Tests unitaires avec **JUnit 5** et **Mockito 5**

Achref El Mouelhi

Docteur de l'université d'Aix-Marseille
Chercheur en programmation par contrainte (IA)
Ingénieur en génie logiciel

elmouelhi.achref@gmail.com

Plan

Plan

Plan

9 Quelques méthodes pour les mocks

- `mock`
- `when ... thenReturn`
- `doReturn ... when`
- `verify`
- `times`
- `atLeast`
- `any`
- `anyInt`
- `never`
- `only`
- `inOrder`
- `when ... thenAnswer`
- `doAnswer ... when`
- `when ... thenThrow`
- `doThrow ... when`
- `spy`

Plan

JUnit

Deux catégories de tests

- tests manuels :
  - coûteux
  - répétitif (corriger et retester)
- tests automatiques (via des programmes)

JUnit

Plusieurs niveaux de tests

- tests unitaires
- tests d'intégration
- tests fonctionnels
- tests de charge
- tests d'acceptation

JUnit

tests unitaires

- Permettent de tester une **unité (partie) isolée** de l'application (souvent une fonction ou une méthode).
- Indiquent au développeur que le code fait **bien les choses**.

JUnit

tests unitaires

- Permettent de tester une **unité (partie) isolée** de l'application (souvent une fonction ou une méthode).
- Indiquent au développeur que le code fait **bien les choses**.

Remarque

Les tests qui communiquent avec une base de données
ne sont pas des tests unitaires, mais plutôt des tests d'intégration.

JUnit

tests d'intégration

- permettent de vérifier que les **unités isolées** fonctionnent correctement ensemble.
- peuvent nécessiter de composants extérieurs (Service web, base de données...).

JUnit

Autres niveaux de tests

- **tests fonctionnels** (de bout-en-bout ou **end-to-end** en anglais) : sont écrits du point de vue de l'utilisateur final depuis l'interface utilisateur, pour vérifier que l'application répond aux exigences.
  Ils indiquent au développeur que le code fait **les bonnes choses**.

- **tests d'acceptation** : permettent de vérifier que l'**application** respecte bien le besoin fonctionnel.

- **tests de charge** (système) : permettent de vérifier si un système peut gérer une charge spécifiée : le nombre d'utilisateurs simultanés...

JUnit

Et les tests de non-régression ?

Ils permettent de vérifier que la livraison de nouvelles fonctionnalités n'aura pas d'effet de bord sur les fonctionnalités existantes (programme préalablement testé).

JUnit

Autres tests

- **tests de performance**
- **tests de fiabilité**
- ...

JUnit

JUnit ?

- Framework open-source pour **Java** créé par Kent Beck et Erich Gamma.
- Permettant d'automatiser les tests et de s'assurer que le programme répond toujours aux besoins.
- Basé sur les assertions qui vérifient si les résultats de tests correspondent aux résultats attendus.
- Membre de la famille **XUnit** (**CPPUnit** pour **C++**, **CUnit** pour **C**, **PHPUnit** pour **PHP**...).

JUnit

JUnit ?

- Framework open-source pour **Java** créé par Kent Beck et Erich Gamma.
- Permettant d'automatiser les tests et de s'assurer que le programme répond toujours aux besoins.
- Basé sur les assertions qui vérifient si les résultats de tests correspondent aux résultats attendus.
- Membre de la famille **XUnit** (**CPPUnit** pour **C++**, **CUnit** pour **C**, **PHPUnit** pour **PHP**...).

Objectif

Trouver un maximum de bugs pour les corriger.

JUnit

JUnit 5 est composé de 3 modules

- **JUnit Jupiter** : nouveau module de tests
- **JUnit Vintage** : support d'exécution pour **JUnit 3** et **JUnit 4**
- **JUnit Platform** : support d'exécution pour différents environnements (**Maven**, **Eclipse**, **IntelliJ**, **Jenkins**...)

JUnit

TestCase (cas de test)

Classe Java

- Contenant quelques méthodes de test (annotées par `@Test`)
- Permettant de tester le bon fonctionnement d'une classe (en testant ses méthodes)

Remarque

Si le test ne détecte pas d'erreur $\not\Rightarrow$ il n'y en a pas.

JUnit

Étape

- Création d'un `Java Project`
- Création de deux `Package` : `org.eclipse.main` et `org.eclipse.test`
- Pour chaque classe créée dans `org.eclipse.main`, on lui associe une classe de test (dans `org.eclipse.test`)
- On prépare le test et ensuite on le lance : s'il y a une erreur, on la corrige et on relance le test.

JUnit

Création d'une première classe `Calcul`

```
package org.eclipse.main;

public class Calcul {
  public int somme(int x, int y) {
    return x + y;
  }
  public int division(int x, int y) {
    if (y == 0)
      throw new  ArithmeticException();
    return   x / y;
  }
}
```

JUnit

Pour créer une classe de test

- Faire un clic droit sur le package `org.eclipse.test`
- Aller dans `New > JUnit Test Case`
- Saisir le nom `CalculTest` dans `Name`
- Cocher les 4 cases de `Which method stubs would you like to create ?`
- Cliquer sur `Browse` en face de `Class under test`
- Chercher `calcul`, sélectionner **`Calcul`** `- org.eclipse.main` et valider
- Cliquer sur `Next` puis cocher les cases correspondantes de `somme` et `division` dans `Calcul`
- Cliquer sur `Finish` puis sur `ok` (pour valider `Add JUnit 5 library to the build path` dans `Perform the following action:`)

Code généré

```
package org.eclipse.test;

import static org.junit.jupiter.api.Assertions.fail;
import org.junit.jupiter.api.AfterAll;
import org.junit.jupiter.api.AfterEach;
import org.junit.jupiter.api.BeforeAll;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;

class CalculTest {

    @BeforeAll
    static void setUpBeforeClass() throws Exception { }

    @AfterAll
    static void tearDownAfterClass() throws Exception { }

    @BeforeEach
    void setUp() throws Exception { }

    @AfterEach
    void tearDown() throws Exception { }

    @Test
    void testSomme() {
        fail("Not yet implemented");
    }

    @Test
    void testDivision() {
        fail("Not yet implemented");
    }
}
```

JUnit

Nous parlerons de ces quatre méthodes dans une autre section

```
@BeforeAll
static void setUpBeforeClass() throws Exception {
}

@AfterAll
static void tearDownAfterClass() throws Exception {
}

@BeforeEach
void setUp() throws Exception {
}

@AfterEach
void tearDown() throws Exception {
}
```

JUnit

Pour tester

- Faire un clic droit sur le la classe de test
- Aller dans `Run As > JUnit Test`

JUnit

Pour tester

- Faire un clic droit sur le la classe de test
- Aller dans `Run As > JUnit Test`

Résultat

2 Exécutions : 2 Échecs : car les deux méthodes de test sont vides.

JUnit

Implémentons `testSomme()` en ciblant chaque fois les cas particuliers

```
void testSomme() {
  Calcul calcul = new Calcul();
  if (calcul.somme(2, 3) != 5)
    fail("faux pour deux entiers positifs");
  if (calcul.somme(-2, -3) != -5)
    fail("faux pour deux entiers négatifs");
  if (calcul.somme(-2, 3) != 1)
    fail("faux pour deux entiers de signe différent");
  if (calcul.somme(0, 3) != 3)
    fail("faux pour x nul");
  if (calcul.somme(2, 0) != 2)
    fail("faux pour y nul");
  if (calcul.somme(0, 0) != 0)
    fail("faux pour x et y nuls");
}
```

JUnit

Implémentons `testSomme()` en ciblant chaque fois les cas particuliers

```
void testSomme() {
  Calcul calcul = new Calcul();
  if (calcul.somme(2, 3) != 5)
    fail("faux pour deux entiers positifs");
  if (calcul.somme(-2, -3) != -5)
    fail("faux pour deux entiers négatifs");
  if (calcul.somme(-2, 3) != 1)
    fail("faux pour deux entiers de signe différent");
  if (calcul.somme(0, 3) != 3)
    fail("faux pour x nul");
  if (calcul.somme(2, 0) != 2)
    fail("faux pour y nul");
  if (calcul.somme(0, 0) != 0)
    fail("faux pour x et y nuls");
}
```

En testant, plus d'échec pour `somme`.

