Génie logiciel et Qualité

Lucile Torres

lucile.torres@univ-amu.fr

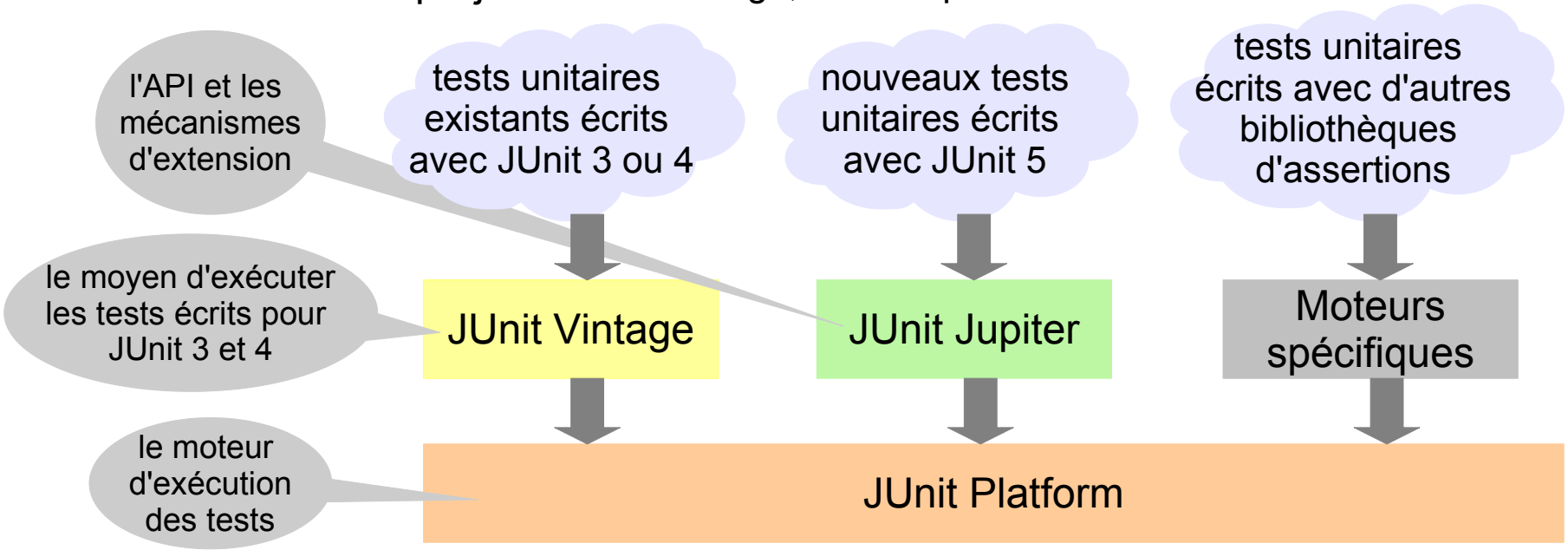
Plan

- Outil JUnit 5
  - Outil JUnit 5 d'écriture et d'exécution de tests automatisés d'applications Java
  - Techniques Java utilisées par JUnit
    - Assertions
    - Annotations : Types d'annotations (annotations standards, annotations utilisateur, méta-annotations) - Exploitation des annotations
- Le test logiciel
  - Qualité et test logiciel, test automatisé, tests unitaires
  - Types de test logiciel : statique / dynamique, structurel / fonctionnel, aléatoire
  - Développement piloté par les tests
- Autres outils pour le test
  - Test statique : SpotBugs, PMD, CheckStyle, JDepend, etc.
  - Couverture de code : Emma
  - Création de doublures de test : Mockito
- *Introduction à la POA (Programmation Orientée Aspect)*
- *Synthèse sur la méta-programmation*

JUnit 5

JUnit 5

- Environnement open source pour **automatiser des tests (unitaires et autres)** sur du code Java, dans l'objectif de s'assurer que le code répond toujours au besoin même après d'éventuelles modifications (tests de non régression).
- A la différence des versions précédentes, JUnit 5 est composé de plusieurs modules issus de trois sous-projets : Junit Vintage, JUnit Jupiter et JUnit Platform.



Ce découpage favorise l'intégration de JUnit dans les outils (IDE, build, intégration continue, etc.) et surtout la customisation de JUnit, tout en permettant l'exécution des tests unitaires existants écrits avec JUnit 3 et 4.

- JUnit 5 nécessite au minimum Java 8 car il utilise des expressions lambdas.

JUnit 5

- L'outil de test unitaire automatisé JUnit 5 utilise :
  - des **annotations** spécifiques, comme @Test ou @BeforeAll, définies dans le paquet org.junit.jupiter.api,
  - des **assertions** spécifiques, comme assertEquals ou assertNonNull, définies dans la classe org.junit.jupiter.api.Assertions
- Les tests sont exprimés dans des **classes de test** sous la forme de cas de tests qui expriment une concordance entre un **résultat obtenu** et un **résultat attendu**. Junit exécute les tests en vérifiant les concordances, le test échoue dès qu'une concordance n'est pas vérifiée. En cas d'échec, une exception de type **java.lang.AssertionFailedError** est levée.
- Une classe de test JUnit 5 crée des instances de la classe à tester et de tout autre objet nécessaire aux tests et organise les cas de test sous forme de méthodes.
- Le code de la classe à tester est séparé du code qui permet de la tester. Ceci évite de créer du code superflu aux traitements proprement dits qui doit être exécuté "à la main", par exemple une méthode main() supplémentaire contenant les traitements de tests.

Procédure de construction des tests

- Création d'un projet Java
- Création d'un package src (avec différents sous-packages) pour les classes de l'application
- Création d'un package test pour les classes de test. On y place les tests unitaires de chaque classe de l'application, mais aussi des tests de module, des tests de recette, des tests de charge, etc.
- Selon le type de test, on peut écrire les classes de test avant ou après le code à tester.
- On construit les cas de test de manière incrémentale, c'est-à-dire qu'on ajoute (ou active) les cas de test à l'intérieur d'une classe de test au fur et à mesure. On lance un cas de test, tant qu'il échoue, on corrige le code et on relance le test. Lorsque le cas de test "passe", on s'occupe du cas de test suivant jusqu'à ce que tous les tests de la classe passent. On fait de même pour chaque classe de test.

Exemple de classe de test JUnit 5

```
package calcul ;
public class Calcul {
        public static int calculer(int a, int b)
        {
                int res = a + b;
                if (a == 0) { res = b * 2; }
                if (b == 0) { res = a * a; }
                return res;
        }
}
```

classe à tester

classe de test
de la classe Calcul

```
package test ;
import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.Test;
Import calcul.Calcul ;

class CalculTest {
        @Test
        void testCalculer( ) {
                assertEquals(2,Calcul.calculer(1,1));
                assertEquals(6,Calcul.calculer(0,3));
                assertEquals(9,Calcul.calculer(3,0));
        }
}
```

Une méthode de test
- ne doit renvoyer aucune valeur
- ne doit pas posséder de paramètres
- doit avoir la visibilité publique ou par défaut
- doit être annotée avec @Test

Les assertions de JUnit 5

- Les tests utilisent des **assertions** sous la forme de méthodes publiques statiques de la classe org.junit.jupiter.api.Assertions :
  - ***assertEquals***, ***assertNotEquals*** pour tester l'égalité/la différence par equals d'instances ou pour tester l'égalité/la différence de nombres primitifs (float, long)
  - ***assertSame***, ***assertNotSame*** pour tester l'égalité/la différence de références
  - ***assertNull***, **assertNotNull** pour vérifier qu'un objet est null/non null
  - ***assertTrue***, ***assertFalse*** pour vérifier qu'une condition est vraie/fausse
  - ***assertArrayEquals*** pour tester l'égalité de tableaux
  - ***assertThrows*** pour vérifier qu'une exception est levée
  - ***assertTimeout***, ***assertTimeoutPreemptively*** pour vérifier que le temps d'exécution ne dépasse par une limite fixée
  - ***assertLinesMatch*** pour vérifier la correspondance de deux List<String>
  - **assertIterableEquals** pour vérifier l'égalité de deux itérations
  - ***assertAll*** pour réaliser des assertions multiples
  - ***fail*** pour forcer le test à échouer

Les assertions de JUnit5

- Chacune de ces méthodes possède une version surchargée qui accepte un paramètre supplémentaire sous la forme d'une chaîne de caractères indiquant un message qui sera affiché en cas d'échec du cas de test.
  - *public static void assertTrue(boolean condition)*
  - *public static void assertTrue(boolean condition, **String message**)*

    exemple : assertTrue(x > 0);
    assertTrue(x > 0, "x doit être positif");
  - *public static void assertEquals(Object attendu, Object obtenu)*
  - *public static void assertEquals(Object attendu, Object obtenu, **String message**)*

    exemple : assertEquals("Durand", nom);
    assertEquals("Durand", nom, "nom n'est pas Durand !");
  - *public static void assertEquals(double attendu, double obtenu, double delta)*
    *public static void assertEquals(double attendu, double obtenu, double delta, **String message**)*

    exemple : assertEquals(0.3, 0.1 + 0.2, 0.0001);
    assertEquals(0.3, 0.1 + 0.2, 0.0001, "Valeurs différentes avec la tolérance donnée");
  - *public static void fail()*
    *public static void fail(**String message**)*

Exemples d'assertions JUnit5

```
int i = 2;
assertEquals(2,i);
assertEquals(2,i,"i doit être égal à 2");
assertNotEquals(0,i);
assertNotEquals(0,i,"i doit être différent de 0");
assertTrue( i > 0);
assertFalse( i <= 0, "i doit être positif");
```

```
Integer a = new Integer(2);
Integer b = a;
assertSame(b,a);
assertEquals(b,a);
assertNotNull(a, "a doit être non null");
```

```
String s = null;
assertNull( s);
```

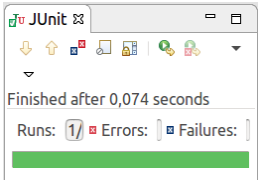
```
int [ ] c = {1,2,3};
int [ ] d = {1,2,3};
assertNotSame(c,d);
assertNotEquals(c,d);
assertArrayEquals(c,d);
```

```
ArrayList<Integer> liste1 =
      new ArrayList<>(Arrays.asList(1, 2, 3));
ArrayList<Integer> liste2 =
      new ArrayList<>(Arrays.asList(1, 2));
HashSet<Integer> ensemble = new HashSet<>(liste1);
assertNotEquals(liste1, ensemble);
assertNotEquals(liste1, liste2);

assertIterableEquals(liste1, ensemble);
assertIterableEquals(liste1, liste2); //AssertionFailedError : iterable lengths
                                      //differ, expected <3> but was : <2>
liste2.add(3);
assertIterableEquals(liste1, liste2);
```

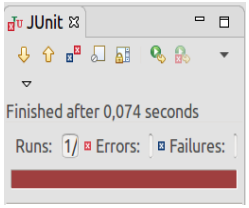
assertIterableEquals vérifie que le nombre et l'ordre des éléments lors de l'itération sont les mêmes, mais aussi que les éléments sont equals.

