

# Programmation Orientée Objets avec Java

Prof. Yassin Rekik et Prof. Stéphane Malandain - 2022-2023



## Chapitre 9

### Les tests unitaires



- Fiabilité des logiciels :
  - Probabilité qu'un système logiciel ne provoque pas de défaillance dans des conditions spécifiées.
  - Mesurée par le temps de fonctionnement, le MTTF (mean time till failure), les données de crash.
- Les bugs sont inévitables dans tout système logiciel complexe.
  - Estimations de l'industrie : 10 à 50 bugs pour 1000 lignes de code.
- Un bug peut être visible ou se cacher dans votre code jusqu'à une date ultérieure.
- Tests unitaires : Une tentative systématique de révéler les erreurs.
  - Test échoué : une erreur a été démontrée.
  - Test réussi : aucune erreur n'a été trouvée (pour cette situation particulière).

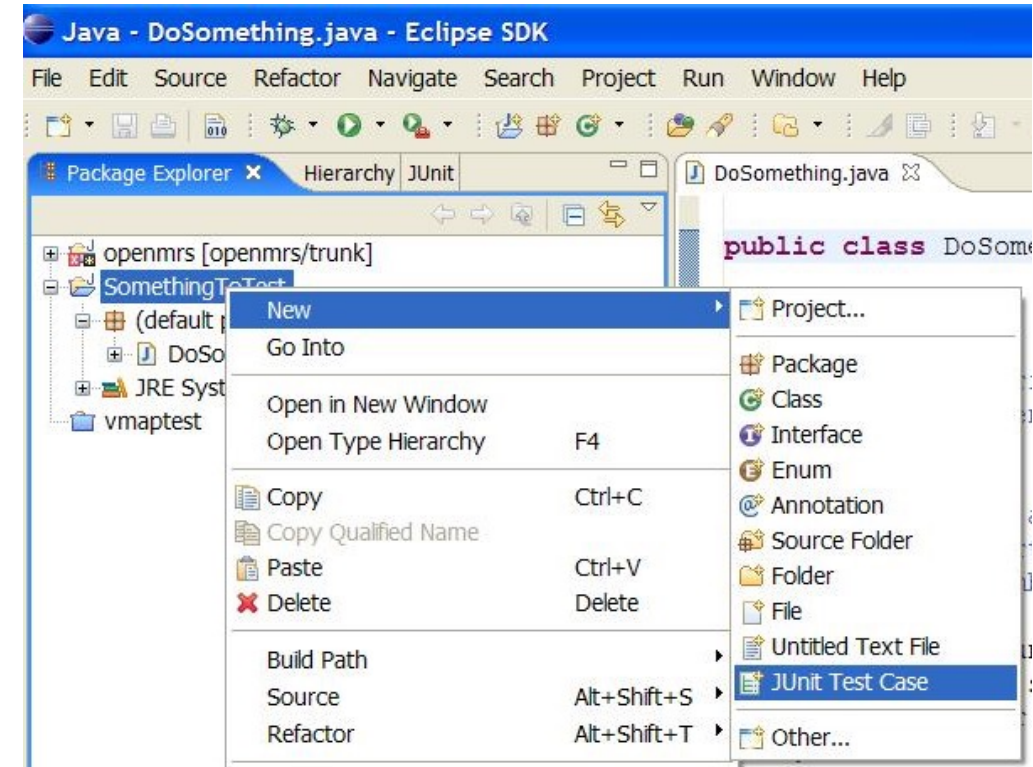
- Perception de certains développeurs et managers :
  - Les tests sont considérés comme un travail de novice.
  - Il est confié aux membres les moins expérimentés de l'équipe.
  - Effectués après coup (voir jamais).
  - "Mon code est bon, il n'y a pas de bugs, je n'ai pas besoin de le tester. Je n'ai pas besoin de le tester".
  - "Je trouverai les bugs en exécutant le programme client".
- Limites de ce que les tests peuvent vous montrer :
  - Il est impossible de tester complètement un système.
  - Les tests ne révèlent pas toujours directement les bugs réels dans le code.
  - Les tests ne prouvent pas l'absence d'erreurs dans les logiciels.