JUnit

Remarques

Pour tester une méthode, on peut aussi utiliser les assertions ou les assomptions.

JUnit

Remarques

Pour tester une méthode, on peut aussi utiliser les assertions ou les assomptions.

Assertions

- Méthodes statiques définies dans la classe `Assertions`.
- Permettant de vérifier le bon déroulement d'un test : si la vérification échoue, l'assertion lève une exception et le test échoue.
- Acceptant depuis la version 5 les expressions Lambda.

Quelques assertions

- `assertTrue(condition, message)` : permet de vérifier que la condition fournie en paramètre est vraie.
- `assertFalse(condition, message)` : permet de vérifier que la condition fournie en paramètre est fausse.
- `assertEquals(valeur_attendue, appel_de_méthode, message)` : permet de vérifier l'égalité de valeur (sa réciproque est `assertNotEquals`).
- `assertSame(objet_attendu, objet_retourné, message)` : permet de vérifier si `objet_attendu` et `objet_retourné` réfèrent le même objet (sa réciproque est `assertNotSame`).
- `assertArrayEquals(tableau_attendu, tableau_retourné, message)` : permet de vérifier l'égalité de deux tableaux.
- `assertIterableEquals(iterable_attendu, iterable_retourné, message)` : permet de vérifier l'égalité de deux itérables.
- `assertNotNull(message, object)` permet de vérifier, pour les paramètres utilisés, qu'une méthode ne retourne pas la valeur `null` (sa réciproque est `assertNull`).
- ...

JUnit

Remarques

- En l'absence d'un message explicite, un message d'erreur par défaut sera affiché.
- Les méthodes `assertX()` peuvent aussi avoir la signature suivante : `assertX(message, valeurAttendue, appelDeMéthodeATester)`

Implémentons la méthode `testDivision()`

```
void testDivision() {
  Calcul calcul = new Calcul();
  assertFalse(calcul.division(6, 3) == 0, "2 entiers positifs");
  assertEquals(2, calcul.division(-6, -3), "2 entiers négatifs");
  assertNotNull(calcul.division(-6, 3), "2 entiers de signe diff");
  assertTrue(calcul.division(0, 3) == 0, "entier x nul");
  Throwable e = null;
  try {
    calcul.division(2, 0);
  }
  catch (Throwable ex) {
    e = ex;
  }
  assertTrue(e instanceof ArithmeticException);
  e = null;
  try {
    calcul.division(0, 0);
  }
  catch (Throwable ex) {
    e = ex;
  }
  assertTrue(e instanceof ArithmeticException);
}
```

JUnit

Les imports nécessaires

```
import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.junit.jupiter.api.Assertions.assertFalse;
import static org.junit.jupiter.api.Assertions.assertTrue;
import static org.junit.jupiter.api.Assertions.assertNotNull;
```

JUnit

Ajouter des valeurs erronées pour avoir un échec

- Le message défini sera affiché dans le panneau `Failure Trace` (vous pouvez cliquer sur l'icône d'écran, `Show Stack Trace in Console View`, en face `Failure Trace` pour visualiser les détails de l'erreur dans la console).
- Dans ce cas, il faut localiser l'erreur, la corriger et relancer.

Par exemple, les deux premières assertions ne sont plus correctes

```
void testDivision() {
  Calcul calcul = new Calcul();
  assertFalse(calcul.division(6, 3) == 2, "2 entiers positifs");
  assertEquals(1, calcul.division(-6, -3), "2 entiers négatifs");
  assertNotNull(calcul.division(-6, 3), "2 entiers de signe diff");
  assertTrue(calcul.division(0, 3) == 0, "entier x nul");
  Throwable e = null;
  try {
    calcul.division(2, 0);
  }
  catch (Throwable ex) {
    e = ex;
  }
  assertTrue(e instanceof ArithmeticException);
  e = null;
  try {
    calcul.division(0, 0);
  }
  catch (Throwable ex) {
    e = ex;
  }
  assertTrue(e instanceof ArithmeticException);
}
```

JUnit

Constats

Deux assertions incorrectes mais une seule exception retournée.

JUnit

Constats

Deux assertions incorrectes mais une seule exception retournée.

Solution

Regrouper les assertions avec `assertAll`

JUnit

Regroupons les assertions avec `assertAll`

```
void testDivision() {
  Calcul calcul = new Calcul();
  assertAll("premier regroupement",
    () -> assertFalse(calcul.division(6, 3) == 2, "2 entiers positifs"),
    () -> assertEquals(1, calcul.division(-6, -3), "2 entiers négatifs"),
    () -> assertNotNull(calcul.division(-6, 3), "2 entiers de signe diff"),
    () -> assertTrue(calcul.division(0, 3) == 0, "entier x nul")
  );
  Throwable e = null;
  try {
    calcul.division(2, 0);
  } catch (Throwable ex) {
    e = ex;
  }
  assertTrue(e instanceof ArithmeticException);
  e = null;
  try {
    calcul.division(0, 0);
  } catch (Throwable ex) {
    e = ex;
  }
  assertTrue(e instanceof ArithmeticException);
}
```

JUnit

Les imports nécessaires

```
import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.junit.jupiter.api.Assertions.assertFalse;
import static org.junit.jupiter.api.Assertions.assertTrue;
import static org.junit.jupiter.api.Assertions.assertNotNull;
import static org.junit.jupiter.api.Assertions.assertAll;
```

JUnit

Les imports nécessaires

```
import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.junit.jupiter.api.Assertions.assertFalse;
import static org.junit.jupiter.api.Assertions.assertTrue;
import static org.junit.jupiter.api.Assertions.assertNotNull;
import static org.junit.jupiter.api.Assertions.assertAll;
```

En lançant les tests précédents, les deux échecs sont signalés ainsi que le nom du regroupement

```
org.opentest4j.MultipleFailuresError: premier regroupement (2 failures)
    org.opentest4j.AssertionFailedError: 2 entiers positifs ==> expected: <false> but was: <
      true>
    org.opentest4j.AssertionFailedError: 2 entiers négatifs ==> expected: <1> but was: <2>
```

JUnit

Assumptions (suppositions)

- Méthodes statiques définies dans la classe `Assumptions`.
- Permettant de conditionner l'exécution d'un bloc de test : si l'évaluation de l'assumption échoue, le test sera désactivé sans être échoué.

