

Test Driven Development en Java

Objectifs

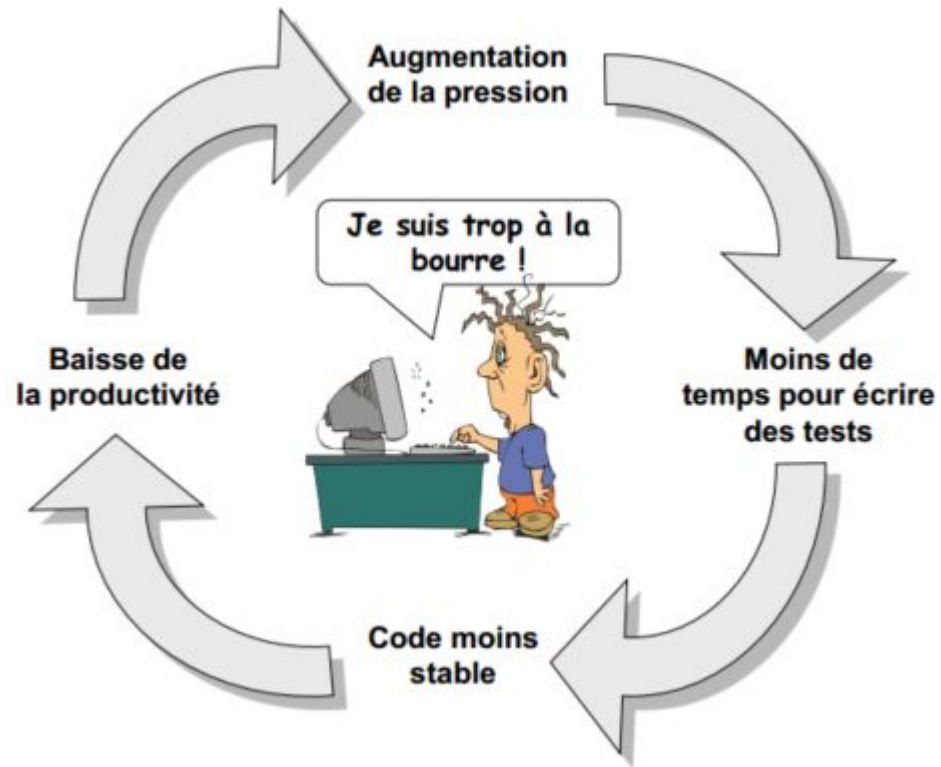
- Maîtriser la démarche et la mise en œuvre du Test Driven Development
- Intégrer les tests dans le cycle de développement d'une application Java
- Prendre en main les principaux outils de tests et d'intégration continue

Plan

1. Introduction
2. Rappels des Tests en général
3. Junit
4. Maven et Junit
5. Les Mock

1. Introduction

- Tous les programmeurs savent qu'ils doivent écrire des tests mais peu le font...



Rappels des Tests en général

- Objectifs :
 - Valider le comportement du code
 - Empêcher les régressions
 - Valider les impacts de modifications
 - Faciliter la maintenance et l'évolution
 - Il existe même des techniques de développement piloté par les tests (Test Driven Development)

Les Tests : objectifs (2)

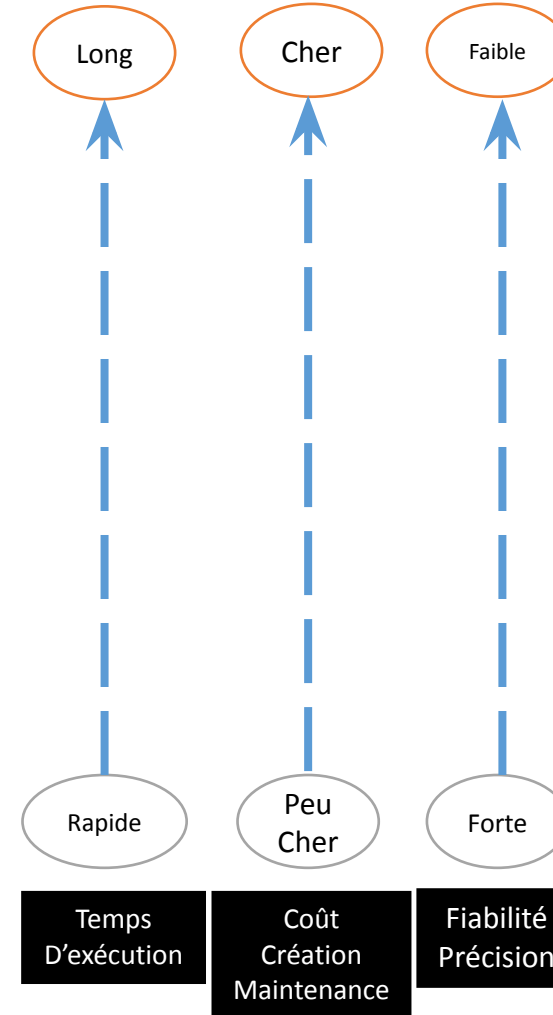
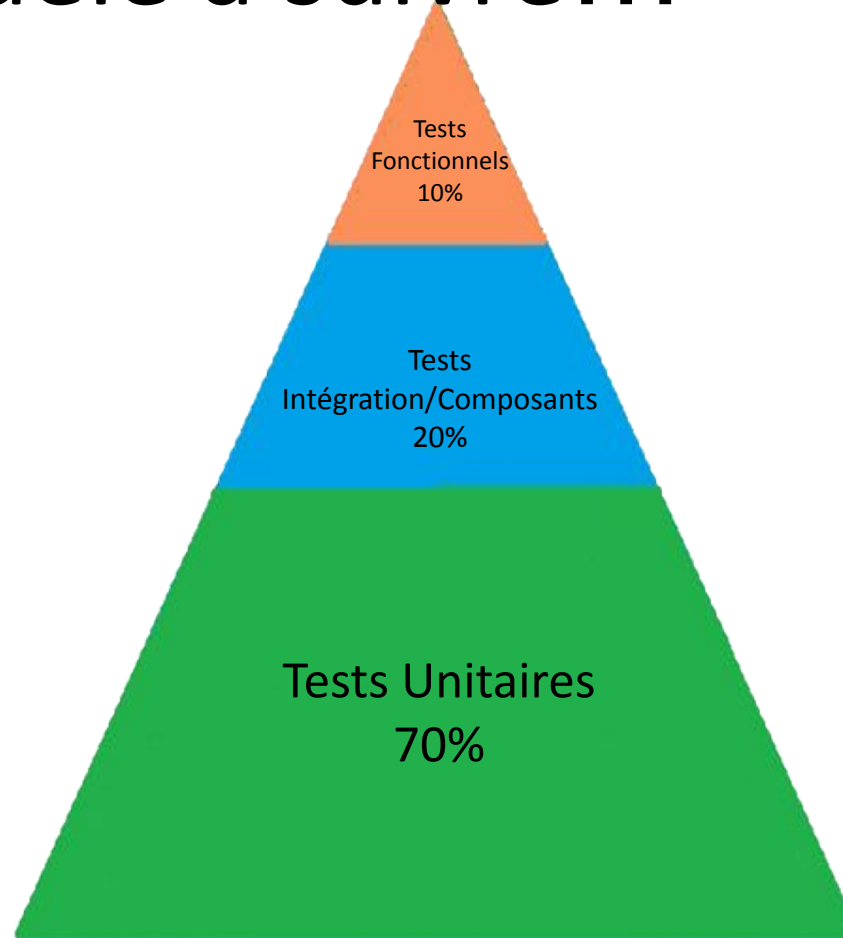
- Dans l'environnement :
 - Les tests doivent être **représentatifs** des **fonctionnalités** demandées.
 - Ils peuvent ainsi se lancer **de manière automatique** avant chaque **build** ou **release**
- Les tests ont une **valeur monétaire**
 - Ils **prennent du temps** à écrire (33% à 50% d'overhead sur un développement)
 - Ils doivent **permettre d'en gagner** sur la maintenance et l'évolutivité du code (33% à 50% de gain)
 - Ils permettent aussi de **garder une bonne compréhension** du code \Leftrightarrow **facilitent la passation** de connaissance

Des niveaux et des phases

- **Tests unitaires** : ciblent le niveau opération. Ils prennent **chaque méthode** de chaque objet et vérifient que **tout fonctionne correctement**.
- **Tests d'intégration** : ciblent le niveau module. Ils vérifient les **interactions entre les composants** unitaires, les **différents modules** et **dans leur environnement d'exploitation définitif**. Ils permettent de mettre en évidence des problèmes d'interfaces.
- **Tests fonctionnels** : ciblent le niveau de granularité du cas d'utilisation ou scénario. Ils permettent de vérifier la **conformité de l'application** développée avec **le cahier des charges** initial. Ils sont donc basés sur les **spécifications fonctionnelles** et techniques.
- **Tests de non régression** : permettent de vérifier que **les modifications** du code source n'ont pas **altérées le fonctionnement** de l'application.
- **Tests métiers** : cible le niveau processus et consiste à simuler l'exécution d'un **processus** avec l'application / le logiciel.

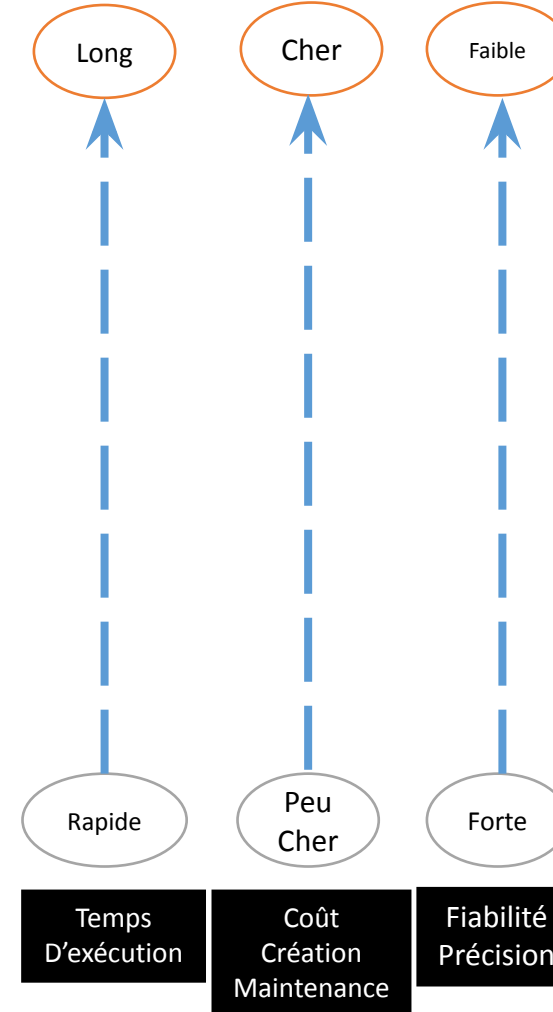
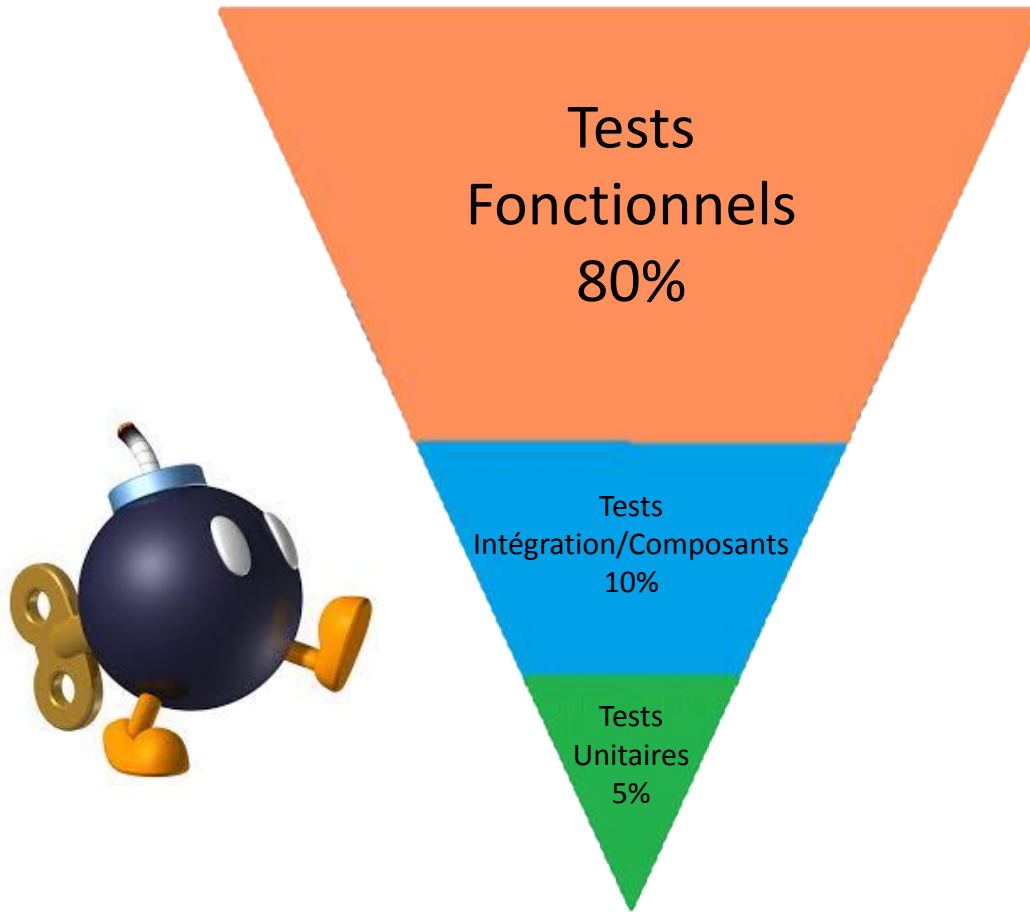
Des niveaux et des phases