Les assertions de JUnit 5

assertEquals(2,2);

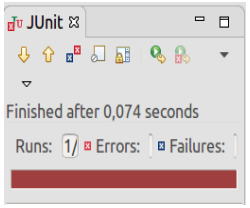


assertEquals(2,3);



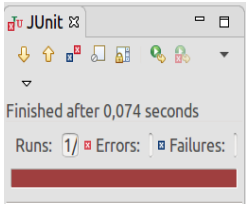
AssertionFailedError: expected <2> but was: 3

assertEquals(2,3,"Valeur attendue 2");



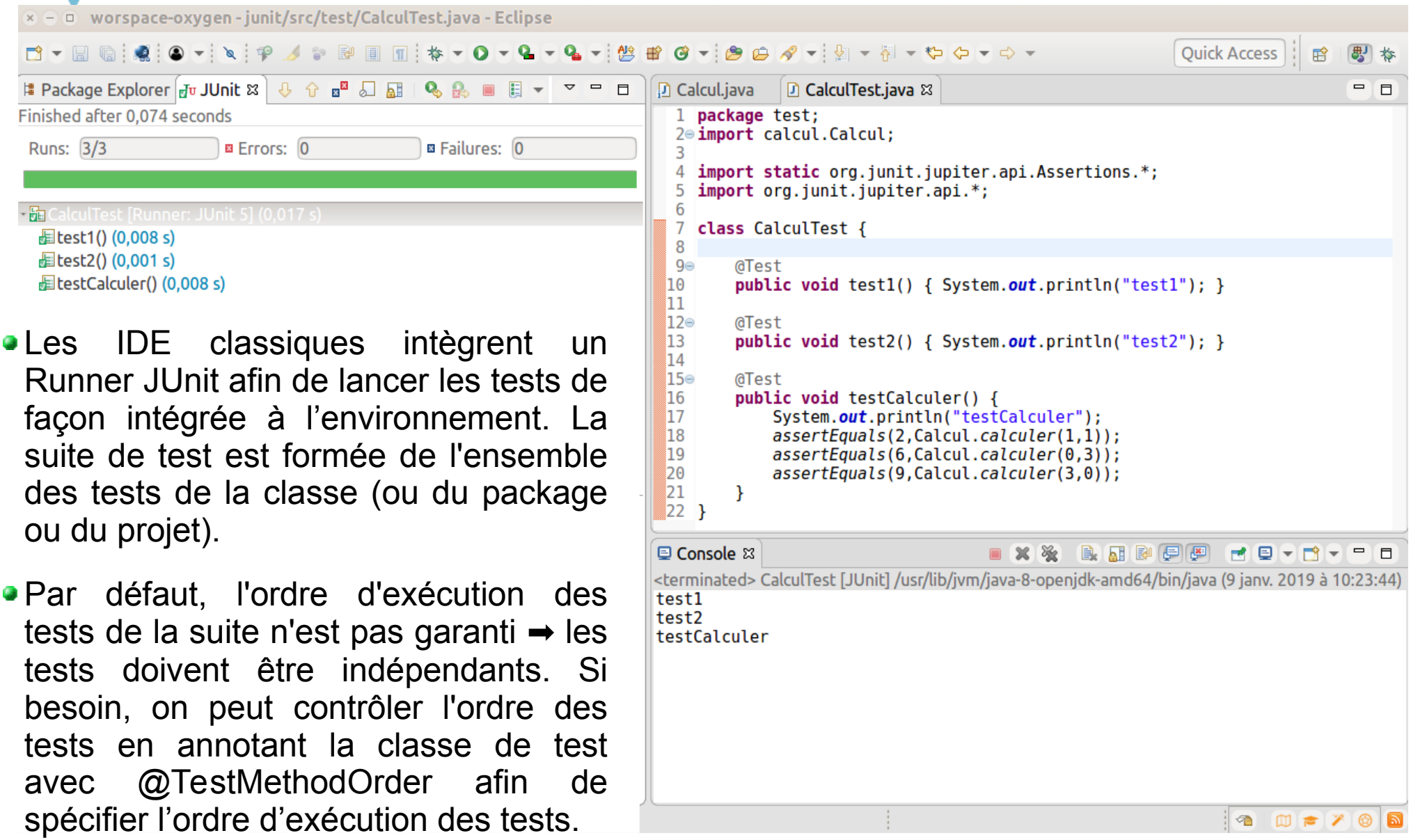
AssertionFailedError: Valeur attendue 2
==> expected <2> but was: 3

fail("problème") ;



AssertionFailedError: problème

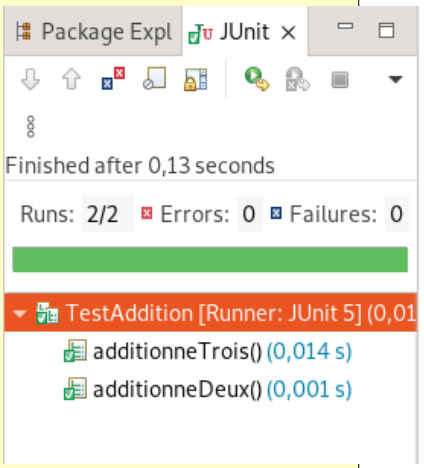
Exécution de la suite de tests

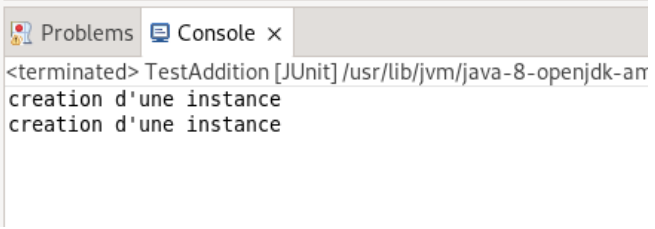


- Les IDE classiques intègrent un Runner JUnit afin de lancer les tests de façon intégrée à l'environnement. La suite de test est formée de l'ensemble des tests de la classe (ou du package ou du projet).
- Par défaut, l'ordre d'exécution des tests de la suite n'est pas garanti ➡ les tests doivent être indépendants. Si besoin, on peut contrôler l'ordre des tests en annotant la classe de test avec @TestMethodOrder afin de spécifier l'ordre d'exécution des tests.

Cycle de vie de la classe de test

```
package test;
import static org.junit.Assert.assertEquals;
import org.junit.jupiter.api.Test;

class TestAddition {
    private int somme;

    TestAddition() {
        System.out.println("creation d'une instance");
        somme = 0;
    }

    @Test
    void additionneDeux() {
        somme += 2;
        assertEquals(2, somme);
    }

    @Test
    void additionneTrois() {
        somme += 3;
        assertEquals(3, somme);
    }
}
```





- **Une nouvelle instance de TestAddition est créée avant chaque appel de méthode de test.** ➡ La variable somme vaut toujours 0 avant l'exécution de chaque test.
- C'est le mode PER_METHOD : une instance de la classe de test par (méthode de) test. C'est le mode d'exécution des tests de JUnit 5 par défaut. **L'objectif est d'exécuter chaque test de manière isolée, en évitant les effets de bords liés à l'exécution des autres tests.**
- Il reste possible toutefois de modifier ce mode d'exécution en précisant le mode PER_CLASS : une instance de la classe de test par classe de test.

Cycle de vie par défaut (PER_METHOD)

```
package test;
import static org.junit.Assert.assertEquals;
import org.junit.jupiter.api.Test;
class TestAddition {
    private int somme;

    TestAddition() {
        System.out.println("coucou");
        somme = 0;
    }

    @Test
    void additionneDeux() {
        somme += 2;
        assertEquals(2, somme);
    }

    @Test
    void additionneTrois() {
        somme += 3;
        assertEquals(3, somme);
    }
}
```

contexte

même exécution

```
package test;
import static org.junit.Assert.assertEquals;
import org.junit.jupiter.api.*;
class TestAddition {
    private int somme;

    @BeforeEach
    void init1() {
        System.out.println("coucou");
        somme = 0;
    }

    @Test
    void additionneDeux() {
        somme += 2;
        assertEquals(2, somme);
    }

    @Test
    void additionneTrois() {
        somme += 3;
        assertEquals(3, somme);
    }
}
```

- Chaque test s'exécute dans le même **contexte**. Les tests sont ainsi indépendants.
- En JUnit, on va rendre ce contexte explicite dans une ou plusieurs méthodes annotées par @BeforeEach.

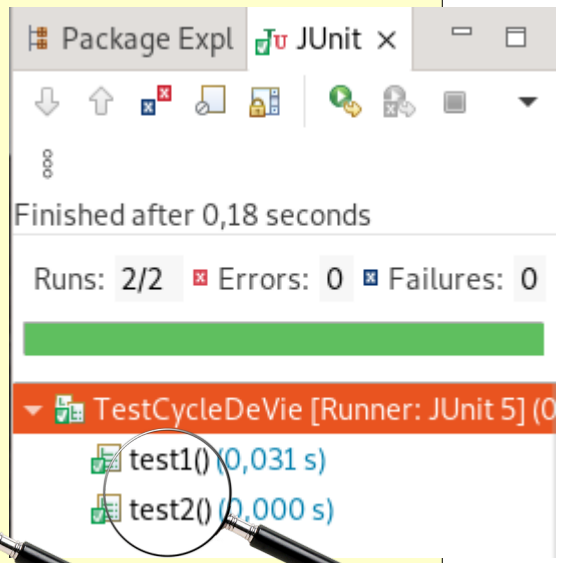
Cycle de vie PER_METHOD ou PER_CLASS

- Il est possible de modifier le cycle de vie de telle sorte que toutes les méthodes soient exécutées sur la même instance de la classe de test. C'est le mode PER_CLASS, obtenu en annotant la classe de test avec **@TestInstance(Lifecycle.PER_CLASS)**. Dans ce mode, une seule instance de la classe de test est créée pour exécuter toutes les méthodes de test :
  - Pour garantir l'indépendance des tests, vous devrez obligatoirement réinitialiser l'état des variables d'instances dans les méthodes @BeforeEach et @AfterEach.
  - Pour des tests qui ne sont pas indépendants, vous pouvez contrôler l'ordre d'exécution des tests avec **@TestMethodOrder**.
  - Les méthodes annotées par @BeforeAll et @AfterAll n'ont plus besoin d'être statiques.
  - Il devient possible d'utiliser des méthodes annotées par @BeforeAll et @AfterAll sur des classes imbriquées (annotées avec @Nested).
- Il existe 2 modes d'exécution : PER_METHOD et PER_CLASS. Le mode d'exécution par défaut est le mode PER_METHOD, qui revient à annoter la classe de test avec @TestInstance(Lifecycle.PER_METHOD).