Modifions `testDivision()` pour que l'ensemble de tests précédents ne soit exécuté que si on travaille dans un environnement Windows

```
void testDivision() {
  Calcul calcul = new Calcul();
  assumeTrue(System.getenv("OS").startsWith("Windows"));
  assertFalse(calcul.division(6, 3) == 0, "2 entiers positifs");
  assertEquals(2, calcul.division(-6, -3), "2 entiers négatifs");
  assertNotNull(calcul.division(-6, 3), "2 entiers de signe diff");
  assertTrue(calcul.division(0, 3) == 0, "entier x nul");
  Throwable e = null;
  try {
    calcul.division(2, 0);
  } catch (Throwable ex) {
    e = ex;
  }
  assertTrue(e instanceof ArithmeticException);
  e = null;
  try {
    calcul.division(0, 0);
  } catch (Throwable ex) {
    e = ex;
  }
  assertTrue(e instanceof ArithmeticException);
}
```

Modifions `testDivision()` pour que l'ensemble de tests précédents ne soit exécuté que si on travaille dans un environnement Windows

```
void testDivision() {
  Calcul calcul = new Calcul();
  assumeTrue(System.getenv("OS").startsWith("Windows"));
  assertFalse(calcul.division(6, 3) == 0, "2 entiers positifs");
  assertEquals(2, calcul.division(-6, -3), "2 entiers négatifs");
  assertNotNull(calcul.division(-6, 3), "2 entiers de signe diff");
  assertTrue(calcul.division(0, 3) == 0, "entier x nul");
  Throwable e = null;
  try {
    calcul.division(2, 0);
  } catch (Throwable ex) {
    e = ex;
  }
  assertTrue(e instanceof ArithmeticException);
  e = null;
  try {
    calcul.division(0, 0);
  } catch (Throwable ex) {
    e = ex;
  }
  assertTrue(e instanceof ArithmeticException);
}
```

L'import nécessaire

```
import static org.junit.jupiter.api.Assumptions.assumeTrue;
```

Ou en utilisant `assumingThat()` fonctionnant avec une expression Lambda

```
void testDivision() {
  Calcul calcul = new Calcul();
  assumingThat(System.getenv("OS").startsWith("Windows"), () -> {
    assertFalse(calcul.division(6, 3) == 0, "2 entiers positifs");
    assertEquals(2, calcul.division(-6, -3), "2 entiers négatifs");
    assertNotNull(calcul.division(-6, 3), "2 entiers de signe diff");
    assertTrue(calcul.division(0, 3) == 0, "entier x nul");
    Throwable e = null;
    try {
      calcul.division(2, 0);
    } catch (Throwable ex) {
      e = ex;
    }
    assertTrue(e instanceof ArithmeticException);
    e = null;
    try {
      calcul.division(0, 0);
    } catch (Throwable ex) {
      e = ex;
    }
    assertTrue(e instanceof ArithmeticException);
  });
}
```

JUnit

Explication

- Si `assumeX()` échoue, les assertions ne seront pas vérifiées et par conséquent le test passera toujours.
- Sinon les assertions seront vérifiées.

JUnit

Quelques autres assumptions

- `assumeTrue()`
- `assumeFalse()`
- `assumeThat()`
- `assumingThat()`
- `assumeNoException()`
- `assumeNotNull()`

JUnit

Dans certains cas

- Avant de démarrer un test, il faut faire certains traitements :
  - instancier un objet de la classe,
  - se connecter à une base de données,
  - ouvrir un fichier...
- Après le test, il faut aussi fermer certaines ressources : connexion à une base de données, socket...

JUnit

Dans certains cas

- Avant de démarrer un test, il faut faire certains traitements :
  - instancier un objet de la classe,
  - se connecter à une base de données,
  - ouvrir un fichier...
- Après le test, il faut aussi fermer certaines ressources : connexion à une base de données, socket...

Solution

On peut utiliser les méthodes `setUp()` et `tearDown()` qui sont respectivement exécutées avant et après chaque appel d'une méthode de test.

Reprenons les quatre méthodes précédentes et modifions le code

```
@BeforeAll
static void setUpBeforeClass() throws Exception {
  System.out.println("BeforeAll");
}

@AfterAll
static void tearDownAfterClass() throws Exception {
  System.out.println("AfterAll");
}

@BeforeEach
void setUp() throws Exception {
  System.out.println("BeforeEach");
}

@AfterEach
void tearDown() throws Exception {
  System.out.println("AfterEach");
}
```

JUnit

Pour mieux comprendre

Lancer le test **JUnit**

JUnit

Pour mieux comprendre

Lancer le test **JUnit**

Le résultat

```
BeforeAll
BeforeEach
AfterEach
BeforeEach
AfterEach
AfterAll
```

JUnit

Comprenons les annotations de méthodes précédentes

- `@BeforeAll` : la méthode annotée sera exécutée seulement avant le premier test
- `@AfterAll` : la méthode annotée sera exécutée seulement après le dernier test
- `@BeforeEach` : la méthode annotée sera exécutée avant chaque test
- `@AfterEach` : la méthode annotée sera exécutée après chaque test

JUnit

Remarques

- `@BeforeAll` était `@BeforeClass` dans **JUnit 4.**
- `@AfterAll` était `@AfterClass` dans **JUnit 4.**
- `@BeforeEach` était `@Before` dans **JUnit 4.**
- `@AfterEach` était `@After` dans **JUnit 4.**

JUnit

Utilisons ces méthodes pour restructurer la classe `CalculTest`

```
class CalculTest {

  Calcul calcul;

  @BeforeAll
  static void setUpBeforeClass() throws Exception { }

  @AfterAll
  static void tearDownAfterClass() throws Exception { }

  @BeforeEach
  void setUp() throws Exception {
    calcul = new Calcul();
  }

  @AfterEach
  void tearDown() throws Exception {
    calcul = null;
  }
  @Test
  void testSomme() {
    // le code précédent sans l'instanciation de calcul
  }
  @Test
  void testDivision() {
    // le code précédent sans l'instanciation de calcul
  }
}
```

JUnit

Commençons par créer un **Java Project** avec **Maven**

- Aller dans `File > New > Other`
- Chercher puis sélectionner `Maven Project`
- Cliquer sur `Next`
- Choisir `maven-archetype-quickstart`
- Remplir les champs
  - `Group Id` avec `org.eclipse`
  - `Artifact Id` avec `cours-junit-maven`
  - `Package` avec `org.eclipse.main`

JUnit

Vérifier l'existence des deux répertoires

- `/src/main/java` : code source
- `/src/test/java` : code source de test
- ...

JUnit

Vérifier l'existence des deux répertoires

- `/src/main/java` : code source
- `/src/test/java` : code source de test
- ...

S'il n'y a pas de `src/main/java` ou `src/test/java`

- Faire clic droit sur le projet
- Aller dans `Build Path > Configure Build Path...`
- Cliquer sur `Order and Export`
- Cocher les cases `Maven Dependencies` et `JRE System Library`
- Cliquer sur `Apply and Close`

JUnit

Ajouter les propriétés suivantes dans la section `properties` de `pom.xml`

```
<properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <maven.compiler.source>18</maven.compiler.source>
    <maven.compiler.target>18</maven.compiler.target>
    <junit-jupiter.version>5.9.3</junit-jupiter.version>
    <junit-platform.version>1.9.3</junit-platform.version>
</properties>
```

JUnit

Ajouter les dépendances suivantes dans la section `dependencies` de `pom.xml`

```
<dependency>
    <groupId>org.junit.platform</groupId>
    <artifactId>junit-platform-runner</artifactId>
    <version>${junit-platform.version}</version>
    <scope>test</scope>
</dependency>

<dependency>
        <groupId>org.junit.jupiter</groupId>
        <artifactId>junit-jupiter</artifactId>
        <version>${junit-jupiter.version}</version>
        <scope>test</scope>
</dependency>
```

JUnit

Pour tester

- Déplacez le package `org.eclipse.main` du projet précédent dans `src/main/java`.
- Déplacez le package `org.eclipse.test` dans `src/test/java` et renommez le `org.eclipse.main`.
- Lancez le test et vérifie qu'il passe

JUnit

Pour modifier l'affichage de la classe de test, on utilise `@DisplayName`

```
package org.eclipse.main;

import org.junit.jupiter.api.DisplayName;

// + les imports précédents

@DisplayName("Test de la classe Calcul")
class CalculTest {

  // + tout le code précédent

}
```

JUnit

`@DisplayName` **peut être utilisée aussi avec les méthodes**

```
@Test
@DisplayName("Test de la méthode somme")
void testSomme() {
    ...
}
```

JUnit

Pour désactiver un test, on utilise l'annotation `@Disabled` (ou `@Ignore` dans JUnit 4)

```
@Disabled
@Test
void testDivision() {
    ...
}
```

JUnit

Pour désactiver un test, on utilise l'annotation `@Disabled` (ou `@Ignore` dans JUnit 4)

```
@Disabled
@Test
void testDivision() {
    ...
}
```


Remarque

`@Disabled` peut être aussi utilisée au niveau de la classe.

