Tests Unitaires TP0
Préparation de l'environnement

Objectifs

Ce TP a pour objectifs de vous apprendre à:

- Installer et configurer JUnit 5 sur Intellij IDEA
- Écrire et exécuter des tests unitaires avec JUnit 5
- Utiliser les annotations, les assertions et les hypothèses de JUnit 5
- Écrire des tests dynamiques et paramétrés avec JUnit 5

Prérequis

Pour réaliser ce TP, vous devez avoir:

- Une version de Java 8 ou supérieure
- Un IDE Intellij IDEA
- Une connexion internet

Installation et configuration de JUnit 5

JUnit 5 est composé de plusieurs modules qui appartiennent à trois sous-projets:

- JUnit Platform: le module qui fournit le support pour lancer les frameworks de test sur la JVM
- JUnit Jupiter: le module qui contient le nouveau modèle de programmation et d'extension pour écrire les tests avec JUnit 5
- JUnit Vintage: le module qui permet de faire tourner les tests écrits avec les versions antérieures de JUnit sur la plateforme JUnit 5

Pour utiliser JUnit 5 dans votre projet, vous devez ajouter les dépendances Maven correspondantes dans votre fichier pom.xml. Par exemple, pour utiliser le module JUnit Jupiter, vous devez ajouter la dépendance suivante:

```
<dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-engine</artifactId>
    <version>5.9.2</version>
    <scope>test</scope>
</dependency>
```

Vous pouvez aussi utiliser le Bill Of Materials (BOM) de JUnit 5 pour gérer les versions des

différents modules de JUnit 5. Pour cela, vous devez ajouter la dépendance suivante dans la section `<dependencyManagement>` de votre fichier pom.xml:

```
<dependency>
    <groupId>org.junit</groupId>
    <artifactId>junit-bom</artifactId>
    <version>5.9.2</version>
    <type>pom</type>
    <scope>import</scope>
</dependency>
```

Ensuite, vous pouvez ajouter les dépendances des modules de JUnit 5 sans spécifier leur version, par exemple:

```
<dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-engine</artifactId>
    <scope>test</scope>
</dependency>
```

Pour configurer JUnit 5 sur Intellij IDEA, vous devez suivre les étapes suivantes:

- Créer un nouveau projet Maven avec Intellij IDEA
- Ajouter les dépendances de JUnit 5 dans le fichier pom.xml
- Créer un dossier `src/test/java` pour y placer vos classes de test
- Créer une classe de test avec l'annotation `@Test` sur une méthode
- Exécuter la classe de test en faisant un clic droit sur la classe ou la méthode et en choisissant `Run ...`
- Vérifier le résultat du test dans la fenêtre `Run`

Un test unitaire est une méthode qui vérifie le comportement d'une unité de code, c'est-à-dire une méthode, une classe ou un composant. Un test unitaire doit être:

- Isolé: il ne doit pas dépendre d'autres tests ou de l'état du système
- Répétable: il doit produire le même résultat à chaque exécution
- Automatisé: il doit être exécuté sans intervention humaine
- Rapide: il doit s'exécuter en un temps raisonnable

Pour écrire un test unitaire avec JUnit 5, vous devez utiliser l'annotation `@Test` sur une méthode publique sans paramètre et sans valeur de retour. Par exemple, voici un test unitaire qui vérifie que la méthode `add` de la classe `Calculator` renvoie la somme de deux nombres:

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

class CalculatorTest {

    @Test
    void addShouldReturnTheSumOfTwoNumbers() {
        // Arrange
        Calculator calculator = new Calculator();
        int a = 2;
        int b = 3;

        // Act
        int result = calculator.add(a, b);