Le modèle à suivre...



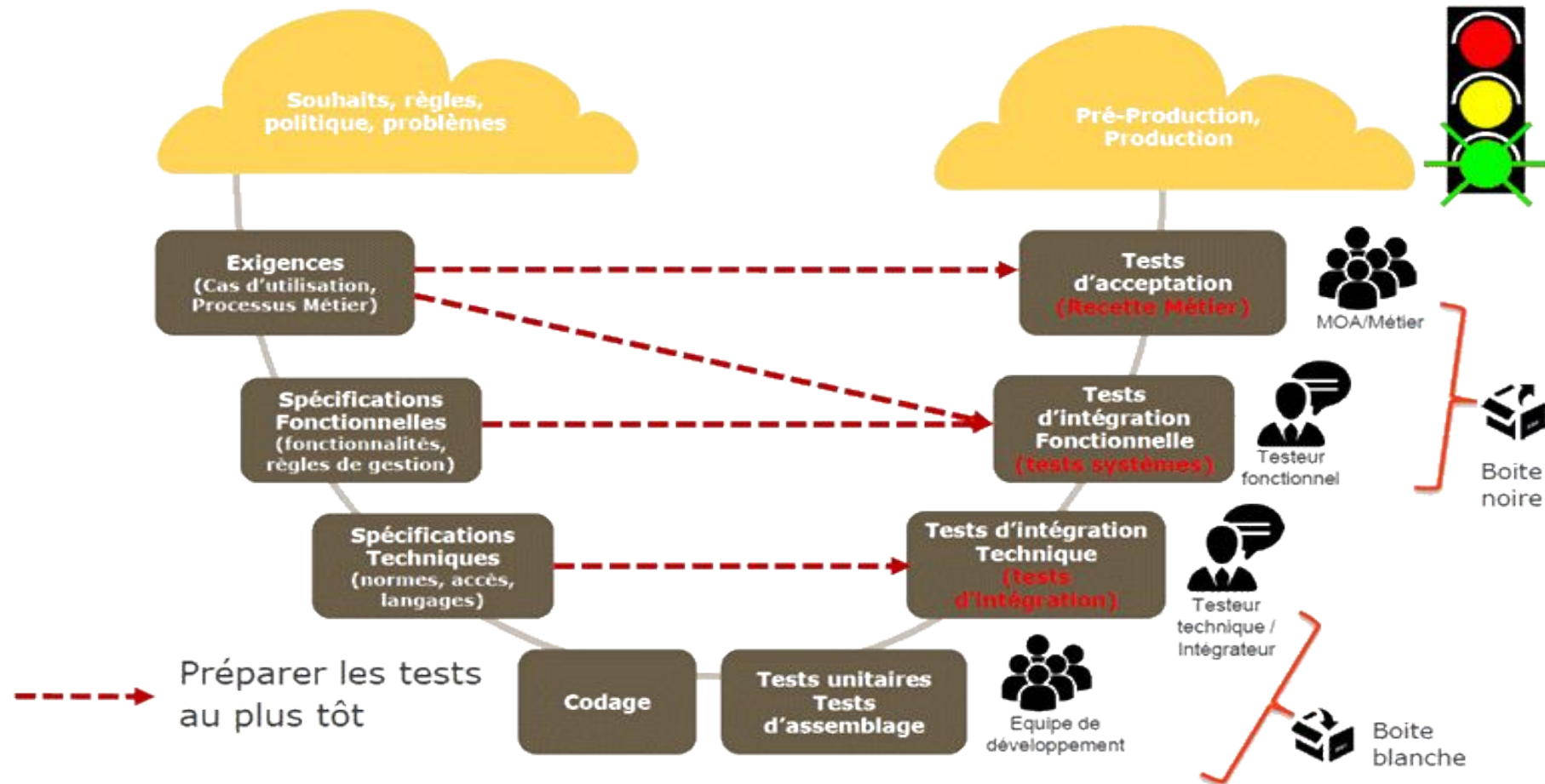
Des niveaux et des phases

... Et CELUI à éviter



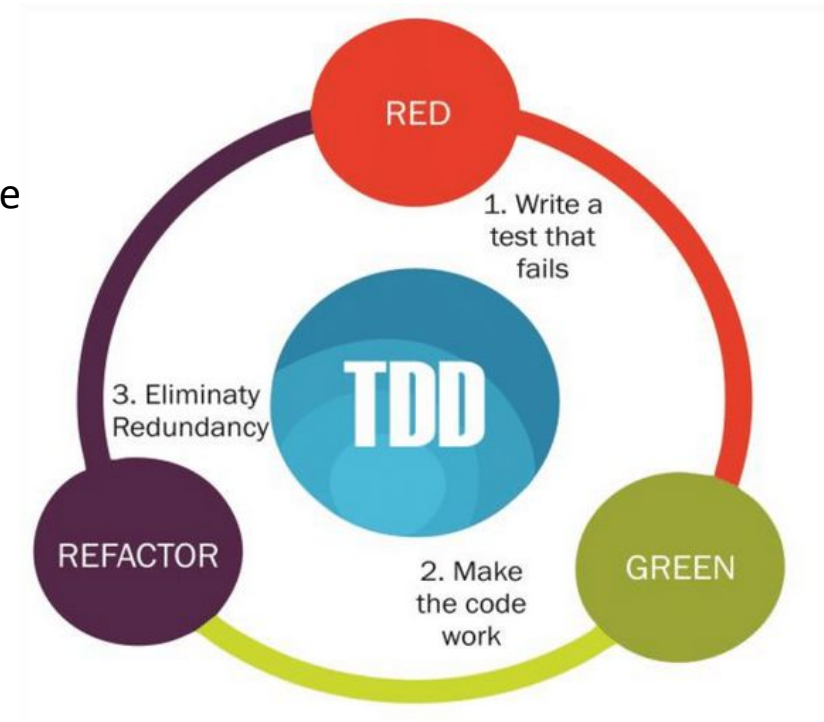
AUTOMATISATION DES TESTS

Cycle de Vie et Niveaux de test



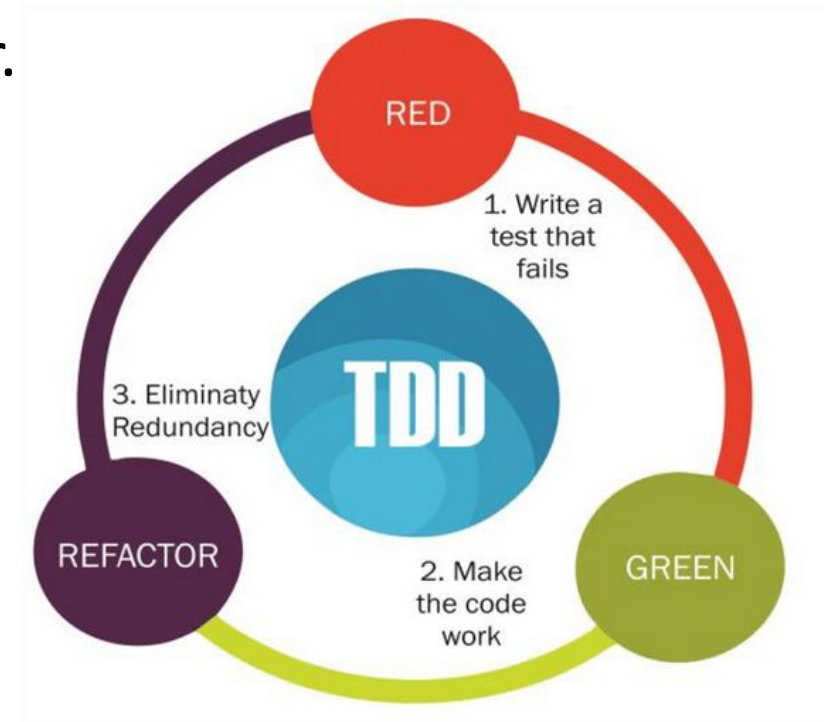
2. Développement piloté par les tests TDD

- Il existe même des techniques de développement pilotées par les tests (Test Driven Development TDD).
- Constatations
 - Les tests d'un logiciel sont généralement **automatisables**
 - Le temps nécessaire à la création des tests et à l'automatisation e
 - Une fois fait : **moins de temps consacré aux tests**
 - **Moins de défauts** dans le code
 - **Moins de régressions**
- Ecrire les tests en premier
 - Position a priori paradoxale, mais :
 - Elimination du dilemme habituel en fin de projet
 - Code plus **facilement testable**
 - **Meilleure conception**
 - Tests fonctionnels : forme de spécialisation
 - Tests unitaires : forme de conception détaillée



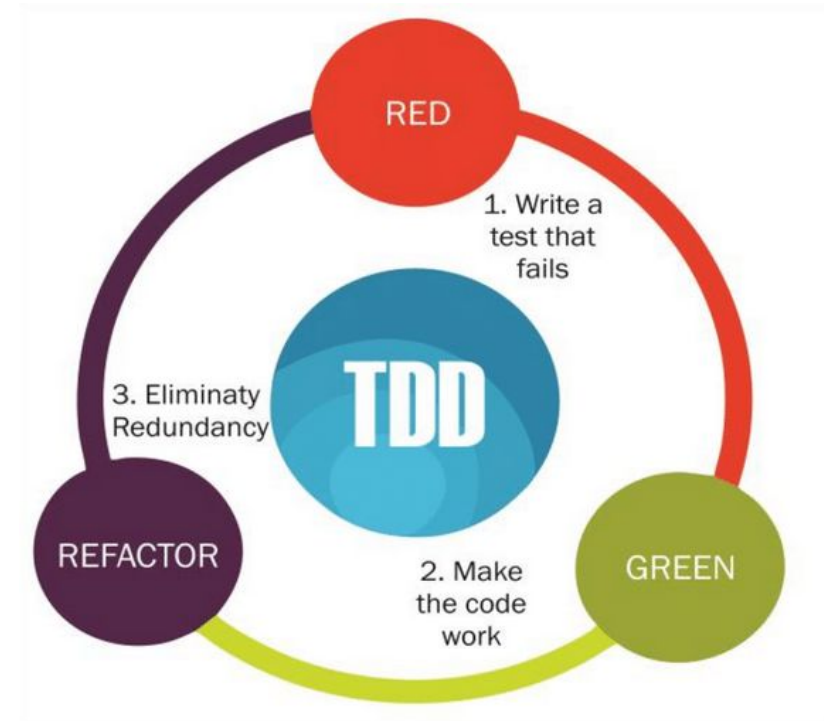
2. Développement piloté par les tests TDD

