Junit

- JUnit est une famille de bibliothèques de tests unitaires. Le principe a été repris dans beaucoup d'autres langages pour générer les bibliothèques faisant partie du système *xUnit*.
- Un peu d'histoire
  - La première version de Junit largement utilisée fut Junit 3. Cette version est désormais considérée comme obsolète.
  - La version suivante de Junit fut Junit 4. C'est la version qui a vraiment vu l'explosion de Junit. Elle n'est pas compatible avec Junit 3. Elle reste encore la version la plus présente dans les projets industriels.
  - La dernière version de Junit est **Junit 5**. Elle n'est pas compatible avec Junit 4, mais présente de nombreuses similarités, et des méthodes particulières permettent de lancer depuis Junit 5 des tests Junit 4. Le développement de cette nouvelle version a pris du temps, mais elle semble avoir atteint l'âge de la maturité.

Junit 5

- Référence : https://junit.org
- Javadoc : https://junit.org/junit5/docs/current/api/index.html
- Structure de base : 3 “blocs” :
  - **Junit Platform** : le cœur du système ; cadre général permettant d'exécuter des tests
  - **Junit Jupiter** : la bibliothèque nécessaire à l'exécution de tests *Junit 5*
  - **Junit Vintage** : pour exécuter des tests Junit 3 et Junit 4 depuis Junit 5.
- Intégration dans les outils de développement
  - Junit 5 est maintenant pris en charge par les divers IDE (Eclipse, Netbeans, IntelliJ Idea)
  - Junit 5 est pris en charge par les outils de *build* (Maven, Gradle)

Contenu du présent support

Ce que ce document est :

- un support relativement détaillé pour un cours sur les fonctionnalités de base de Junit 5

Ce que ce document n'est pas :

- Un cours sur l'écriture des tests
- Un document autonome
- Une description exhaustive des fonctionnalités de Junit 5 car :
  - cela prendrait trop de temps
  - Un certain nombre de fonctionnalités de Junit 5 sont encore *expérimentales*

Tests élémentaires : principes de base

- Structure d'une classe de test
  - 1 classe de test = un ensemble de méthodes de test
  - 1 classe de test par classe à tester
  - 1 méthode de test = 1 cas de test
  - 1 cas de test = (description, données d'entrée, résultat attendu)
- Structure d'une méthode de test de base
  - méthode d'instance publique
  - annotée avec `@Test`
  - ne prend aucun paramètre
  - ne renvoie rien
  - lève une AssertionError en cas de test échoué
- Conventions
  - nom d'une classe de test : *NomClasseTestée*Test
  - nom d'une méthode de test : test*NomMethodeTestee*