        // Assert
        assertEquals(5, result);
    }
}
```

La méthode de test suit le schéma AAA (Arrange, Act, Assert):

- Arrange: on initialise les objets et les données nécessaires au test
- Act: on invoque la méthode à tester avec les paramètres appropriés
- Assert: on vérifie que le résultat obtenu est conforme à l'attendu

Pour vérifier le résultat du test, on utilise les méthodes de la classe `org.junit.jupiter.api.Assertions`, qui permettent de faire des assertions sur les valeurs, les exceptions, les conditions, etc. Par exemple, la méthode `assertEquals` vérifie que deux valeurs sont égales. Si l'assertion échoue, le test est marqué comme échoué et un message d'erreur est affiché.
Pour exécuter le test, on peut faire un clic droit sur la classe ou la méthode de test et choisir `Run ...`. On peut aussi utiliser le raccourci `Ctrl+Shift+F10`. Le résultat du test s'affiche dans la fenêtre `Run`, avec un symbole vert si le test réussit, ou un symbole rouge si le test échoue.
Pour ce TP, il faut réaliser les tests unitaires selon le category class equivalence testing et les pusher sur vos comptes Github.

Exercice 0
Implémenter les tests pour les exemples vus en cours.

Exercice 1
La classe Person représente une personne avec un nom, un prénom et un âge. La méthode getFullName renvoie le nom complet de la personne, sous la forme “prénom nom”. La méthode isAdult renvoie true si la personne est majeure, c’est-à-dire si son âge est supérieur ou égal à 18 ans, et false sinon.

```
public class Person {

    private String firstName;
    private String lastName;
    private int age;

    public Person(String firstName, String lastName, int age) {
        this.firstName = firstName;
        this.lastName = lastName;
        this.age = age;
    }

    public String getFullName() {
        return firstName + " " + lastName;
    }

    public boolean isAdult() {
        return age >= 18;
    }
}
```

Exercice 2
La classe Stack implémente une pile d’entiers avec les opérations push, pop et peek. La méthode push ajoute un élément au sommet de la pile. La méthode pop retire et renvoie l’élément au sommet de la pile. La méthode peek renvoie l’élément au sommet de la pile sans le retirer. La méthode isEmpty renvoie true si la pile est vide, et false sinon. La méthode size renvoie le nombre d’éléments dans la pile. La pile est basée sur un tableau interne, dont la capacité initiale est de 10. Si la pile est pleine, le tableau est agrandi automatiquement.

```
public class Stack {
```

```
private int[] array;
private int top;
private static final int INITIAL_CAPACITY = 10;

public Stack() {
    array = new int[INITIAL_CAPACITY];
    top = -1;
}

public void push(int element) {
    if (top == array.length - 1) {
        expandArray();
    }
    array[++top] = element;
}

public int pop() {
    if (isEmpty()) {
        throw new IllegalStateException("Stack is empty");
    }
    return array[top--];
}

public int peek() {
    if (isEmpty()) {
        throw new IllegalStateException("Stack is empty");
    }
    return array[top];
}

public boolean isEmpty() {
    return top == -1;
}

public int size() {
    return top + 1;
```

```
    }

    private void expandArray() {
        int[] newArray = new int[array.length * 2];
        System.arraycopy(array, 0, newArray, 0, array.length);
        array = newArray;
    }
}
```

Exercice 3

La classe Fibonacci contient une méthode statique fibonacci qui calcule le n-ième terme de la suite de Fibonacci, définie par la relation de récurrence: F(n) = F(n-1) + F(n-2), avec F(0) = 0 et F(1) = 1. La méthode utilise un algorithme récursif.

```
public class Fibonacci {

    public static int fibonacci(int n) {
        if (n < 0) {
            throw new IllegalArgumentException("n must be positive");
        }
        if (n == 0) {
            return 0;
        }
        if (n == 1) {
            return 1;
        }
        return fibonacci(n - 1) + fibonacci(n - 2);
    }
}
```

Exercice 4

La classe Prime contient une méthode statique isPrime qui vérifie si un nombre entier est premier, c'est-à-dire s'il n'a que deux diviseurs positifs: 1 et lui-même. La méthode utilise un algorithme naïf qui teste tous les diviseurs potentiels de 2 à la racine carrée du nombre.

```
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;
    }
}
```

Exercice 5

La classe Factorial contient une méthode statique factorial qui calcule la factorielle d'un nombre entier positif, définie par la relation: n! = n * (n-1) * ... * 2 * 1, avec 0! = 1. La méthode utilise un algorithme itératif.