- Tests unitaires
 - Écrire les tests avant de faire le code à tester.
 - Plusieurs rôles :
 - **Rythmer** la programmation
 - **Guider** la conception
 - **Documenter** le code produit



2. Développement piloté par les tests TDD

- Les étapes des tests unitaires :
 - Identifier une **sous-partie** du problème
 - Ecrire un test indiquant le **résultat si le problème est résolu**
 - Exécuter le test... qui doit **échouer**
 - **Ecrire** le code
 - Exécuter le test pour **vérifier que le code fonctionne** correctement
- Lors d'une modification de conception :
 - **Modifier le test** concerné
 - Exécuter le test... qui doit **échouer**.
 - **Modifier le code**
 - Exécuter le test pour **vérifier que le code fonctionne** correctement



L'intégration continue

- L'intégration continue est un ensemble de pratiques utilisées en génie logiciel consistant à **vérifier à chaque modification** de code sources que le **résultat** des modifications ne produit **pas de régression** dans l'application développée.
- L'objectif est de **détecter les problèmes** d'intégration **au plus tôt** lors du développement.
- L'intégration continue permet **d'automatiser** l'exécution **des suites de tests** et de **voir l'évolution** du développement du logiciel.

Les outils de tests de base

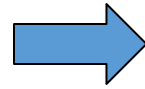
- Pour chaque problématique de tests on peut trouver un ou plusieurs outils de base, un ou plusieurs frameworks
- Junit : pour les tests unitaires
 - Intégrer dans les IDE, la référence aujourd'hui
- Mockito : pour les bouchons
 - Surcharger le comportement de certains éléments sur lesquels on a pas la main
- JMeter : permet d'enregistrer des comportements et de les rejouer plusieurs fois en même temps
 - Idéal pour les tests de charge Web ou d'IHM
- Selenium : permet de tester des scénarios d'IHM
 - Idéal pour les tests fonctionnels

Définition

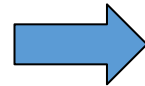
- Un test unitaire est une procédure permettant de **vérifier** le bon fonctionnement d'une **partie précise** d'un logiciel ou d'une portion d'un programme en lançant des **exécutions** selon un certain **critère d'arrêt**.
- L'observation de **chaque exécution** est **comparée** au **critère d'arrêt** pour statuer en vérifiant que :
 - **Si conforme** : test passé (**ACCEPT**)
 - **Sinon** : test échoué (**FAIL**)
 - Ce ne sont **pas des tests exhaustifs** mais des tests de validation suivant **certaines conditions**
!!!

Exemple de test unitaire

- **TestVoitureCouleurRouge(Voiture A)**
 - Si conforme : test passé (**ACCEPT**)
 - Sinon : test échoué (**FAIL**)



Test FAIL



Test ACCEPT

Les bonnes pratiques

- **Quelques recommandations**

- Le **nom** des tests devrait permettre de facilement fournir une indication sur **le but du test**
- Il est préférable de n'avoir **qu'un seul assert par test** car un test ne devrait avoir qu'une **seule raison d'échouer**
- Le code des **tests unitaires** doit être **maintenu** au même titre que le code qu'il teste : la même attention doit être portée dans leur écriture (respect des normes, commentaires, refactoring, ...)
- Il ne faut surtout **pas livrer du code** dont au moins **un test unitaire échoue** quelques soient les raisons.

Mettre en œuvre des tests unitaires (1)

- **Quand ?**

- Idéalement, écrire les tests **avant le code** à tester
- Ecrire les tests **juste après** avoir écrit une méthode
- Ecrire les tests, écrire le code pour faire échouer les tests, vérifier que les tests échouent, **corriger le code**, vérifier que les **tests sont OK**

- **Pourquoi ?**

- Permettre de **détecter des bugs** le plus **rapidement** possible...
- Coder des **cas oubliés** dans la méthode,
- Ecrire du code testable **évitant ainsi un refactoring** parfois conséquent

Mettre en œuvre des tests unitaires (2)

- **Trois règles à appliquer avec les tests unitaires :**

- **Tester le plus possible** : afin d'augmenter les chances de découvrir des bugs
- **Tester le plus tôt possible** : plus les tests sont faits tôt plus les bugs sont rapidement détectés
- **Tester le plus souvent possible** : en les automatisant et si possible en les intégrant dans un processus d'intégration continue

Mettre en œuvre des tests unitaires (3)

- **Les principes à respecter :**

- **Le test doit être le plus petit et le plus simple possible** : un test, une méthode de test. Il doit être simple, compréhensible et maintenable
 - **Chaque test doit être isolé** : un test ne doit pas dépendre d'un autre. Ceci permet aussi de garantir qu'une modification d'un test n'aura pas d'impact sur un autre
 - **Les tests unitaires doivent être automatisés** pour pouvoir être facilement exécutés régulièrement,
- Mais le problème est que souvent le code d'une méthode peut avoir besoin d'autres objets ou de ressources externes. Penser aux **design patterns**, aux **mocks**.

Remarques

- **Les deux assumptions de base**

- Si cela fonctionne une fois, cela fonctionne toujours
- Si cela fonctionne pour quelques valeurs clés, cela fonctionnera pour toutes les autres.

- **Observations**

- Le test **ne dit pas quelle est l'erreur**, il dit seulement qu'il y en a une
- Le test **ne corrige pas l'erreur**
- **Ce n'est pas parce que le test passe qu'il n'y a pas d'erreur**
- **Ce n'est pas parce que vous corrigez l'erreur qu'il n'y en a plus.**

Junit

- **Framework** de référence **pour réaliser des tests unitaires.**
- Gratuit, et téléchargeable **<http://junit.org/>**
- Simple, on écrit une classe Java on utilise des **annotations** pour **indiquer les tests** et on lance le tout par exemple dans son IDE
- **Le plus difficile** n'est pas dans l'écriture du code mais dans **sa réflexion**
 - Je veux tester quoi ? Pourquoi ?
 - Le comment est simple

Ecriture des cas de tests

1. **Création** d'une instance de la **classe** et de **tout autre objet** nécessaire aux tests
2. **Appel de la méthode à tester** avec les paramètres du cas de tests
3. **Comparaison** du résultat **attendu** avec le résultat **obtenu** : en cas d'échec, une exception est levée
 - Il est important de se souvenir lors de l'écriture de cas de tests que ceux-ci doivent être **indépendants** les uns des autres. Junit **ne garantit pas l'ordre d'exécution** des cas de tests puisque ceux-ci sont obtenus par introspection.

Junit Error ou Fail

- Un test (i.e. une méthode testXxx) est :
 - Un *échec* si une exception de type Junit.framework.*AssertionFailedError* est levée
 - Une *erreur* si une exception de type java.lang.*Exception* est levée
- Vous pouvez utiliser les méthodes d'*assert* fournies par la classe *Assert* afin de *mettre en évidence un échec*.
- Ne pas confondre la classe *Assert*, avec le mot clé *assert* en Java

JUnit 4 Tester Quoi ? (1)

```
package fr.exemple;

public class Personne {
    private String nom;
    private String prenom;
    private int age;

    public Personne(String unNom,
                    String unPrenom,
                    int unAge) {
        super();
        this.setNom(unNom);
        this.setPrenom(unPrenom);
        this.setAge(unAge);
    }

    public String getNom() {
        return this.nom;
    }

    public void setNom(String aNom) {
        this.nom = aNom;
    }
}
```

```
    public String getPrenom() {
        return this.prenom;
    }

    public void setPrenom(String aPrenom) {
        this.prenom = aPrenom;
    }

    public int getAge() {
        return this.age;
    }

    public void setAge(int aAge) {
        if (this.age < 0) {
            this.age = 0;
        } else {
            this.age = aAge;
        }
    }
}
```

JUnit 4 Tester Quoi ? (2)

```
@Override
public String toString() {
    StringBuilder lcBuilder = new StringBuilder();
    lcBuilder.append("Personne [");
    if (this.nom != null) {
        lcBuilder.append("nom=");
        lcBuilder.append(this.getNom());
        lcBuilder.append(", ");
    }
    if (this.prenom != null) {
        lcBuilder.append("prenom=");
        lcBuilder.append(this.getPrenom());
        lcBuilder.append(", ");
    }
    lcBuilder.append("age=");
    lcBuilder.append(this.getAge());
    lcBuilder.append("]");
    return lcBuilder.toString();
}
```

```
@Override
public boolean equals(Object obj) {
    if (this == obj) {    return true;    }
    if (obj == null) {    return false;    }
    if (this.getClass() != obj.getClass()) {
        return super.equals(obj);
    }
    Personne other = (Personne) obj;
    if (this.age != other.age) {    return false;    }
    if (this.nom == null) {
        if (other.nom != null) {    return false;    }
    } else if (!this.nom.equals(other.nom)) {
        return false;
    }
    if (this.prenom == null) {
        if (other.prenom != null) {    return false;    }
    } else if (!this.prenom.equals(other.prenom)) {
        return false;
    }
    return true;
}

} // Fin de la classe
```

Junit 4

Annotation @Test

- Après avoir défini ce que l'on souhaite tester, il suffit de réaliser une **classe Java** qui aura comme rôle de **contenir nos tests**
- Le nom de la classe n'a pas d'importance, son héritage non plus
 - Cependant, il est de coutume que son nom termine par Test
 - Ex: ServiceImplTest.java
- Pour chaque méthode dans cette classe qui représentera **un test** on utilisera l'annotation **@Test**

Junit 4 - Annotation @Test

```
package fr.test.exemple;  
import org.junit.*;  
Import static org.junit.Assert.*;  
import fr.exemple.Personne;
```

Attention aux import

```
public class PersonneTest {
```

Pensez à l'annotation

@Test

```
public void nom1Test() {  
    final String nom = "Dupont", prenom = "Jean";  
    final int age = 25;  
    Personne p = new Personne(nom, prenom, age);  
    Assert.assertEquals("Les deux noms doivent être identiques", nom, p.getNom());  
}
```

Pensez à l'annotation

@Test

```
public void nom2Test() {  
    final String nom = "Dupont", prenom = "Jean";  
    final int age = 25;  
    Personne p = new Personne(null, prenom, age);  
    p.setNom(nom);  
    Assert.assertEquals("Les deux noms doivent être identiques", nom, p.getNom());  
}  
}
```

Junit

Assert

- Le nom de la méthode qui représente votre test n'a techniquement aucune importance.
 - Cependant, il est conseillé de lui donner un nom qui explique sur quoi porte le test et de la faire terminer par `xxxxxTest`
- Notez que dans le corps des méthodes, on ne fait **pas de if pour valider un test**
 - On fait usage **d'Assert** qui remplacent les tests

```
Assert.assertEquals("Les deux noms doivent être identiques", nom, p.getNom());
```

- Fait que si `nom.equals(p.getNom())` n'est pas vrai alors le test échoue (*fail*).
- Il existe une série d'`assertXxxx`, il est **capital** d'en faire usage. (<http://junit.sourceforge.net/javadoc/>)