- Les tests unitaires : Recherche d'erreurs dans un sous-système isolé.
  - En général, un "sous-système" désigne une classe ou un objet particulier.
  - La bibliothèque Java JUnit nous aide à effectuer facilement des tests unitaires.
- L'idée de base :
  - Pour une classe donnée Foo, créez une autre classe FooTest pour la tester, contenant diverses méthodes de "cas de test" à exécuter.
  - Chaque méthode recherche des résultats particuliers et réussit ou échoue.
  - JUnit fournit des commandes "assert" pour nous aider à écrire des tests.
  - L'idée : Placez des appels d'assertion dans vos méthodes de test pour vérifier les choses que vous vous attendez à voir vraies. Si ce n'est pas le cas, le test échouera.

# JUnit / Eclipse

5

- Pour ajouter JUnit à un projet Eclipse, cliquez sur :
  - Project → Properties → Build Path → Libraries → Add Library... → JUnit → JUnit 4 → Finish
- Pour créer un cas de test :
  - cliquez avec le bouton droit de la souris sur un fichier et choisissez : New → Test Case
  - Ou en general : File → New → JUnit Test Case
- Eclipse peut créer des squelettes de tests pour vous.



# Classe de test (a exécuter en tant que test)

6

```
import org.junit.*;
import static org.junit.Assert.*;

public class name {
    ...

    @Test
    public void name() { // a test case method
        ...
    }
}
```

- Une méthode tagguée @Test sera considérée comme un “JUnit test case”.
  - Toutes les @Test méthodes seront exécutées quand Junit teste la classe.

# JUnit assertion methods

7

<code>assertTrue(<b>test</b>)</code>	fails if the boolean test is <code>false</code>
<code>assertFalse(<b>test</b>)</code>	fails if the boolean test is <code>true</code>
<code>assertEquals(<b>expected</b>, <b>actual</b>)</code>	fails if the values are not equal
<code>assertSame(<b>expected</b>, <b>actual</b>)</code>	fails if the values are not the same (by <code>==</code> )
<code>assertNotSame(<b>expected</b>, <b>actual</b>)</code>	fails if the values <i>are</i> the same (by <code>==</code> )
<code>assertNull(<b>value</b>)</code>	fails if the given value is <i>not</i> <code>null</code>
<code>assertNotNull(<b>value</b>)</code>	fails if the given value is <code>null</code>
<code>fail()</code>	causes current test to immediately fail

- Chaque méthode peut également recevoir une chaîne de caractères à afficher en cas d'échec :
  - par exemple, `assertEquals("message", expected, actual)`

# ArrayList JUnit test

8

```
import org.junit.*;
import static org.junit.Assert.*;
public class TestArrayList {
    @Test
    public void testAddGet1() {
        ArrayList list = new ArrayList();
        list.add(42);
        list.add(-3);
        list.add(15);
        assertEquals(42, list.get(0));
        assertEquals(-3, list.get(1));
        assertEquals(15, list.get(2));
    }
    @Test
    public void testIsEmpty() {
        ArrayList list = new ArrayList();
        assertTrue(list.isEmpty());
        list.add(123);
        assertFalse(list.isEmpty());
        list.remove(0);
        assertTrue(list.isEmpty());
    }
    ...
}
```