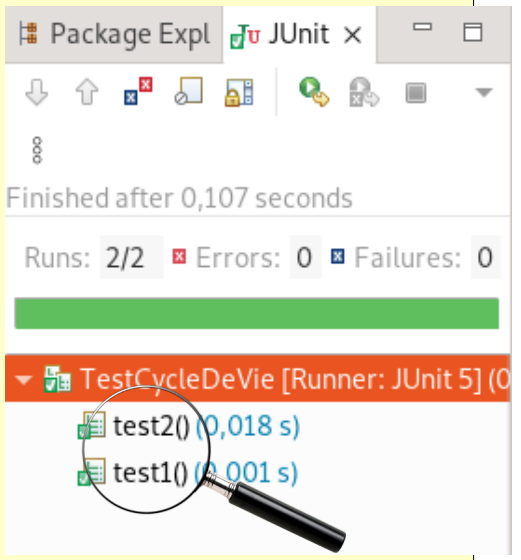
Cycle de vie PER_METHOD ou PER_CLASS

L'annotation @TestInstance est optionnelle si utilisée avec la valeur PER_METHOD)

```
package test;
import static org.junit.Assert.assertEquals;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.TestInstance;

@TestInstance(TestInstance.Lifecycle.PER_METHOD)
class TestCycleDeVie {
    private int a;
    private int b = 0;

    @BeforeEach
    void defContexte() { a = 0;  }

    @Test
    void test2() {
        a += 2;
        assertEquals(2, a);
        b++;
        assertEquals(1, b);
    }

    @Test
    void test1() {
        a += 3;
        assertEquals(3, a);
        b++;
        assertEquals(1, b);
    }
}
```




```
package test;
import static org.junit.Assert.assertEquals;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.TestInstance;
import org.junit.jupiter.api.TestMethodOrder;
import org.junit.jupiter.api.MethodOrderer;
import org.junit.jupiter.api.Order;

@TestInstance(TestInstance.Lifecycle.PER_CLASS)
@TestMethodOrder(MethodOrderer.OrderAnnotation.class)
class TestCycleDeVie {
    private int a;
    private int b = 0;

    @BeforeEach
    void defContexte() { a = 0;  }

    @Test @Order(1)
    void test2() {
        a += 2;
        assertEquals(2, a);
        b++;
        assertEquals(1, b);
    }

    @Test @Order(2)
    void test1() {
        a += 3;
        assertEquals(3, a);
        b++;
        assertEquals(2, b);
    }
}
```



Cycle de vie PER_CLASS

Ce mode d'exécution permet :

- **de réduire le nombre d'instances créées pendant l'exécution des tests**, ce qui peut être utile lorsque la classe contient de nombreuses méthodes de test et que l'instanciation de cette classe est coûteuse
- **de partager l'état de certaines variables entre les méthodes de test** (qui ne sont plus de ce fait indépendantes)
  - Le mode PER_CLASS permet de réaliser des tests séquentiels qui partagent intentionnellement l'état des variables. Le partage d'état entre les tests (le fait qu'un test utilise l'état des variables tel que modifié par un test précédent) est généralement considéré comme une mauvais pratique dans les tests unitaires. Il peut toutefois être utile dans les tests d'intégration. Cela peut permettre d'éviter que les tests ultérieurs répètent les étapes des tests précédents lorsque ces étapes sont lentes.
  - L'état des variables peut évidemment être partagé partiellement : il est possible que certaines variables partagées entre les différentes méthodes de test soient nettoyées entre les tests (grâce aux méthodes @BeforeEach et @AfterEach), tandis que d'autres sont maintenues dans l'état où les tests les mettent, pendant toute la durée du cas de test.
- de partager l'état de certaines variables entre les méthodes non statiques @BeforeAll et @AfterAll (les méthodes @BeforeAll et @AfterAll peuvent être statiques ou non statiques en mode PER_CLASS)
- de déclarer des méthodes @BeforeAll et @AfterAll dans des classes imbriquées (annotées avec @Nested)
- de réaliser des tests paramétrés avec MethodSource dans des classes de test imbriquées : dans ce cas, le mode PER_CLASS est obligatoire.

Cycle de vie PER_CLASS

Ce mode d'exécution doit être utilisé avec prudence, seulement pour réaliser des tests d'intégration.

Les tests unitaires doivent toujours être isolés les uns des autres.

Ecriture des méthodes de test

```
package calcul;

public class Calcul {
   private int a, b;
   public Calcul( int a, int b) {
      this.a = a;
      this.b = b;
   }
   public int getA() { return a; }
   public int getB() { return b; }
   public void setA( int  c) { a = c; }
   public void setB( int  c) { b = c; }
   public int calculer() {
      if ( a == 0 ) return 2 * b;
      if ( b == 0)
         throw new
IllegalStateException();
      return a+b;
   }
}
```

Dans l'IDE Eclise, pour obtenir un squelette de la classe de test :
New > JUnit Test Case

```
package test;
import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.*;
import calcul.Calcul;

class CalculTest {

      @BeforeAll static void setUpBeforeClass()
      throws Exception { }

      @AfterAll static void tearDownAfterClass()
      throws Exception { }

      @BeforeEach void setUp() throws Exception { }

      @AfterEach void tearDown() throws Exception { }

      @Test void testGetA() { fail("Not yet implemented"); }

      @Test void testGetB() { fail("Not yet implemented"); }

      @Test void testSetA() { fail("Not yet implemented"); }

      @Test void testSetB() { fail("Not yet implemented"); }

      @Test void testCalculer() { fail("Not yet implemented"); }

}
```

Ecriture des méthodes de test

```
package calcul;
public class Calcul {
   private int a, b;
   public Calcul( int a, int b) {
      this.a = a;
      this.b = b;
   }
   public int getA() { return a; }
   public int getB() { return b; }
   public void setA( int  c) { a = c; }
   public void setB( int  c) { b = c; }
   public int calculer() {
      if ( a == 0 ) return 2 * b;
      if ( b == 0)
         throw new IllegalStateException();
      return a+b;
   }
}
```

contexte commun à toutes les méthodes de test

```
package test;
import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.*;
import calcul.Calcul;
class CalculTest {
      Calcul c1, c2, c3;
      @BeforeEach void setUp() throws Exception {
            c1 = new Calcul(3,4);
            c2 = new Calcul(0,5);
            c3 = new Calcul(2,0);  }
      @AfterEach void tearDown() throws Exception {
            c1 = c2 = c3 = null ; }
      @Test void testGetA() {
            assertEquals(3,c1.getA()); }
      @Test void testGetB() {
            assertEquals(4,c1.getB());}
      @Test void testSetA() {
            c3.setA(7); assertEquals(7,c3.getA()); }
      @Test void testSetB() {
            c2.setB(2); assertEquals(2,c2.getB()); }
      @Test void testCalculer() { ... }
}
```

Ecriture des méthodes de test

```
package calcul;

public class Calcul {
   private int a, b;
   public Calcul( int a, int b) {
      this.a = a;
      this.b = b;
   }
   public int getA() { return a; }
   public int getB() { return b; }
   public void setA( int  c) { a = c; }
   public void setB( int  c) { b = c; }
   public int calculer() {
      if ( a == 0 ) return 2 * b;
      if ( b == 0)
         throw new IllegalStateException();
      return a+b;
   }
}
```

contexte commun à toutes les méthodes de test

```
package test;
import static org.junit.Assert.*;
import org.junit.*;
import calcul.Calcul;

class CalculTest {
   Calcul c1, c2, c3;
   @BeforeEach void setUp1() throws Exception {
      c1 = new Calcul(3,4);
      c2 = new Calcul(0,5);
      c3 = new Calcul(2,0);  }
   @AfterEach void tearDown1() throws Exception { ... }
   @Test void testGetA() { ... }
   @Test void testGetB()  { ... }
   @Test void testSetA() { ... }
   @Test void testSetB() { ... }
   @Test void testCalculer() {
      assertEquals(7,c1.calculer());
      assertEquals(10,c2.calculer());
         try {
            c3.calculer();
            fail("Exception IllegalStateException non levee");
         } catch (IllegalStateException ise) { }
   }
}
```

Exécution des tests

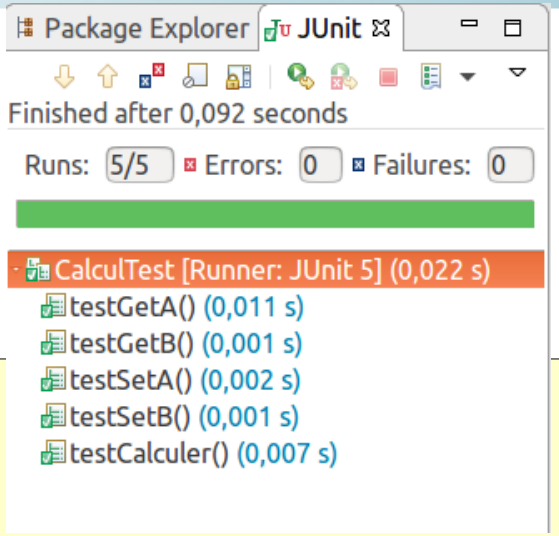
```
package test;
import static org.junit.Assert.*;
import org.junit.*;
import calcul.Calcul;

class CalculTest {
    Calcul c1, c2, c3;

    @BeforeEach void setUp1()
    throws Exception {
        c1 = new Calcul(3,4);
        c2 = new Calcul(0,5);
        c3 = new Calcul(2,0);  }

    @AfterEach void tearDown1()
        throws Exception { ... }

    @Test void testGetA() { ... }

    @Test void testGetB()  { ... }

    @Test void testSetA() { ... }

    @Test void testSetB() { ... }

    @Test void testCalculer() { … }
}
```

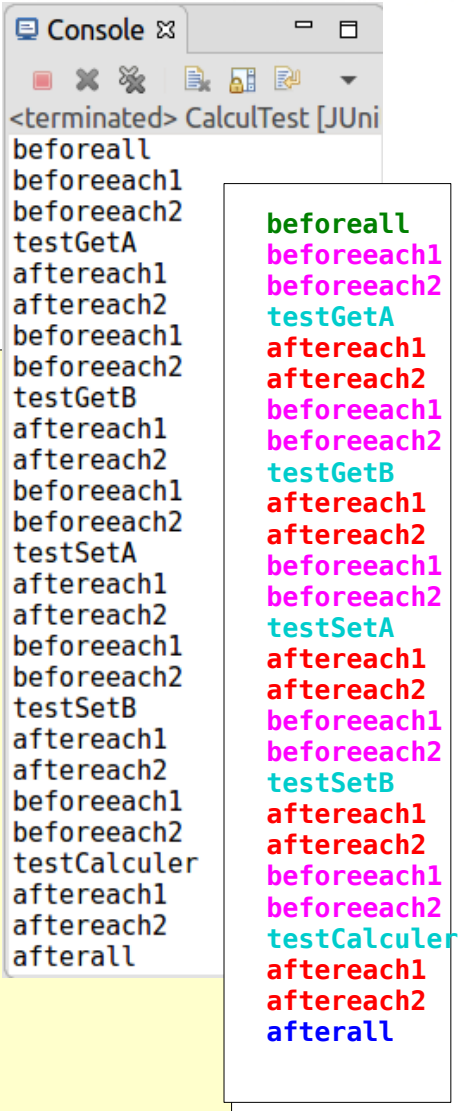
- Les méthodes annotées par **@Test** sont les méthodes de test dont l'ensemble constitue le cas de test.
- Les méthodes annotées par **@BeforeEach** sont exécutées avant CHAQUE méthode de test. Elles permettent de définir un contexte d'exécution commun à toutes les méthodes de test.
- Les méthodes annotées par **@AfterEach** sont exécutées après CHAQUE méthode de test.
- Les méthodes annotées par **@BeforeAll** sont exécutées au lancement du cas de test.
- Les méthodes annotées par **@AfterAll** sont exécutées à la fin du cas de test.
- Les méthodes annotées par **@Disabled** ne sont pas exécutées.