Junit

Assert

- Toutes les méthodes assertXxx
 - Sont **statiques**
 - Prennent un **message en premier paramètre**
- Le message sert à afficher une information/explication que l'on verra
 - Dans le code
 - Dans le cas où le test ne se passe pas correctement (*fail*)
- Il est **important** d'indiquer un message **clair**, **simple** et **explicite** sur ce que vous vouliez tester
- Ne jamais mettre une chaîne vide

Junit

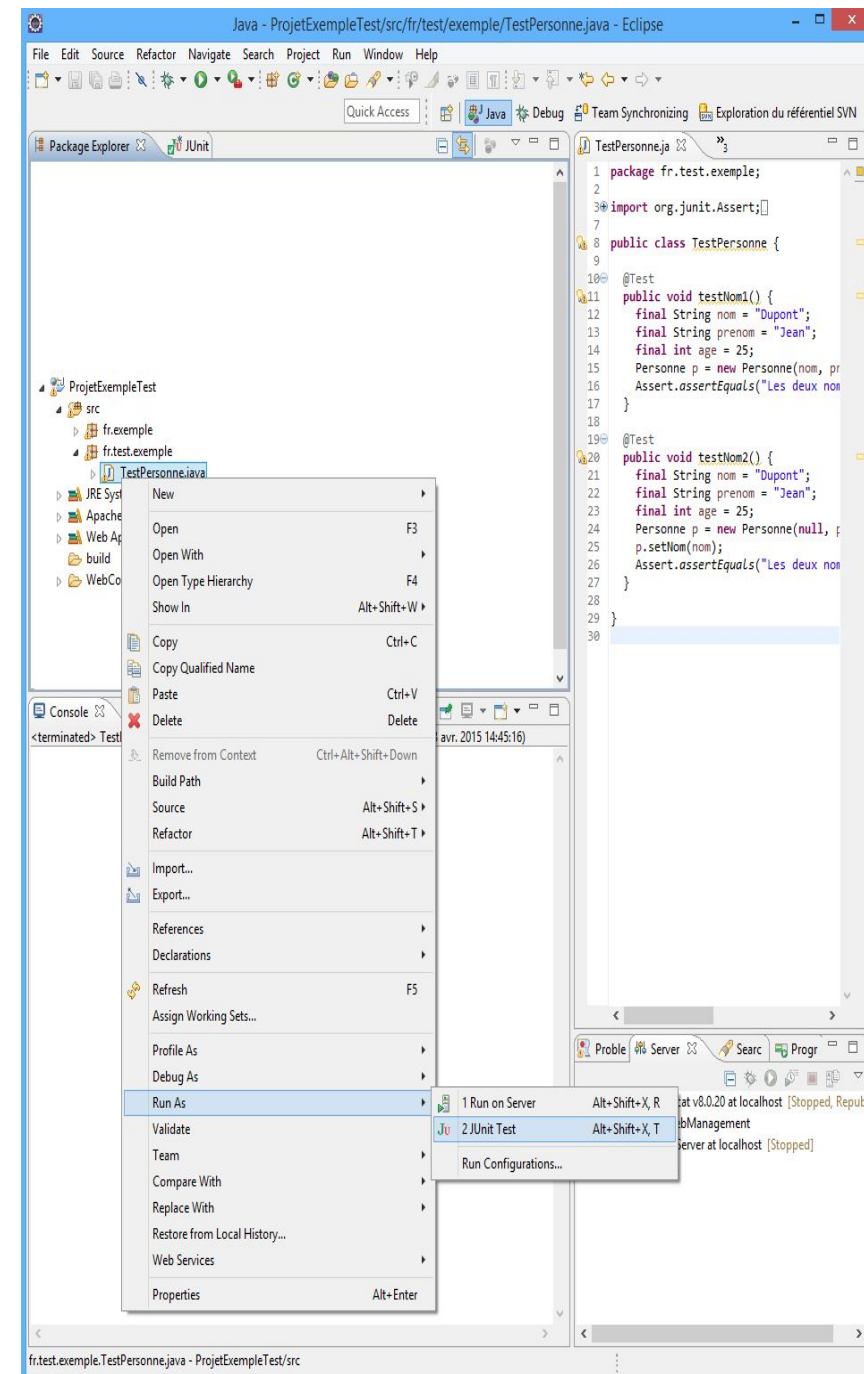
Assert

- Les méthodes **assertEquals** pour l'égalité et sur tout les **type primitifs** :
 - `static void assertEquals(String msg, boolean expected, boolean actual)`
 - `static void assertEquals (String msg, int expected, int actual)`
- Les méthodes **assertTrue** / **false** pour l'égalité sur condition **booléenne** :
 - `static void assertTrue(String msg, boolean condition)`
- Les méthodes **assertSame** pour l'égalité sur les **références** (**==**):
 - `static void assertSame(String msg, Object expt, Object actu)`
- Les méthodes **assertNull** pour l'égalité sur **null** :
 - `static void assertNotNull(String msg, Object object)`

Junit

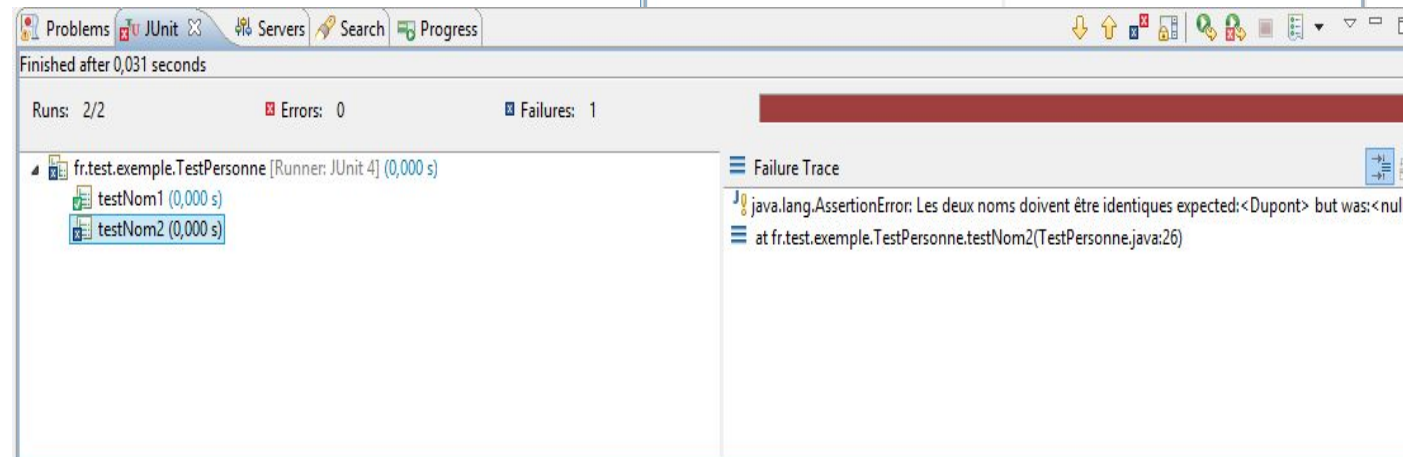
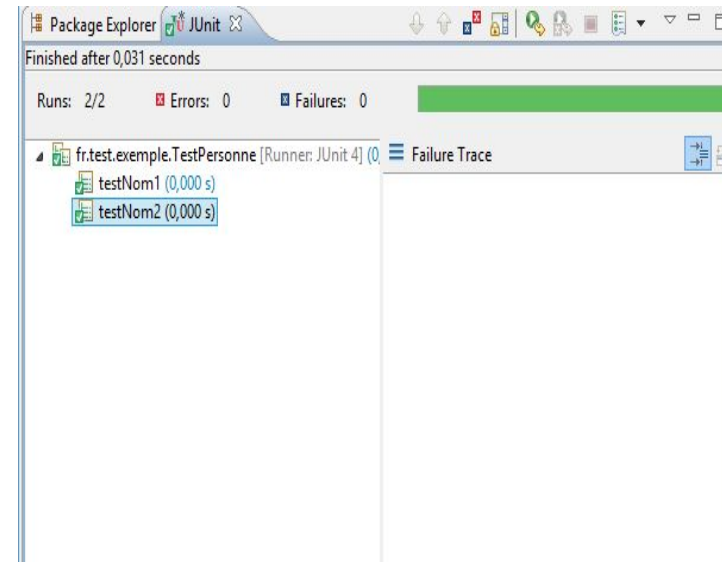
Lancement IDE

- Pour lancer son test unitaire, il suffit de faire un clic droit sur
 - sa méthode @Test
 - ou sa classe Java de test
 - ou son package
- Puis Run As/Junit Test



Junit Résultat

- Un écran spécial s'ouvrira faisant apparaître l'outil JUnit d'Eclipse
- Si tous les tests sont passés, tout est vert.
- Sinon, vous aurez du rouge



4. Les annotation JUnit

Après avoir défini ce que l'on souhaite tester, il suffit de réaliser une **classe Java** qui aura comme rôle de contenir nos tests.

Le nom de la classe n'a pas d'importance, son héritage non plus, il est cependant de coutume que son nom termine par Test.

Pour chaque méthode dans cette classe qui représentera un test on utilisera l'annotation @Test.

Annotation	Description
@Test	L'annotation TEST indique que la méthode public auquel il est attaché peut être exécutée comme un cas de test.
@BeforeAll	La méthode annotée sera exécutée seulement avant le premier test
@AfterAll	La méthode annotée sera exécutée seulement après le dernier test
@BeforeEach	La méthode annotée sera exécutée avant chaque test
@AfterEach	La méthode annotée sera exécutée après chaque test

4. Les assertions JUnit

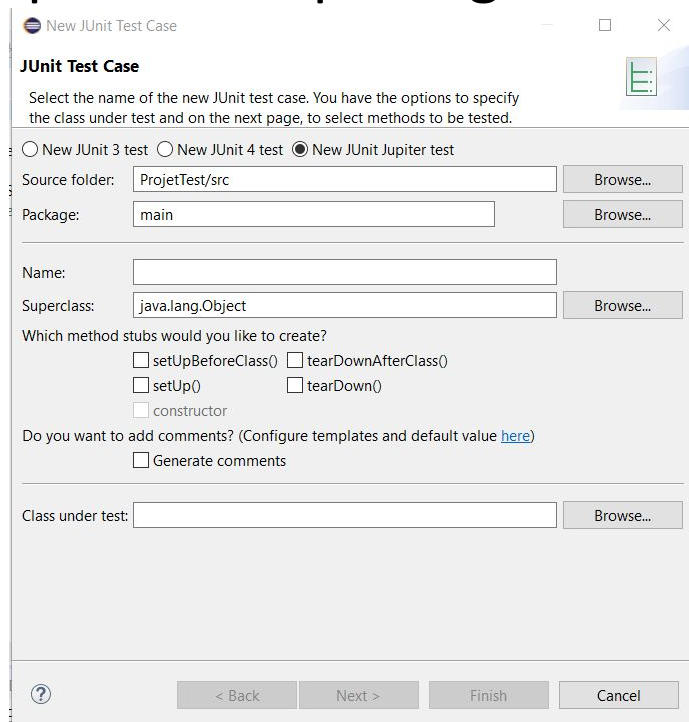
Nous pouvons faire des tests en utilisant d'autres méthodes qui peuvent remplacer `if ... fail...`

- `assertTrue(message, condition)` : permet de vérifier que la condition fournie en paramètre est vraie.
- `assertFalse(message, condition)` : permet de vérifier que la condition fournie en paramètre est fausse.
- `assertEquals(message, expected, actual)` : permet de vérifier l'égalité (sa réciproque est `assertNotEquals`).
- `assertNotNull(message, object)` permet de vérifier, pour les paramètres utilisés, qu'une méthode ne retourne pas la valeur null (sa réciproque est `assertNull`).

