# Cours sur l'utilisation de JUnit5 et Mockito dans le contexte des tests unitaires

## 1. Introduction

Dans ce cours, nous allons explorer l'utilisation de **JUnit5** pour tester un portefeuille électronique (`Purse`) et un code secret (`CodeSecret`). Nous allons également utiliser **Mockito** pour simuler des comportements (mocks) afin de rendre les tests plus isolés et contrôlés. Nous allons aborder les éléments suivants :

- **Tests unitaires avec JUnit5**
- **Mocking avec Mockito**
- **Assertions et vérifications**

## 2. Test Unitaire avec JUnit5

### 2.1. Structure d'un test unitaire avec JUnit5

Un test unitaire en JUnit5 est structuré de la manière suivante :

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

public class MonTest {

    @BeforeEach
    public void setup() {
        // Initialisation avant chaque test
    }

    @Test
    public void testFonctionnalite() {
        // Test de la fonctionnalité
        Assertions.assertEquals(valeurAttendue, valeurTestee);
    }
}
```

### 2.2. Exemple : Test de la classe `Purse`

Voici un exemple de test de la classe `Purse` qui teste un portefeuille électronique.

```java
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.Assertions;
import org.mockito.Mockito;

public class PurseUnitTest {

    private String codeJuste = "9876";  // Code secret valide pour les tests
    private CodeSecret mockPinCode;  // Mock pour le code secret

    @BeforeEach
    public void setup() {
        mockPinCode = Mockito.mock(CodeSecret.class);
        Mockito.when(mockPinCode.verifierCode(codeJuste)).thenReturn(true);  // Code secret valide
    }

    @Test
    public void testCredit() throws MotifBlocageTransaction {
        Purse purse = new Purse(100, 100, mockPinCode);  // Créer un portefeuille avec un plafond de 100
        double solde = purse.getSolde();  // Récupérer le solde initial
        purse.credit(10);  // Effectuer un crédit de 10
        Assertions.assertEquals(solde + 10, purse.getSolde());  // Vérifier si le solde est bien mis à jour
```

```
    }
}
```

Dans cet exemple, nous testons la méthode `credit()` de la classe `Purse`. Le test vérifie que le solde du portefeuille est bien mis à jour après un crédit de 10 unités.

### 2.3. Tests d'assertion

Les **assertions** permettent de vérifier que le comportement observé est celui attendu.

Quelques exemples d'assertions courantes : - `Assertions.assertEquals(valeurAttendue, valeurTestee)` - `Assertions.assertTh () -> ... )` - `Assertions.assertTrue(condition)` - `Assertions.assertFalse(condition)`

### 2.4. Test de la classe `CodeSecret`

Voici un exemple pour tester un code secret avec **JUnit5** :

```java
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.Assertions;
import org.mockito.Mockito;

public class CodeSecretUnitTest {

    private CodeSecret codeSecret;

    @BeforeEach
    public void setup() {
        MyRandom random = Mockito.mock(MyRandom.class);
        Mockito.when(random.nextInt(10)).thenReturn(5, 4, 3, 2);
        codeSecret = CodeSecret.createCode(random);
    }

    @Test
    public void testRevelerCode() {
        Assertions.assertTrue(isCode(codeSecret.revelerCode()));
        Assertions.assertEquals("xxxx", codeSecret.revelerCode());
    }

    private boolean isCode(String code) {
        if (code.length() != 4) return false;
        try {
            Integer.parseInt(code);
        } catch (NumberFormatException e) {
            return false;
        }
        return true;
    }
}
```

Dans ce test, nous créons un objet `CodeSecret` et nous simulons le comportement du générateur de nombres aléatoires avec **Mockito** pour contrôler la génération du code secret.

## 3. Mocking avec Mockito

Mockito permet de simuler des objets ou des comportements. Dans les tests, les objets réels peuvent être difficiles à tester directement, donc Mockito nous permet de créer des **mocks** qui imitent ces objets.

### 3.1. Création d'un mock avec Mockito

```java
MyRandom random = Mockito.mock(MyRandom.class);
Mockito.when(random.nextInt(10)).thenReturn(5, 4, 3, 2);  // On définit la réponse attendue
```

Ici, nous simulons la méthode `nextInt(10)` pour qu'elle retourne successivement `5`, `4`, `3`, `2`.

### 3.2. Vérification des appels

Vous pouvez aussi vérifier qu'une méthode a été appelée un certain nombre de fois :

```
Mockito.verify(random, Mockito.times(4)).nextInt(10);  // Vérifie que nextInt a été appelé 4 fois
```

## 4. Utilisation des Mocks pour les Tests

Les **mocks** sont utilisés pour remplacer des composants externes qui pourraient rendre les tests complexes ou non isolés. Dans le cas du portefeuille (`Purse`), nous utilisons un mock de `CodeSecret` pour éviter la gestion réelle des codes secrets.

Exemple de test d'un portefeuille (`Purse`) avec un mock de `CodeSecret` :

```
@Test
public void testDebitRejeteSurCodeFaux() throws Exception {
    String codeFaux = "1234";  // Code secret incorrect
    Purse purse = new Purse(100, 50, mockPinCode);
    purse.credit(50);  // Créer un portefeuille et effectuer un crédit de 50
    Assertions.assertThrows(CodeFauxException.class, () -> purse.debit(20, codeFaux));  // Tentative de débit
}
```

Ici, nous simulons un débit dans le portefeuille avec un code incorrect, et nous attendons qu'une exception `CodeFauxException` soit lancée.

## 5. Tests de comportements complexes avec Mockito

Mockito est aussi utile pour tester des comportements complexes comme les tentatives multiples d'entrer un mauvais code, ce qui peut entraîner un blocage du code.

```
@Test
public void testCodeBloqueApres3essaisFauxNonSuccessifs() {
    Assertions.assertFalse(codeSecret.verifierCode("2345"));
    Assertions.assertFalse(codeSecret.verifierCode("2345"));
    Assertions.assertFalse(codeSecret.verifierCode("2345"));
    Assertions.assertTrue(codeSecret.isBlocked());
}
```

Dans cet exemple, nous simulons trois tentatives avec un mauvais code. Après trois tentatives infructueuses, le code doit être bloqué.

## 6. Bonnes Pratiques

- **Isolation des tests** : Utilisez les mocks pour simuler des dépendances extérieures (ex. : code secret, génération de nombres aléatoires).
- **Utilisation des annotations JUnit5** : `@BeforeEach` pour l'initialisation avant chaque test, `@Test` pour les méthodes de test, et `@Disabled` pour ignorer certains tests.
- **Assertions claires et explicites** : Utilisez des assertions pour tester des valeurs spécifiques, des exceptions attendues, et des comportements.

## 7. Conclusion

Les tests unitaires avec **JUnit5** et **Mockito** sont essentiels pour garantir la fiabilité du code, en particulier dans des systèmes complexes comme un portefeuille électronique (`Purse`). Grâce à **Mockito**, nous pouvons simuler des comportements et tester des scénarios de manière isolée. Les assertions de **JUnit5** nous permettent de vérifier les résultats attendus de manière précise.

En combinant **JUnit5** pour la structure des tests et **Mockito** pour le mocking, nous pouvons tester efficacement des composants tout en réduisant les dépendances externes et en contrôlant les comportements complexes.