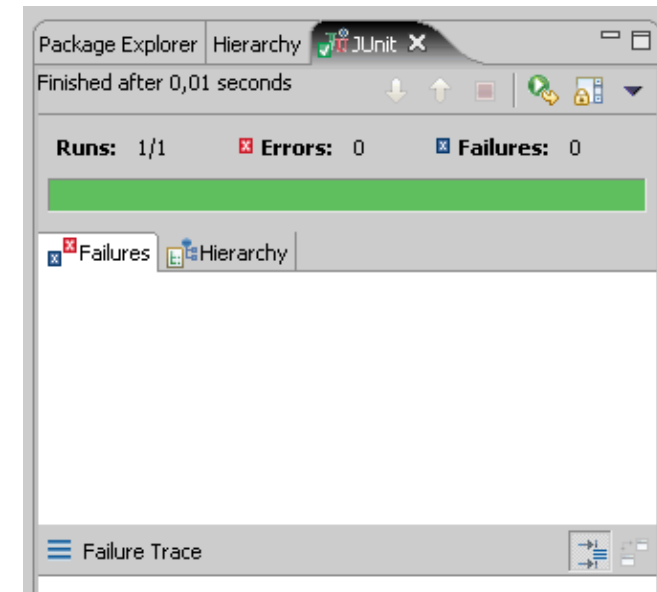
# Exécuter les tests

9

- Cliquez avec le bouton droit de la souris sur le paquet dans l'explorateur de paquets d'Eclipse à gauche ; choisissez :
  - Run As → JUnit Test



- La barre JUnit s'affiche en vert si tous les tests passent, en rouge s'ils échouent.
- La trace d'échec indique quels tests ont échoué, le cas échéant, et pourquoi.



- On suppose une classe Date avec les méthodes suivantes :
  - `public Date(int year, int month, int day)`
  - `public Date() // today`
  - `public int getDay(), getMonth(), getYear()`
  - `public void addDays(int days) // advances by days`
  - `public int daysInMonth()`
  - `public String dayOfWeek() // e.g. "Sunday"`
  - `public boolean equals(Object o)`
  - `public boolean isLeapYear()`
  - `public void nextDay() // advances by 1 day`
  - `public String toString()`

# Ce qui ne va pas !

11

```
public class DateTest {  
    @Test  
    public void test1() {  
        Date d = new Date(2050, 2, 15);  
        d.addDays(4);  
        assertEquals(d.getYear(), 2050); // il faut inverser  
        assertEquals(d.getMonth(), 2);  
        assertEquals(d.getDay(), 19);  
    }  
  
    @Test  
    public void test2() {  
        Date d = new Date(2050, 2, 15);  
        d.addDays(14);  
        assertEquals(d.getYear(), 2050);  
        assertEquals(d.getMonth(), 3);  
        assertEquals(d.getDay(), 1);  
    }  
}
```

# Assertions bien structurées

12

```
public class DateTest {  
    @Test  
    public void test1() {  
        Date d = new Date(2050, 2, 15);  
        d.addDays(4);  
        assertEquals(2050, d.getYear()); // expected  
        assertEquals(2, d.getMonth());   // value should  
        assertEquals(19, d.getDay());     // be at LEFT  
    }  
  
    @Test  
    public void test2() {  
        Date d = new Date(2050, 2, 15);  
        d.addDays(14);  
        assertEquals("year after +14 days", 2050, d.getYear());  
        assertEquals("month after +14 days", 3, d.getMonth());  
        assertEquals("day after +14 days", 1, d.getDay());  
    } // test cases should usually have messages explaining  
} // what is being checked, for better failure output
```

# Utilisation d'objet attendue

13

```
public class DateTest {  
    @Test  
    public void test1() {  
        Date d = new Date(2050, 2, 15);  
        d.addDays(4);  
        Date expected = new Date(2050, 2, 19);  
        assertEquals(expected, d);           // use an expected answer  
    }                                         // object to minimize tests  
  
                                           // (Date must have toString  
                                           // and equals methods)  
    @Test  
    public void test2() {  
        Date d = new Date(2050, 2, 15);  
        d.addDays(14);  
        Date expected = new Date(2050, 3, 1);  
        assertEquals("date after +14 days", expected, d);  
    }  
}
```