5. Premier exemple

On souhaite écrire une méthode qui retourne le plus grand nombre entier dans une liste d'entier.

1. Créer un nouveau projet comprenant un package main contenant la classe principale et un package test qui va contenir les tests unitaires.



```
package test;

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

class ClasseTest {

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

}
```

Importation

5. Premier exemple

2. Écrire un premier test, avant même d'écrire la fonction qui renvoie le plus grand nombre le plus simplement possible.

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

import main.GererNombre;

import org.junit.jupiter.api.Test;

class ClasseTest {

    /* debut du test*/
    @Test
    public void testPlusGRandNombreBasic() {
        int[] list = new int[] {1, 7 , 9}; //declare une liste exemple
        GererNombre nb=new GererNombre(); // instanciation object classe GererNombre a écri
        int max = nb.plusGrandNombre(list); // appel de la methode biggestNumber à coder da
        assertEquals(9,max); // resultat attendu par rapport à list exemple af
    }
}
```

5. Premier exemple

3. Écrire la classe GererNombre et la méthode plusGrandNombre qui renvoi un entier (le plus grand) et prend en paramètre un tableau d'entier. Écrire la méthode la plus simple possible même si dans un premier temps elle ne fait pas ce que l'on souhaite.

```
public class GererNombre {  
    public int plusGrandNombre(int[] list) {  
        return 0;  
    }  
}
```

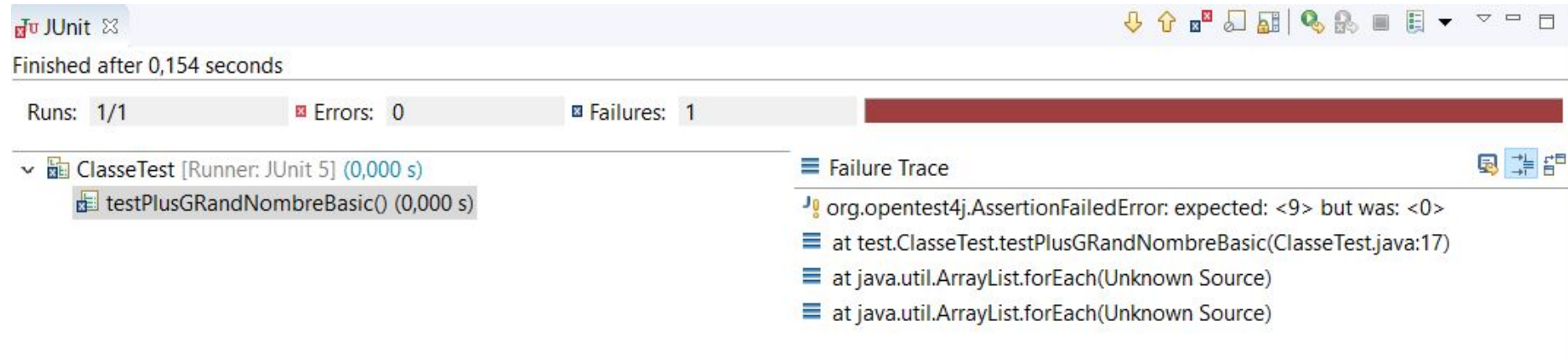
Pour le TDD on code le minimum à chaque fois donc pour le moment :

return 0;

Ne pas oublier d'importer GererNombre dans la classe de test.

5. Premier exemple

4. Lancer le test. Clic droit sur le nom de la fonction Test -> Run As -> JUnit Test



Ouverture de la fenêtre Junit : le test a échoué.

Information sur l'échec: il attendait 9 mais il a obtenu 0.

Normal puisque la fonction plusGrandNombre retourne 0.

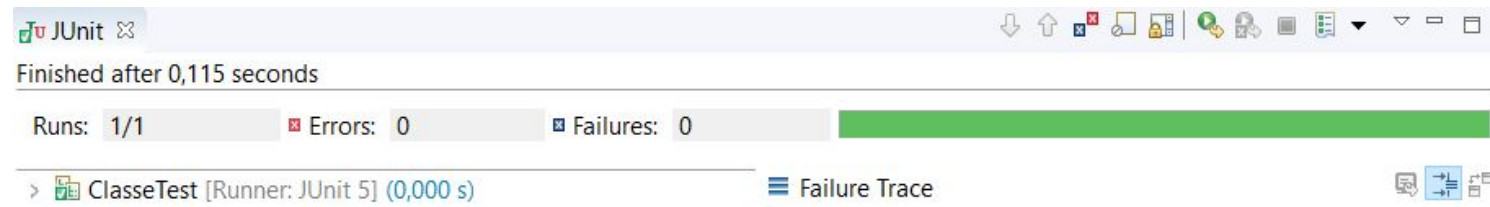
5. Premier exemple

5. Modifier la fonction plusGrandNombre pour que celle-ci retourne ce qu'on attend c'est-à-dire 9 et relancer le test.

```
public class GererNombre {  
  
    public int plusGrandNombre(int[] list) {  
        return 9;  
    }  
}
```

Le test passe.... Mais la fonction retourne 9 et s'applique à notre exemple de test. L'intérêt du TDD est d'essayer de penser à toutes les éventualités. Parmi elles, si on utilise une liste dont le plus grand nombre n'est pas 9, le test devrait échouer.

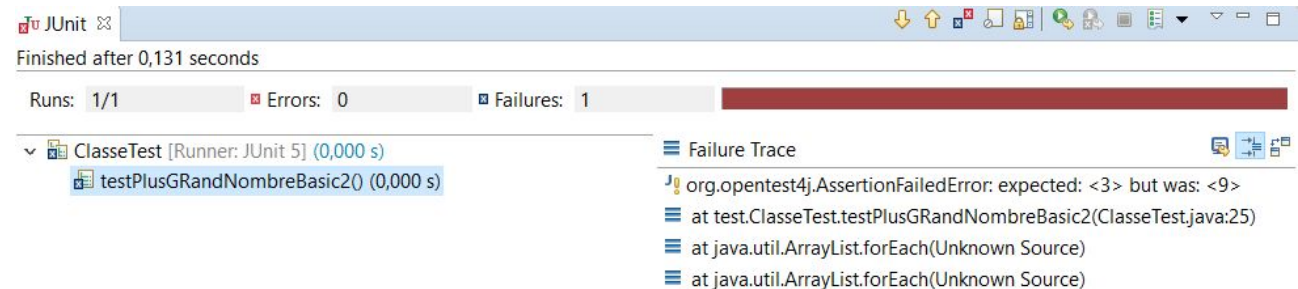
Modifions le test pour vérifier.



5. Premier exemple

6. Un autre test basic avec une liste différente :

```
@Test
public void testPlusGRandNombreBasic2() {
    int[] list = new int[] {1, 2, 3}; //de
    GererNombre nb=new GererNombre(); // i
    int max = nb.plusGrandNombre(list); //
    assertEquals(3,max); // r
}
```

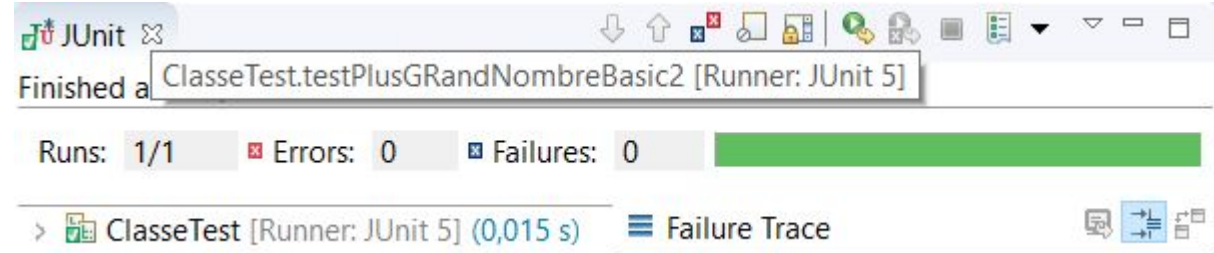


On voit bien que le test ne passe pas. Donc il va falloir modifier la fonction plusGrandNombre. Donc à chaque fois on ne va pas détailler pour tous les nombres possibles tous les tests, mais il est intéressant de voir au moins une fois pour débiter en détail la précision des tests que l'on peut faire.

5. Premier exemple

7. Code plus général de la fonction plusGrandNombre :

```
public int plusGrandNombre(int[] list) {  
    int max=0;  
    for (int i=0; i<list.length; i++) {  
        if(list[i]>max) {  
            max=list[i];  
        }  
    }  
    return max;  
}
```



Et relancer le test.

5. Premier exemple

8. Nous avons réalisé un premier cycle de TDD! Pouvez-vous maintenant imaginer dans quel cas le code de cette fonction pourrait ne pas fonctionner ?

```
public int plusGrandNombre(int[] list) {  
    int max=0;  
    for (int i=0; i<list.length; i++) {  
        if(list[i]>max) {  
            max=list[i];  
        }  
    }  
    return max;  
}
```

Par exemple les nombres négatifs ou nuls, écrire un nouveau test ...

5. Premier exemple

9. Un nouveau test pour les nombres négatifs

```
@Test
public void testPlusGrandNombreAvecNombreNegatif() {
    int[] list = new int[] {-1, -2, -3};
    GererNombre nb = new GererNombre();
    int max = nb.plusGrandNombre(list);
    assertEquals(-1, max);
}
```

The screenshot shows the JUnit test runner interface. At the top, it says "JUnit" with a logo and a close button. Below that, it says "Finished after 0,138 seconds". A progress bar shows the test results: "Runs: 3/3", "Errors: 0", and "Failures: 1". The progress bar is red, indicating a failure. Below the progress bar, there is a list of test cases under the heading "ClasseTest [Runner: JUnit 5] (0,016 s)". The list includes three test cases: "testPlusGRandNombreBasic() (0,000 s)", "testPlusGRandNombreBasic2() (0,000 s)", and "testPlusGrandNombreAvecNombreNegatif() (0,016 s)". The last test case is highlighted with a red 'x' icon, indicating it failed. To the right of the test cases, there is a "Failure Trace" section. It shows the following error message: "org.opentest4j.AssertionFailedError: expected: <-1> but was: <0>". The stack trace includes the following lines: "at test.ClasseTest.testPlusGrandNombreAvecNombreNegatif(Clas", "at java.util.ArrayList.forEach(Unknown Source)", and "at java.util.ArrayList.forEach(Unknown Source)".

5. Premier exemple

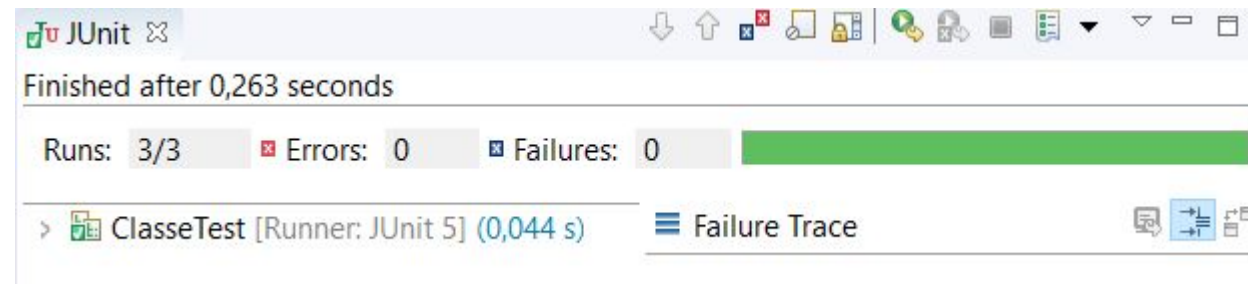
9. Un nouveau test pour les nombres négatifs

Dans la fonction `max = 0`, les valeurs négatives ne peuvent pas être prises en compte. Il faut donc redéfinir `max`.

