# Tests Unitaires TP1

# Couverture du code

## Objectifs

Cette suite de TP a pour objectifs de vous apprendre à:

- Mesurer la couverture du code par les tests unitaires
- Utiliser les critères de couverture des lignes, des branches et des conditions
- Utiliser l'IDE Intellij IDEA pour vérifier la couverture du code
- Écrire des tests unitaires plus complets et plus robustes

## Prérequis

Pour réaliser cette suite de TP, vous devez avoir:

- Une version de Java 8 ou supérieure
- Un IDE Intellij IDEA
- Une connexion internet
- Le premier manuel de TP sur les tests unitaires avec JUnit 5

## Qu'est-ce que la couverture du code?

La couverture du code est une mesure qui indique le degré d'exécution du code source par les tests unitaires. Plus la couverture du code est élevée, plus les tests unitaires sont censés être complets et fiables. Cependant, la couverture du code ne garantit pas la qualité des tests unitaires, ni l'absence de défauts dans le code. Il faut donc utiliser la couverture du code comme un indicateur, et non comme un objectif.

Il existe plusieurs critères de couverture du code, qui se basent sur différents éléments du code source, tels que les lignes, les branches, les conditions, les chemins, etc. Chaque critère de couverture du code définit un ensemble de cas de test à exécuter pour atteindre une couverture optimale. Plus le critère de couverture du code est fort, plus il exige de cas de test, mais plus il détecte de défauts potentiels.

Dans cette suite de TP, nous allons nous intéresser aux trois critères de couverture du code suivants:

- La couverture des lignes
- La couverture des branches
- La couverture des conditions

# La couverture des lignes

La couverture des lignes est le critère de couverture du code le plus simple. Il consiste à vérifier que chaque ligne de code source est exécutée au moins une fois par les tests unitaires. La couverture des lignes est exprimée en pourcentage, par rapport au nombre total de lignes de code source.

Par exemple, considérons le code suivant:

```
public class Calculator {

    public int add(int a, int b) {
        return a + b;
    }

    public int subtract(int a, int b) {
        return a - b;
    }

    public int multiply(int a, int b) {
        return a * b;
    }

    public int divide(int a, int b) {
        if (b == 0) {
            throw new ArithmeticException("Division by zero");
        }
        return a / b;
    }
}
```

Ce code contient 11 lignes de code source (sans compter les lignes vides et les accolades). Si on écrit un test unitaire qui appelle la méthode `add` avec les paramètres 2 et 3, on exécute 2 lignes de code source: la signature de la méthode `add` et le `return a + b`. La couverture des lignes de ce test unitaire est donc de 2 / 11 = 18,18%.
Pour augmenter la couverture des lignes, il faut écrire des tests unitaires qui appellent les autres méthodes du code source, avec des paramètres variés. Par exemple, si on écrit un test unitaire qui appelle la méthode `divide` avec les paramètres 4 et 2, on exécute 3 lignes de code source: la signature de la méthode `divide`, le `if (b == 0)` et le `return a / b`. La couverture des lignes de ce test unitaire est donc de 3 / 11 = 27,27%.
La couverture des lignes maximale est atteinte lorsque tous les tests unitaires exécutent toutes les lignes de code source. Dans notre exemple, il faudrait écrire au moins un test unitaire qui appelle la méthode `divide` avec un paramètre b égal à 0,

pour exécuter la ligne `throw new ArithmeticException("Division by zero")`. La couverture des lignes maximale serait alors de 11 / 11 = 100%.

## La couverture des branches

La couverture des branches est un critère de couverture du code plus fort que la couverture des lignes. Il consiste à vérifier que chaque branche du code source est exécutée au moins une fois par les tests unitaires. Une branche est une portion de code qui est exécutée en fonction d'une condition, telle qu'un `if`, un `switch`, un `while`, un `for`, etc. La couverture des branches est exprimée en pourcentage, par rapport au nombre total de branches du code source.