Ordre d'exécution des méthodes de test

L'ordre d'exécution des méthodes annotées par @Test (respectivement @BeforeEach puis @AfterEach) entre elles n'est pas garanti

➡ elles doivent être indépendantes !






```
beforeall
beforeeach1
beforeeach2
testGetA
aftereach1
aftereach2
beforeeach1
beforeeach2
testGetB
aftereach1
aftereach2
beforeeach1
beforeeach2
testSetA
aftereach1
aftereach2
beforeeach1
beforeeach2
testSetB
aftereach1
aftereach2
beforeeach1
beforeeach2
testCalculer
aftereach1
aftereach2
afterall
```

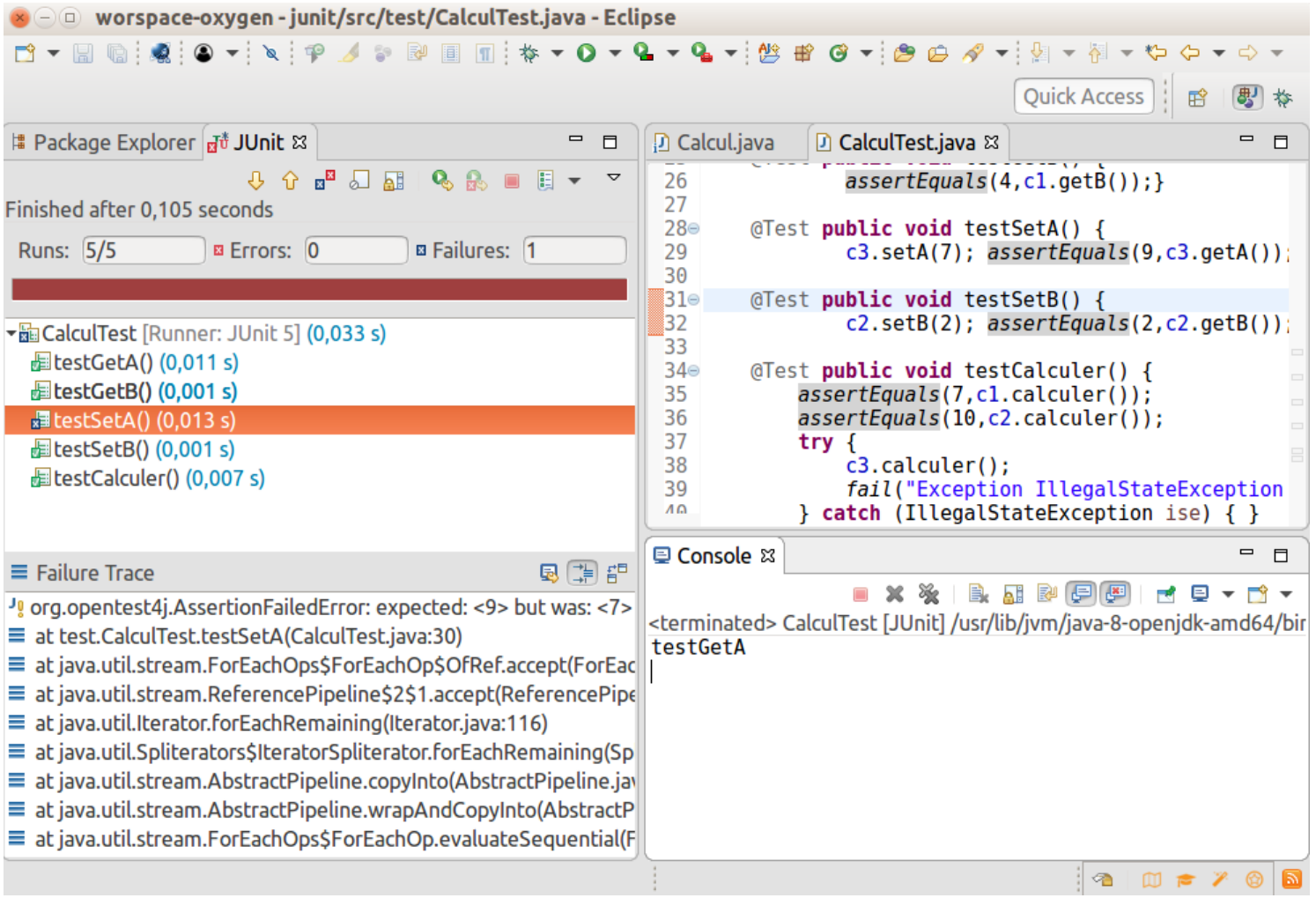
```
package test;
import static org.junit.Assert.*;
import org.junit.*;
public class CalculTest {
    @BeforeAll static void deb() throws Exception { System.out.println("beforeall"); }
    @AfterAll static void fin() throws Exception { System.out.println("afterall"); }
    @BeforeEach void setUp2() throws Exception { System.out.println("beforeeach2");  }
    @BeforeEach void setUp1() throws Exception { System.out.println("beforeeach1"); }
    @AfterEach void tearDown2() throws Exception { System.out.println("aftereach2"); }
    @AfterEach void tearDown1() throws Exception { System.out.println("aftereach1"); }
    @Test void testGetB() { System.out.println("testGetB"); }
    @Test void testSetA() { System.out.println("testSetA");  }
    @Test void testSetB() { System.out.println("testSetB"); }
    @Test void testCalculer() {  System.out.println("testCalculer"); }
    @Test void testGetA() { System.out.println("testGetA");  }
}
```

Echec d'un test : cas d'une assertion non vérifiée

JUnit 5

@Test void testSetA() { c3.setA(7); assertEquals(9,c3.getA()); }

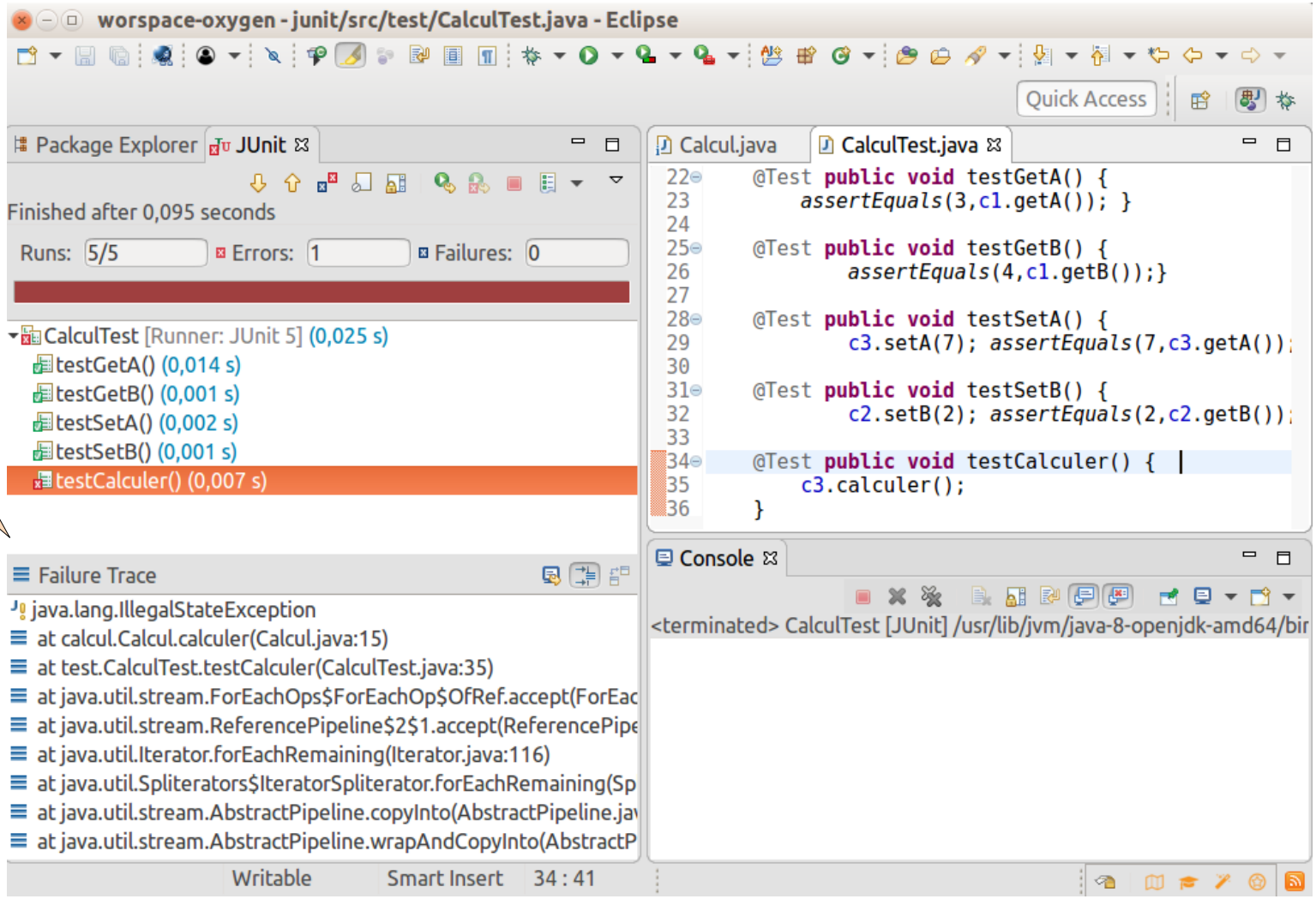
1 failure : l'erreur AssertionFailedError a été lancée par la méthode de test testSetA() : 9 était attendu mais 7 a été obtenu



Echec d'un test : cas d'une exception non récupérée

*public final void testCalculer() { **c3.calculer();** }*

1 error : l'exception IllegalStateException a été lancée par la méthode de test testCalculer()



Tester que la levée d'une exception est un comportement attendu

JUnit 5

CalculTest.java

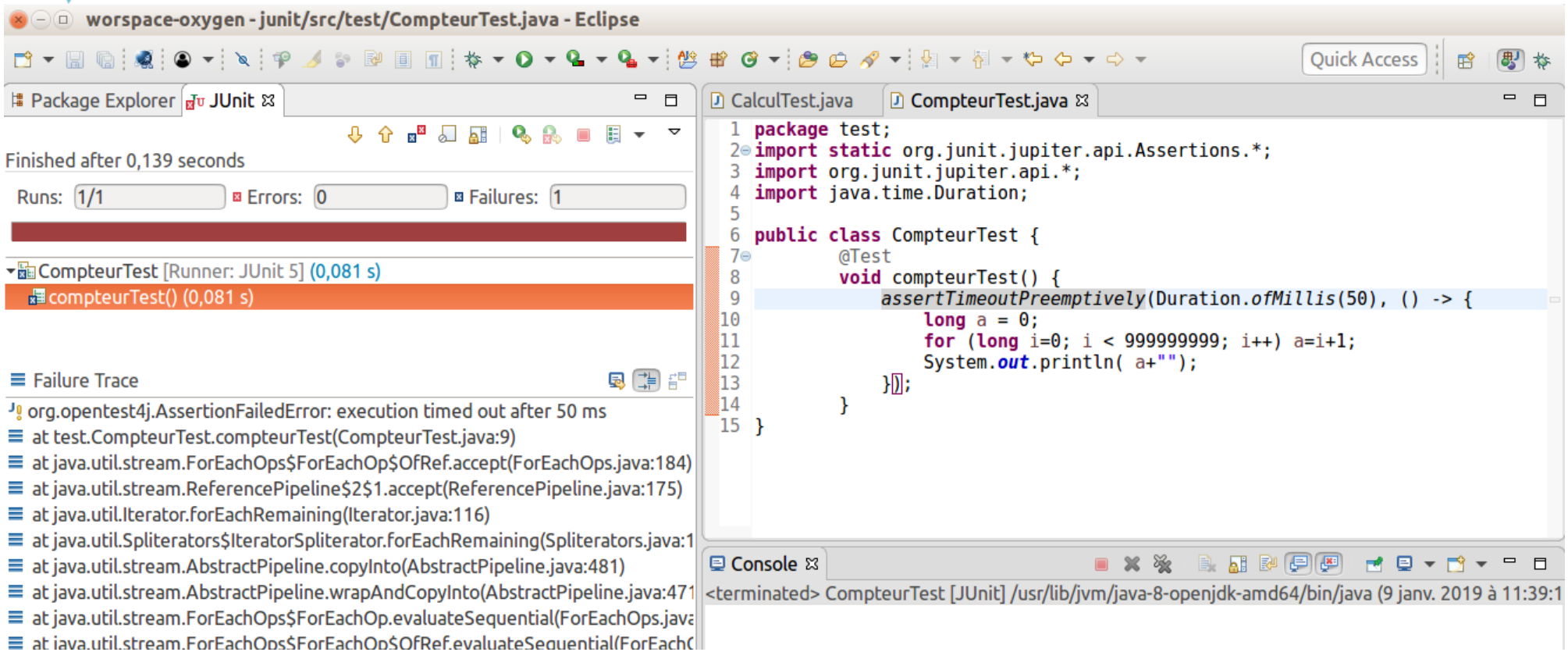
```
28
29        @Test public void testCalculer() {
30                assertEquals(7,c1.calculer());
31                assertEquals(10,c2.calculer());
32                try {
33                    c3.calculer();
34                    fail("Exception IllegalStateException non levee");
35                } catch (IllegalStateException ise) { }
36        }
}
```

CalculTest.java

```
28
29        @Test public void testCalculer() {
30                assertEquals(7,c1.calculer());
31                assertEquals(10,c2.calculer());
32                assertThrows( IllegalStateException.class, () -> { c3.calculer(); });
33        }
}
```

- L'assertion **assertThrows** permet de tester la levée d'une exception.