```
public int plusGrandNombre(int[] list) {  
    int max=Integer.MIN_VALUE;  
    for (int i=0; i<list.length; i++) {  
        if(list[i]>max) {  
            max=list[i];  
        }  
    }  
    return max;  
}
```

On relance le test.

OK!



5. Premier exemple

10. Écrire les tests suivants et vérifier qu'ils fonctionnent. Si ce n'est pas le cas il faudra modifier le code de la fonction `plusGrandNombre` :

- S'il y a un zéro dans la liste.
- Si les valeurs de la liste sont désordonnées.
- S'il y a des valeurs répétées dans la liste.
- S'il n'y a qu'une seule valeur dans la liste.
- Si la liste ou tableau d'entier de départ est null.

5. Premier exemple

10. Écrire les tests suivants et vérifier qu'ils fonctionnent. Si ce n'est pas le cas il faudra modifier le code de la fonction plusGrandNombre:

```
@Test
public void testPlusGrandNombreNull() {
    int[] list = null;
    GererNombre nb=new GererNombre();
    int max = nb.plusGrandNombre(list);
}
```

Le test ne fonctionne pas et renvoie une exception de type NullPointerException, il faut donc créer une exception.

5. Premier exemple

11. Modifier le code en créant une classe qui va gérer l'exception

```
public class InvalidListException extends RuntimeException {  
    public InvalidListException(String arg0) {  
        super(arg0);  
    }  
}
```

```
public int plusGrandNombre(int[] list) {  
    if(list==null)  
        throw new InvalidListException("la liste ne doit pas etre nulle");  
  
    int max=Integer.MIN_VALUE;  
    for (int i=0; i<list.length; i++) {  
        if(list[i]>max) {  
            max=list[i];  
        }  
    }  
    return max;  
}
```

Permet de définir
l'exception

```
@Test (expected = InvalidListException.class)  
public void testPlusGrandNombreNull(){  
    int[] list = null;  
    GererNombre nb=new GererNombre();  
    int max = nb.plusGrandNombre(list);  
}
```

Exercice

- Créer un nouveau projet Java.
- Créer deux Package : org.eclipse.main et org.eclipse.test.
Pour chaque classe créée dans org.eclipse.main, on lui associe une classe de test (dans org.eclipse.test).
- On prépare le test et ensuite on le lance; s'il y a une erreur, on la corrige et on relance le test.
- Créer une première classe Calcul contenant une méthode pour faire une somme, une soustraction, une multiplication et une division.

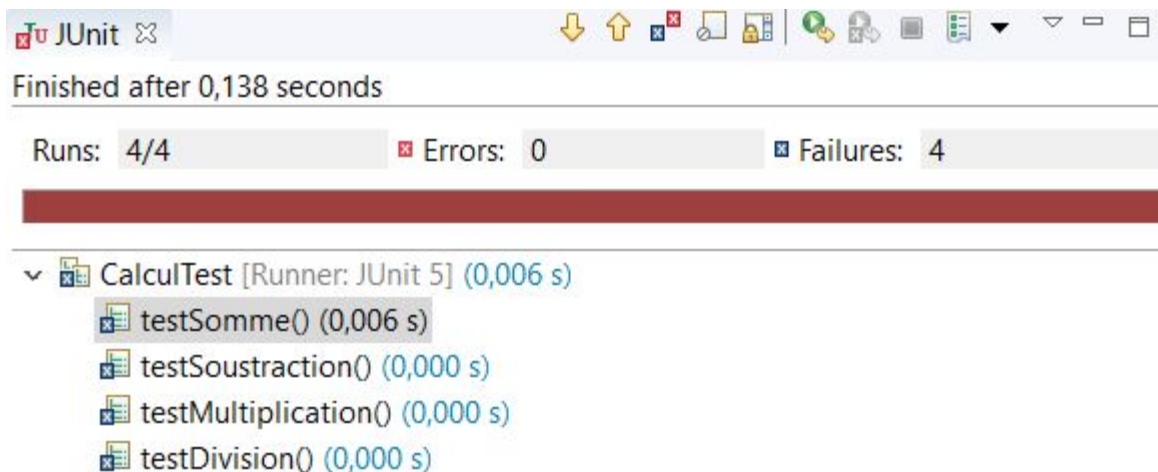
```
public class Calcul {  
    public Calcul() {  
    }  
  
    public int somme(int x, int y) {  
        return x+y;  
    }  
  
    public int soustraction(int x, int y) {  
        return x-y;  
    }  
  
    public int multiplication(int x, int y) {  
        return x*y;  
    }  
  
    public int division(int x, int y) {  
        return x/y;  
    }  
}
```

Exercice

- créer la Classe de Test :
 - Clic droit sur le package org.eclipse.test
 - New -> JUnit Test Case
 - Saisir le nom « CalculTest » dans Name
 - Cochés les 4 cases « Which method stubs would like to create? »
 - Cliquer sur « Browse » en face de « Class under test »
 - Chercher « calcul », sélectionner Calcul-org.eclipse.main et valider
 - Cliquer sur « Next »
 - Cocher les cases correspondantes de somme, soustraction, multiplication et division dans Calcul
 - Cliquer sur « finish »
 - Cliquer sur Ok pour valider Add Junit 5 library

6. Deuxième exemple

- Clic droit sur la classe de test
- Run As -> JUnit Test



```
package org.eclipse.test;

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

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 testSomme() {
        fail("Not yet implemented");
    }

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

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

Exercice

- Créer les cas de tests de la méthode somme

```
@Test
void testSommeEntiersIdentiques() {
    assertEquals(2, calcul.somme(1, 1));
}
```

assertEquals(expected, actual)

```
@Test
void testSomme() {
    assertEquals("2 entiers identiques", 2, calcul.somme(1, 1));
    assertEquals("2 entiers positifs", 5, calcul.somme(2, 3));
    assertEquals("2 entiers negatifs", -5, calcul.somme(-2, -3));
    assertEquals("2 entiers de signes opposés", 1, calcul.somme(-2, 3));
    assertEquals("1er entier null", 2, calcul.somme(2, 0));
    assertEquals("2eme entier null", 3, calcul.somme(0, 3));
    assertEquals("2 entiers null", 0, calcul.somme(0, 0));
}
```

assertEquals(message, expected, actual)

Annotations @BEFORE et @AFTER

- Dans certain cas, on veut **initialiser** des informations **avant** chaque test
 - Par exemple : ouvrir un fichier, une base de données , ...
- De même, on veut réaliser une action **à la fin de chaque test**
 - Par exemple : fermer la base de données, un fichier, ...
- Deux annotations :
 - **@Before** : à placer sur une méthode. Cette dernière sera appelée avant chaque @Test
 - **@After** : à placer sur une méthode. Cette dernière sera appelée après chaque @Test

Annotations @BEFORE et @AFTER

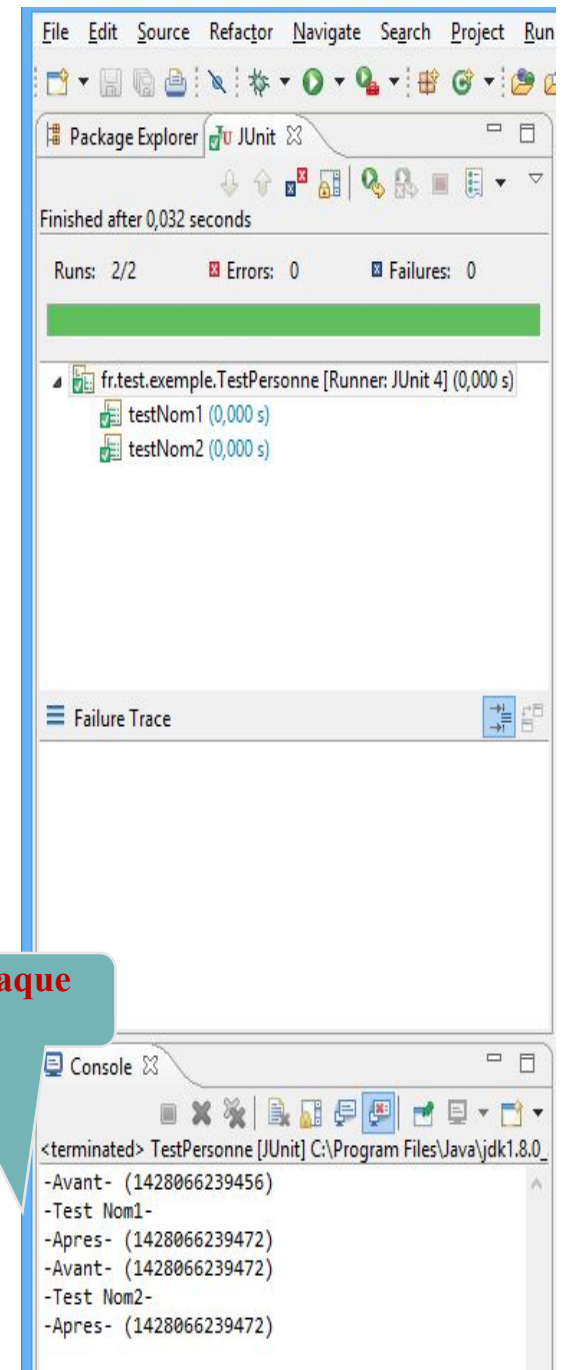
```
1 package fr.exemple;
2
3 import org.junit.*;
4
5 public class PersonneTest {
6
7     @Before
8     public void setUpBefore() throws Exception {
9         System.out.println("-Avant- (" + System.currentTimeMillis() + ")");
10    }
11
12    @After
13    public void tearDownAfter() throws Exception {
14        System.out.println("-Après- (" + System.currentTimeMillis() + ")");
15    }
16
17    @Test
18    public void testNom1() {
19        System.out.println("testNom1");
20        final String nom = "Dupont", prenom = "Jean";
21        final int age = 25;
22        Personne p = new Personne(nom, prenom, age);
23        Assert.assertEquals("Les deux noms doivent être identiques",
24            nom, p.getNom());
25    }
26
27    @Test
28    public void testNom2() {
29        System.out.println("testNom2");
30        final String nom = "Dupont", prenom = "Jean";
31        final int age = 25;
32        Personne p = new Personne(null, prenom, age);
33        p.setNom(nom);
34        Assert.assertEquals("Les deux noms doivent être identiques",
35            nom, p.getNom());
36    }
37 }
```

Attention à vos import

Initialise chaque test

Finalise chaque test

Exécution de chaque test



Annotations @BEFOREclass et @AFTERclass

- Dans la même idée, deux annotations permettent d'exécuter un code en début de tous les tests ainsi qu'en fin
 - @BeforeClass : à placer sur une méthode. Cette dernière sera appelée avant le tout premier @Test
 - @AfterClass : à placer sur une méthode. Cette dernière sera appelée après le dernier @Test
- Dans ce cas, les méthodes annotées doivent impérativement être static.