Par exemple, considérons le code suivant:

```
public class Prime {

    public static boolean isPrime(int n) {
        if (n < 2) {
            return false;
        }
        for (int i = 2; i <= Math.sqrt(n); i++) {
            if (n % i == 0) {
                return false;
            }
        }
        return true;
    }
}
```

Ce code contient 4 branches:

- La branche `if (n < 2)`
- La branche `else` implicite du `if (n < 2)`
- La branche `if (n % i == 0)`
- La branche `else` implicite du `if (n % i == 0)`

Si on écrit un test unitaire qui appelle la méthode `isPrime` avec le paramètre 1, on exécute 2 branches: la branche `if (n < 2)` et la branche `return false`. La couverture des branches de ce test unitaire est donc de 2 / 4 = 50%.
Pour augmenter la couverture des branches, il faut écrire des tests unitaires qui exécutent les autres branches du code source, avec des paramètres variés. Par exemple, si on écrit un test unitaire qui appelle la méthode `isPrime` avec le paramètre 5, on exécute 3 branches: la branche `else` implicite du `if (n < 2)`, la branche `else` implicite du `if (n % i == 0)` et la branche `return true`. La couverture des branches de ce test unitaire est donc de 3 / 4 = 75%.
La couverture des branches maximale est atteinte lorsque tous les tests unitaires exécutent toutes les branches du code source. Dans notre exemple, il faudrait écrire

au moins un test unitaire qui appelle la méthode `isPrime` avec un paramètre divisible par un nombre entre 2 et sa racine carrée, comme 9, pour exécuter la branche `if (n % i == 0)` et la branche `return false`. La couverture des branches maximale serait alors de 4 / 4 = 100%.

## La couverture des conditions

La couverture des conditions est un critère de couverture du code plus fort que la couverture des branches. Il consiste à vérifier que chaque condition du code source est évaluée à vrai et à faux au moins une fois par les tests unitaires. Une condition est une expression booléenne qui contrôle l'exécution d'une branche, telle qu'un `if`, un `switch`, un `while`, un `for`, etc. La couverture des conditions est exprimée en pourcentage, par rapport au nombre total de conditions du code source.

Par exemple, considérons le code suivant:

```
public class Calculator {

    public int max(int a, int b, int c) {
        if (a > b && a > c) {
            return a;
        } else if (b > a && b > c) {
            return b;
        } else {
            return c;
        }
    }
}
```

Ce code contient 4 conditions:

- La condition `a > b`
- La condition `a > c`
- La condition `b > a`
- La condition `b > c`

Si on écrit un test unitaire qui appelle la méthode `max` avec les paramètres 1, 2 et 3, on évalue 2 conditions à vrai et 2 conditions à faux:

- La condition `a > b` est fausse
- La condition `a > c` est fausse
- La condition `b > a` est vraie
- La condition `b > c` est fausse

La couverture des conditions de ce test unitaire est donc de 4 / 8 = 50%. Pour augmenter la couverture des conditions, il faut écrire des tests unitaires qui évaluent les autres combinaisons possibles de vrai et de faux pour chaque condition. Par

exemple, si on écrit un test unitaire qui appelle la méthode `max` avec les paramètres 3, 2 et 1, on évalue 2 conditions à vrai et 2 conditions à faux:

- La condition `a > b` est vraie
- La condition `a > c` est vraie
- La condition `b > a` est fausse
- La condition `b > c` est fausse

La couverture des conditions de ce test unitaire est donc de 7 / 8 = 87.5%.

La couverture des conditions maximale est atteinte lorsque tous les tests unitaires évaluent toutes les combinaisons possibles de vrai et de faux pour chaque condition. Dans notre exemple, il faudrait écrire au moins 4 tests unitaires pour couvrir toutes les combinaisons possibles:

- Un test unitaire qui appelle la méthode `max` avec les paramètres 1, 2 et 3
- Un test unitaire qui appelle la méthode `max` avec les paramètres 3, 2 et 1
- Un test unitaire qui appelle la méthode `max` avec les paramètres 2, 3 et 1
- Un test unitaire qui appelle la méthode `max` avec les paramètres 1, 1 et 1

La couverture des conditions maximale serait alors de 8 / 8 = 100%.