Premier exemple

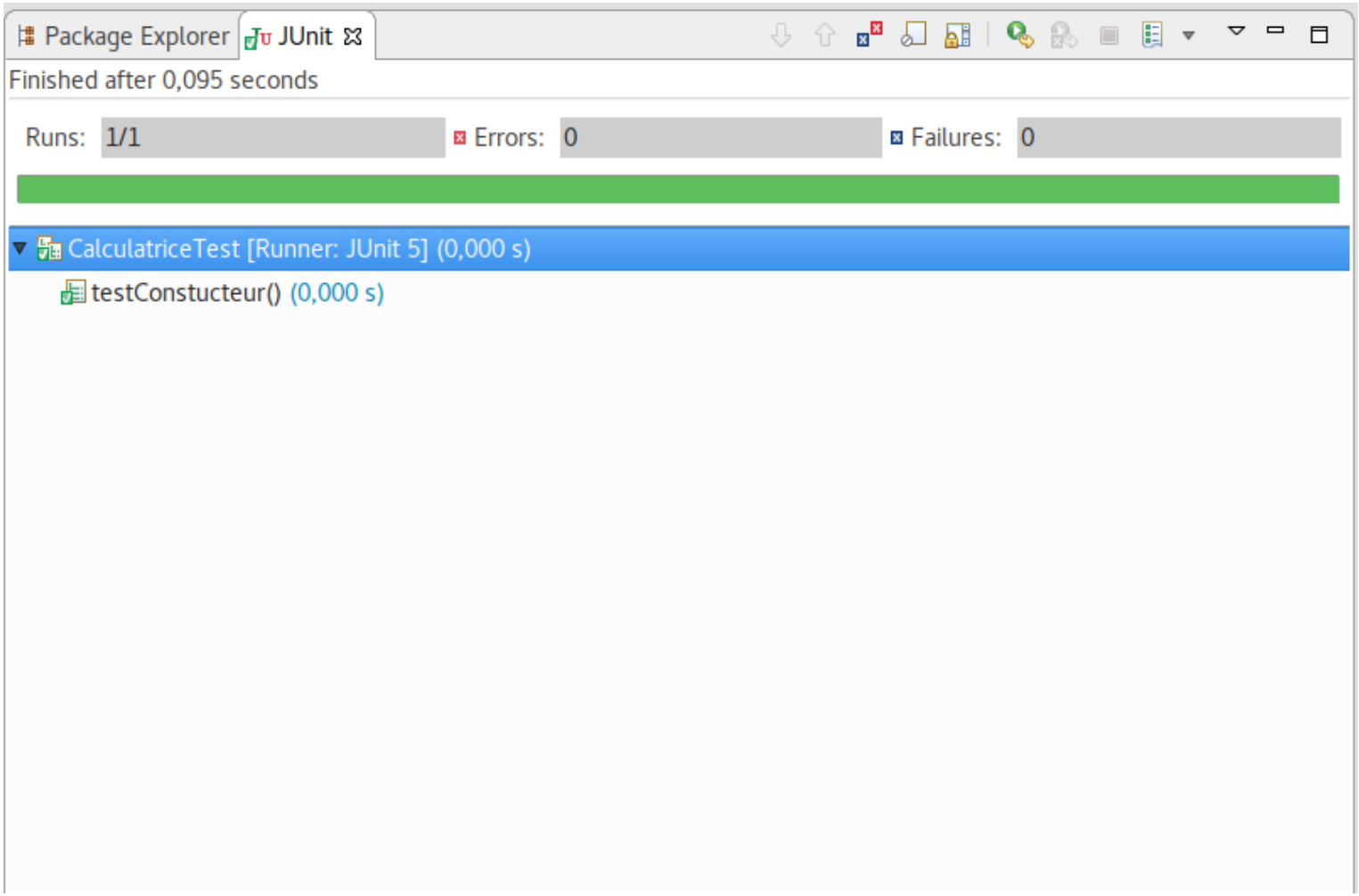
Classe à tester

```java
public class Calculatrice {
  private int x;
  private int y;

  public Calculatrice(int a, int b) {
    x = a;
    y = b;
  }

  public int ajouter() {
    x -= y;
    return x;
  }

  @Override
  public String toString() {
    return "x = " + x + "; y = " + y;
  }
}
```

Classe de test

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

import org.junit.jupiter.api.Test;