# Nommage des test cases

14

```
public class DateTest {  
    @Test  
    public void test_addDays_withinSameMonth_1() {  
        Date actual = new Date(2050, 2, 15);  
        actual.addDays(4);  
        Date expected = new Date(2050, 2, 19);  
        assertEquals("date after +4 days", expected, actual);  
    }  
    // give test case methods really long descriptive names  
  
    @Test  
    public void test_addDays_wrapToNextMonth_2() {  
        Date actual = new Date(2050, 2, 15);  
        actual.addDays(14);  
        Date expected = new Date(2050, 3, 1);  
        assertEquals("date after +14 days", expected, actual);  
    }  
    // give descriptive names to expected/actual values  
}
```

# Ce qui ne va pas !

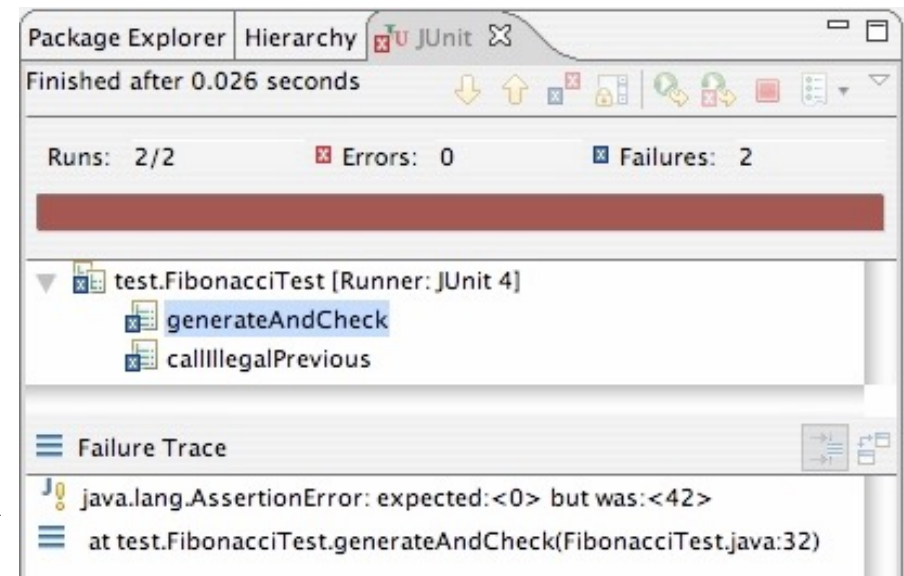
15

```
public class DateTest {  
    @Test  
    public void test_addDays_addJustOneDay_1() {  
        Date actual = new Date(2050, 2, 15);  
        actual.addDays(1);  
        Date expected = new Date(2050, 2, 16);  
        assertEquals(  
            "should have gotten " + expected + "\n" +  
            " but instead got " + actual + "\n",  
            expected, actual);  
    }  
    ...  
}
```

# Les messages des assertions

16

```
public class DateTest {  
    @Test  
    public void test_addDays_addJustOneDay_1() {  
        Date actual = new Date(2050, 2, 15);  
        actual.addDays(1);  
        Date expected = new Date(2050, 2, 16);  
        assertEquals("adding one day to 2050/2/15",  
            expected, actual);  
    }  
    ...  
}  
  
// JUnit affichera déjà  
// les valeurs attendues et réelles  
// dans sa sortie ;  
//  
// Il n'est pas nécessaire de les répéter  
// dans le message de l'assertion
```





# Tests avec Timeout

17

```
@Test(timeout = 5000)
    public void name() { ... }
    // La méthode ci-dessus sera considérée comme un échec si
    // elle ne s'exécute pas dans les 5000 ms.

    private static final int TIMEOUT = 2000;
    ...

    @Test(timeout = TIMEOUT)
    public void name() { ... }
    // Times out / fails after 2000 ms
```

