodes citées précédemment, il existe une surcharge ayant, en plus des paramètres indiqués précédemment, une `String` (p.ex., `assertEquals(type attendu, type calculé,`

`String message)` et une autre surcharge avec un `Supplier<String>` au lieu d'une String (p.ex., ., `assertEquals(type attendu, type calculé, Supplier<String> supplierMessage)`. Ce paramètre permet de mettre un **message** expliquant la raison de l'échec du test (si vous utiliser un Supplier, le message ne sera créé que si c'est nécessaire). **Il est vivement conseillé de définir le supplier !**

La méthode de tous les tests : assertAll

Si, dans une méthode de tests, on a plusieurs assert et qu'il y en a un qui échoue, les asserts suivants ne seront plus exécutés. Si on veut mettre plusieurs asserts dans une méthode de tests et qu'on veut qu'ils soient tous exécutés, indépendamment de la réussite ou de l'échec des asserts qui le précèdent, on peut utiliser la méthode suivante :

```
assertAll(Executable... executables)
```

Exemple :

```
@DisplayName("Test des getters")
@Test
void testGetters() {
      assertAll(
                  () -> assertEquals(date, commande.getDate(),"La méthode
                     getDate ne renvoie pas la bonne date"),
                  () -> assertEquals("Leconte", commande.getClient(),"La
                     méthode getDate ne renvoie pas le bon nom de client")
            );
}
```

Pour une documentation complète concernant la classe `Assertions`, consultez :
https://junit.org/junit5/docs/5.0.1/api/org/junit/jupiter/api/Assertions.html

N'oubliez pas que chaque méthode de test ne doit se concentrer que sur un seul aspect de la méthode testée, assertAll() est donc une technique à utiliser avec parcimonie et il vaut souvent mieux écrire deux méthodes de test distinctes contenant chacune un seul assert().

Les tests paramétrés

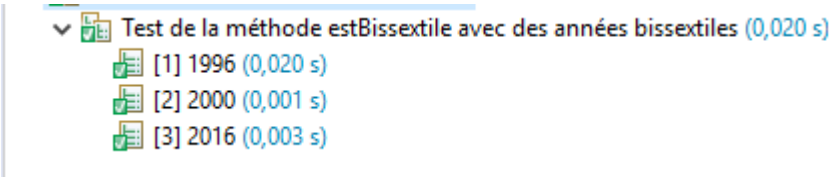
Les tests paramétrés permettent d'exécuter plusieurs fois le même cas de test en utilisant des données différentes. Ces données peuvent être fournies de différentes manières grâce à des annotations. Un test paramétré sera toujours annoté par `@ParameterizedTest`. Cette annotation doit toujours être utilisée conjointement avec une autre annotation (`@ValueSource`, `@MethodSource`, `@EnumSource`, …) qui sert à « fournir » les données.

Exemples :

Si on veut tester une méthode statique `estBissextile(int annee)` située dans une classe `Date` et qu'on veut vérifier qu'elle renvoie vraie pour différentes valeurs, on peut écrire la méthode de tests suivante :

```
@DisplayName("Test de la méthode estBissextile avec des années bissextiles")
@ParameterizedTest
@ValueSource(ints = {1996,2000,2016})
void testEstBissextile(int annee) {
      assertTrue(Date.estBissextile(annee),"L'année " + annee + " est bissextile
                  mais la méthode renvoie false");
}
```

L'annotation `@ValueSource(…)` permet de préciser les données qui seront passées à la méthode `testEstBissextile(int année)`. JUnit exécutera la méthode autant de fois qu'il y a de données spécifiées et présentera une sortie détaillée :



Les types supportés par l'annotation `@ValueSource` sont les types primitifs à l'exception des booléens, `String` et `Class`.

Si on a besoin de passer un autre type de données ou si on a besoin de paramétrer plusieurs données, on utilisera plutôt l'annotation `@MethodSource` qui permet de préciser une méthode qui fournira les données.

Exemple :

On veut maintenant tester que le constructeur de la classe `Date`, qui reçoit le jour, le mois et l'année en paramètres, lance une `IllegalArgumentException` dans divers cas. On peut faire cela en écrivant deux méthodes comme ci-dessous :

```
@DisplayName("Test du constructeur Date avec des données invalides")
@ParameterizedTest
/*La chaîne de caractères mise dans l'annotation MethodSource correspond au
 * nom de la méthode fournissant les données.
 */
@MethodSource("invalidDateProvider")
void testDate(int jour, int mois, int annee) {
	assertThrows(IllegalArgumentException.class,
			() -> new Date(jour,mois,annee),
			"Création de la date " + jour +"/" + mois +"/" + annee);
}

/* La méthode fournissant les données (provider) doit renvoyer un Stream<Arguments>
 * car la méthode de tests (MethodSource) prend plusieurs paramètres
 */
static Stream<Arguments> invalidDateProvider(){
	return Stream.of(
		/*permet de créer un Arguments avec les valeurs pour les paramètres*/
		Arguments.of(0,1,2000),
		Arguments.of(31,4,2016),
		Arguments.of(29,2,2017),
		Arguments.of(1,0,2000),
		Arguments.of(1,13,2017)
	);
}
```

JUnit exécutera la méthode autant de fois qu'il y a de données dans le `Stream` renvoyé et présentera à nouveau une sortie détaillée.

Remarquez que, si la méthode de tests n'a qu'un paramètre, la méthode qui fournit les données renverra un `Stream<Type>` ou `Type` est le type du paramètre indiqué dans la méthode de tests :

```java
@ParameterizedTest
@MethodSource("typeProvider")
void testDate(Type date) {
	...
}

static Stream<Type> typeProvider (){
	return Stream.of(
		...
	);
}
```

JUnit 5 fournit encore d'autres techniques pour passer des données à un test paramétré et fournit également d'autres types de tests (Repeated test, Dynamic test) que ceux vus jusqu'à présent. Vous pouvez trouver toutes les informations sur ces différents points, et même plus encore, dans le guide utilisateur :
https://junit.org/junit5/docs/current/user-guide/

Bonnes pratiques pour l'écriture de tests

JUnit offre un outil facilitant l'écriture mais il n'offre pas de recettes pour écrire de bons tests. Quand on écrit des tests, il y a certaines règles à respecter le plus possible afin d'améliorer la qualité des tests :

- écrire une classe de tests par classe testée;
- écrire une méthode de test par aspect testé.
- utiliser `@DisplayName` afin d'avoir un message clair sur ce qui est testé ;
- expliquer la raison d'un échec à l'aide d'un message ;
- tester tout ce qui peut échouer ;
- tester que tout ce qui doit être mis à jour par une méthode l'a vraiment été ;
- tester tous les cas où une exception doit être lancée.
- ne pas « catcher » les exceptions qui ne doivent pas se produire mais les propager.
- mettre la classe de tests dans le même package que la classe testée mais dans des sources folders différents !
- écrire les tests avant d'écrire le code ;
- grouper les tests s'appuyant sur un même @BeforeEach
- utiliser des données d'exemple pour valider la logique du code, ne pas recopier la logique du code dans le test.

L'exécution des tests

Lorsqu'on exécute une classe de tests JUnit, toutes les méthodes de tests sont exécutées même si certains tests échouent. Lorsqu'un test échoue, JUnit le signale par :

- une **failure** si on est passé par un **fail()** ou si un « **assert** » a raté ;
- une **error** si une exception inattendue s'est produite (`NullPointerException`,…).

Alternatives à JUnit :

- TestNG (www.testng.org)
- JTiger (www.jtiger.org)