class CalculatriceTest {

    @Test
    void testConstucteur() {
        // données d'entrées
        Calculatrice calc = new Calculatrice(4, 5);
        // résultat attendu
        String resultatAttendu = "x = 4; y = 5";
        // résultat effectif
        String resultatEffectif = calc.toString();
        // vérification
        assertEquals(resultatAttendu, resultatEffectif,
                "construction simple");
    }
}
```

Premier exemple : exécution dans Eclipse



Deuxième Exemple

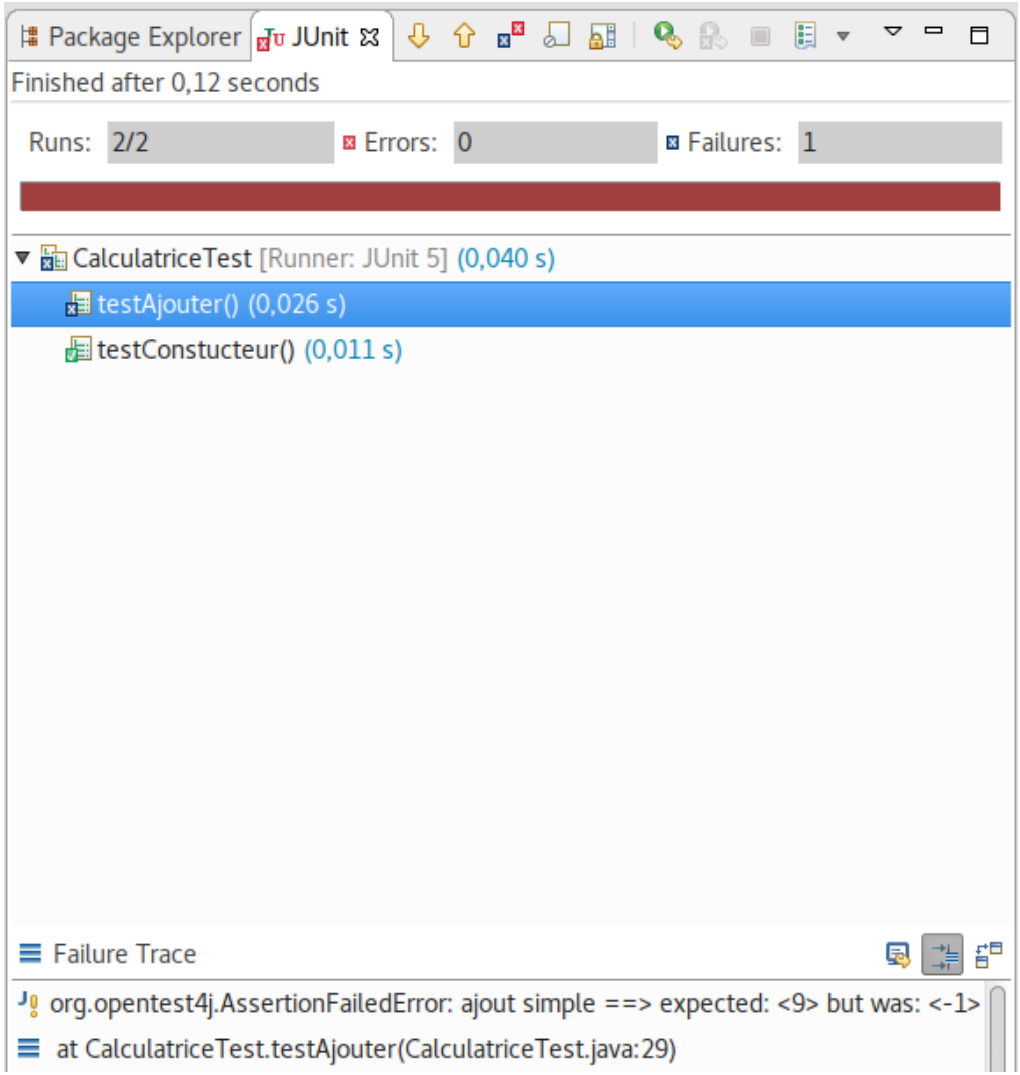
Classe à tester

```java
public class Calculatrice {
  private int x;
  private int y;

  public Calculatrice(int a, int b) {
    x = a;
    y = b;
  }

  public int ajouter() {
    x -= y;
    return x;
  }

  @Override
  public String toString() {
    return "x = " + x + "; y = " + y;
  }
}
```

Classe de test

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

import org.junit.jupiter.api.Test;

class CalculatriceTest {

  @Test
  void testConstucteur() {...

  @Test
  void testAjouter() {
    // données d'entrées
    Calculatrice calc = new Calculatrice(4, 5);
    // résultat attendu
    int resultatAttendu = 9;
    // résultat effectif
    int resultatEffectif = calc.ajouter();
    // vérification
    assertEquals(resultatAttendu, resultatEffectif,
                 "ajout simple");
  }

}
```

Deuxième exemple : exécution dans Eclipse



La classe Assertions

Rôle

Ensemble de méthodes statiques aidant à écrire des tests. Ces méthodes lèvent des `AssertionFailedError` (sous-type de `AssertionError`) en cas d'échec.