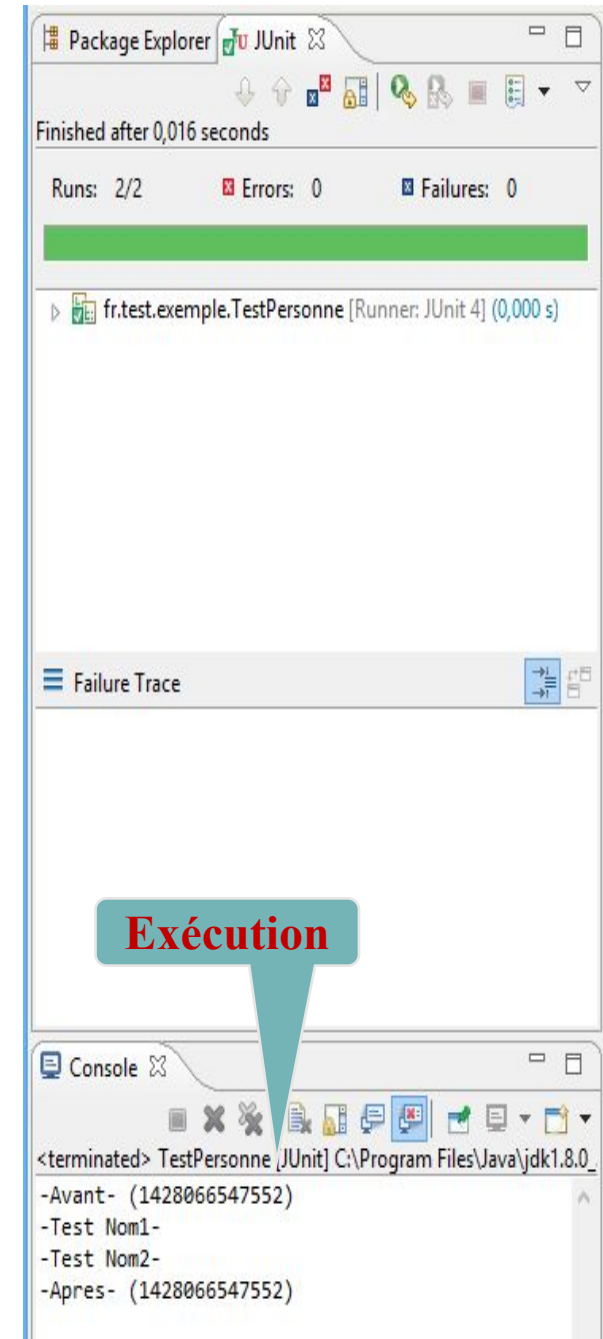
@Beforeclass et @Afterclass

```
1 package fr.exemple;
2
3 import org.junit.*;
4
5
6
7
8
9
10
11 public class PersonneTest {
12
13     @BeforeClass
14     public static void setUpBeforeClass() throws Exception {
15         System.out.println("-Avant- (" + System.currentTimeMillis() + ")");
16     }
17
18     @AfterClass
19     public static void tearDownAfterClass() throws Exception {
20         System.out.println("-Après- (" + System.currentTimeMillis() + ")");
21     }
22
23     @Test
24     public void testNom1() {
25         System.out.println("testNom1");
26         final String nom = "Dupont", prenom = "Jean";
27         final int age = 25;
28         Personne p = new Personne(nom, prenom, age);
29         Assert.assertEquals("Les deux noms doivent être identiques",
30             nom, p.getNom());
31     }
32
33     @Test
34     public void testNom2() {
35         System.out.println("testNom2");
36         final String nom = "Dupont", prenom = "Jean";
37         final int age = 25;
38         Personne p = new Personne(null, prenom, age);
39         p.setNom(nom);
40         Assert.assertEquals("Les deux noms doivent être identiques",
41             nom, p.getNom());
42     }
43 }
44 }
```

Attention à vos import

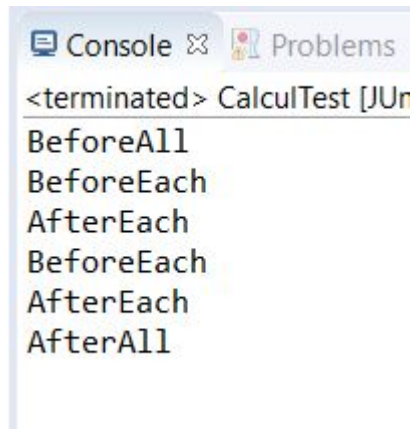
Initialise une fois

Finalise



Pré/post test

- Nous allons pouvoir restructurer la classe de Test



```
Console Problems
<terminated> CalculTest [JUn
BeforeAll
BeforeEach
AfterEach
BeforeEach
AfterEach
AfterAll
```

```
@BeforeAll
static void setUpBeforeClass() throws Exception {
    System.out.println("BeforeAll");
}

@AfterAll
static void tearDownAfterClass() throws Exception {
    System.out.println("AfterAll");
}

@BeforeEach
void setUp() throws Exception {
    System.out.println("BeforeEach");
}

@AfterEach
void tearDown() throws Exception {
    System.out.println("AfterEach");
}

@Test
void testSomme() {
    Calcul calcul = new Calcul();
    assertEquals("2 entiers identiques", 2, calcul.somme(1, 1));
}
```

Exercice

- Dans notre exemple nous allons pouvoir:
 - instancier un objet en dehors des cas de test
 - Initialiser l'objet dans la méthode de test appelant l'annotation `@BeforeClass`
 - Remettre l'objet à null dans la méthode de test appelant l'annotation `@AfterClass`
 - On peut donc se dispenser d'instancier un objet dans les cas de tests

```
class CalculTest {  
  
    Calcul calcul;  
  
    @BeforeClass  
    static void setUpBeforeClass() throws Exception {  
    }  
  
    @AfterClass  
    static void tearDownAfterClass() throws Exception {  
    }  
  
    @Before  
    void setUp() throws Exception {  
        calcul=new Calcul();  
    }  
  
    @After  
    void tearDown() throws Exception {  
        calcul=null;  
    }  
  
    @Test  
    void testSomme() {  
        assertEquals("2 entiers identiques", 2, calcul.somme(1, 1));  
    }  
}
```

Exercice

- Cas de test de la méthode division

```
public int division(int x, int y) throws DivisionArithmeticException {  
    if(y==0)  
        throw new DivisionArithmeticException("pas de division par 0");  
    return x/y;  
}
```

```
public class DivisionArithmeticException extends RuntimeException {  
  
    public DivisionArithmeticException(String arg0) {  
        super(arg0);  
    }  
}
```

```
@Test  
void testDivision() {  
    assertEquals("2 entiers identiques", 1, calcul.division(2, 2));  
    assertEquals("2 entiers positifs", 2, calcul.division(6, 3));  
    assertEquals("2 entiers negatifs", 2, calcul.division(-6, -3));  
    assertEquals("2 entiers de signes opposés", -2, calcul.division(-6, 3));  
  
    //1er entier est null  
    boolean th = false;  
    try {  
        calcul.division(2,0);  
    }catch(DivisionArithmeticException e) {  
        th=true;  
    }  
    assertTrue(th);  
    th=false;  
  
    //2 entiers null  
    try {  
        calcul.division(0,0);  
    }catch(DivisionArithmeticException e) {  
        th=true;  
    }  
    assertTrue(th);  
}
```

Le test de la levée d'exceptions

- La gestion des tests d'erreurs \Leftrightarrow tests qui doivent générer une **exception**
- Utiliser le paramètre **expected** de l'annotation `@Test`. Ce dernier prend un objet `Class` comme valeur
- Ne **jamais** faire de **try / catch**

Exception – Par annotation

```
1 package fr.exemple;
2
3+ import org.junit.*;
10
11 public class PersonneTest {
12
14+ public void setUpBefore() throws Exception {
17
19+ public void tearDownAfter() throws Exception {
22
24+ public void testNom1() {
32
34+ public void testNom2() {
43
44- @Test(expected = NullPointerException.class)
45 public void testException1() {
46     System.out.println("-Test Exception1-");
47     final String nom = "Dupont";
48     final String prenom = "Jean";
49     final int age = 25;
50     Personne p = new Personne(null, prenom, age);
51     p.getNom().equals(nom);
52 }
53 }
54
```

Définition de l'exception

Potentiellement à throws à indiquer
si l'exception n'est pas de
type RuntimeException

Pas de try / catch

Limitation du temps d'exécution

- Junit 4 propose une fonctionnalité rudimentaire pour **vérifier** qu'un cas de tests s'exécute **dans un temps maximum donné**.
 - L'attribut *timeout* de l'annotation `@Test` attend comme valeur un délai maximum d'exécution exprimé en millisecondes.

Limitation du temps d'exécution

```
1 package fr.exemple;  
2  
3 import org.junit.*;  
10  
11 public class PersonneTest {  
12  
14 public void setUpBefore() throws Exception {  
17  
19 public void tearDownAfter() throws Exception {  
22  
24 public void testNom1() {  
32  
34 public void testNom2() {  
43  
45 public void testException1() {  
53  
54 @Test(timeout=100)  
55 public void testcompteur() {  
56     System.out.println("-Test Compteur");  
57     for(long i = 0 ; i < 999999999; i++) {  
58         long a = i + 1;  
59     }  
60 }  
61 }  
62
```

Exception sur le temps

FINISHED: after 0,122 seconds

Runs: 4/4 Errors: 1 Failures: 0

fr.exemple.PersonneTest [Runner: JUnit 4] (0,121 s)

- testException1 (0,000 s)
- testNom1 (0,000 s)
- testNom2 (0,000 s)
- testcompteur (0,121 s)

Failure Trace

org.junit.runners.model.TestTimedOutException: test timed out after 100 milliseconds

at fr.exemple.PersonneTest.testcompteur(PersonneTest.java:56)

at java.util.concurrent.FutureTask.run(FutureTask.java:266)

at java.lang.Thread.run(Thread.java:745)

Junit et Maven

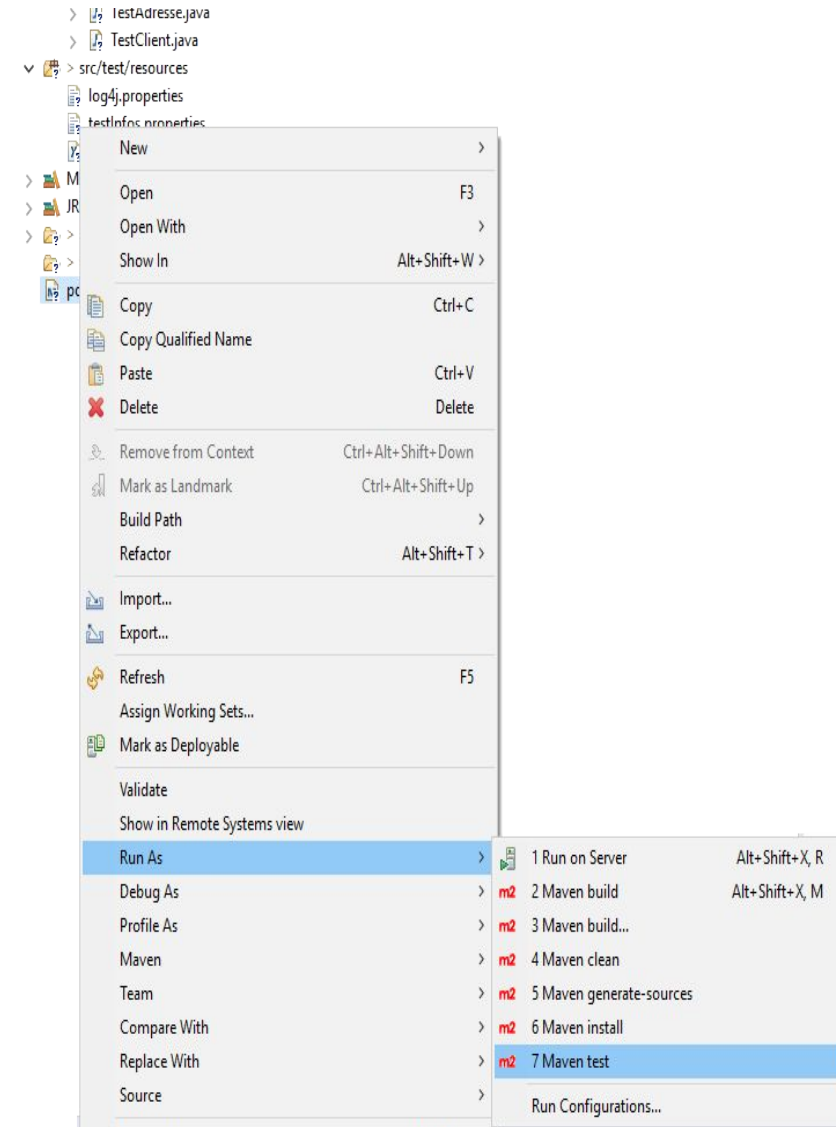
- Maven va lancer vos tests Junit **automatiquement** si
 - vous en avez fait et qu'ils sont dans le bon dossier
 - vous avez ajoutés les dépendances nécessaires à votre framework de test
 - vous avez indiqué le plugin qui doit gérer vos tests
- Le traitement des ressources de tests et la compilation des sources de tests sont effectuées à l'identique
 - `src/test/java` ⇔ Endroit par défaut où trouver vos classes de tests
 - `src/test/resources` ⇔ Endroit par défaut où trouver vos fichiers de configuration pour les tests