JUnit

Utiliser `@RepeatedTest` pour répéter un test plusieurs fois

```
@RepeatedTest(3)
void testSomme(RepetitionInfo repetitionInfo) {
  assertNotEquals(7, calcul.somme(repetitionInfo.getCurrentRepetition(), 3));
}
```

JUnit

Utiliser `@RepeatedTest` pour répéter un test plusieurs fois

```
@RepeatedTest(3)
void testSomme(RepetitionInfo repetitionInfo) {
  assertNotEquals(7, calcul.somme(repetitionInfo.getCurrentRepetition(), 3));
}
```


Explication

- Il faut remplacer `@Test` par `@RepeatedTest`.
- Pour récupérer l'index de l'itération courante, on déclare un objet de type `RepetitionInfo`.

JUnit

On peut utiliser `@RepeatedTest` pour personnaliser le message afficher

```
@RepeatedTest(value = 3, name = RepeatedTest.LONG_DISPLAY_NAME)
void testSomme(RepetitionInfo repetitionInfo) {
        assertNotEquals(7, calcul.somme(repetitionInfo.getCurrentRepetition(), 3));
}
```

JUnit

Utiliser `@ParameterizedTest` pour paramétrer un test

```
@DisplayName("Test de la methode somme")
@ParameterizedTest
@ValueSource(ints = { 2, 3 })
void testSomme(int t) {
    assertNotEquals(5, calcul.somme(t, 1));
}
```

JUnit

Utiliser `@ParameterizedTest` pour paramétrer un test

```
@DisplayName("Test de la methode somme")
@ParameterizedTest
@ValueSource(ints = { 2, 3 })
void testSomme(int t) {
    assertNotEquals(5, calcul.somme(t, 1));
}
```

Explication

- Il faut remplacer `@Test` par `@ParameterizedTest`.
- Cette méthode sera testée deux fois : une fois pour la valeur `2` et une fois pour la valeur `3`.
- `@ValueSource` indique les valeurs à injecter dans `t` à chaque appel.
- Il existe aussi `strings`, `longs` et `doubles`.

JUnit

Si vous utilisez une version de Jupiter inférieure à 5.9, vous devez rajouter également la dépendance suivante

```
<dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-params</artifactId>
    <version>${junit-jupiter.version}</version>
    <scope>test</scope>
</dependency>
```

JUnit

Types supportés par `@ValueSource`

- Tous les types primitifs en **Java**
- `java.lang.String`
- `java.lang.Class`

JUnit

Annotations à utiliser avec `@ParameterizedTest`

- `@ValueSource`
- `@EnumSource`
- `@CsvFileSource`
- `@MethodSource`
- ...

JUnit

Pour activer ou désactiver un test selon le système d'exploitation, on utilise soit `@DisabledOnOs` soit `@EnabledOnOs`

```
@Test
@DisabledOnOs(value = OS.MAC)
void testSomme() {
    ...
}
```

JUnit

Pour activer ou désactiver un test selon le système d'exploitation, on utilise soit `@DisabledOnOs` soit `@EnabledOnOs`

```
@Test
@DisabledOnOs(value = OS.MAC)
void testSomme() {
    ...
}
```


Autres constantes possibles : `LINUX`, `WINDOWS`

JUnit

Pour activer ou désactiver un test selon la version de `JRE`, on utilise soit `@DisabledOnJre` soit `@EnabledOnJre`

```
@Test
@DisabledOnOs(MAC)
@EnabledOnJre(value = JRE.JAVA_18)
void testSomme() {
    ...
}
```

JUnit

Pour activer ou désactiver un test selon la version de `JRE`, on utilise soit `@DisabledOnJre` soit `@EnabledOnJre`

```
@Test
@DisabledOnOs(MAC)
@EnabledOnJre(value = JRE.JAVA_18)
void testSomme() {
    ...
}
```

Il existe une constante pour chaque version de **Java**.

JUnit

Pour activer ou désactiver un test selon la valeur d'un test logique (on peut même utiliser une expression régulière), on utilise soit `@DisabledIf` soit `@EnabledIf`

```
class CalculTest {

    // le code précédent

    @Test
    @EnabledIf("checkNight")
    void testDivision() {
        // les assertions précédentes
    }

    // La méthode qui vérifie si c'est la nuit
    boolean checkNight() {
        return (LocalDateTime.now().getHour() >= 22 || LocalDateTime.now().getHour() <= 6);
    }

}
```

JUnit

Pour modifier l'ordre d'exécution de nos méthodes de test

```
class CalculTest {

    // code précédent

    @Test
    @Order(2)
    void testSomme() {
        // code précédent
    }

    @Test
    @Order(1)
    void testDivision() {
        // code précédent
    }

}
```

JUnit

Pour activer l'annotation `@Oder`, on utilise `@TestMethodOrder`

```
@TestMethodOrder(OrderAnnotation.class)
class CalculTest {

    // code précédent

    @Test
    @Order(2)
    void testSomme() {
        // code précédent
    }

    @Test
    @Order(1)
    void testDivision() {
        // code précédent
    }

}
```

JUnit

Pour activer l'annotation `@Oder`, on utilise `@TestMethodOrder`

```
@TestMethodOrder(OrderAnnotation.class)
class CalculTest {

    // code précédent

    @Test
    @Order(2)
    void testSomme() {
        // code précédent
    }

    @Test
    @Order(1)
    void testDivision() {
        // code précédent
    }

}
```

Relancez le test et vérifiez que `somme` a été testée après `division`.

JUnit

Mock ?

- Doublure, objet factice (fake object)
- permettant de reproduire le comportement d'un objet réel non implémenté

JUnit

Mock ?

- Doublure, objet factice (fake object)
- permettant de reproduire le comportement d'un objet réel non implémenté

Mockito ?

