Centre Universitaire de Mila
Institut des sciences et de la technologie

Informatique - Semestre 02
Test logiciel - TP N° 04

TP 4

JUint 5

1. Les assertions

Le package **org.junit.jupiter.api.Assertions.*** de **JUnit Jupiter (JUnit)** contient plusieurs types d'assertions utilisées pour comparer le résultat attendu avec le résultat obtenu d'un test. Le tableau suivant illustre quelques types d'assertions.

Type d'assertion	Description
assertEquals()	Cette assertion est utilisée pour vérifier que la valeur (ou objet) attendue est égale à la valeur (ou objet) réelle ou obtenue.
assertNotEquals()	Cette assertion permet de vérifier que la valeur (ou objet) attendue est différente de la valeur (ou objet) réelle ou obtenue.
assertTrue()	Cette assertion est utilisée pour vérifier qu'une valeur booléenne est vraie.
assertFalse()	Cette assertion est utilisée pour vérifier qu'une valeur booléenne est fausse.
assertArrayEquals	Cette assertion permet de vérifier que deux tableaux sont égaux. Deux tableaux sont considérés comme égaux si : - Ils sont tous les deux nuls ou vides. - Les deux tableaux contiennent les "mêmes" objets ou valeurs. Pour être plus précis, JUnit 5 itère les deux tableaux un élément à la fois et s'assure que les éléments trouvés à partir de l'index donné sont égaux.
assertIterableEquals()	Cette assertion permet de vérifier que deux itérables sont profondément égaux. Deux itérables sont considérés comme égaux si : - Ils sont tous les deux nuls ou vides.

	- Les deux itérables contiennent les "mêmes" objets ou valeurs. Pour être plus précis, JUnit 5 itère les deux itérables un élément à la fois et s'assure que les éléments trouvés à partir de l'index donné sont égaux.
assertNull()	La méthode annotée un objet nul.
assertNotNull()	La méthode annotée un objet n'est pas nul.
assertSame()	Cette assertion est utilisée pour assurer que deux objets font référence au même objet.
assertNotSame()	Cette assertion est utilisée pour assurer que deux objets ne font pas référence au même objet.
assertTimeout()	Cette assertion est utilisée pour assurer que l'exécution du système sous test est terminée avant que le délai spécifié ne soit dépassé. Cette assertion utilise les paramètres de méthode suivants: - Un objet Duration qui spécifie le délai d'expiration. - Un objet Executable ou ThrowingSupplier qui appelle le système testé. - Un message d'erreur facultatif qui s'affiche si le délai d'attente spécifié est dépassé. Dans cette assertion, l'exécutable fourni ou ThrowingSupplier sera exécuté dans le même thread que le code qui l'appelle. De plus, cette méthode n'interrompt pas l'exécution si le délai d'attente est dépassé.
assertTimeoutPreemptively()	Cette assertion est la même que la méthode précédente et elle est utilisée dans le cas où l'exécutable fourni ou ThrowingSupplier sera exécuté dans un thread différent du code qui l'appelle. De plus, cette méthode interrompt l'exécution si le délai d'attente est dépassé.
assertAll()	Cette assertion est utilisée pour écrire une assertion pour un état qui nécessite plusieurs assertions. Cette méthode prend les paramètres de méthode suivants : - Un en-tête facultatif qui identifie l'état affirmé. - Un tableau, une collection ou un flux d'objets exécutables qui invoquent nos assertions.

	Lorsque nous invoquons la méthode assertAll(), elle appelle toutes les assertions spécifiées et signale tous les échecs d'assertion une fois que toutes les assertions ont été exécutées.

Exemple1 :