# Pervasive timeouts

18

```
public class DateTest {  
    @Test(timeout = DEFAULT_TIMEOUT)  
    public void test_addDays_withinSameMonth_1() {  
        Date d = new Date(2050, 2, 15);  
        d.addDays(4);  
        Date expected = new Date(2050, 2, 19);  
        assertEquals("date after +4 days", expected, d);  
    }  
    @Test(timeout = DEFAULT_TIMEOUT)  
    public void test_addDays_wrapToNextMonth_2() {  
        Date d = new Date(2050, 2, 15);  
        d.addDays(14);  
        Date expected = new Date(2050, 3, 1);  
        assertEquals("date after +14 days", expected, d);  
    }  
    // presque tous les tests devraient avoir un timeout afin de ne pas  
    // conduire à une boucle infinie. Il est bon de fixer une valeur par défaut, aussi  
    private static final int DEFAULT_TIMEOUT = 2000;  
}
```

# Tester les exceptions

19

```
@Test(expected = ExceptionType.class)
public void name() {
    ...
}
```

```
// Le test est réussi s'il déclenche l'exception donnée.
// Si l'exception n'est pas levée, le test échoue.
// Utilisez cette fonction pour tester les erreurs attendues.
```

```
@Test(expected = ArrayIndexOutOfBoundsException.class)
public void testBadIndex() {
    ArrayList list = new ArrayList();
    list.get(4);    // should fail
}
```

# Cycle de vie d'exécution

20

```
@Before
```

```
public void name() { ... }
```

```
@After
```

```
public void name() { ... }
```

```
// méthodes à exécuter avant/après l'appel de chaque méthode de test
```

```
// dans JUnit 5 la syntaxe a changé : beforeEach / afterEach
```

```
@BeforeClass
```

```
public static void name() { ... }
```

```
@AfterClass
```

```
public static void name() { ... }
```

```
// méthodes à exécuter une fois avant/après l'exécution de l'ensemble de la classe de test
```

```
// dans JUnit 5 la syntaxe a changé : beforeAll / afterAll
```

- Vous ne pouvez pas tester toutes les entrées possibles, toutes les valeurs de paramètres, etc.
  - Vous devez donc penser à un ensemble limité de tests susceptibles de révéler des bugs.
- Pensez aux cas limites
  - les nombres positifs, nuls et négatifs
  - juste à la limite de la taille d'un tableau ou d'une collection
- Pensez aux cas vides et aux cas d'erreurs
  - 0, -1, null ; une liste ou un tableau vide
- tester le comportement en combinaison
  - peut-être que add fonctionne habituellement, mais échoue après avoir appelé remove
  - faire plusieurs appels ; peut-être que la taille échoue la deuxième fois seulement

# Ce qui ne va pas !

22

```
public class DateTest {
    // test every day of the year
    @Test(timeout = 10000)
    public void tortureTest() {
        Date date = new Date(2050, 1, 1);
        int month = 1;
        int day = 1;
        for (int i = 1; i < 365; i++) {
            date.addDays(1);
            if (day < DAYS_PER_MONTH[month]) {day++;}
            else {month++; day=1;}
            assertEquals(new Date(2050, month, day), date);
        }
    }

    private static final int[] DAYS_PER_MONTH = {
        0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31
    }; // Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec
}
```

- Testez une chose à la fois par méthode de test.
  - 10 petits tests valent bien mieux qu'un test 10 fois plus important.
- Chaque méthode de test doit comporter peu d'affirmations (probablement 1).
  - Si vous faites plusieurs assertions, la première qui échoue arrête le test.
  - Vous ne saurez pas si une assertion ultérieure aurait échoué.
- Les tests doivent éviter la logique.
  - minimiser les if/else, les boucles, les switchs, etc.
  - éviter les try/catch
  - S'il est supposé être lancé, utilisez `expected= ...` sinon, laissez JUnit l'attraper.
- Les “Torture Tests” sont acceptables, mais seulement en complément de tests simples.