- Framework open-source pour **Java**
- Générateur automatique de doublures
- Un seul type de Mock possible et une seule façon de le créer

JUnit

Exemple, supposant qu'on

- a une interface `CalculService` ayant une méthode `carre()`
- veut développer une méthode `sommeCarre()` dans `Calcul` qui utilise la méthode `carre()` de cette interface `CalculService`

JUnit

L'interface `CalculService`

```
package org.eclipse.main;

public interface CalculService {

  public int carre(int x);

}
```

JUnit

La classe `Calcul`

```
package org.eclipse.main;

public class Calcul {

        CalculService calculService;

        public Calcul(CalculService calculService) {
                this.calculService = calculService;
        }

        public int sommeCarre(int x, int y) {
                return somme(calculService.carre(x), calculService.carre(y));
        }

        public int somme(int x, int y) {
                return x + y;
        }

        public int division(int x, int y) {
                if (y == 0)
                        throw new ArithmeticException();
                return x / y;
        }
}
```

JUnit

Pour tester la classe `Calcul` dans `TestCalcul`, il faut commencer par instancier `CalculService`

```
class CalculTest {

  Calcul calcul;
  CalculService calculService;

  @BeforeEach
  void setUp() throws Exception {
    calcul = new Calcul(calculService);
  }

  @Test
  void testSommeCarre() {
    assertTrue(calcul.sommeCarre(2, 3) == 13, "calcul exact");
  }

  // + le code précédent

}
```

JUnit

En testant, on aura l'erreur suivante

```
java.lang.NullPointerException
    at org.eclipse.main.Calcul.sommeCarre(Calcul.java:6)
    at org.eclipse.test.CalculTest.testSommeCarre(CalculTest.java:26)
```

JUnit

En testant, on aura l'erreur suivante

```
java.lang.NullPointerException
    at org.eclipse.main.Calcul.sommeCarre(Calcul.java:6)
    at org.eclipse.test.CalculTest.testSommeCarre(CalculTest.java:26)
```

Explication

- La source de l'erreur est l'appel de la méthode `carre(x)` qui n'est pas implémenté.
- Pour corriger cette erreur, on peut utiliser les STUB.
- STUB (les bouchons en français) : code qui renvoie le même résultat pour une méthode invoquée.

JUnit

Pour tester la classe `Calcul` dans `TestCalcul`, il faut commencer par instancier un objet d'une classe anonyme implémentant `CalculService`

```
class CalculTest {
  Calcul calcul;

  CalculService calculService =  new CalculService() {

    @Override
    public int carre(int x) {
      // TODO Auto-generated method stub
      return x * x;
    }
  };

  // + le code précédent

}
```

JUnit

Ou en utilisant les expressions Lambda

```
class CalculTest {
  Calcul calcul;

  CalculService calculService = (int x) -> x * x;

  // + le code précédent

}
```

JUnit

En testant

Tout se passe bien et le test est passé.

JUnit

En testant

Tout se passe bien et le test est passé.

On peut utiliser les mocks

pour créer un objet factice de `CalculService`.

JUnit

Ajouter la propriété suivante dans la section `properties` de `pom.xml`

```
<properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <maven.compiler.source>18</maven.compiler.source>
    <maven.compiler.target>18</maven.compiler.target>
    <junit-jupiter.version>5.9.3</junit-jupiter.version>
    <junit-platform.version>1.9.3</junit-platform.version>
    <mockito.version>5.3.1</mockito.version>
</properties>
```

JUnit

Ajouter la propriété suivante dans la section `properties` de `pom.xml`

```
<properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <maven.compiler.source>18</maven.compiler.source>
    <maven.compiler.target>18</maven.compiler.target>
    <junit-jupiter.version>5.9.3</junit-jupiter.version>
    <junit-platform.version>1.9.3</junit-platform.version>
    <mockito.version>5.3.1</mockito.version>
</properties>
```


Ajouter la dépendance suivante dans la section `dependencies` de `pom.xml`

```
dependency>
        <groupId>org.mockito</groupId>
        <artifactId>mockito-junit-jupiter</artifactId>
        <version>${mockito.version}</version>
        <scope>test</scope>
</dependency>
```

JUnit

Commençons par importer `mockito` dans la classe `CalculTest`

```
import static org.mockito.Mockito.*;
```

JUnit

Commençons par importer `mockito` dans la classe `CalculTest`

```
import static org.mockito.Mockito.*;
```

Remplaçons l'instanciation de `CalculService` par un `mock`

```
CalculService calculService = mock(CalculService.class);
```

JUnit

Utilisons `when ... thenReturn` pour indiquer ce qu'il faut retourner lorsque la méthode `carre` est appelée avec les paramètres 2 ou 3

```
@Test
void testSommeCarre() {
  when(calculService.carre(2)).thenReturn(4);
  when(calculService.carre(3)).thenReturn(9);
  assertTrue(calcul.sommeCarre(2, 3) == 13, "calcul exact");
}
```

JUnit

Utilisons `when ... thenReturn` pour indiquer ce qu'il faut retourner lorsque la méthode `carre` est appelée avec les paramètres 2 ou 3

```
@Test
void testSommeCarre() {
  when(calculService.carre(2)).thenReturn(4);
  when(calculService.carre(3)).thenReturn(9);
  assertTrue(calcul.sommeCarre(2, 3) == 13, "calcul exact");
}
```

Lancez le test et vérifiez qu'il se termine correctement.

JUnit

Nous pouvons également utiliser `doReturn ... when` de la manière suivante

```
@Test
void testSommeCarre() {
  when(calculService.carre(2)).thenReturn(4);
  doReturn(9).when(calculService).carre(3);
  assertTrue(calcul.sommeCarre(2, 3) == 13, "calcul exact");
}
```

JUnit

Nous pouvons également utiliser `doReturn ... when` de la manière suivante

```
@Test
void testSommeCarre() {
  when(calculService.carre(2)).thenReturn(4);
  doReturn(9).when(calculService).carre(3);
  assertTrue(calcul.sommeCarre(2, 3) == 13, "calcul exact");
}
```


Lancez le test et vérifiez qu'il se termine correctement.

JUnit

Remarques

- `when ... thenReturn` n'accepte pas les méthodes `void`.
- `doReturn` accepte toutes les méthodes quelle que soit la valeur de retour.

JUnit

Pour vérifier que le mock a bien été appelé, on peut utiliser la méthode `verify`

```
@Test
void testSommeCarre() {
  when(calculService.carre(2)).thenReturn(4);
  when(calculService.carre(3)).thenReturn(9);
  when(calculService.carre(4)).thenReturn(16);
  assertTrue(calcul.sommeCarre(2, 3) == 13, "calcul exact");
  verify(calculService).carre(2);
}
```

JUnit

Pour vérifier que le mock a bien été appelé, on peut utiliser la méthode `verify`

```
@Test
void testSommeCarre() {
  when(calculService.carre(2)).thenReturn(4);
  when(calculService.carre(3)).thenReturn(9);
  when(calculService.carre(4)).thenReturn(16);
  assertTrue(calcul.sommeCarre(2, 3) == 13, "calcul exact");
  verify(calculService).carre(2);
}
```


Remarque

Remplacez `verify(calculService).carre(2)` par `verify(calculService).carre(4)` et vérifiez que le test échoue.

JUnit

Et si on n'appelle plus la méthode `carre()` de `CalculService` dans `Calcul`

```
public int sommeCarre(int x, int y) {
  return somme(x * x, y * y);
}
```

JUnit

Et si on n'appelle plus la méthode `carre()` de `CalculService` dans `Calcul`

```
public int sommeCarre(int x, int y) {
  return somme(x * x, y * y);
}
```

Testons de nouveau `testSommeCarre()`

```
@Test
void testSommeCarre() {
  when(calculService.carre(2)).thenReturn(4);
  when(calculService.carre(3)).thenReturn(9);
  assertTrue(calcul.sommeCarre(2, 3) == 13, "calcul exact");
  verify(calculService).carre(2);
}
```

JUnit

Le résultat est

```
Wanted but not invoked:
calculService.carre(2);
-> at org.eclipse.test.CalculTest.testSommeCarre(CalculTest.java:39)
Actually, there were zero interactions with this mock.
```

JUnit

Le résultat est

```
Wanted but not invoked:
calculService.carre(2);
-> at org.eclipse.test.CalculTest.testSommeCarre(CalculTest.java:39)
Actually, there were zero interactions with this mock.
```


Explication

Tout mock créé doit être invoqué.