- Créez une nouvelle classe de **Junit test** qui s'appelle «ObjectAssertionTest » avec la spécification suivante :

```
@DisplayName("Writing assertions for objects")
class ObjectAssertionTest {

  //----------------------assertTrue-----------------------------
       @Test
       @DisplayName("Should be true")
       void shouldBeTrue() {
        assertTrue(true);
        }

  //----------------------assertFalse----------------------------
       @Test
       @DisplayName("Should be false")
       void shouldBeFalse() {
          assertFalse(false);
         }

  //----------------------assertNull-----------------------------
       @Test
    @DisplayName("Should be null")
    void shouldBeNull() {
       assertNull(null);
      }

  //----------------------assertNotNull--------------------------
    @Test
    @DisplayName("Should not be null")
    void shouldNotBeNull() {
       assertNotNull(new Object());
       }

  //----------------------assertEquals---------------------------
    private final Integer ACTUAL = 9;
    private final Integer EXPECTED = 9;
    @Test
    @DisplayName("Should be equal")
    void shouldBeEqual() {
       assertEquals(EXPECTED, ACTUAL);
      }


  //----------------------assertNotEquals------------------------
    private final Integer ACTUAL2 = 9;
    private final Integer EXPECTED2 = 4;
    @Test
    @DisplayName("Should not be equal")
    void shouldNotBeEqual() {
```

```
        assertNotEquals(EXPECTED2, ACTUAL2);
      }

  //----------------------assertSame-----------------------------
    private final Object ACTUAL3 = new Object();
    private final Object EXPECTED3 = ACTUAL3;
    @Test
    @DisplayName("Should refer to the same object")
    void shouldReferToSameObject() {
       assertSame(EXPECTED3, ACTUAL3);
      }


  //----------------------assertNotSame-----------------------------
    private final Object ACTUAL4 = new Object();
    private final Object EXPECTED4 = new Object();
    @Test
    @DisplayName("Should not refer to the same object")
    void shouldNotReferToSameObject() {
       assertNotSame(EXPECTED4, ACTUAL4);
      }


  //----------------------assertArrayEquals-----------------------------
    private final int[] ACTUAL5 = new int[]{2, 5, 7};
    private final int[] EXPECTED5 = new int[]{2, 5, 7};
      @Test
      @DisplayName("Should contain the same integers")
      void shouldContainSameIntegers() {
          assertArrayEquals(EXPECTED5, ACTUAL5);
         }


  //----------------------assertIterableEquals-----------------------------
      private final List<Integer> FIRST = Arrays.asList(1, 2, 3);
      private final List<Integer> SECOND = Arrays.asList(1, 2, 3);
      @Test
      @DisplayName("Should contain the same elements")
      void shouldContainSameElements() {
          assertIterableEquals(FIRST, SECOND);
         }


  //----------------------assertTimeout-----------------------------

      @Test
      @DisplayName("Should return the correct message before timeout is
exceeded")
      void shouldReturnCorrectMessageBeforeTimeoutIsExceeded1() {
      final String message = assertTimeout(Duration.ofMillis(50), () -> {
          Thread.sleep(20);
          return "Hello World!";
           });
           assertEquals("Hello World!", message);
          }


  //----------------------assertTimeoutPreemptively-------------------------
---
```

```
	@Test
	@DisplayName("Should return the correct message before timeout is
exceeded")
	void shouldReturnCorrectMessageBeforeTimeoutIsExceeded2() {
	final String message = assertTimeoutPreemptively(Duration.ofMillis(50),
() -> {
	    Thread.sleep(20);
	    return "Hello World!";
	     });
	    assertEquals("Hello World!", message);
	   }
}
```

Exemple 2 :

- Créez une nouvelle classe JAVA qui s'appelle «Person» avec la spécification suivante :

```
public class Person {

    private String firstName;
    private String lastName;

    public Person() {}

    public String getFirstName() {
        return firstName;
    }

    public String getLastName() {
        return lastName;
    }

    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    public void setLastName(String lastName) {
        this.lastName = lastName;
    }

}
```

- Créez une nouvelle classe JUnit Test qui s'appelle «GroupAssertionsTest» avec la spécification suivante :

```
@DisplayName("Group multiple assertions")
class GroupAssertionsTest {

	private static final String FIRST_NAME = "Jane";
    private static final String LAST_NAME = "Doe";

    private Person person;

    @BeforeEach
    void createPerson() {
```

```
        person = new Person();
        person.setFirstName(FIRST_NAME);
        person.setLastName(LAST_NAME);
    }

    @Test
    @DisplayName("Should have the correct name")
    void shouldHaveCorrectName() {
        assertAll("name",
                () -> assertEquals(FIRST_NAME,
                        person.getFirstName(),
                        "The first name is incorrect"
                ),
                () -> assertEquals(LAST_NAME,
                        person.getLastName(),
                        "The last name is incorrect"
                )
        );
    }
}
```

Exercice :

- Créez une classe java appelée « TriEtRechecheTable» qui contient deux méthodes : la première méthode consiste à trier un tableau d'entier, et la deuxième méthode est utilisé pour chercher un nombre dans le tableau trié. Les deux algorithmes seront donnés en dessous.

- En utilisant les assertions précédentes, testez ce programme.

Algorithme de tri

```
procedure triSelection(entier[] tab)
  entier i, k;
  entier min;
  entier tmp;
  pour (i de 1 à N-1 en incrémentant de 1) faire  */    recherche du numéro du minimum/*
     min ← i;
     pour (k de i+1 à N en incrémentant de 1) faire
          si (tab[k] < tab[min]) alors
               min ← k;
          fin si
      fin pour
 */    échange des valeurs entre la case courante et le minimum/*
      tmp ← tab[i];
      tab[i] ← tab[min];
      tab[min] ← tmp;
  fin pour
fin procédure
```

Algorithme de recherche dichotomique

```
function rechercheDichotomique(tab: array of Integer; valeur: Integer): Integer;
var
   gauche, droite, milieu: Integer;
begin
  gauche := 0;
  droite := Length(tab) - 1;
  while gauche <= droite do
    begin
       milieu := (gauche + droite) div 2;
       if tab[milieu] = valeur then
         begin
            Result := milieu;
           Exit;
         end
         else
            if tab[milieu] < valeur then
                gauche := milieu + 1
            else
              droite := milieu - 1;
       end;
  Result := -1; // La valeur n'a pas été trouvée dans le tableau
end;
```