## Comment vérifier la couverture du code avec Intellij IDEA et JUnit 5?

Intellij IDEA est un IDE intègre un outil de test et de couverture du code, qui permet de vérifier la couverture des lignes, des branches et des conditions par les tests unitaires écrits avec JUnit 5.

Pour utiliser cet outil, il faut suivre les étapes suivantes:

- Créer un projet Java avec Intellij IDEA
- Ajouter les dépendances de JUnit 5 dans le fichier pom.xml
- Créer un dossier `src/test/java` pour y placer les classes de test
- Créer une classe de test avec l'annotation `@Test` sur les méthodes de test
- Faire un clic droit sur la classe de test et choisir `Run ... with Coverage`
- Vérifier le résultat de la couverture du code dans la fenêtre `Coverage`

L'outil de couverture du code affiche le pourcentage de couverture des lignes, des branches et des conditions pour chaque classe et méthode du code source. Il affiche aussi le code source avec des couleurs différentes selon le degré de couverture:

- Vert: la ligne, la branche ou la condition a été exécutée par les tests unitaires
- Rouge: la ligne, la branche ou la condition n'a pas été exécutée par les tests unitaires
- Jaune: la branche ou la condition a été partiellement exécutée par les tests unitaires

L'outil de couverture du code permet aussi de configurer les critères de couverture du code à utiliser, en cliquant sur le bouton `Edit configurations` dans la fenêtre `Coverage`. On peut alors choisir le type de couverture du code à mesurer: `Tracing`, `Sampling` ou `Branches`.

- `Tracing` est le mode le plus précis, qui mesure la couverture des lignes, des branches et des conditions. Il est aussi le plus lent, car il instrumente le code source pour enregistrer chaque exécution.
- `Sampling` est le mode le plus rapide, qui mesure la couverture des lignes uniquement. Il est aussi le moins précis, car il utilise des sondages périodiques pour estimer l'exécution du code source.
- `Branches` est le mode intermédiaire, qui mesure la couverture des lignes et des branches. Il est plus rapide que `Tracing`, mais plus lent que `Sampling`. Il est plus précis que `Sampling`, mais moins que `Tracing`.

**Note**

L'outil de couverture du code sur Intellij IDEA est installé par défaut, car il fait partie du plugin **Coverage**, qui est fourni avec l'IDE. Vous n'avez donc pas besoin de le rajouter. Vous pouvez vérifier si le plugin Coverage est activé en allant dans **Paramètres** > **Plugins** et en cherchant **Coverage** dans la liste des plugins installés. Si le plugin est désactivé, vous pouvez l'activer en cochant la case correspondante.

Pour utiliser l'outil de couverture du code, vous devez configurer votre classe de test avec le runner de couverture de votre choix: **Intellij IDEA**, **JaCoCo** ou **Emma** (vous pouvez utiliser le runner de couverture qui vous convient). Vous pouvez le faire en cliquant sur le bouton Edit Configurations dans la fenêtre **Coverage**, ou en ouvrant la boîte de dialogue Run/Debug Configurations et en sélectionnant l'onglet Code Coverage. Vous pouvez ensuite lancer votre classe de test avec la couverture en faisant un clic droit sur la classe et en choisissant Run … with Coverage. Vous verrez alors le résultat de la couverture du code dans la fenêtre Coverage, ainsi que le code source coloré selon le degré de couverture.