Limitation du temps d'exécution d'un test



- Les assertions **assertTimeout** et **assertTimeoutPreemptively** permettent de tester que l'exécution d'un ensemble d'instructions dans une méthode de test ne dépasse pas un temps limite. On utilise l'un ou l'autre selon que l'on souhaite attendre ou non la fin d'exécution du traitement dès que le temps limite est dépassé.

Limitation du temps d'exécution d'un test

Plutôt que de tester qu'un ensemble d'instructions à l'intérieur d'une méthode de test ne dépasse pas une durée donnée (comme dans la diapositive précédente), il est possible de vérifier qu'une méthode de test ne dépasse pas une durée donnée. La durée est indiquée à l'aide de l'annotation **@TimeOut**. L'unité de temps est par défaut en secondes mais est configurable.

```
@Test
@TimeOut( value = 100, unit = TimeUnit.MILLISECONDS)
void testAjouter() {
   // échoue si le temps d'excéution dépasse 100 millisecondes
   …
}


@TimeOut(2)
@Test
void testEnlever() {
   // échoue sir le temps d'excéution dépasse 2 secondes
   …
}
```

L'assertion assertLinesMatch

- L'assertion **assertLinesMatch** s'applique aux listes de chaînes de caractères. Elle vérifie que les éléments d'une List<String> correspondent à ceux d'une autre List<String> :
  - soit les éléments sont deux à deux égaux (avec *equals()*)
  - soit leur correspondance est établie au moyen d'une expression régulière (avec *String.matches(String)*),
  - soit des éléments obtenus sont ignorés (grâce au marqueur d'avance rapide).

```
package test;
import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.*;
import java.util.Arrays;

class MonTest {
    @Test
    void test() {
      assertLinesMatch(Arrays.asList("un", "(.*)@(.*)"),    Arrays.asList("un", "test@gmail.com")) ;
      assertLinesMatch(Arrays.asList(".* en Provence"),     Arrays.asList("Aix en Provence")) ;
      assertLinesMatch(Arrays.asList("un", "\\d+", "3"),    Arrays.asList("un", "11", "3"))) ;
      assertLinesMatch(Arrays.asList("oui", "\\w+", "non"),    Arrays.asList("oui", "bof", "non"))) ;
      assertLinesMatch(Arrays.asList("un", "deux", "trois"),    Arrays.asList("un", "deux", "trois"))) ;
    }
}
```

Les expressions régulières utilisées sont celles vérifiées par la méthode matches(String regex) de String (voir les expressions régulières en Java, par exemple : https://www.vogella.com/tutorials/JavaRegularExpressions/article.html)

L'assertion assertLinesMatch

- Dans l'assertion assertLinesMatch , un marqueur d'avance rapide commence et termine par >>, il permet d'ignorer des éléments pendant la comparaison (par exemple ceux dont la valeur change à chaque exécution).

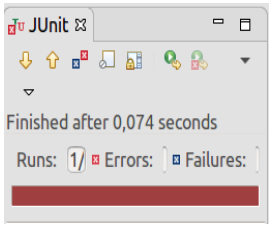
```
package test;
import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.*;
import java.util.Arrays;

class MonTest {
    @Test
    void test() {
      assertLinesMatch( Arrays.asList("un", ">> commentaire sans nombres >>", "six", "sept") ;
                        Arrays.asList("un", "deux", "trois", "quatre", "cinq", "six, "sept") ;
      assertLinesMatch( Arrays.asList("(.*)@(.*)", ">>>>", "(.*)@(.*)"),
          Arrays.asList("truc@yahoo.fr", "un", "deux", "trois", "quatre", "cinq", "test@gmail.com")) ;
      assertLinesMatch(Arrays.asList("a(.*)", ">>2>>", "a(.*)"),
          Arrays.asList("abbbb", "un", "deux", "aaaa")) ;
    }
}
```

Le nombre dans le marqueur d'avance rapide précise le nombre exact d'éléments à ignorer, ici 2.

Groupe d'assertions

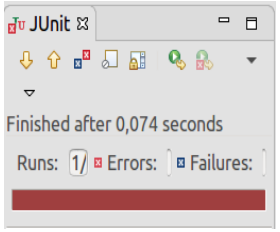
```
package test;
import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.*;
import java.awt.Point;

public class MonTest {
   @Test void test() {
      Point p1 = new Point(10,20);
      assertAll ( "p1",
         () -> assertEquals(11, p1.x, "ERR1"),
         () -> assertEquals(21, p1.y, "ERR2"),
         () -> assertTrue( p1.y == 0, "ERR3") );
   }
}
```

Dans une assertion groupée, toutes les assertions sont exécutées et les échecs sont reportés ensemble.




```
MultipleFailureErrors: p1 (3 failures)
    ERR1 ==> expected : <11> but was: <10>
    ERR2 ==> expected : <21> but was: <20>
    ERR3
```

```
package test;
import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.*;
import java.awt.Point;

public class MonTest {
   @Test void test() {
      Point p2 = new Point(10,20);
      assertAll ( "p2",
       () -> { assertTrue(p2.x == p2.y, "ERR1");
               assertAll ( "coordonnées entières",
                  () -> assertEquals(0, p2.x, "ERR2"),
                  () -> assertEquals(0, p2.y, "ERR3") );
             },
       () -> assertTrue(p2.getX()<5, "ERR4") ) ;
   }
}
```




```
MultipleFailureErrors: p2 (2 failures)
    ERR1
    ERR4
```

Dans un bloc, les assertions sont dépendantes : si une assertion échoue, le code suivant est sauté.

Assertions supplémentaires

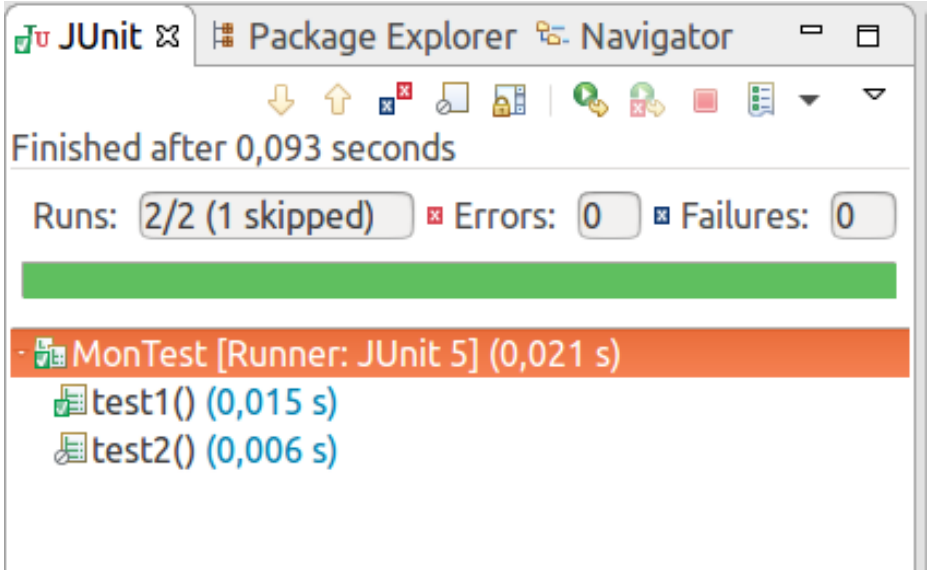
- Les assertions fournies par JUnit Jupiter sont suffisantes pour la plupart des cas de test. Toutefois, d'autres bibliothèques d'assertions (AssertJ, Hamcrest, Truth, etc.) offrent des fonctionnalités supplémentaires plus puissantes comme les "matchers".
- Exemple l'assertion assertThat de Hamcrest :

```
package test;
import static org.hamcrest.CoreMatchers.*;
import static org.hamcrest.MatcherAssert.assertThat;
import static org.hamcrest.core.CombinableMatcher.either;
import static org.hamcrest.core.StringEndsWith.endsWith;
import static org.hamcrest.core.StringContains.containsString;
import org.junit.jupiter.api.Test;
import java.util.*;

class MonTest {
    @Test
    void test() {
        assertThat(2 + 1, equalTo(3));
        assertThat( x, is(3));
        assertThat( 3, is(not(4)));
        assertThat(Arrays.asList("un", "deux", "trois".size(), is(3));
        assertThat(Arrays.asList("un", "deux", "trois", hasItem("deux"));
        assertThat(Arrays.asList("un", "deux", "trois"), hasItems("deux","un"));
        assertThat(Arrays.asList("un", "deux", "trois"), hasItems(endsWith("x"),endsWith("s")));
        assertThat( "colour", either( containsString("color")).or ( containsString("colour")));
        assertThat( "Bonjour", is( anyOf(nullValue(), instanceOf( String.class), equalTo("Salut"))));
        assertThat( "Bonjour", is(not(anyOf(nullValue(),instanceOf(Integer.class), equalTo("Salut")))));
    }
}
```

L'ordre des paramètres obtenu-attendu de assertThat est intuitif (il correspond à "je voudrais que ce que j'obtiens soit égal à ça") alors qu'il est inversé dans assertEquals.

Les suppositions

- Une supposition permet de conditionner l'exécution de tout ou partie d'un test. Si l'évaluation de la supposition échoue, alors le test est interrompu et considéré comme désactivé.
- Les suppositions sont définies dans la classe org.junit.jupiter.api.Assumptions : assumeTrue, assumeFalse, assumingThat.