JUnit

Pour vérifier que `carre(2)` a été appelé une seule fois, on peut utiliser la méthode `times`

```
@Test
void testSommeCarre() {
  when(calculService.carre(2)).thenReturn(4);
  when(calculService.carre(3)).thenReturn(9);
  assertTrue(calcul.sommeCarre(2, 3) == 13, "calcul exact");
  verify(calculService, times(1)).carre(2);
}
```

JUnit

Pour vérifier que `carre(2)` a été appelé une seule fois, on peut utiliser la méthode `times`

```
@Test
void testSommeCarre() {
  when(calculService.carre(2)).thenReturn(4);
  when(calculService.carre(3)).thenReturn(9);
  assertTrue(calcul.sommeCarre(2, 3) == 13, "calcul exact");
  verify(calculService, times(1)).carre(2);
}
```


Remarque

Remplacez `times(1)` par `times(2)` et vérifiez que le test échoue.

JUnit

Pour vérifier que `carre(2)` a été appelé au moins une fois, on peut utiliser la méthode `atLeast`

```
@Test
void testSommeCarre() {
  when(calculService.carre(2)).thenReturn(4);
  when(calculService.carre(3)).thenReturn(9);
  assertTrue(calcul.sommeCarre(2, 3) == 13, "calcul exact");
  verify(calculService, atLeast(1)).carre(2);
}
```

JUnit

Pour vérifier que `carre(2)` a été appelé au moins une fois, on peut utiliser la méthode `atLeast`

```
@Test
void testSommeCarre() {
  when(calculService.carre(2)).thenReturn(4);
  when(calculService.carre(3)).thenReturn(9);
  assertTrue(calcul.sommeCarre(2, 3) == 13, "calcul exact");
  verify(calculService, atLeast(1)).carre(2);
}
```


Remarque

Remplacez `atLeast(1)` par `atLeast(2)` et vérifiez que le test échoue.

JUnit

Autres méthodes similaires

- `atLeastOnce()` : au moins une fois .
- `atMost(n)` : au plus `n` fois.
- `atMostOnce()` : au plus une fois.
- ...

JUnit

Pour vérifier que `carre` a été appelé deux fois avec des paramètres de type `Integer`, on peut utiliser la méthode `any`

```
@Test
void testSommeCarre() {
  when(calculService.carre(2)).thenReturn(4);
  when(calculService.carre(3)).thenReturn(9);
  when(calculService.carre(4)).thenReturn(16);
  assertTrue(calcul.sommeCarre(2, 3) == 13, "calcul exact");
  verify(calculService, times(2)).carre(Mockito.any(Integer.class));
}
```

JUnit

Pour vérifier que `carre` a été appelé deux fois avec des paramètres de type `Integer`, on peut utiliser la méthode `any`

```
@Test
void testSommeCarre() {
  when(calculService.carre(2)).thenReturn(4);
  when(calculService.carre(3)).thenReturn(9);
  when(calculService.carre(4)).thenReturn(16);
  assertTrue(calcul.sommeCarre(2, 3) == 13, "calcul exact");
  verify(calculService, times(2)).carre(Mockito.any(Integer.class));
}
```


Remarque

Remplacez `times(2)` par `times(3)` ou `times(1)` et vérifiez que le test échoue.

JUnit

Pour vérifier que `carre` a été appelé deux fois avec des paramètres de type `Integer`, on peut aussi utiliser la méthode `anyInt`

```
@Test
void testSommeCarre() {
  when(calculService.carre(2)).thenReturn(4);
  when(calculService.carre(3)).thenReturn(9);
  when(calculService.carre(4)).thenReturn(16);
  assertTrue(calcul.sommeCarre(2, 3) == 13, "calcul exact");
  verify(calculService, times(2)).carre(Mockito.anyInt());
}
```

JUnit

Pour vérifier que `carre` a été appelé deux fois avec des paramètres de type `Integer`, on peut aussi utiliser la méthode `anyInt`

```
@Test
void testSommeCarre() {
  when(calculService.carre(2)).thenReturn(4);
  when(calculService.carre(3)).thenReturn(9);
  when(calculService.carre(4)).thenReturn(16);
  assertTrue(calcul.sommeCarre(2, 3) == 13, "calcul exact");
  verify(calculService, times(2)).carre(Mockito.anyInt());
}
```


Remarque

Remplacez `times(2)` par `times(3)` et vérifiez que le test échoue.

JUnit

Autres méthodes similaires

- `anyByte()`
- `anyShort()`
- `anyChar()`
- `anyFloat()`
- `anyDouble()`
- ...

JUnit

Pour vérifier que `carre(4)` n'a jamais été appelé, on peut utiliser la méthode `never`

```
@Test
void testSommeCarre() {
  when(calculService.carre(2)).thenReturn(4);
  when(calculService.carre(3)).thenReturn(9);
  assertTrue(calcul.sommeCarre(2, 3) == 13, "calcul exact");
  verify(calculService, never()).carre(4);
}
```

JUnit

Pour vérifier que `carre(4)` n'a jamais été appelé, on peut utiliser la méthode `never`

```
@Test
void testSommeCarre() {
  when(calculService.carre(2)).thenReturn(4);
  when(calculService.carre(3)).thenReturn(9);
  assertTrue(calcul.sommeCarre(2, 3) == 13, "calcul exact");
  verify(calculService, never()).carre(4);
}
```


Remarque

Remplacez `verify(calculService, never()).carre(4)` par `verify(calculService, never()).carre(2)` et vérifiez que le test échoue.

JUnit

Pour vérifier si `carre()` a été appelé une seule fois quel que soit le nombre de paramètres, on peut utiliser la méthode `only`

```
@Test
void testSommeCarre() {
  when(calculService.carre(2)).thenReturn(4);
  when(calculService.carre(3)).thenReturn(9);
  assertTrue(calcul.sommeCarre(2, 3) == 13, "calcul exact");
  verify(calculService, only()).carre(4);
}
```

JUnit

Pour vérifier l'ordre d'interactions avec le mock, on peut utiliser la méthode `inOrder`

```
@Test
void testSommeCarre() {
  when(calculService.carre(2)).thenReturn(4);
  when(calculService.carre(3)).thenReturn(9);
  assertTrue(calcul.sommeCarre(2, 3) == 13, "calcul exact");
  InOrder ordre = Mockito.inOrder(calculService);
  ordre.verify(calculService).carre(2);
  ordre.verify(calculService).carre(3);
}
```

JUnit

Pour vérifier l'ordre d'interactions avec le mock, on peut utiliser la méthode `inOrder`

```
@Test
void testSommeCarre() {
  when(calculService.carre(2)).thenReturn(4);
  when(calculService.carre(3)).thenReturn(9);
  assertTrue(calcul.sommeCarre(2, 3) == 13, "calcul exact");
  InOrder ordre = Mockito.inOrder(calculService);
  ordre.verify(calculService).carre(2);
  ordre.verify(calculService).carre(3);
}
```


Remarque

Inverser `ordre.verify(calculService).carre(2)` par `ordre.verify(calculService).carre(3)` et vérifiez que le test échoue.