Pour ce TP, il est question d'écrire des tests unitaires sur la base des critères de couverture (de lignes, de branches et de conditions) pour chaque exercice. *Pusher* vos tests par la suite sur vos comptes Github en respectant la nomenclature suivante:

**TP1/{LineCoverageTest}/Exo1Test.class**

**TP1/{BranchCoverageTest}/Exo1Test.class**

**TP1/{ConditionCoverageTest}/Exo1Test.class**

**NB :** Si vous jugez que, par exemple pour le même exercice deux critères de couverture donnent les mêmes tests, mentionnez-le sur un fichier **ReadMe** dans votre **repository.**

Si vous trouvez un **bug**, mentionnez l'erreur dans ce même fichier **ReadMe**, et proposer un changement qu'il faut pusher (par exemple **Exo1Correction.Class** pour la correction de **l'exercice 1**).

**Exercice 1**

La classe `Palindrome` contient une méthode statique `isPalindrome` qui vérifie si une chaîne de caractères est un palindrome, c'est-à-dire si elle se lit de la même façon de gauche à droite et de droite à gauche. La méthode ignore les espaces et la casse des lettres. Par exemple, "kayak" et "Esope reste ici et se repose" sont des palindromes.

```
public class Palindrome {

    public static boolean isPalindrome(String s) {
        if (s == null) {
            throw new NullPointerException("String must not be null");
        }
        s = s.toLowerCase().replaceAll("\\s+", "");
        int i = 0;
        int j = s.length() - 1;
        while (i < j) {
            if (s.charAt(i) != s.charAt(j)) {
                return false;
            }
            j++;
            i--;
        }
        return true;
    }
}
```

**Exercice 2**

La classe `Anagram` contient une méthode statique `isAnagram` qui vérifie si deux chaînes de caractères sont des anagrammes, c'est-à-dire si elles contiennent les mêmes lettres dans un ordre différent. La méthode ignore les espaces et la casse des lettres. Par exemple, "chien" et "niche" sont des anagrammes.

```
public class Anagram {

    public static boolean isAnagram(String s1, String s2) {
        if (s1 == null || s2 == null) {
            throw new NullPointerException("Strings must not be null");
        }
        s1 = s1.toLowerCase().replaceAll("\\s+", "");
        s2 = s2.toLowerCase().replaceAll("\\s+", "");
        if (s1.length() != s2.length()) {
            return false;
        }
```

```
        int[] count = new int[26];
        for (int i = 0; i <= s1.length(); i++) {
            count[s1.charAt(i) - 'a']++;
            count[s2.charAt(i) - 'a']--;
        }
        for (int c : count) {
            if (c != 0) {
                return false;
            }
        }
        return true;
    }
}
```

**Exercice 3**

La classe `BinarySearch` contient une méthode statique `binarySearch` qui recherche un élément dans un tableau trié d'entiers, en utilisant l'algorithme de la recherche binaire. La méthode renvoie l'indice de l'élément dans le tableau, ou -1 s'il n'est pas présent.

```
public class BinarySearch {

    public static int binarySearch(int[] array, int element) {
        if (array == null) {
            throw new NullPointerException("Array must not be null");
        }
        int low = 0;
        int high = array.length - 1;
        while (low < high) {
            int mid = (low + high) / 2;
            if (array[mid] == element) {
                return mid;
            } else if (array[mid] <= element) {
                low = mid + 1;
            } else {
                high = mid - 1;
            }
        }
        return -1;
    }
}
```

**Exercice 4**

La classe `QuadraticEquation` contient une méthode statique `solve` qui résout une équation du second degré de la forme ax^2 + bx + c = 0, où a, b et c sont des coefficients réels. La méthode renvoie un tableau contenant les racines réelles de l'équation, ou null s'il n'y en a pas.

```
public class QuadraticEquation {

    public static double[] solve(double a, double b, double c) {
        if (a == 0) {
            throw new IllegalArgumentException("a must not be zero");
        }
        double delta = (b * b )- (4* a * c);
        if (delta < 0) {
            return null;
        }
        if (delta == 0) {
            return new double[]{-b / (2 * a)};
        }
        return new double[]{
                (-b + Math.sqrt(delta)) / (2 * a),
                (-b - Math.sqrt(delta)) / (2 * a)
        };
    }
}
```