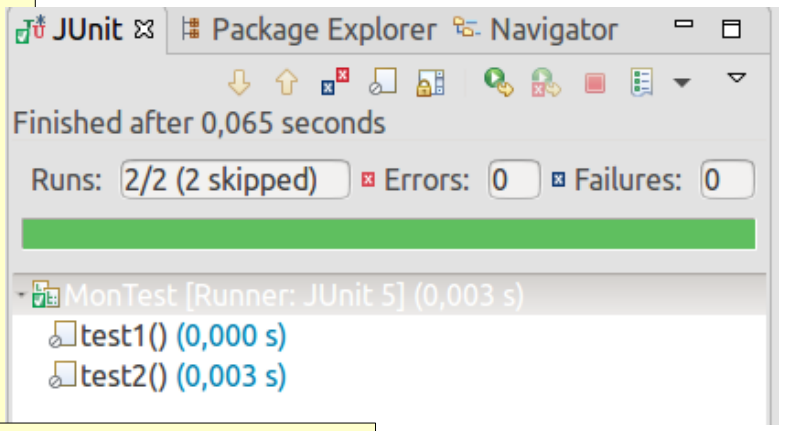



```
package test;
import static org.junit.jupiter.api.Assertions.*;
import static org.junit.jupiter.api.Assumptions.*;
import org.junit.jupiter.api.*;
import java.awt.Point;

class MonTest {
	Point p1, p2;
	@BeforeEach
	void init() {
		p1 = new Point(0,0);
		p2 = null;
	}
	@Test
	void test1() {
		assumeTrue(p1 != null);
		assertEquals(0, p1.x);
	}
	@Test
	void test2() {
		assumeTrue(p2 != null);
		assertEquals(0, p2.x);
	}
}
```

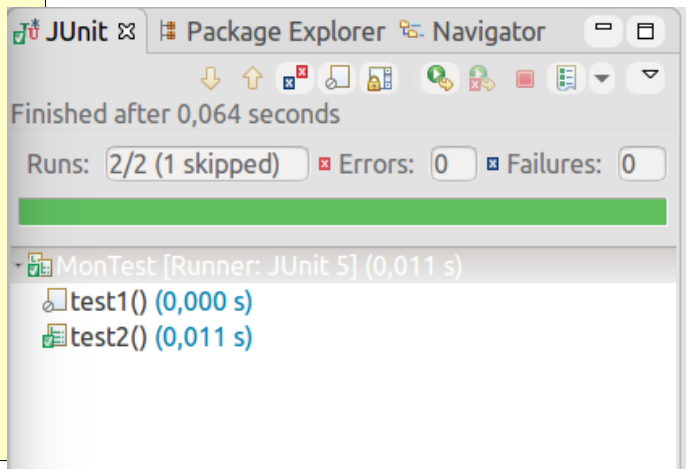
Désactivation des tests

- Une classe entière de test peut être désactivée avec l'annotation **@Disabled**.
- @Disabled permet aussi de désactiver les méthodes de test individuellement.
- Cette fonctionnalité est utilisée pendant la mise au point des programmes.

```
package test;
import org.junit.jupiter.api.*;
import org.junit.jupiter.api.Assertions.fail;

@Disabled
class MonTest {
        @Test
        void test1() {  fail("A faire");}
        @Test
        void test2() { fail("A faire");}
}
```



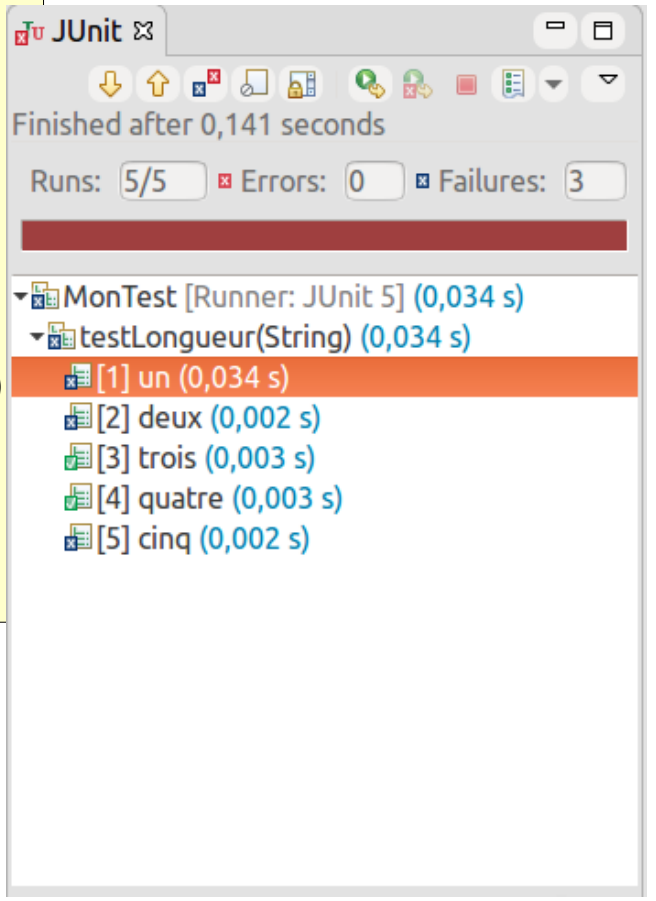

```
package test;
import org.junit.jupiter.api.Assertions.fail;
import org.junit.jupiter.api.*;

class MonTest {
        @Test @Disabled
        void test1() { fail("A faire"); }
        @Test
        void test2() { }
}
```



Test paramétré

- Un test paramétré permet d'exécuter plusieurs fois un cas de test avec des valeurs différentes. Le test doit être annoté avec **@ParameterizedTest**. Il faut déclarer une source qui fournira les paramètres de chaque exécution. La méthode de test d'un test paramétré a nécessairement des paramètres.

```
package test;
import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.ValueSource;

public class MonTest {
	@ParameterizedTest
	@ValueSource(strings = { "un", "deux", "trois", "quatre", "cinq" })
	void testLongueur(String candidat) {
	    assertTrue(candidat.length() > 4);
	}
}
```

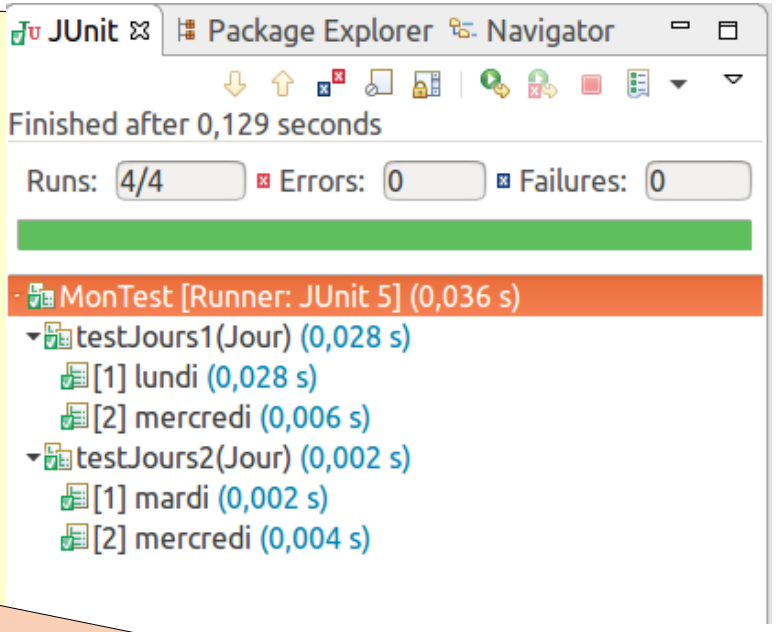


La source d'arguments la plus simple est **@ValueSource**. Elle permet de spécifier les valeurs d'**un argument unique** (ici *candidat*) dans un tableau de littéraux de types primitifs ou de type String ou Class.

Test paramétré

- Les paramètres des tests sont fournis grâce à une source de données. JUnit Jupiter propose plusieurs annotations pour différents types de source dans le package org.junit.jupiter.params.provider
  - @ValueSource : données sous forme d'un tableau de chaînes de caractères ou de primitifs
  - @EnumSource : données sous forme d'une énumération
  - @MethodSource : données fournies par une méthode
  - @CsvSource : données sous forme de chaînes de caractères séparées par des virgules
  - @CsvSourceFile : données sous forme d'un ou plusieurs fichiers CSV
  - @ArgumentsSource : données sous forme d'une méthode d'un ArgumentProvider

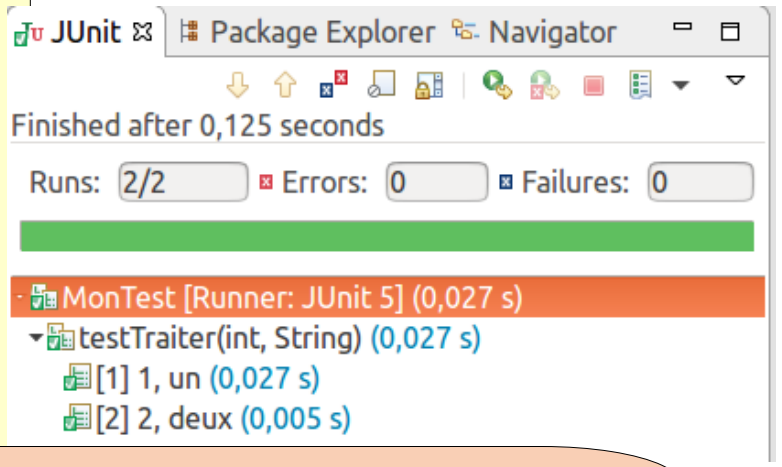
```
package test;
import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.*;
import org.junit.jupiter.params.provider.EnumSource.Mode;
public class MonTest {
        enum Jour { lundi,mardi,mercredi,jeudi,vendredi,samedi,
                        dimanche }
        @ParameterizedTest  @EnumSource(value=Jour.class,
                                                names= {"lundi","mercredi"})
        void testJours1(Jour j) { assertNotNull(j); }
        @ParameterizedTest @EnumSource(value=Jour.class,
                        mode=Mode.MATCH_ALL, names = {"^m.+$"})
        void testJours2(Jour j) { assertNotNull(j); }
}
```



@EnumSource est une autre source d'**argument unique**. Les valeurs de l'argument sont les constantes de l'énumération.

Test paramétré

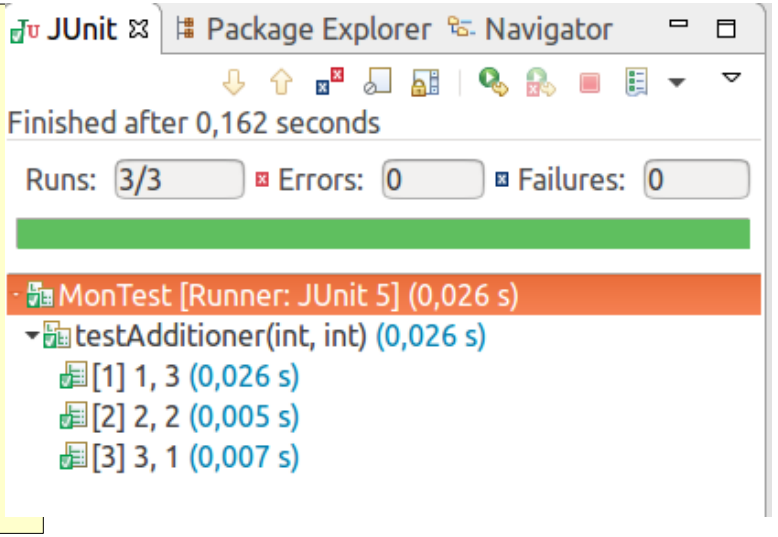
```java
package test;
import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.*;
import java.util.*;
public class MonTest {
	static List<Object[]> fournirDonnees() {
		return Arrays.asList(new Object[][]{{1,"un"}, {2,"deux"}} );
	}
	@ParameterizedTest
	@MethodSource("fournirDonnees")
	void testTraiter(int index, String element) {
		assertTrue(index > 0);
		assertTrue(element.length() > 1);
	}
}
```



La méthode appelée par @MethodSource doit renvoyer un type de collection, qui peut être Stream, Iterable, Iterator ou un tableau. Elle doit être statique et peut être privée.