JUnit

Utilisons `when ... thenAnswer` pour définir le comportement de la méthode simulée tout en pouvant récupérer les paramètres et retourner un résultat

```
@Test
void testSommeCarre() {
  when(calculService.carre(Mockito.anyInt())).thenAnswer(
  (invocation) -> {
    Integer entier = invocation.getArgument(0);
    return entier * entier;
  });
  assertTrue(calcul.sommeCarre(2, 3) == 13, "calcul exact");
}
```

JUnit

Utilisons `when ... thenAnswer` pour définir le comportement de la méthode simulée tout en pouvant récupérer les paramètres et retourner un résultat

```
@Test
void testSommeCarre() {
  when(calculService.carre(Mockito.anyInt())).thenAnswer(
  (invocation) -> {
    Integer entier = invocation.getArgument(0);
    return entier * entier;
  });
  assertTrue(calcul.sommeCarre(2, 3) == 13, "calcul exact");
}
```

Lancez le test et vérifiez qu'il se termine correctement.

JUnit

Nous pouvons également utiliser `doAnswer ... when` de la manière suivante

```
@Test
void testSommeCarre() {
  doAnswer((invocation) -> {
    Integer entier = invocation.getArgument(0);
    return entier * entier;
  }).when(calculService).carre(Mockito.anyInt());
  assertTrue(calcul.sommeCarre(2, 3) == 13, "calcul exact");
}
```

JUnit

Nous pouvons également utiliser `doAnswer ... when` de la manière suivante

```
@Test
void testSommeCarre() {
  doAnswer((invocation) -> {
    Integer entier = invocation.getArgument(0);
    return entier * entier;
  }).when(calculService).carre(Mockito.anyInt());
  assertTrue(calcul.sommeCarre(2, 3) == 13, "calcul exact");
}
```

Lancez le test et vérifiez qu'il se termine correctement.

JUnit

Remarques

- `when ... thenAnswer` n'accepte pas les méthodes `void`.
- `doAnswer` accepte toutes les méthodes quelle que soit la valeur de retour.

JUnit

Remarques

- Un mock est initialement introduit pour simuler des objets externes ou non instanciables (interface, classe abstraite...)
- Mais il peut aussi simuler le comportement d'une classe concrète.

JUnit

Considérons la classe `Point` suivante

```
package org.eclipse.main;

public class Point {

    private double abs;
    private double ord;

    public Point(double abs, double ord) {
        this.abs = abs;
        this.ord = ord;
    }

    public Point deplacerOrd(double d) {
        ord += d;
        return this;
    }

    public void deplacerAbs(double d) {
        abs += d;
    }

    public final void deplacer(double d) {
        abs += d;
        ord += d;
    }
}
```

Pour tester la classe `Point`, commençons par créer la classe `PointTest` et préparer le mock

```
class PointTest {
    Point pointMock = Mockito.mock(Point.class);

    @BeforeAll
    static void setUpBeforeClass() throws Exception { }

    @AfterAll
    static void tearDownAfterClass() throws Exception { }

    @BeforeEach
    void setUp() throws Exception { }

    @AfterEach
    void tearDown() throws Exception {
    }

    @Test
    void testDeplacerOrd() {
        fail("Not yet implemented");
    }

    @Test
    void testDeplacerAbs() {
        fail("Not yet implemented");
    }

    @Test
    void testDeplacer() {
        fail("Not yet implemented");
    }

}
```

JUnit

Utilisons `when ... thenThrow` pour lever une exception si certains paramètres sont présents

```
@Test
void testDeplacerOrd() {

    when(pointMock.deplacerOrd(Double.MAX_VALUE))
        .thenThrow(ArithmeticException.class);
    Throwable e = null;
    try {
        pointMock.deplacerOrd(Double.MAX_VALUE);
    } catch (Exception ex) {
        e = ex;
    }
    assertTrue(e instanceof ArithmeticException);
}
```

JUnit

Ajoutons l'affichage suivant dans la méthode `deplacerOrd`

```
public Point deplacerOrd(double d) {
        System.out.println(d);
        ord += d;
        return this;
}
```

JUnit

Ajoutons l'affichage suivant dans la méthode `deplacerOrd`

```
public Point deplacerOrd(double d) {
        System.out.println(d);
        ord += d;
        return this;
}
```

Remarques

- En lançant le test, le message ajouté ne s'affiche pas.
- En effet, appeler une méthode après un `when` permet de retourner le résultat indiqué par `thenReturn` ou `thenThrow`.
- Le mock renvoie `null` si la méthode appelée n'est pas définie avec un `when`.

JUnit

Remarques

- `when ... thenThrow` n'accepte pas les méthodes `void`.
- Mais on peut utiliser `doThrow`.

JUnit

Utilisons `doThrow` pour lever une exception si certains paramètres sont présents pour tester la méthode `déplacerAbs` ne retournant pas de valeurs

```
@Test
void testDeplacerAbs() {
    doThrow(ArithmeticException.class)
    .when(pointMock).deplacerAbs(Double.MAX_VALUE);
    Throwable e = null;
    try {
        pointMock.deplacerAbs(Double.MAX_VALUE);
    } catch (Exception ex) {
        e = ex;
    }
    assertTrue(e instanceof ArithmeticException);
}
```

JUnit

Remarques

- Un mock ne peut jamais simuler une méthode finale.
- Il ne peut utiliser la version définie dans la classe réelle car il retourne `null` si la méthode n'est pas mentionnée dans un `when`.

JUnit

Remarques

- Un mock ne peut jamais simuler une méthode finale.
- Il ne peut utiliser la version définie dans la classe réelle car il retourne `null` si la méthode n'est pas mentionnée dans un `when`.

Solution

Utiliser les espions (Spy).

JUnit

Spy ?

- Un spy est un mock partiel créé à partir d'une instance réelle.
- Une partie de l'objet sera simulée et une partie utilisera des invocations de méthodes réelles.
- Si un spy appelle une méthode non mentionnée dans un `when`, c'est la méthode originale définie dans la classe réelle qui sera appelée.

JUnit

Remplaçons la déclaration du mock

```
Point pointMock = mock(Point.class);
```

JUnit

Remplaçons la déclaration du mock

```
Point pointMock = mock(Point.class);
```

Par celle d'un spy construit à partir d'une instance réelle

```
Point pointMock = Mockito.spy(new Point(2, 2));
```

JUnit

Remplaçons la déclaration du mock

```
Point pointMock = mock(Point.class);
```

Par celle d'un spy construit à partir d'une instance réelle

```
Point pointMock = Mockito.spy(new Point(2, 2));
```


Vérifiez que les tests précédents passent toujours.

Définissons le test de la méthode finale

```
@Test
void testDeplacer() {
    Point resultat = new Point(6.0, 6.0);
    pointMock.deplacer(4);
    assertTrue(resultat.equals(pointMock));
}
```

Définissons le test de la méthode finale

```
@Test
void testDeplacer() {
    Point resultat = new Point(6.0, 6.0);
    pointMock.deplacer(4);
    assertTrue(resultat.equals(pointMock));
}
```

Et redéfinissons la méthode `equals` dans `Point`

```
@Override
public boolean equals(Object obj) {
        Point p = (Point) obj;
        return p.abs == abs && p.ord == ord;
}
```

Définissons le test de la méthode finale

```
@Test
void testDeplacer() {
    Point resultat = new Point(6.0, 6.0);
    pointMock.deplacer(4);
    assertTrue(resultat.equals(pointMock));
}
```

Et redéfinissons la méthode `equals` dans `Point`

```
@Override
public boolean equals(Object obj) {
        Point p = (Point) obj;
        return p.abs == abs && p.ord == ord;
}
```

Vérifiez que le test passe correctement.