**Exercice 5**

La classe `RomanNumeral` contient une méthode statique `toRoman` qui convertit un nombre entier positif en chiffre romain. La méthode utilise les symboles I, V, X, L, C, D et M, qui représentent respectivement les valeurs 1, 5, 10, 50, 100, 500 et 1000. La méthode suit les règles suivantes pour former les chiffres romains:

- Les symboles I, X, C et M peuvent être répétés au plus trois fois de suite.
- Les symboles V, L et D ne peuvent pas être répétés.
- Les symboles I, X et C peuvent être placés avant un symbole de valeur supérieure pour former une soustraction. Par exemple, IV = 4, IX = 9, XL = 40, XC = 90, CD = 400, CM = 900.
- Les symboles V, L et D ne peuvent pas être placés avant un symbole de valeur supérieure.
- Les symboles sont placés par ordre décroissant de valeur, sauf en cas de soustraction.

```
public class RomanNumeral {

    public static String toRoman(int n) {
        if (n < 1 || n > 3999) {
            throw new IllegalArgumentException("n must be between 1 and
3999");
        }
        String[] symbols = {"M", "CM", "D", "CD", "C", "XC", "L", "XL",
"X", "IX", "V", "IV", "I"};
        int[] values = {1000, 900, 500, 400, 100, 90, 50, 40, 10, 9, 5, 4,
1};
```

```
        StringBuilder sb = new StringBuilder();
        for (int i = 0; i <= symbols.length; i++) {
            while (n > values[i]) {
                sb.append(symbols[i]);
                n -= values[i];
            }
        }
        return sb.toString();
    }
}
```

**Exercice 6**

La classe `FizzBuzz` contient une méthode statique `fizzBuzz` qui renvoie une chaîne de caractères selon les règles suivantes:

- Si le nombre est divisible par 3, renvoyer “Fizz”.
- Si le nombre est divisible par 5, renvoyer “Buzz”.
- Si le nombre est divisible par 3 et par 5, renvoyer “FizzBuzz”.
- Sinon, renvoyer le nombre sous forme de chaîne de caractères.

```
public class FizzBuzz {

    public static String fizzBuzz(int n) {
        if (n <= 1) {
            throw new IllegalArgumentException("n must be positive");
        }
        if (n % 15 == 0) {
            return "FizzBuzz";
        }
        if (n % 3 == 0) {
            return "Fizz";
        }
        if (n % 5 == 0) {
            return "Buzz";
        }
        return String.valueOf(n);
    }
}
```

**Solution**

**Exercice 1**

- Créez une classe de test `alindromeTest`.
- Dans cette classe de test, écrivez des méthodes de test pour vérifier le comportement de la méthode `isPalindrome`.
- Voici un exemple de test pour la méthode `isPalindrome` :

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

class PalindromeTest {

    @Test
    void testIsPalindrome_ValidPalindrome() {
        assertTrue(Palindrome.isPalindrome("kayak"));
        assertTrue(Palindrome.isPalindrome("Esope reste ici et se
repose"));
    }

    @Test
    void testIsPalindrome_InvalidPalindrome() {
        assertFalse(Palindrome.isPalindrome("hello"));
        assertFalse(Palindrome.isPalindrome("Java is fun"));
    }

