

Test et mocks

JUnit est installé par défaut dans Eclipse.

1 Première classe de test simple pour `OrderedDictionary`

Nous allons ici mettre en place une première classe de test simple pour tester la classe `OrderedDictionary`.

1.1 Création de la classe de test

Dans votre projet :

- clic droit → new JUnit test case
- Choisissez bien JUnit Jupiter.
- Nommez votre classe de test. Cochez la case permettant de générer la méthode `setUp`. Ne sélectionnez aucune autre option.

1.2 Environnement de test

Dans la classe test nouvellement créée, déclarez un attribut de type `OrderedDictionary`. Créez-le dans la méthode `setUp` qui a été générée. Vous noterez que cette méthode est annotée par l'annotation `@BeforeEach`, c'est donc une méthode de préambule qui sera appelée avant chaque méthode de test.

1.3 Une première méthode de test

Ajoutez à votre classe une première méthode de test `testAddOneElementToEmptyDico()`. N'oubliez pas d'ajouter l'annotation `@Test` à cette méthode, afin qu'elle soit bien considérée par JUnit comme une méthode de test. Dans le corps de cette méthode, écrivez le code permettant d'ajouter le couple (clef-valeur) de votre choix. Puis, vérifiez que tout s'est bien passé :

- Vérifiez que la taille du dictionnaire est bien 1 (par une assertion du type `AssertEquals(1, dico.size())`)
- Vérifiez que l'élément ajouté existe bien dans votre dictionnaire (par une assertion du type `AssertTrue(dico.containsKey(clef))`)
- Vérifiez que vous pouvez bien retrouver l'élément ajouté dans votre dictionnaire.

Exécutez votre test (`run as JUnit application`).

1.4 Fin de la classe de test pour `OrderedDictionary`


Complétez le test de la classe `OrderedDictionary` en ajoutant autant de méthodes de test qu'il vous semble nécessaire. Corrigez les erreurs que vous détecterez au fur et à mesure.

2 Test des autres dictionnaires

Testez les 2 autres types de dictionnaires. Vous vous rendrez compte que de nombreuses méthodes de test sont identiques à celles de la classe de test pour `OrderedDictionary`. Il pourra être intéressant de créer une super-classe encapsulant ces tests.

Finalement, créez une suite de test permettant d'enchaîner l'exécution des tests des 3 classes de test.

3 Couverture de code

Quand vous pensez avoir terminé vos tests, ré-exécutez-les, cette fois avec `run as Coverage Application` (vous pouvez utiliser l'icône : ). **Coverage** est un outil provenant du plugin Eclipse **ECLEmma**, qui permet d'analyser le code couvert par les tests, c'est-à-dire de déterminer quelles parties du code sont exécutées lors de l'exécution des tests, et quelles parties ne le sont pas. On obtient une mesure de la couverture de code. Attention, avoir 100% de couverture de code ne signifie pas nécessairement d'avoir correctement testé votre code. Si **Coverage** n'est pas disponible sur la version d'Eclipse que vous utilisez, installez le plug-in **ECLEmma** depuis le marketplace d'Eclipse.

Observez les parties de votre code couvertes et non couvertes. (Re-)Testez en conséquence.

4 Installation de Mockito

Téléchargez depuis cette adresse : <https://mvnrepository.com/artifact/org.mockito> les fichiers jar : pour mockito core et mockito junit jupiter. Lorsque vous serez sur la page de mockito core, regardez les dépendances et téléchargez les jar correspondants (byte-buddy, byte-buddy-agent, objenesis).

Placez les jar dans votre buildpath (dans le projet Eclipse dans lequel vous allez travailler).

5 Premiers pas avec Mockito

On suppose que l'on doit mettre en place un système de gestion d'ouvrages dans un groupement de médiathèques. Vous implémentez une partie de recherche d'ouvrage, et votre collègue implémente la classe Ouvrage. La classe Ouvrage dispose d'une méthode indiquant le nombre total d'exemplaires disponibles pour l'ouvrage, et une méthode qui retourne une liste de paires nom de médiathèques / nombre d'ouvrage disponible.