Structure des différentes méthodes

La plupart de ces méthodes existent sous 3 formats :

- avec les paramètres de base
- avec les paramètres de base et une chaîne de caractères correspondant au message à afficher en cas d'échec
- avec les paramètres de base et un `Supplier<String>`, méthode construisant la chaîne de caractères à afficher en cas d'échec

Méthodes de base de la classe `Assertions` (1)

- `assertTrue(boolean)` : Le paramètre doit être vrai
- `assertEquals(Object attendu, Object effectif)` : `attendu.equals(effectif)` doit être vrai
- `assertEquals(`*typeDeBase* `attendu,` *typeDeBase* `effectif)` : `attendu == effectif` doit être vrai
- `assertEquals(`*typeReel* `attendu,` *typeReel* `effectif,` *typeReel* `delta)` : `abs(attendu - effectif) < delta` doit être vrai
- `assertSame(Object attendu, Object effectif)` : `attendu == effectif` doit être vrai
- `assertNull(Object objNul)` : `objNul == null` doit être vrai

Méthodes de base de la classe `Assertions` (2)

- `assertArrayEquals(`*Type[]* `attendu,` *Type[]*`effectif)` : égal “profond” sur les tableaux
- `assertIterableEquals(Iterable<?> attendu, Iterable<?> effectif)` : égal “profond” sur les itérables
- `assertThrows(Class<T> typeException, Executable exécutable)` : `exécutable` doit lever une exception du type spécifié

Méthodes de la classe `Assertions` : mais aussi...

- `assertFalse(boolean)` : No comment...
- `assertNotEquals(...)` : No comment...
- `assertNotSame(Object attendu, Object effectif)` : No comment...
- `assertNotNull(Object obj)`
- `assertArrayEquals(`*TypeReel[]* `attendu,` *TypeReel[]* `effectif,` *TypeReel* `delta)` : égal “profond” sur les tableaux de réels
- `assertDoesNotThrows(Class<T> typeException, Executable exécutable)` : No comment...

Contrôle du temps d'exécution

- `assertTimeout(Duration durée, Executable exécutable)`
  - l'`exécutable` s'exécute complètement, mais il faut que sa durée d'exécution soit inférieure à `durée` pour que le test soit considéré comme réussi
- `assertTimeout(Duration durée, ThrowingSupplier<T> exécutable)`
  - l'`exécutable` s'exécute complètement, mais il faut que sa durée d'exécution soit inférieure à `durée` pour que le test soit considéré comme réussi. Par ailleurs, s'il n'y a pas eu d'exception levée, le résultat de `exécutable` est renvoyé.
- `assertTimeoutPreemptively(Duration durée, Executable exécutable)`
- `assertTimeoutPreemptively(Duration durée, ThrowingSupplier<T> exécutable)`
  - Idem que précédemment, mais l'exécution est interrompue si elle n'est pas terminée au bout du délai spécifié

Temps d'exécution : exemple

```
@Test
void testDureeExecution() {
  Calculatrice calc = new Calculatrice(4,5);
  int res = assertTimeout(Duration.ofMillis(1),
                    () -> {return calc.soustraire();});
  assertEquals(-1, res);
}
```

Détecter plusieurs erreurs en une fois

Problème

Comme un échec se traduit par une levée d'exception, cela interrompt l'exécution de la méthode de test. Du coup, si une méthode de test fait plusieurs vérifications, Le premier échec qui survient bloc l'exécution des tests suivants.