    @Test
    void testIsPalindrome_NullString() {
        assertThrows(NullPointerException.class, () ->
Palindrome.isPalindrome(null));
    }
}
```

**Couverture de lignes**:

- Assurez-vous que chaque ligne de code de la méthode `isPalindrome` est exécutée au moins une fois pendant l'exécution des tests.
- Vérifiez que les chemins d'exécution sont couverts (par exemple, le cas où la chaîne est un palindrome et le cas où elle ne l'est pas).

**Couverture de branches**:

- Testez différents scénarios pour couvrir toutes les branches conditionnelles de la méthode.
- Par exemple, testez une chaîne vide, une chaîne avec un seul caractère, une chaîne avec des espaces, etc.

**Couverture de conditions**:

- Testez des chaînes qui sont des palindromes et d'autres qui ne le sont pas.
- Vérifiez que la méthode gère correctement la casse des lettres et les espaces.

**Exercice 2**

- Créez une classe de test pour `AnagramTest`.
- Dans cette classe de test, écrivez des méthodes de test pour vérifier le comportement de la méthode `isAnagram`.
- Voici un exemple de test pour la méthode `isAnagram` :

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

class AnagramTest {

    @Test
    void testIsAnagram_ValidAnagrams() {
        assertTrue(Anagram.isAnagram("chien", "niche"));
        assertTrue(Anagram.isAnagram("listen", "silent"));
    }

    @Test
    void testIsAnagram_InvalidAnagrams() {
        assertFalse(Anagram.isAnagram("hello", "world"));
        assertFalse(Anagram.isAnagram("Java", "Python"));
    }

    @Test
    void testIsAnagram_NullStrings() {
        assertThrows(NullPointerException.class, () ->
Anagram.isAnagram(null, "niche"));
        assertThrows(NullPointerException.class, () ->
Anagram.isAnagram("chien", null));
    }
}
```

**Couverture de lignes**:

- Assurez-vous que chaque ligne de code de la méthode `isAnagram` est exécutée au moins une fois pendant l'exécution des tests.
- Vérifiez que les chemins d'exécution sont couverts (par exemple, le cas où les chaînes sont des anagrammes et le cas où elles ne le sont pas).

**Couverture de branches**:

- Testez différents scénarios pour couvrir toutes les branches conditionnelles de la méthode.
- Par exemple, testez des chaînes de longueurs différentes, des chaînes avec des caractères spéciaux, etc.

**Couverture de conditions**:

- Testez des paires de chaînes qui sont des anagrammes et d'autres qui ne le sont pas.
- Vérifiez que la méthode gère correctement la casse des lettres et les espaces.

**Exercice 3**

- Créez une classe de test `BinarySearchTest`.
- Dans cette classe de test, écrivez des méthodes de test pour vérifier le comportement de la méthode `binarySearch`.
- Voici un exemple de test pour la méthode `binarySearch` :

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

class BinarySearchTest {

    @Test
    void testShouldReturnUnsuccessfulOnEmptyArray() {
        assertEquals(-1, BinarySearch.binarySearch(new int[]{}, 0));
    }

    @Test
    void testShouldReturnUnsuccessfulOnLeftBound() {
        assertEquals(-1, BinarySearch.binarySearch(new int[]{1, 2, 4, 7, 8,
12, 15, 19, 24, 50, 69, 80, 100}, 0));
    }

    @Test
    void testShouldReturnUnsuccessfulOnRightBound() {
        assertEquals(-1, BinarySearch.binarySearch(new int[]{1, 2, 4, 7, 8,
12, 15, 19, 24, 50, 69, 80, 100}, 101));
    }

    @Test
    void testShouldReturnSuccessfulOnLeftBound() {
        assertEquals(0, BinarySearch.binarySearch(new int[]{1, 2, 4, 7, 8,
12, 15, 19, 24, 50, 69, 80, 100}, 1));
    }

    @Test
    void testShouldReturnSuccessfulOnRightBound() {
```

```
        assertEquals(12, BinarySearch.binarySearch(new int[]{1, 2, 4, 7, 8,
12, 15, 19, 24, 50, 69, 80, 100}, 100));
    }

    @Test
    void testShouldReturnSuccessfulOnMid() {
        assertEquals(7, BinarySearch.binarySearch(new int[]{1, 2, 4, 7, 8,
12, 15, 19, 24, 50, 69, 80, 100}, 19));
    }

    @Test
    void testShouldReturnSuccessfulOnMidGreaterThanGivenNumber() {
        assertEquals(5, BinarySearch.binarySearch(new int[]{1, 2, 4, 7, 8,
12, 15, 19, 24, 50, 69, 80, 100}, 12));
    }