De votre côté, vous devez mettre au point une classe Requete qui dispose d'une méthode de classe qui, à partir d'un ouvrage donné (qui a été récupéré dans un catalogue mais on ne s'intéresse pas à cette partie), retourne la liste de tous les noms de médiathèques dans lesquelles l'ouvrage peut être emprunté actuellement.

Mettez en place l'interface de la classe Ouvrage :

```
public interface IOuvrage {
    public int nbExemplairesDisponibles() ;

    public ArrayList<Pair<String , Integer>> listeDesMediathequesAvecNbOuvrageDisponible() ;
}
```

Pair est soit la classe Pair de javafx Mettez en place votre classe Requete. Vous voulez maintenant tester votre classe (son unique méthode de classe).

Pour cela, vous utiliserez dans votre test des mocks d'ouvrages.

On commence simplement par ce genre de test :

```
import static org.junit.jupiter.api.Assertions.*;
import static org.mockito.Mockito.mock;
import static org.mockito.Mockito.when;

import org.junit.jupiter.api.Test;

class TestRequete {

    @Test
    void test1() {
        IOuvrage ouvrage = mock(IOuvrage.class);
        ArrayList<String> res=Requete.listesMediatheques(ouvrage);
        assertTrue(res.isEmpty());
    }
}
```

Première constatation, ce test compile (il devrait) et peut même s'exécuter, bien que la classe Ouvrage n'existe pas. Un mock de l'interface a été créé, et votre méthode listeMediatheques y a fait appel. Deuxième constatation, on ne va pas pouvoir tester grand chose si on s'arrête là, car le mock d'ouvrage que nous avons retourne toujours 0 et une liste vide ... On note de ce fait que l'assertion res.isEmpty() ne provoque pas d'erreur. Conclusion : avant d'appeler la méthode listeMediatheques, il faut configurer le mock. Ainsi, il retournera des valeurs pertinentes pour votre test de getMediatheques (notamment des listes non vides, avec des valeurs intéressantes). Mettez cela en place avec des instructions de type :

```
when(ouvrage.nbExemplairesDisponibles()).thenReturn(2);
when(ouvrage.listeDesMediathequesAvecNbOuvrageDisponible()).thenReturn(11);
```

Vous pourrez ainsi utiliser des assertions pour vérifier que ce que retourne listeMediatheques est correct.

6 Paramétrage un peu plus fin avec Mockito

Question 1. Créez l'interface suivante.

```
public interface I {
    public void methodeVoid() throws Exception;
    public int methodeInt() throws Exception;
    public int methodeParam(int i);
    public int methodeParamArrayList(ArrayList<String> l);
}
```

Dans une classe de test, déclarez et créez un mock sur I (avec l'annotation @Mock ou la méthode mock()). Ecrivez les méthodes de test suivantes :

- vérification de la valeur par défaut pour une méthode mockée retournant un int : vérifiez (par une assertion) que l'appel de `methodeInt` retourne 0.
- mocker une méthode retournant un résultat, utilisation de `verify`. Faites retourner au mock successivement les valeurs 1, 2, 3 et 4 lors de l'appel à la méthode `methodeInt`. Puis appelez 4 fois `methodeInt`. Vérifiez (`verify`) que la méthode `methodeInt` a bien été appelée 4 fois. Vérifiez ensuite la valeur retournée par le mock lors des appels suivants à `methodeInt`.
- mocker une méthode avec type de retour qui retourne une exception. Mockez `MethodeInt` de manière à ce qu'elle jette systématiquement une exception. Appelez la méthode `methodeInt` et assurez-vous que l'exception est lancée (utilisez le `expected=Exception.class` de l'annotation `Test` de Junit).
- mocker une méthode void qui retourne une exception. Mockez `methodeVoid` de manière à ce qu'elle jette une exception. Appelez la méthode `methodeVoid` et assurez-vous que l'exception est lancée.
- utilisation de paramètres dans les mocks. Mockez la méthode `methodeParam` de manière à ce que si on l'appelle avec 3, elle retourne 3 et que si on l'appelle avec 5 elle retourne 10. Appelez `methodeParam` successivement avec pour paramètres 1, 3 et 5 et assurez-vous que vous obtenez bien 0, 3 puis 10.
- utilisation de matchers sur les entiers. Mockez la méthode `methodeParam` de manière à ce que si on l'appelle avec n'importe quel entier strictement plus grand que 10, elle retourne 42, sinon elle retourne 0. On utilisera pour cela les additional matchers `gt` et `leq` (*greater than* et *lower or equal to*).
- utilisation de matchers sur les listes. Mockez la méthode `methodeParamArrayList` de manière à ce qu'elle retourne 42 si sa liste en paramètre contient la chaîne "42" ou est de taille 1 (et 0 sinon). On utilisera un `argThat` et un `ArgumentMatcher` personnalisé, avec une lambda (voir <https://static.javadoc.io/org.mockito/mockito-core/2.1.0/org/mockito/ArgumentMatcher.html>).