# Éliminer la redondance

24

```
public class DateTest {  
    @Test(timeout = DEFAULT_TIMEOUT)  
    public void addDays_withinSameMonth_1() {  
        addHelper(2050, 2, 15, +4, 2050, 2, 19);  
    }  
    @Test(timeout = DEFAULT_TIMEOUT)  
    public void addDays_wrapToNextMonth_2() {  
        addHelper(2050, 2, 15, +14, 2050, 3, 1);  
    }  
    // utiliser beaucoup d'aides pour rendre les tests extrêmement courts  
    private void addHelper(int y1, int m1, int d1, int add,  
        int y2, int m2, int d2) {  
        Date act = new Date(y, m, d);  
        actual.addDays(add);  
        Date exp = new Date(y2, m2, d2);  
        assertEquals("after +" + add + " days", exp, act);  
    }  
    // peut également utiliser des "tests paramétrés" dans certains frameworks  
    ...  
}
```



# Flexible helpers

25

```
public class DateTest {

    @Test(timeout = DEFAULT_TIMEOUT)

    public void addDays_multipleCalls_wrapToNextMonth2x() {

        Date d = addHelper(2050, 2, 15, +14, 2050, 3, 1);

        addhelper(d, +32, 2050, 4, 2);

        addhelper(d, +98, 2050, 7, 9);

    }

    // Les helper peuvent vous enfermer ; il est difficile de tester de nombreux appels/combinaisons.
    // Créer des variations qui permettent une meilleure flexibilité

    private Date addHelper(int y1, int m1, int d1, int add, int y2, int m2, int d2) {

        Date date = new Date(y, m, d);

        addHelper(date, add, y2, m2, d2);

        return d;

    }

    private void addHelper(Date date, int add, int y2, int m2, int d2) {

        date.addDays(add);

        Date expect = new Date(y2, m2, d2);

        assertEquals("date after +" + add + " days", expect, d);

    }

    ...
}
```

# Tests avec conditions

26

- Tests exécutés selon les OS ou les Version Java

```
@Test
@EnabledOnJre(JAVA_8)
void onlyOnJava8() {
    // ...
}

@Test
@EnabledOnJre({ JAVA_9, JAVA_10 })
void onJava9Or10() {
    // ...
}

@Test
@EnabledForJreRange(min = JAVA_9, max = JAVA_11)
void fromJava9to11() {
    // ...
}
```

```
@Test
@EnabledOnOs(MAC)
void onlyOnMacOs() {
    // ...
}

@TestOnMac
void testOnMac() {
    // ...
}

@Test
@EnabledOnOs({ LINUX, MAC })
void onLinuxOrMac() {
    // ...
}

@Test
@DisabledOnOs(WINDOWS)
void notOnWindows() {
    // ...
}
```

# Tests avec conditions

27

- Conditionner selon des paramètres système ou des variables d'environnement

```
@Test
@EnabledIfSystemProperty(named = "os.arch", matches = ".*64.*")
void onlyOn64BitArchitectures() {
    // ...
}

@Test
@DisabledIfSystemProperty(named = "ci-server", matches = "true")
void notOnCiServer() {
    // ...
}
```

```
@Test
@EnabledIfEnvironmentVariable(named = "ENV", matches = "staging-server")
void onlyOnStagingServer() {
    // ...
}

@Test
@DisabledIfEnvironmentVariable(named = "ENV", matches = ".*development.*")
void notOnDeveloperWorkstation() {
    // ...
}
```

# Tests répétés

28

- Exécution répétée des tests
- Possibilité d'afficher les informations de chaque itération

```
@RepeatedTest(5)
void repeatedTestWithRepetitionInfo(RepetitionInfo repetitionInfo) {
    assertEquals(5, repetitionInfo.getTotalRepetitions());
}

@RepeatedTest(value = 8, failureThreshold = 2)
void repeatedTestWithFailureThreshold(RepetitionInfo repetitionInfo) {
    // Simulate unexpected failure every second repetition
    if (repetitionInfo.getCurrentRepetition() % 2 == 0) {
        fail("Boom!");
    }
}

@RepeatedTest(value = 1, name = "{displayName} {currentRepetition}/{totalRepetitions}")
@DisplayName("Repeat!")
void customDisplayName(TestInfo testInfo) {
    assertEquals("Repeat! 1/1", testInfo.getDisplayName());
}

@RepeatedTest(value = 1, name = RepeatedTest.LONG_DISPLAY_NAME)
@DisplayName("Details...")
void customDisplayNameWithLongPattern(TestInfo testInfo) {
    assertEquals("Details... :: repetition 1 of 1", testInfo.getDisplayName());
}
```