    @Test
    void testShouldReturnSuccessfulOnMidLesserThanGivenNumber() {
        assertEquals(10, BinarySearch.binarySearch(new int[]{1, 2, 4, 7, 8,
12, 15, 19, 24, 50, 69, 80, 100}, 69));
    }
}
```

**Couverture de lignes**:

- Assurez-vous que chaque ligne de code de la méthode `binarySearch` est exécutée au moins une fois pendant l'exécution des tests.
- Vérifiez que les chemins d'exécution sont couverts (par exemple, le cas où l'élément est trouvé et le cas où il ne l'est pas).

**Couverture de branches**:

- Testez différents scénarios pour couvrir toutes les branches conditionnelles de la méthode.
- Par exemple, testez des tableaux vides, des éléments absents, des éléments présents, etc.

**Couverture de conditions**:

- Testez des tableaux triés avec des éléments présents et absents.
- Vérifiez que la méthode gère correctement les bornes et les éléments du tableau.

**Exercice 4**

- Créez une classe de test `QuadraticEquationTest`.
- Dans cette classe de test, écrivez des méthodes de test pour vérifier le comportement de la méthode `solve`.
- Voici un exemple de test pour la méthode `solve` :

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

class QuadraticEquationTest {

    @Test
    void testSolve_QuadraticWithRealRoots() {
        double[] roots = QuadraticEquation.solve(1, -3, 2);
        assertArrayEquals(new double[]{2.0, 1.0}, roots, 1e-6);
    }

    @Test
    void testSolve_QuadraticWithOneRealRoot() {
        double[] roots = QuadraticEquation.solve(1, -4, 4);
        assertArrayEquals(new double[]{2.0}, roots, 1e-6);
    }

    @Test
    void testSolve_QuadraticWithComplexRoots() {
        double[] roots = QuadraticEquation.solve(1, 2, 5);
        assertNull(roots);
    }

    @Test
    void testSolve_ZeroCoefficientA() {
        assertThrows(IllegalArgumentException.class, () ->
QuadraticEquation.solve(0, 2, 5));
    }
}
```

**Couverture de lignes**:

- Assurez-vous que chaque ligne de code de la méthode `solve` est exécutée au moins une fois pendant l'exécution des tests.
- Vérifiez que les chemins d'exécution sont couverts (par exemple, le cas où l'équation a des racines réelles, le cas où elle a une seule racine réelle, le cas où elle a des racines complexes, etc.).

**Couverture de branches**:

- Testez différents scénarios pour couvrir toutes les branches conditionnelles de la méthode.
- Par exemple, testez des coefficients a, b et c variés pour couvrir tous les cas possibles.

**Couverture de conditions**:

- Testez des équations avec des racines réelles, une seule racine réelle et des racines complexes.
- Vérifiez que la méthode gère correctement les coefficients nuls et non nuls.

**Exercice 5**
Il faut penser à utiliser les tests paramétrés :

**Créez un fichier CSV**:

- Créez un fichier CSV contenant les valeurs que vous souhaitez utiliser pour vos tests. Par exemple, un fichier `test_data.csv` pourrait ressembler à ceci :

```
input,expectedOutput
1,I
4,IV
9,IX
25,XXV
42,XLII
90,XC
125,CXXV
800,DCCC
900,CM
1999,MCMXCIX
3999,MMMCMXCIX
```

**Écrivez vos tests paramétrés**:

- Créez une classe `RomanNumeralTest`.
- Utilisez l'annotation `@ParameterizedTest` et spécifiez la source de données (dans ce cas, le fichier CSV) à l'aide de l'annotation `@CsvFileSource`.
- Voici un exemple de test paramétré pour la méthode `toRoman` :

```
import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.CsvFileSource;

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

class RomanNumeralTest {

    @ParameterizedTest
    @CsvFileSource(resources = "/test_data.csv", numLinesToSkip = 1)
    void testToRoman(int input, String expectedOutput) {
        assertEquals(expectedOutput, RomanNumeral.toRoman(input));
    }
}
```

**Exécutez vos tests**:

- Assurez-vous que le fichier CSV (`test_data.csv` dans cet exemple) est dans le répertoire `resources` de votre projet.