7 Gérer le temps ...

On vous donne à tester une classe `Adhérent` (voir sur Moodle). Cette classe `Adhérent` possède une méthode qui est appelée à chaque nouvelle année, la cotisation de l'adhérent devient alors non payée, et dans le cas où il faut radier l'abonné, elle retourne vrai, sinon retourne faux. On décide de radier un abonné au bout de 5 années sans cotisation (si un abonné cotise en 2010 mais pas en 2011 ni 2012 ni 2013 ni 2014 ni 2015, alors il est radié en 2016).

Ecrivez des tests vérifiant si la méthode `nouvelleAnnee` est correcte. Vous serez amenés à modifier le code pour le rendre plus facilement testable.

8 Gérer l'aléatoire ...

On reprend l'exemple classique d'un jeu de dé. Mettez en place une classe `Dice` dont le constructeur détermine le nombre de faces du dé et l'instance de `Random` à utiliser, et qui contient une méthode `roll` qui permet de lancer le dé. Cette méthode lance une exception de type `DiceException` si la valeur qu'elle calcule à partir de l'instance de `random` est en dehors du domaine de valeurs du dé.

```
public class Dice {
    private final int nbFaces;
    private Random randomGen;

    public Dice(int nbFaces, Random randomGen) {
        this.nbFaces=nbFaces;
        this.randomGen = randomGen;
    }

    public int roll() throws DiceException {
        int result = randomGen.nextInt(nbFaces) + 1; // 0 is not a possible value but nbFaces is.
        if (result > nbFaces || result <=0)
            throw new DiceException("Incorrect_value");
        return result;
    }
}
```

Pour tester que l'exception est bien levée, il va falloir transmettre une instance de `Random` pour laquelle le `nextInt` sort de ses bornes. On va pour cela utiliser un mock de `Random`.

Rappel : en JUnit Jupiter, pour tester la levée d'exception, on utilise :

```
assertThrows(DiceException.class, ()->{ dice.roll(); });
```

On imagine maintenant un simulateur de jeu de dés (pas malin) à deux joueurs et trois tours maximum. A chaque tour, les joueurs lancent un dé. En cas d'égalité, les deux joueurs relancent jusqu'à se départager. Le jeu est gagné par 2 tours gagnants, donc le troisième tour n'a lieu que si chaque joueur a gagné l'un des 2 premiers tours. La méthode qui lance le jeu retourne le nom du joueur gagnant. Des méthodes auxiliaires permettent de récupérer les scores aux différents tours et le nombre de tours effectivement joués.

Mettez en place le jeu et testez-le en utilisant des mocks de dés qui vous permettent de contrôler l'aléatoire et donc de tester que le jeu se déroule correctement.

9 Application aux dictionnaires

Question 2. Certaines situations étaient difficiles à faire apparaître lors du test des dictionnaires (comme par exemple le conflit de hachage et la position d'un tel conflit en fin de tableau pour tester la recherche circulaire). Reprenez avec Mockito vos tests sur les dictionnaires de manière à utiliser les mocks pour faire apparaître ces situations artificiellement. Vous serez certainement amenés pour cela à refactoriser une partie de votre code pour le rendre plus facilement testable.