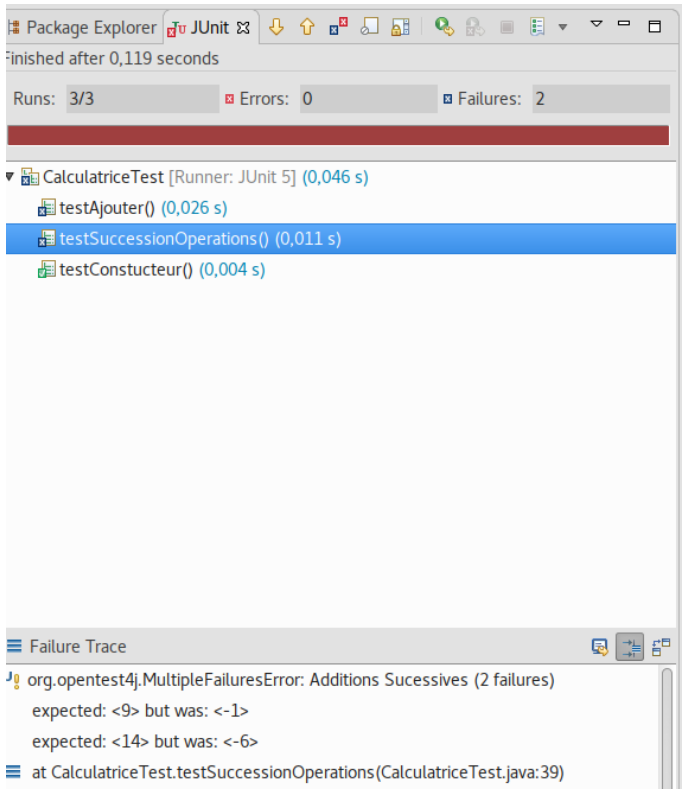
Solution

- `assertAll(String enTete, Collection<Executable> exécutables)`
- `assertAll(String enTete, Stream<Executable> exécutables)`
- `assertAll(String enTete, Executable... exécutables)`
  - Avec ces différentes méthodes, les `AssertionError` pouvant survenir dans les différents `exécutables` sont rassemblées en une seule `MultipleFailuresError`

Plusieurs erreurs à la fois : Exemple

Classe à tester

```
class CalculatriceTest {

  @Test
    void testConstucteur() {...

  @Test
    void testAjouter() {...

  @Test
  void testSuccessionOperations() {
    // données d'entrées
    Calculatrice calc = new Calculatrice(4, 5);
    final int res1, res2;
    res1 = calc.ajouter();
    res2 = calc.ajouter();
    assertAll("Additions Sucessives",
      () -> {assertEquals(9, res1);},
      () -> {assertEquals(14, res2);}
    );
  }
}
```



Méthode `assertLinesMatch(List<String> attendu, List<String> effectif)`

Principe

Vérifier une liste de lignes. Les lignes de `attendu` peuvent être :

- des lignes réellement attendues
- des lignes de la forme >>*entier*>>, spécifiant un nombre de ligne à sauter
- des lignes de la forme >>*nonEntier*>>, pour préciser que l'on peut sauter un nombre quelconque de lignes jusqu'à ce qu'une ligne soit identique à la ligne suivante

Exemple (attendu, effectif 1, effectif 2)

```
a
>> 2 >>
b
>> sans importance >>
c
```

```
a
z
z
b
c
```

```
a
z
z
b
x
y
z
c
```

Alternatives à la classe `Assertions`

Il est possible d'utiliser d'autres bibliothèques d'assertions. Il est notamment possible d'utiliser la méthode `assertThat(T, Matcher<? super T>)` de la bibliothèque *Hamcrest* (`org.hamcrest.core`).

Exemple

```
@Test
void testAjouterHamcrest() {
  // données d'entrées
  Calculatrice calc = new Calculatrice(4, 5);
  // résultat attendu
  int resultatAttendu = 9;
  // résultat effectif
  int resultatEffectif = calc.ajouter();
  // vérification
  assertThat(resultatEffectif, equalTo(9));
}
```

Décorer l'exécution des tests

- Initialisation avant l'exécution de tous les tests
  - Méthode de classe ne renvoyant rien et sans paramètre annotée avec `@BeforeAll`
- Finalisation à la fin de l'exécution de tous les tests
  - Méthode de classe ne renvoyant rien et sans paramètre annotée avec `@AfterAll`
- Code à exécuter avant chaque test
  - Méthode d'instance ne renvoyant rien et sans paramètre annotée avec `@BeforeEach`
- Code à exécuter après chaque test
  - Méthode d'instance ne renvoyant rien et sans paramètre annotée avec `@AfterEach`