JUnit

Remarques

- Pour les spy, `when ... thenReturn` et `when ... thenThrow` appellent la méthode de la classe réelle juste avant que la valeur spécifiée ne soit renvoyée. Donc, si la méthode appelée lève une exception, il faut la gérer / la simuler, etc.
- `doReturn` et `doThrow` n'appelle pas du tout la méthode de la classe réelle.

JUnit

Reprenons la méthode `testSommeCarre()`

```
@Test
void testSommeCarre() {
  doAnswer((invocation) -> {
    Integer entier = invocation.getArgument(0);
    return entier * entier;
  }).when(calculService).carre(Mockito.anyInt());
  assertTrue(calcul.sommeCarre(2, 3) == 13, "calcul exact");
}
```

JUnit

Reprenons la méthode `testSommeCarre()`

```
@Test
void testSommeCarre() {
  doAnswer((invocation) -> {
    Integer entier = invocation.getArgument(0);
    return entier * entier;
  }).when(calculService).carre(Mockito.anyInt());
  assertTrue(calcul.sommeCarre(2, 3) == 13, "calcul exact");
}
```


Question

Comment capturer les arguments utilisées dans une méthode simulée ?

JUnit

Réponse

Utiliser `ArgumentCaptor`

JUnit

Réponse

Utiliser `ArgumentCaptor`

`ArgumentCaptor` ?

- utilisé avec la méthode `verify` pour capturer les paramètres passés à une méthode simulée.
- particulièrement utile lorsque nous ne pouvons pas accéder à un paramètre en dehors de la méthode simulée.

JUnit

Exemple

```
@Test
void testSommeCarre() {
  ArgumentCaptor<Integer> nbr = ArgumentCaptor.forClass(Integer.class);
  doAnswer((invocation) -> {
    Integer entier = invocation.getArgument(0);
    return entier * entier;
  }).when(calculService).carre(Mockito.anyInt());
  assertTrue(calcul.sommeCarre(2, 3) == 13, "calcul exact");
  verify(calculService, times(2)).carre(nbr.capture());
  assertEquals(2, nbr.getAllValues().get(0));
  assertEquals(3, nbr.getAllValues().get(1));
}
```

JUnit

Exemple

```
@Test
void testSommeCarre() {
  ArgumentCaptor<Integer> nbr = ArgumentCaptor.forClass(Integer.class);
  doAnswer((invocation) -> {
    Integer entier = invocation.getArgument(0);
    return entier * entier;
  }).when(calculService).carre(Mockito.anyInt());
  assertTrue(calcul.sommeCarre(2, 3) == 13, "calcul exact");
  verify(calculService, times(2)).carre(nbr.capture());
  assertEquals(2, nbr.getAllValues().get(0));
  assertEquals(3, nbr.getAllValues().get(1));
}
```

Lancez le test et vérifiez qu'il se termine correctement.

JUnit

Explication

- `ArgumentCaptor<Integer> nbr = ArgumentCaptor.forClass(Integer.class)` permet de déclarer un capteur d'argument de type `Integer`.
- `nbr.capture()` permet de capturer les arguments de type entier.
- `nbr.getAllValues()` retourne les arguments capturés de type entier.

JUnit

Explication

- `ArgumentCaptor<Integer> nbr = ArgumentCaptor.forClass(Integer.class)` permet de déclarer un capteur d'argument de type `Integer`.
- `nbr.capture()` permet de capturer les arguments de type entier.
- `nbr.getAllValues()` retourne les arguments capturés de type entier.

Remarques

- Si une méthode prend plusieurs arguments de types différents, alors il faut déclarer un capteur d'arguments par type.
- Si le capteur capture une seule valeur, on peut utiliser `capteur.getValue()`.

JUnit

On peut aussi utiliser les annotations pour créer un mock

```
@Mock
CalculService calculService;
```

JUnit

On peut aussi utiliser les annotations pour créer un mock

```
@Mock
CalculService calculService;
```


Remarques

Pour activer l'annotation `@Mock`, il faut soit :

- appeler la méthode `initMocks` (ou `openMocks` depuis **Mockito** 3).
- annoter la classe de test par `@ExtendWith`.

JUnit

Activons l'annotation `@Mock` (Pour Mockito 2, remplacez `openMocks` par `initMocks`)

```
class CalculTest {

  Calcul calcul;

  @Mock
  CalculService calculService;

  @BeforeEach
  void setUp() throws Exception {
    MockitoAnnotations.openMocks(this);
    calcul = new Calcul(calculService);
  }

  // + le code précédent
```

JUnit

On peut aussi remplacer l'appel de la méthode `openMocks` par l'annotation `@ExtendWith(MockitoExtension.class)`

```
@ExtendWith(MockitoExtension.class)
class CalculTest {

  Calcul calcul;

  @Mock
  CalculService calculService;

  @BeforeEach
  void setUp() throws Exception {
    calcul = new Calcul(calculService);
  }

  // + le code précédent
```

JUnit

On peut aussi automatiser l'injection du mock dans le constructeur de la classe `Calcul` en utilisant l'annotation `@InjectMocks`

```
@ExtendWith(MockitoExtension.class)
class CalculTest {

  @Mock
  CalculService calculService;

  @InjectMocks
  Calcul calcul;

  @BeforeEach
  void setUp() throws Exception {
  }

  // + le code précédent
```

On peut aussi remplacer l'appel de la méthode `spy` par l'annotation `@Spy`

```
@ExtendWith(MockitoExtension.class)
class PointTest {

        @Spy
        Point pointMock = new Point(2, 2);

        @BeforeAll
        static void setUpBeforeClass() throws Exception {
        }

        @AfterAll
        static void tearDownAfterClass() throws Exception {
        }

        @BeforeEach
        void setUp() throws Exception {
        }

        @AfterEach
        void tearDown() throws Exception {
        }

        // + les tests
```

JUnit

On peut aussi déclarer le capteur avec `@Captor`

```
@ExtendWith(MockitoExtension.class)
class CalculTest {

  @Mock
  CalculService calculService;

  @InjectMocks
  Calcul calcul;

  @Captor
  ArgumentCaptor<Integer> nbr;

  @BeforeEach
  void setUp() throws Exception {
  }

  @Test
  void testSommeCarre() {
    doAnswer((invocation) -> {
      Integer entier = invocation.getArgument(0);
      return entier * entier;
    }).when(calculService).carre(Mockito.anyInt());
    assertTrue(calcul.sommeCarre(2, 3) == 13, "calcul exact");
    verify(calculService, times(2)).carre(nbr.capture());
    assertEquals(2, nbr.getAllValues().get(0));
    assertEquals(3, nbr.getAllValues().get(1));
  }

  // + les autres tests
```

JUnit

Objectif

- On veut vérifier si nos tests couvrent l'intégralité de notre code.
- Ou s'il y a des zones inaccessibles dans notre code.

JUnit

Objectif

- On veut vérifier si nos tests couvrent l'intégralité de notre code.
- Ou s'il y a des zones inaccessibles dans notre code.

Pour cela, il faut installer un plugin **Eclipse** `EclEmma`

.

- Aller dans `Help > Install New Software > Add`.
- Saisir `EclEmma` dans `Name` et `http://update.eclemma.org/` dans `Location`.
- Cocher la case `EclEmma` et cliquer sur `Next`.
- Terminer l'installation et redémarrer **Eclipse**.

JUnit

Pour tester le recouvrement du code

- Aller dans `Window > Show View > Other ...`
- Saisir `Coverage` et valider.
- Aller dans `Run`.
- Cliquer sur `Coverage` et vérifier si votre code est bien couvert par les tests.