Ces méthodes de test ont plusieurs arguments. @ValueSource et @EnumSource ne peuvent donc pas être utilisées.

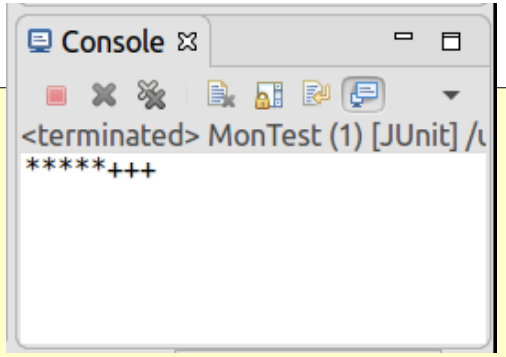
```java
package test;
import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.*;
public class MonTest {
	@ParameterizedTest
	@CsvSource({ "1, 3", "2, 2", "3, 1" })
	void testAdditioner(int a, int b) {
		assertEquals(4, a + b);
	}
}
```

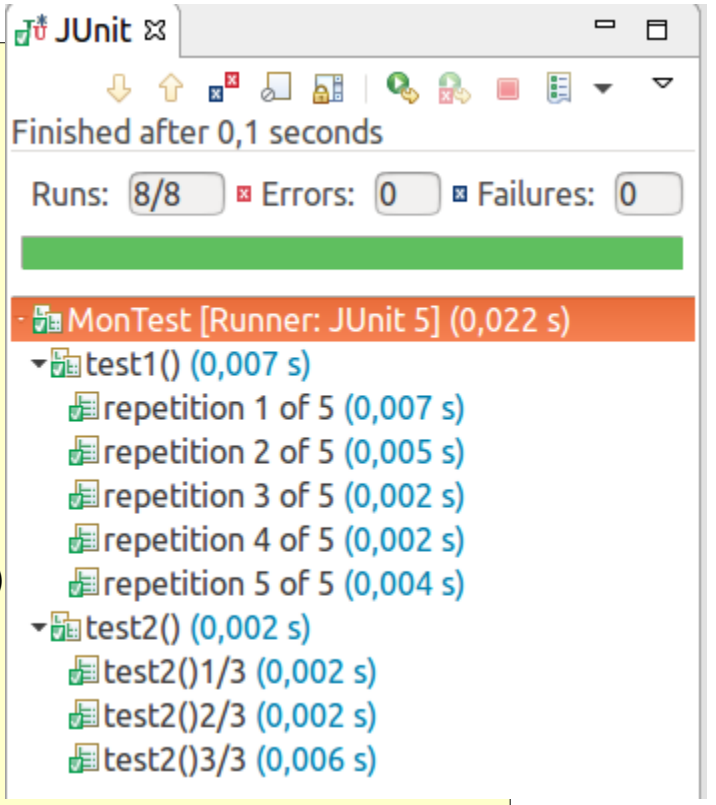


Test répété

- Il est possible de répéter un test plusieurs fois en utilisant l'annotation **@RepeatedTest**. En plus de préciser le nombre de répétitions, un nom peut être affiché à chaque répétition ; ce nom peut être composé du nom de la méthode de test, du numéro courant de répétition, du nombre total de répétitions.

```
package test;
import org.junit.jupiter.api.RepeatedTest;

public class MonTest {
	@RepeatedTest(5)
	void test1() {
	   System.out.print("*");
	}
	@RepeatedTest(value=3,
	      name="{displayName}{currentRepetition}/{totalRepetitions}")
	void test2() {
	   System.out.print("+");
	}
}
```

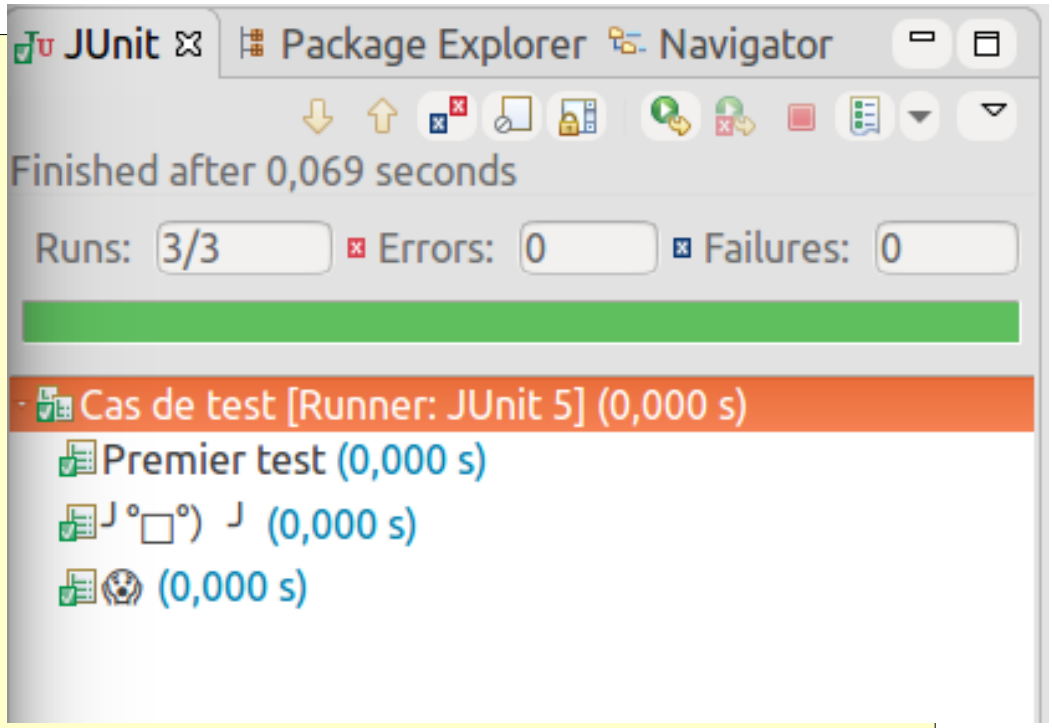




Affichage du nom du test

- Les classes et les méthodes de test peuvent déclarer, avec l'annotation **@DisplayName**, des noms spécifiques (avec des espaces, des caractères spéciaux et des emojis) qui sont affichés par le lanceur de tests et sur les rapports de test

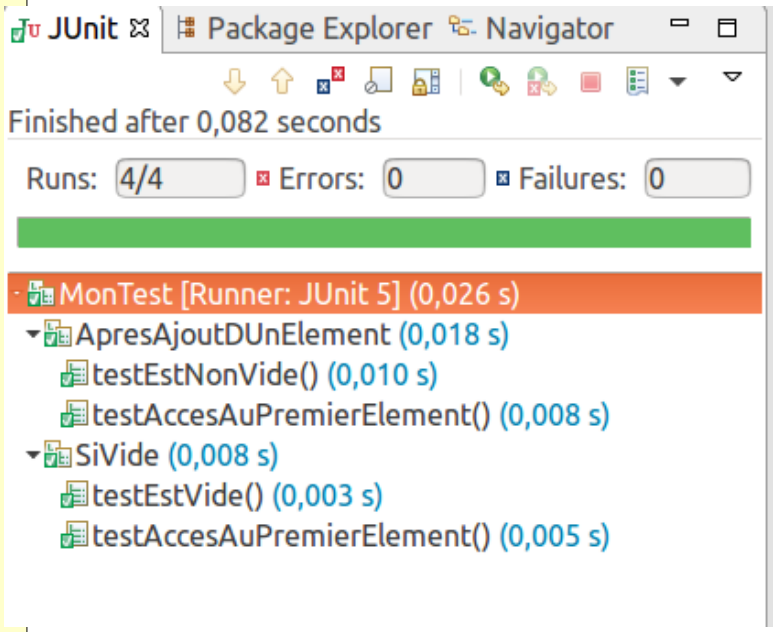
```
package test;
import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.api.Test;

@DisplayName("Cas de test")
public class MonTest {
	@Test
	@DisplayName("Premier test")
	void test1() {  }
	@Test
	@DisplayName("ノ °□°) ノ")
	void test2() { }
	@Test
	@DisplayName("  ")
	void test3() { }
}
```



Classe de test imbriquée

- Les classes de test imbriquées donnent la possibilité d'exprimer des relations entre plusieurs groupes de tests. Une classe imbriquée est une classe interne (non statique) déclarée avec l'annotation **@Nested**.

```
package test;
import org.junit.jupiter.api.*;
import static org.junit.jupiter.api.Assertions.*;
import java.util.ArrayList;
public class MonTest {
   ArrayList liste;

   @Nested
   class SiVide {
      @BeforeEach
      void testCreerVide() { liste = new ArrayList(); }
      @Test
      void testEstVide() { assertTrue(liste.isEmpty()); }
      @Test
       void testAccesAuPremierElement() {
         assertThrows(indexOutOfBoundsException.class, () -> liste.get(0)); }
   }

   @Nested
   class ApresAjoutDUnElement {
      @BeforeEach
      void testCreer() { liste = new ArrayList(); liste.add(1); }
      @Test
      void testEstNonVide() { assertFalse(liste.isEmpty()); }
      @Test
       void testAccesAuPremierElement() { assertEquals(1, liste.get(0));  }
   }
}
```



Suite de tests

- Une **suite de tests** est une agrégation de plusieurs classes de test provenant de différents packages, qui pourront être exécutées ensemble.

```
package test;
import org.junit.platform.suite.api.SelectClasses;
import org.junit.platform.suite.api.Suite;

@SelectClasses({EtudiantTest.class,PersonneTest.class})
@Suite
public class SuiteDeTests {
   // exécute toutes les méthodes de test
   // des classes EtudiantTest et PersonneTest
}
```

```
package suite;
import org.junit.platform.suite.api.SelectPackages;
import org.junit.platform.suite.api.Suite;

@SelectPackages({"testU","testI"})
@Suite
public class SuiteDeTests {
   // exécute toutes les méthodes de test
   // des classes de test des packages testU et testI
}
```

Suite de tests

- D'autres annotations que **@SelectClasses** et **@SelectPackages** permettent d'inclure ou d'exclure des tests dans les suites : **@IncludePackages**, **@ExcludePackages**, **@IncludeClassNamePatterns**, **@ExcludeClassNamePatterns**, **@IncludeTags**, **@ExcludeTags**.

```
@SelectPackages("test")
@ExcludePackages("test.unit")
@Suite
public class SuiteDeTests {
   // exécute les méthodes de test du package
   // test (et de ses sous-packages) en excluant
   // celles du sous-package test.unit
}
```

```
@Suite
@SelectPackages("test")
@IncludeTags("production")
public class SuiteDeTests {
   // exécute les méthodes de test du package
   // test (et ses sous-packages) marquées avec
   // le tag production
}
```

```
@SelectPackages("exo3.test")
@IncludeClassNamePatterns({"^.*ATests?$"})
@Suite
public class SuiteDeTests {
   // exécute toutes les méthodes de test des classes de
   // test du package exo3.test (et ses sous-packages)
   // dont les noms se terminent par ATest ou Atests
}
```

Il faut au préalable avoir balisé certaines méthodes de test ou certaines classes de test avec l'annotation @Tag.

