JUnit 5 (Jupiter) — Guide pratique

> Objectif : comprendre **comment écrire, organiser et exécuter** des tests unitaires avec **JUnit 5** dans un projet **Java + Maven**.

---

1) Pré-requis rapides

- **JDK** : 11+ (idéalement 17).
- **Maven** : utiliser le **Wrapper** (`mvnw`) pour la reproductibilité.
- **Dépendance JUnit 5** (dans `pom.xml`) :

```
<properties>
  <maven.compiler.release>17</maven.compiler.release>
  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  <junit.jupiter.version>5.10.2</junit.jupiter.version>
</properties>

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

<build>
  <plugins>
    <!-- Surefire exécute les tests JUnit 5 -->
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-surefire-plugin</artifactId>
      <version>3.2.5</version>
      <configuration>
        <useModulePath>false</useModulePath>
      </configuration>
    </plugin>
  </plugins>
</build>
```

---

2) Arborescence standard

```
src/
 ├─ main/java/...        # code “production”
 └─ test/java/...        # code de test (JUnit)
```

- Un **test** valide le comportement d’une **unité** (classe/méthode).
- Convention : `MaClasseTest.java` (un fichier de test par classe cible).

---

3) Annotations et API clés

- `@Test` : marque une méthode comme test.
- `@BeforeEach` / `@AfterEach` : setup/teardown **par test**.
- `@BeforeAll` / `@AfterAll` : setup/teardown **global** (méthodes `static`).
- `@ParameterizedTest` + `@CsvSource` / `@MethodSource` : tests paramétrés (table de cas).
- `@Disabled("raison")` : désactiver temporairement (à éviter en CI).
- **Assertions** (import statique recommandé) :
  - `assertEquals` , `assertNotNull` , `assertTrue/False` , `assertThrows` , `assertAll` , etc.

---

4) Écrire un test : le pattern AAA

AAA = Arrange / Act / Assert

1. **Arrange** : préparer l'objet et les données.
2. **Act** : appeler la méthode à tester.
3. **Assert** : vérifier un résultat **précis** (ou une **exception** attendue).

Exemple minimal :

```
// src/main/java/com/acme/Calc.java
package com.acme;

public class Calc {
  public long factorial(int n) {
    if (n < 0) throw new IllegalArgumentException("n must be >= 0");
    long r = 1;
    for (int i = 2; i <= n; i++) r *= i;
    return r;
  }
}
```

```
// src/test/java/com/acme/CalcTest.java
package com.acme;

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

class CalcTest {

  @Test
  void should_return1_when_nIsZero() {           // Arrange + Act
    long result = new Calc().factorial(0);
    assertEquals(1, result);                     // Assert
  }

  @Test
  void should_throw_when_nIsNegative() {         // Assert sur exception
    assertThrows(IllegalArgumentException.class,
      () -> new Calc().factorial(-1));
  }
}
```

---

5) Tests paramétrés (rapides et expressifs)

Idéal pour **multiplier les cas** sans dupliquer le code :

```
// src/test/java/com/acme/CalcParamTest.java
package com.acme;

import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.CsvSource;
import static org.junit.jupiter.api.Assertions.*;

class CalcParamTest {

  @ParameterizedTest
  @CsvSource({ "1,1", "2,2", "3,6", "4,24" })
  void factorial_known_values(int n, long expected) {
    assertEquals(expected, new Calc().factorial(n));
  }
}
```

6) Organisation & lisibilité

- **Noms de tests** parlants : `should_<comportement>_when_<condition>()` .
- Un test = **un comportement** vérifié.
- Évitez les dépendances réseau/IO/temps réel dans les **tests unitaires** (flaky tests).
- Pour isoler des dépendances : utilisez un framework de mocks (ex. Mockito).