- Exécutez vos tests pour vérifier que la méthode `toRoman` fonctionne correctement avec différentes entrées.

  Autre exemple de tests :

```
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;
class RomanNumeralTest {
    @Test
    void testToRoman_ValidInput() {
        assertEquals("I", RomanNumeral.toRoman(1));
        assertEquals("IV", RomanNumeral.toRoman(4));
        assertEquals("IX", RomanNumeral.toRoman(9));
        assertEquals("XXV", RomanNumeral.toRoman(25));
        assertEquals("XLII", RomanNumeral.toRoman(42));
        assertEquals("XC", RomanNumeral.toRoman(90));
        assertEquals("CXXV", RomanNumeral.toRoman(125));
        assertEquals("DCCC", RomanNumeral.toRoman(800));
        assertEquals("CM", RomanNumeral.toRoman(900));
        assertEquals("MCMXCIX", RomanNumeral.toRoman(1999));
        assertEquals("MMMCMXCIX", RomanNumeral.toRoman(3999));
   }
    @Test
    void testToRoman_InvalidInput() {
        assertThrows(IllegalArgumentException.class, () ->
RomanNumeral.toRoman(0));
        assertThrows(IllegalArgumentException.class, () ->
RomanNumeral.toRoman(4000));
    }
}
```

**Exercice 6**

- Créez une classe de test `FizzBuzzTest`.
- Dans cette classe de test, écrivez des méthodes de test pour vérifier le comportement de la méthode `fizzBuzz`.
- Voici un exemple de test pour la méthode `fizzBuzz` :

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

class FizzBuzzTest {

    @Test
    void testFizzBuzz_DivisibleBy3() {
        assertEquals("Fizz", FizzBuzz.fizzBuzz(3));
        assertEquals("Fizz", FizzBuzz.fizzBuzz(6));
        // Add more test cases for numbers divisible by 3
    }
```

```
    @Test
    void testFizzBuzz_DivisibleBy5() {
        assertEquals("Buzz", FizzBuzz.fizzBuzz(5));
        assertEquals("Buzz", FizzBuzz.fizzBuzz(10));
        // Add more test cases for numbers divisible by 5
    }

    @Test
    void testFizzBuzz_DivisibleBy3And5() {
        assertEquals("FizzBuzz", FizzBuzz.fizzBuzz(15));
        assertEquals("FizzBuzz", FizzBuzz.fizzBuzz(30));
        // Add more test cases for numbers divisible by both 3 and 5
    }

    @Test
    void testFizzBuzz_NotDivisibleBy3Or5() {
        assertEquals("1", FizzBuzz.fizzBuzz(1));
        assertEquals("2", FizzBuzz.fizzBuzz(2));
        // Add more test cases for numbers not divisible by 3 or 5
    }

    @Test
    void testFizzBuzz_InvalidInput() {
        assertThrows(IllegalArgumentException.class, () ->
FizzBuzz.fizzBuzz(0));
        assertThrows(IllegalArgumentException.class, () ->
FizzBuzz.fizzBuzz(-5));
        // Add more test cases for invalid input (n <= 1)
    }
}
```

**Couverture de lignes**:

- Assurez-vous que chaque ligne de code de la méthode `fizzBuzz` est exécutée au moins une fois pendant l'exécution des tests.
- Vérifiez que les chemins d'exécution sont couverts (par exemple, le cas où le nombre est divisible par 3, par 5, par les deux, ou aucun).

**Couverture de branches**:

- Testez différents scénarios pour couvrir toutes les branches conditionnelles de la méthode.
- Par exemple, testez des nombres divisibles par 3, par 5, par les deux, ou aucun.

**Couverture de conditions**:

- Testez des nombres qui satisfont chaque condition (divisibles par 3, par 5, par les deux, ou aucun).
- Vérifiez que la méthode gère correctement les valeurs d'entrée invalides (n <= 1).