L'utilisation conjointe de @Tag pour le balisage et de @includeTags et @ExcludeTags pour le filtrage permet d'obtenir un filtrage plus spécifique que les filtrages par classe ou par package.

Classe et méthode de test JUnit5

- Une classe de test est une classe qui contient au moins une méthode de test. Elle n'a pas besoin d'être publique. Comme toute classe, une classe de test peut hériter d'une autre classe ou implémenter une interface.
- Une méthode de test est une méthode d'instance annotée par @Test, @RepeatedTest, @ParameterizedTest. Une méthode de test n'a pas besoin d'être publique.
- Les méthodes annotées par @Test, @ParameterizedTest, @RepeatedTest, @BeforeAll, @AfterAll, @BeforeEach ou @AfterEach ne doivent pas renvoyer de valeur.

Tests communs à plusieurs tests

- Pour factoriser les éléments communs à plusieurs tests
  - d'une même classe de test (contexte commun aux différents tests)

    Il s'agit d'initialiser des instances de la classe à tester dans les méthodes annotées par @BeforeEach pour pouvoir tester les différents cas possibles, en définissant un contexte commun à tous les tests ( diapositives n°20 & 21). Ceci garantit l'indépendance des différents tests de la classe.

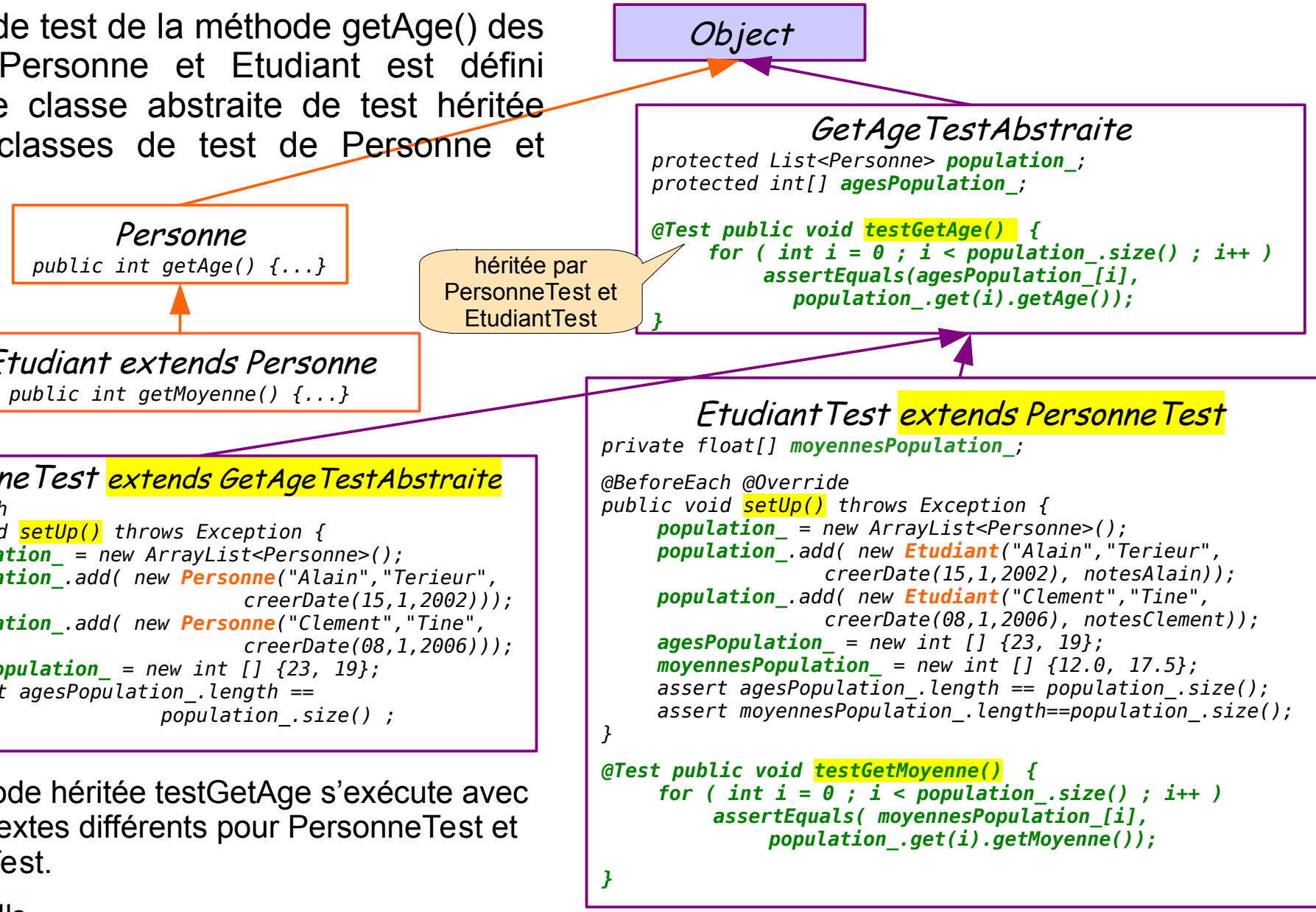
  - de plusieurs classes de test (tests communs à différentes classes)

    Pour tester des classes dont l'une est sous-classe de l'autre ou qui implémentent la même interface, il s'agit de créer une classe abstraite de test avec une variable d'instance du type le plus général, qui regroupe les tests communs, et une classe fille de test pour chaque classe à tester, chacune de ces classes filles ayant une méthode @BeforeEach différente pour initialiser la variable d'instance ( diapositives suivantes n°45 & 46).

Héritage de classes de test

exo5 du TP2

Le code de test de la méthode getAge() des classes Personne et Etudiant est défini dans une classe abstraite de test héritée par les classes de test de Personne et Etudiant.

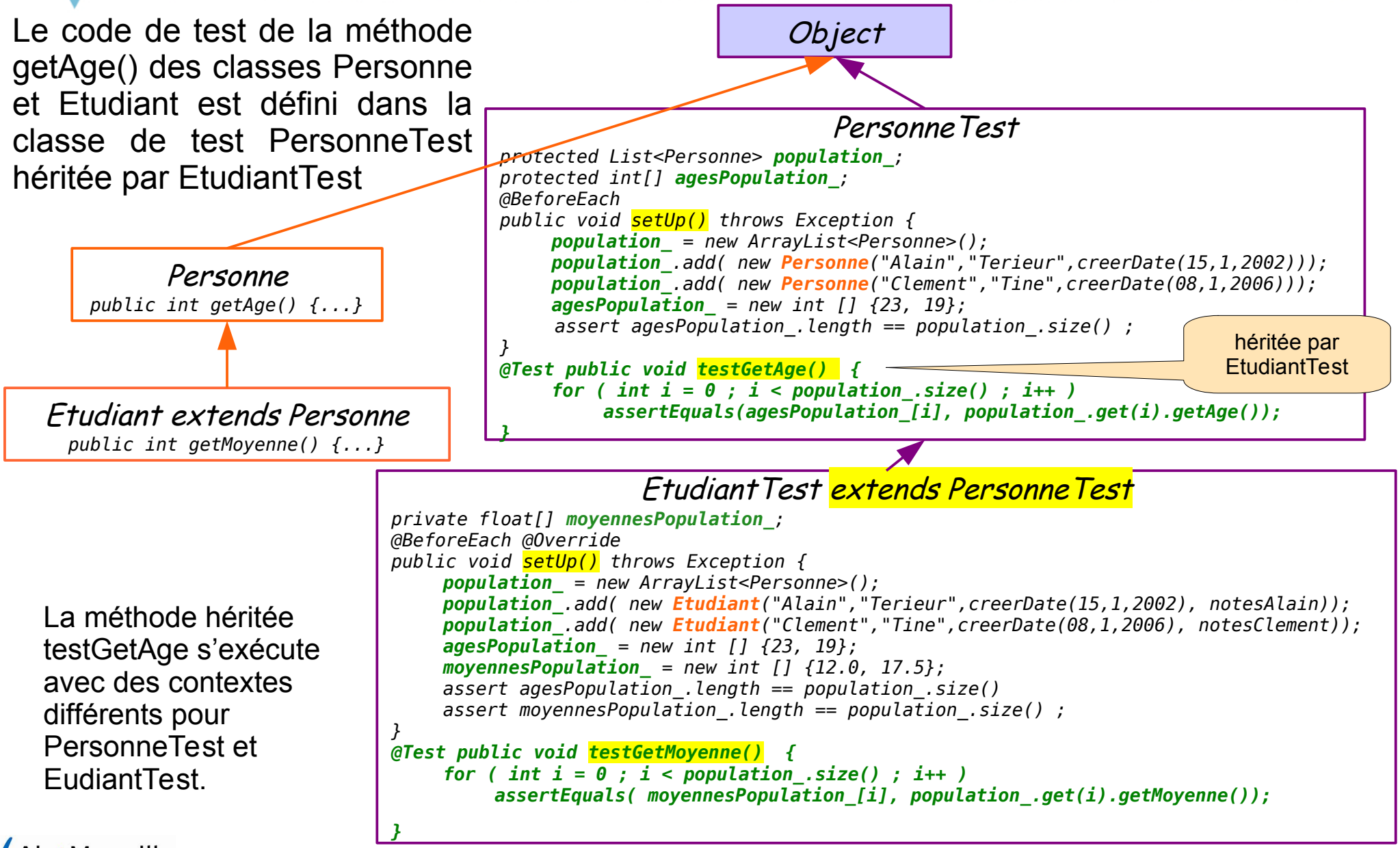


La méthode héritée testGetAge s'exécute avec des contextes différents pour PersonneTest et EudiantTest.

Héritage de classes de test

exo5 du TP2

Le code de test de la méthode getAge() des classes Personne et Etudiant est défini dans la classe de test PersonneTest héritée par EtudiantTest



La méthode héritée testGetAge s'exécute avec des contextes différents pour PersonneTest et EudiantTest.

Comment tester System.out.println() ?

```
package test;
import static org.junit.jupiter.api.Assertions.assertEquals;
import java.io.ByteArrayOutputStream;
import java.io.OutputStream;
import java.io.PrintStream;
import org.junit.jupiter.api.AfterEach;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;

public class MonTest {
    // flux de redirection de la sortie
    private OutputStream sortie;
    // variable utilisée par la methode dernierPrint
    private int indice;

    @BeforeEach
    public void setUp() throws Exception {
        // Redirection du flux de sortie standard vers sortie
        sortie = new ByteArrayOutputStream();
        System.setOut( new PrintStream(sortie));
        indice = 0;
    }

    @AfterEach
    public void tearDown() throws Exception {
        /* Rétablissement de la sortie standard  comme flux de
        sortie */
        System.setOut( System.out);
    }
```

```
    /* Renvoie le texte du flux de sortie à partir d'un certain indice
    et positionne l'indice à la fin du texte de sorte que le prochain
    appel de dernierPrint() renverra le texte généré par le prochain
    appel à System.out.print ou println */
    private String dernierPrint() {
        String resultat =
                sortie.toString().substring(indice);
        indice = sortie.toString().length();
        return resultat ;
    }

    @Test
    void testSuccesAppelPrintln_apresRedirectionDeSystemOut() {
        System.out.print("Hé !");
        assertEquals("Hé !", sortie.toString());

        System.out.println("Ho !");
        assertEquals("Hé !Ho !\n", sortie.toString());

        assertEquals("Hé !Ho !\n", dernierPrint());
        System.out.println("Ca va ?");
        assertEquals("Ca va ?\n", dernierPrint());
    }
}
```

exo5 du TP2