Affichage du nom des tests

Les différentes solutions

- Par défaut : le nom de la méthode
- Modifié statiquement : `@DisplayName(`*nomAAfficher*`)`
- Modifié dynamiquement (**depuis Junit 5.4**): `@DisplayNameGeneration(`*nomClasseGénérateur*`.class)`

Écriture d'un générateur de nom

- écrire une classe implantant `DisplayNameGenerator` (idéalement, hériter de `DisplayNameGenerator.ReplaceUnderscores` ou de `DisplayNameGenerator.Standard`)
- redéfinir la ou les méthodes souhaitées (`generateDisplayNameForMethod()` par exemple)
- N.B. : un générateur de nom pourra souvent être défini sous la forme d'une classe interne statique de la classe de test

Étiquetage des tests

Il est possible d'annoter des méthodes de test avec une ou plusieurs étiquettes. Cela s'effectue avec l'annotation `@Tag(`*étiquette*`)`.

Les étiquettes peuvent être utilisées dans le `pom.xml` d'un projet maven pour requérir/invalider l'exécution des tests en question.

Suppositions (*Assumptions*)

Il est possible d'invalider certains tests à l'exécution en fonction du contexte. Ces tests ne seront alors pas exécutés. Pour ce faire, on peut utiliser les méthodes de classe définies dans la classe `Assumptions` (du package `org.junit.jupiter.api`). Ces méthodes ressemblent aux Méthodes de la classe `Assertions`, mais elles lèvent une `TestAbortedException` afin que le test ne soit pas considéré comme un échec.

Il est également possible d'invalider certains tests avec les annotations définies dans le package `org.junit.jupiter.api.condition` :

- `@DisabledIfEnvironmentVariable` / `@EnabledIfEnvironmentVariable`
- `@DisabledIfSystemProperty` / `@EnabledIfSystemProperty`
- `@DisabledOnOs` / `@EnabledOnOs`
- `@DisabledOnjre` / `@EnabledOnjre`

Injection de dépendance

Si une méthode ou un constructeur d'une classe de test contient des paramètres de type `TestInfo` ou `TestReporter`, des instances adéquates sont automatiquement passées en paramètre.

- TestInfo : permet de fournir des informations sur les conditions de test, notamment grâce aux méthodes suivantes :
  - `getDisplayName() : String`
  - `getTags() : Set<String>`
  - `getTestClass() : Optional<Class<?>>`
  - `getTestMethod() : Optional<Method>`
- TestReporter : permet d'enrichir les messages renvoyés pendant l'exécution des tests grâce aux méthodes suivantes :
  - `publishEntry(String) : void`
  - `publishEntry(String clé, String valeur) : void`
  - `publishEntry(Map<String, String>) : void`

Exécuter plusieurs fois une méthode de test

Dans certains cas, il peut être souhaitable d'exécuter plusieurs fois une méthode de test. Il faut alors utiliser l'annotation `@RepeatedTest(`*nombreRépétitions*`)` au lieu de l'annotation `@Test`.

Premier Exemple

```
@RepeatedTest(10)
void testSoustractionNulle() {
  Random dé = new Random();
  int valeurTest = dé.nextInt(100);
  Calculatrice calc = new Calculatrice(valeurTest, valeurTest);
  int resultatEffectif = calc.soustraire();
  // vérification
  assertEquals(0, resultatEffectif);
}
```

Exemple avec nom plus détaillé

```
@RepeatedTest(value=10, name=RepeatedTest.LONG_DISPLAY_NAME)
void testSoustractionNulle() {...
```

Répétition de test et injection de dépendance

Dans le cas d'un test répété, la présence d'un paramètre de type `RepetitionInfo` pour la méthode de test, comme pour les méthodes `AfterEach` ou `BeforeEach` amènera Junit à passer automatiquement l'objet en question. Sur cet objet, 2 méthodes sont disponibles :

- getCurrentRepetition() : int
- getTotalRepetitions() : int

Tests paramétrés (1)

Principe

Il est possible de paramétrer des méthodes de test pour les faire exécuter avec des valeurs différentes. Il faut alors :