Exemple de **setup** commun :

```
import org.junit.jupiter.api.*;

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

class ExampleLifecycleTest {

  @BeforeAll
  static void initAll() {
    // Ressources globales (DB embarquée, ports dynamiques, etc.)
  }

  @BeforeEach
  void init() {
    // Réinitialiser l'état avant chaque test
  }

  @AfterEach
  void tearDown() {
    // Nettoyer après chaque test si nécessaire
  }

  @AfterAll
  static void tearDownAll() {
    // Libérer les ressources globales
  }

  @Test
  void sample() {
    assertTrue(1 + 1 == 2);
  }
}
```

7) Exécuter les tests

- Tous les tests :

  ```
  ./mvnw test
  ```

- Avec build complet (recommandé) :

  ```
  ./mvnw clean verify
  ```

- Cibler une classe/un test précis :

  ```
  ./mvnw -Dtest=CalcTest test
  ./mvnw -Dtest=CalcTest#should_return1_when_nIsZero test
  ```

Rapports JUnit (Surefire) : `target/surefire-reports/*.xml`
→ lisibles par les serveurs CI (Jenkins, GitLab CI, etc.).

8) Bonnes pratiques & anti-patterns

Bonnes pratiques

- Tests **rapides**, **déterministes**, **localisés** (une seule raison d'échec).
- Couvrez **cas nominaux**, **limites** (0, 1, max) & **erreurs** (exceptions).
- Préférez `assertEquals/Throws` à des `assertTrue` trop génériques.
- Factorisez le setup commun (`@BeforeEach`) si répétitif.

À éviter

- Tests qui dépendent du réseau, de l'heure système, de l'ordre d'exécution.
- Assertions vagues / multiples responsabilités dans un test.
- Mélanger des tests d'intégration lents avec les unitaires **dans le même cycle**.

---

9) Exemples de tests (simples & pédagogiques)

9.1 Calculator — factorielle (nominal & erreur)

```
// src/test/java/com/acme/CalcSimpleTest.java
package com.acme;

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

class CalcSimpleTest {

  @Test
  void factorial_of_0_is_1() {
    assertEquals(1, new Calc().factorial(0));
  }

  @Test
  void factorial_negative_throws_IAE() {
    assertThrows(IllegalArgumentException.class, () -> new Calc().factorial(-5));
  }
}
```

9.2 StringUtils — inversion de chaîne & null-safety

```
// src/main/java/com/acme/StringUtils.java
package com.acme;

public class StringUtils {
  public String reverse(String s) {
    if (s == null) throw new NullPointerException("s is null");
    return new StringBuilder(s).reverse().toString();
  }

  public boolean isBlank(String s) { // "" ou uniquement espaces
    return s == null || s.trim().isEmpty();
  }
}
```

```java
// src/test/java/com/acme/StringUtilsTest.java
package com.acme;

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

class StringUtilsTest {

  @Test
  void reverse_simple_word() {
    assertEquals("tac", new StringUtils().reverse("cat"));
  }

  @Test
  void reverse_null_throws_NPE() {
    assertThrows(NullPointerException.class, () -> new StringUtils().reverse(null));
  }

  @Test
  void isBlank_handles_spaces_and_empty() {
    StringUtils u = new StringUtils();
    assertTrue(u.isBlank(""));
    assertTrue(u.isBlank("   "));
    assertFalse(u.isBlank("a"));
  }
}
```

9.3 FizzBuzz — test paramétré

```java
// src/main/java/com/acme/FizzBuzz.java
package com.acme;

public class FizzBuzz {
  public String of(int n) {
    if (n % 15 == 0) return "FizzBuzz";
    if (n % 3 == 0)  return "Fizz";
    if (n % 5 == 0)  return "Buzz";
    return String.valueOf(n);
  }
}
```

```java
// src/test/java/com/acme/FizzBuzzTest.java
package com.acme;

import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.CsvSource;
import static org.junit.jupiter.api.Assertions.*;

class FizzBuzzTest {

  @ParameterizedTest
  @CsvSource({
    "1,1", "2,2", "3,Fizz", "5,Buzz", "15,FizzBuzz"
  })
  void fizzbuzz_cases(int n, String expected) {
    assertEquals(expected, new FizzBuzz().of(n));
  }
}
```

---

10) TL;DR

- **JUnit 5** = écrire des **tests unitaires** clairs et rapides.
- Placez-les dans `src/test/java`, nommez-les de façon **explicite**.
- Exécutez via **Surefire** : `./mvnw test` (ou `./mvnw clean verify`).
- Couvrez **nominal + limites + erreurs** ; utilisez `assertThrows` pour les exceptions.
- Les **tests paramétrés** simplifient la couverture de multiples cas.

Bonne pratique : gardez vos tests **lisibles et ciblés** – un test = un comportement vérifié.