# Nested Tests

29

- Il est possible d'organiser les tests hiérarchiquement

```
@DisplayName("A stack")
class TestingAStackDemo {

    Stack<Object> stack;

    @Test
    @DisplayName("is instantiated with new Stack()")
    void isInstantiatedWithNew() {
        new Stack<>();
    }

    @Nested
    @DisplayName("when new")
    class WhenNew {

        @BeforeEach
        void createNewStack() {
            stack = new Stack<>();
        }

        @Test
        @DisplayName("is empty")
        void isEmpty() {
            assertTrue(stack.isEmpty());
        }
    }
}
```

# Assumptions

30

- Possible d'ajouter des Assumptions
- Le test est arrêté en cas d'assumption non valide

```
class AssumptionsDemo {  
  
    private final Calculator calculator = new Calculator();  
  
    @Test  
    void testOnlyOnCiServer() {  
        assumeTrue("CI".equals(System.getenv("ENV")));  
        // remainder of test  
    }  
  
    @Test  
    void testOnlyOnDeveloperWorkstation() {  
        assumeTrue("DEV".equals(System.getenv("ENV")),  
            () -> "Aborting test: not on developer workstation");  
        // remainder of test  
    }  
}
```

# Tests paramétrés

31

- Effectuer des tests avec des arguments externes
  - **Source** peut être une **Value Source**
  - Primitive types + java.lang.String + java.lang.Class (instances de la classe Class)

```
@ParameterizedTest
@ValueSource(ints = { 1, 2, 3 })
void testWithValueSource(int argument) {
    assertTrue(argument > 0 && argument < 4);
}
```

# Tests paramétrés

32

- Effectuer des tests avec des arguments externes
  - Source peut être une méthode génératrice

```
@ParameterizedTest
@MethodSource("stringProvider")
void testWithExplicitLocalMethodSource(String argument) {
    assertNotNull(argument);
}

static Stream<String> stringProvider() {
    return Stream.of("apple", "banana");
}
```



# Tests paramétrés

33

- Effectuer des tests avec des arguments externes
  - Source peut être des données en csv

```
@ParameterizedTest
@CsvSource({
    "apple,          1",
    "banana,         2",
    "'lemon, lime', 0xF1",
    "strawberry,     700_000"
})
void testWithCsvSource(String fruit, int rank) {
    assertNotNull(fruit);
    assertNotEquals(0, rank);
}
```

# Tests paramétrés

34

- Effectuer des tests avec des arguments externes
  - Source peut être un fichier csv

```
@ParameterizedTest
@CsvFileSource(resources = "/two-column.csv", numLinesToSkip = 1)
void testWithCsvFileSourceFromClasspath(String country, int reference) {
    assertNotNull(country);
    assertNotEquals(0, reference);
}

@ParameterizedTest
@CsvFileSource(files = "src/test/resources/two-column.csv", numLinesToSkip = 1)
void testWithCsvFileSourceFromFile(String country, int reference) {
    assertNotNull(country);
    assertNotEquals(0, reference);
}
```

- Régression : Lorsqu'une fonctionnalité qui fonctionnait auparavant ne fonctionne plus.
  - Cette situation est susceptible de se produire lorsque le code évolue et se développe au fil du temps.
  - Une nouvelle fonctionnalité/un nouveau correctif peut provoquer un nouveau bug ou réintroduire un ancien bug.
- test de régression : Ré-exécution de tests unitaires antérieurs après un changement.
  - Souvent effectué par des scripts lors de tests automatisés.
  - Utilisé pour s'assurer que les anciens bugs corrigés sont toujours corrigés.
  - Donne à votre application un niveau minimum de fonctionnalité.
- De nombreux produits disposent d'un ensemble de tests de vérification obligatoires qui doivent être réussis avant que le code puisse être ajouté à un dépôt de code source.