- Annoter avec `@ParameterizedTest` au lieu de `@Test`
- Fournir une *source* pour les paramètres

Si la méthode de test prend une seul paramètre

On utilise l'annotation `@ValueSource` et l'on initialise le paramètre optionnel du type requis avec un tableau de valeurs. Exemple :

```
@ParameterizedTest
@ValueSource(ints = {1, 3, 6, 8})
void testParametreSoustraire(int n1) {
  Calculatrice calc = new Calculatrice(5, n1);
  int resultatAttendu = 5 - n1;
  int resultatEffectif = calc.soustraire();
  assertEquals(resultatAttendu, resultatEffectif);
}
```

Tests paramétrés (2)

Si la méthode de test prend plusieurs paramètres (chaînes ou types de base)

On peut utiliser l'annotation `@CsvSource` et fournir en paramètre à cette annotation un tableau de chaînes de caractères contenant les listes de paramètres séparés par des virgules. Exemple :

```
@ParameterizedTest
@CsvSource({"1, 3", "6, 8"})
void test2ParametresSoustraire(int n1, int n2) {
  Calculatrice calc = new Calculatrice(n1, n2);
  int resultatAttendu = n1 - n2;
  int resultatEffectif = calc.soustraire();
  assertEquals(resultatAttendu, resultatEffectif);
}
```

N.B. : l'annotation `CsvFileSource` permet de lire de spécifier un fichier au format CSV comme source des données de test d'une méthode.

Tests paramétrés (3)

La méthode prend un seul paramètre ; cas général

On utilise l'annotation @MethodSource à laquelle on passe en paramètre le nom d'une méthode de la classe courante. Cette méthode doit renvoyer un flux (`Stream`) de données du type de paramètre de la méthode. La méthode de test sera appelée pour chacun des paramètres du flux. Exemple :

```
@ParameterizedTest
@MethodSource("entiers")
void testParametresSoustraireViaMethode(int n2) {
  Calculatrice calc = new Calculatrice(5, n2);
  int resultatAttendu = 5 - n2;
  int resultatEffectif = calc.soustraire();
  assertEquals(resultatAttendu, resultatEffectif);
}

static IntStream entiers() {
  return IntStream.of(1, 3, 5, 7);
}
```

Tests paramétrés (4)

La méthode prend plusieurs paramètres ; cas général

On utilise l'annotation @MethodSource à laquelle on passe en paramètre le nom d'une méthode de la classe courante. Cette méthode doit renvoyer un flux d'arguments (`Stream<Arguments>`). Chaque `Arguments` est construit grâce à la méthode `of` de l'interface `Arguments`. Exemple :

```
@ParameterizedTest
@MethodSource("couplesEntiers")
void testParametresSoustraireViaMethode2(int n1, int n2) {
  Calculatrice calc = new Calculatrice(n1, n2);
  int resultatAttendu = n1 - n2;
  int resultatEffectif = calc.soustraire();
  assertEquals(resultatAttendu, resultatEffectif);
}

static Stream<Arguments> couplesEntiers() {
  return Stream.of(Arguments.of(10, 3), Arguments.of(5, 9));
}
```

N.B. : l'annotation `@ArgumentsSource` permet de spécifier une classe implantant l'interface `ArgumentsProvider` plutôt qu'une méthode comme source des données.

Tests dynamiques

Il est possible de générer dynamiquement des tests en utilisant une méthode génératrice de tests. Une telle méthode doit être annotée avec `@TestFactory` est doit renvoyer un Iterable, un Iterateur ou un flux de tests dynamiques (éventuellement hiérarchisés). Exemple :

```
@TestFactory
Stream<DynamicTest> generateurTests() {
  IntStream fluxEntier = IntStream.range(0, 10);
  Stream<DynamicTest> fluxTests = fluxEntier.mapToObj(
    n -> dynamicTest("test " + n,
          () -> {Calculatrice calc = new Calculatrice(n, n);
                 assertEquals(0, calc.soustraire());
  }));
  return fluxTests;
}
```