```
public class Factorial {

    public static int factorial(int n) {
        if (n < 0) {
            throw new IllegalArgumentException("n must be positive");
        }
        int result = 1;
        for (int i = 2; i <= n; i++) {
            result *= i;
        }
        return result;
    }
}
```

Dans cette partie vous avez une implémentation en Java d'un problème réel avec des erreurs dans le code. Pour chaque exercice, vous devez écrire des tests unitaires avec JUnit 5 pour détecter et corriger les erreurs. Vous devez utiliser le principe du *category equivalence testing*, qui consiste à identifier les classes d'équivalence des entrées et des sorties du code, et à choisir des valeurs représentatives pour chaque classe. Vous devez aussi tester les cas limites et les cas exceptionnels.

Exercice Bonus 1

La classe `BankAccount` représente un compte bancaire avec un solde et un taux d'intérêt annuel. La méthode `deposit` permet de déposer une somme d'argent sur le compte. La méthode `withdraw` permet de retirer une somme d'argent du compte. La méthode `transfer` permet de transférer une somme d'argent d'un compte à un autre. La méthode `addInterest` permet d'ajouter les intérêts annuels au solde du compte. La méthode `getBalance` permet de consulter le solde du compte.

```
public class BankAccount {

    private double balance;
    private double interestRate;

    public BankAccount(double balance, double interestRate) {
        this.balance = balance;
        this.interestRate = interestRate;
    }

    public void deposit(double amount) {
        if (amount < 0) {
            throw new IllegalArgumentException("Amount must be positive");
        }
        balance += amount;
    }

    public void withdraw(double amount) {
        if (amount < 0) {
            throw new IllegalArgumentException("Amount must be positive");
        }
        if (amount > balance) {
            throw new IllegalStateException("Insufficient balance");
        }
        balance -= amount;
    }

    public void transfer(double amount, BankAccount other) {
        if (other == null) {
            throw new NullPointerException("Other account must not be
null");
        }
        withdraw(amount);
        other.deposit(amount);
    }

    public void addInterest() {
        balance = balance * (1 + interestRate);
    }

    public double getBalance() {
        return balance;
    }
}
```

Exercice Bonus 2

La classe `Matrix` représente une matrice carrée d'entiers avec les opérations `add`, `multiply` et `transpose`. La méthode `add` permet d'ajouter une autre matrice à la matrice courante. La méthode `multiply` permet de multiplier la matrice

courante par une autre matrice. La méthode `transpose` permet de transposer la matrice courante, c'est-à-dire d'échanger les lignes et les colonnes. La méthode `toString` permet de renvoyer une représentation textuelle de la matrice.

```
public class Matrix {

    private int[][] array;
    private int size;

    public Matrix(int size) {
        this.size = size;
        array = new int[size][size];
    }

    public void set(int i, int j, int value) {
        array[i][j] = value;
    }

    public int get(int i, int j) {
        return array[i][j];
    }

    public void add(Matrix other) {
        if (other == null) {
            throw new NullPointerException("Other matrix must not be
null");
        }
        if (other.size != this.size) {
            throw new IllegalArgumentException("Matrices must have the same
size");
        }
        for (int i = 0; i < size; i++) {
            for (int j = 0; j < size; j++) {
                array[i][j] += other.array[i][j];
            }
        }
    }

    public void multiply(Matrix other) {
        if (other == null) {
            throw new NullPointerException("Other matrix must not be
null");
        }
        if (other.size != this.size) {
            throw new IllegalArgumentException("Matrices must have the same
size");
        }
        int[][] result = new int[size][size];
        for (int i = 0; i < size; i++) {
```

```
            for (int j = 0; j < size; j++) {
                for (int k = 0; k < size; k++) {
                    result[i][j] += array[i][k] * other.array[k][j];
                }
            }
        }
        array = result;
    }

    public void transpose() {
        for (int i = 0; i < size; i++) {
            for (int j = i + 1; j < size; j++) {
                int temp = array[i][j];
                array[i][j] = array[j][i];
                array[j][i] = temp;
            }
        }
    }

    public String toString() {
        StringBuilder sb = new StringBuilder();
        for (int i = 0; i < size; i++) {
            sb.append("[");
            for (int j = 0; j < size; j++) {
                sb.append(array[i][j]);
                if (j < size - 1) {
                    sb.append(", ");
                }
            }
            sb.append("]\n");
        }
        return sb.toString();
    }
}
```