- Les tests unitaires peuvent être rédigés après, pendant ou même avant le codage.
  - Développement piloté par les tests : Écrire des tests, puis écrire du code pour les réussir.
- Imaginons que nous souhaitons ajouter une méthode `subtractWeeks` à notre classe `Date`, qui décale cette date dans le temps d'un nombre donné de semaines.
  - Écrivez du code pour tester cette méthode avant qu'elle ne soit écrite.
  - Une fois la méthode implémentée, nous saurons si elle fonctionne.

- Besoin de tester beaucoup de tableaux ? Utiliser des Array literals

```
public void exampleMethod(int[] values) { ... }  
...  
exampleMethod(new int[] {1, 2, 3, 4});  
exampleMethod(new int[] {5, 6, 7});
```

- Besoin d'une ArrayList? Utiliser Arrays.asList

```
List<Integer> list = Arrays.asList(7, 4, -2, 3, 9, 18);
```

- Besoin de Set, Queue, etc.? Utiliser les collections à partir d'une liste :

```
Set<Integer> list = new HashSet<Integer>(  
    Arrays.asList(7, 4, -2, 9));
```

# Ce qui ne va pas !

38

```
public class DateTest {
    // shared Date object to test with (saves memory!!1)
    private static Date DATE;

    @Test(timeout = DEFAULT_TIMEOUT)
    public void addDays_sameMonth() {
        DATE = new Date(2050, 2, 15);      // first test;
        addhelper(DATE, +4, 2050, 2, 19); // DATE = 2/15 here
    }

    @Test(timeout = DEFAULT_TIMEOUT)
    public void addDays_nextMonthWrap() { // second test;
        addhelper(DATE, +10, 2050, 3, 1); // DATE = 2/19 here
    }

    @Test(timeout = DEFAULT_TIMEOUT)
    public void addDays_multipleCalls() { // third test;
        addDays_sameMonth();              // go back to 2/19;
        addhelper(DATE, +1, 2050, 2, 20); // test two calls
        addhelper(DATE, +1, 2050, 2, 21);
    }

    ...
}
```

# Test case "smells"

39

- Les tests doivent être autonomes et ne pas se soucier les uns des autres.
- Les "odeurs" (mauvaises choses à éviter) dans les tests :
  - Ordre de test contraint :  
Le test A doit être exécuté avant le test B.  
(généralement une tentative malavisée de tester le flux).
  - Les tests s'appellent les uns les autres : le test A appelle la méthode du test B  
(l'appel d'une aide partagée est acceptable).
  - État partagé mutable : Les tests A/B utilisent tous deux un objet partagé.  
(Si A le casse, qu'arrive-t-il à B ?)



- test suite: One class that runs many JUnit tests.
  - An easy way to run all of your app's tests at once.

```
import org.junit.runner.*;  
import org.junit.runners.*;
```

```
@RunWith(Suite.class)  
@Suite.SuiteClasses({  
    TestCaseName.class,  
    TestCaseName.class,  
    ...  
    TestCaseName.class,  
})  
public class name {}
```

- La syntaxe a changé dans JUnit 5 : @Suite



# Exemple de Test suite

41

```
import org.junit.runner.*;
import org.junit.runners.*;

@RunWith(Suite.class)
@Suite.SuiteClasses({
    WeekdayTest.class,
    TimeTest.class,
    CourseTest.class,
    ScheduleTest.class,
    CourseComparatorsTest.class
})
public class HW2Tests {}
```