Junit et Maven

- Les **tests unitaires** en Junit
 - se font par le plugin surefire
 - sont liés à la phase **test**
- Pour rappel, les **tests d'intégration**
 - se font par le plugin failsafe
 - sont liés à la phase **integration-test**
- On doit distinguer au développement les TU des TI via une **règle de nommage** qui sera utilisée par les deux plugins (configurés en conséquence) pour savoir quels tests exécuter

Maven + Junit + Eclipse

- Dans Eclipse, vous pouvez lancer vos tests comme le fera Maven
 - Le résultat ne sera pas forcément le même que si vous ne lancez qu'une classe dans Eclipse
- Sur votre fichier pom.xml, faites un clique droit
 - Run As (ou Debug As) -> Maven test
- Le résultat ne sera jamais graphique et apparaîtra dans la console Maven



Les Mock

1. Introduction
2. Framework Mockito
3. Mise en pratique de Mockito

Concepts

- Isoler les méthodes testées des services externes
- Limiter la dépendances aux classes
- Vérifier le comportement interne de la méthode testée. Nombre de fois que la méthode appelle un service, vérifier les éléments passés en paramètres des services appelés

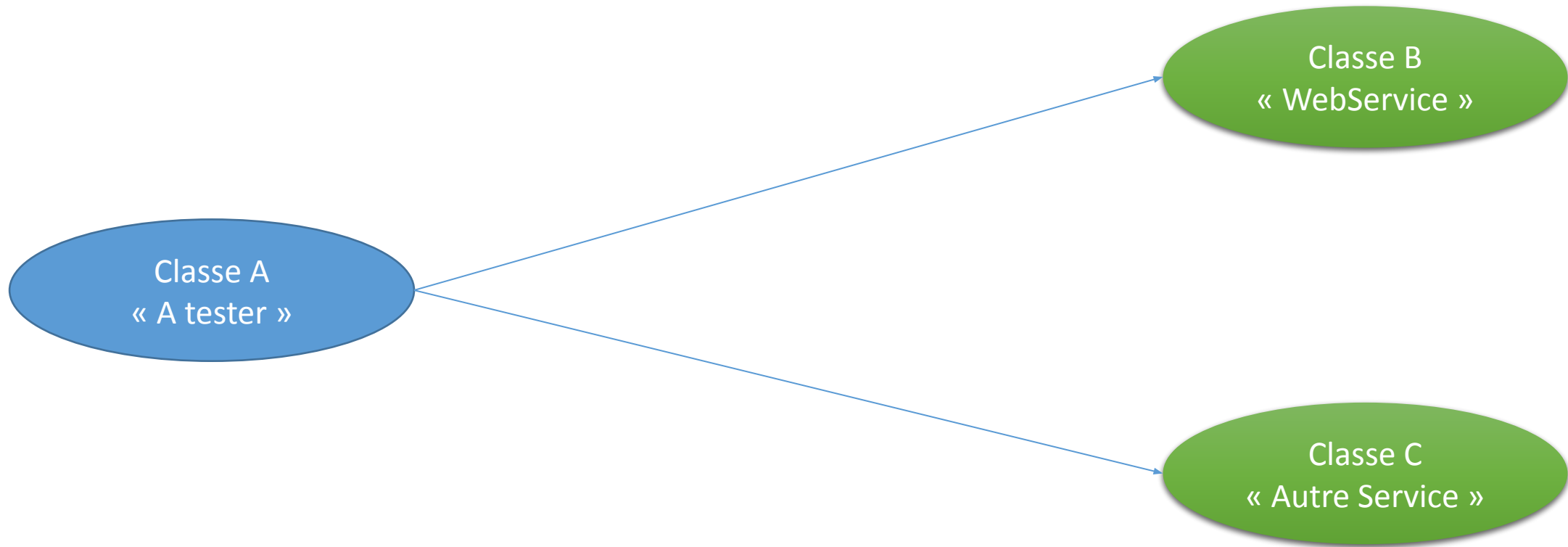
Solution : Mockito

Mockito

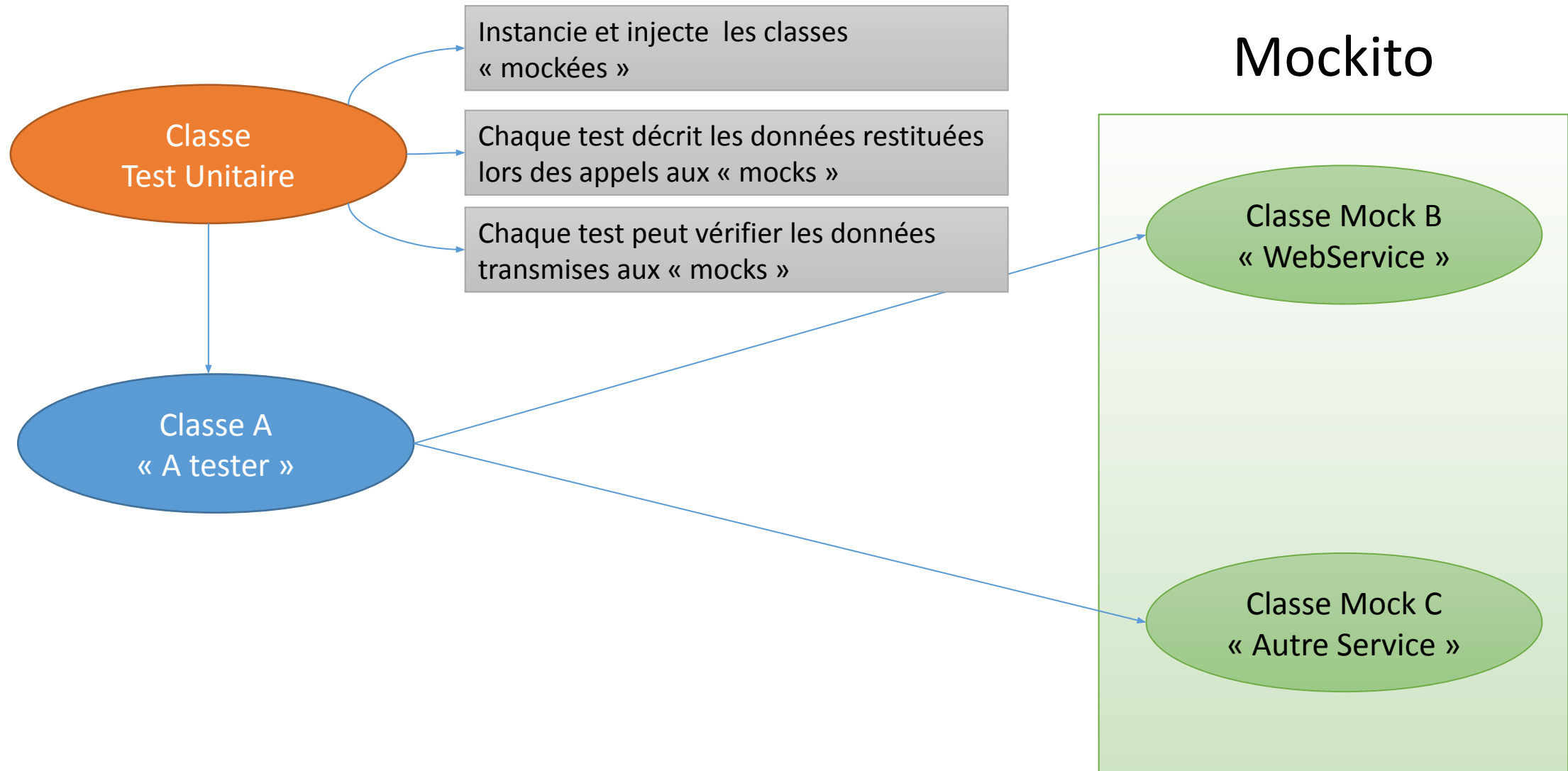
- Solution OpenSource « <https://site.mockito.org/> »
- Modèle « Set-Run-Verify »
 - Préparation des mocks
 - Exécution du test
 - Vérification des données transférés dans les mocks
- Permet de retourner des valeurs et de lever des exceptions
- Faciliter de mise en œuvre
 - Plus besoin de créer des classes de mock spécifiques pour les tests



Principe



Principe



Mise en place Mockito

- Ajout de la dépendance maven

```
<dependency>  
  <groupId>org.mockito</groupId>  
  <artifactId>mockito-core</artifactId>  
  <version>2.27.0</version>  
  <scope>test</scope>  
</dependency>
```



Scope « test »
Mockito n'est utilisé
uniquement pour les
tests

Mise en place Mockito

- Importer Mockito :

```
import static org.mockito.Mockito.*;
```

- Ajout de l'annotation @RunWith sur la classe de test

Junit 4

```
@RunWith(MockitoJUnitRunner.class)
public class UserServiceTest {
    /*...*/
}
```

Junit 5

```
@ExtendWith(MockitoExtension.class)
@RunWith(JUnitPlatform.class)
public class UserServiceTest {
```

- Déclarer les services « mockés » avec @Mock

```
@Mock
private UserJdbcDao userJdbcDao;
```

Mise en place Mockito

- Setter la classe mockée

```
@Before
public void setUp() {
    this.userService = new UserService();
    this.userService.setUserJdbcDao(this.userJdbcDao);
}
```

- Réaliser un test en initialisant les données du mock

```
@Test
public void should_return_all_users() {
    User u1 = new User(1L, "Daenerys", "Targaryen", "0611111111");

    Mockito.when(this.userJdbcDao.findAll()).thenReturn(Arrays.asList(u1));

    Collection<User> ret = this.userService.findAll();
    Assert.assertFalse(ret.isEmpty());
    Assert.assertArrayEquals(new User[] {u1}, ret.toArray());
}
```

Contrôler les réponses des mocks

- Possibilité de contrôler les réponses des mocks avec
 - la classe « ArgumentMatcher » (eq, any, contains...)
 - Avec la librairie « hamcrest »

@Test

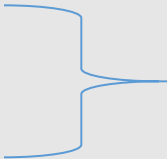
```
public void should_find_user_when_user_exists() {  
    User u1 = new User(1L, "Daenerys", "Targaryen", "0611111111");  
    User u2 = new User(2L, "Jon", "Snow", "0711111111");
```

```
    when(this.userService.findById(eq(1L))).thenReturn(u1);  
    when(this.userService.findById(eq(2L))).thenReturn(u2);
```

```
    User r1 = this.userService.findById(1L).orElseThrow(() -> fail("Utilisateur non trouvé"));  
    Assert.assertEquals((Long)1L, r1.getIdentifiant());  
    Assert.assertEquals("Daenerys", r1.getFirstName());
```

```
    User r2 = this.userService.findById(2L).orElseThrow(() -> fail("Utilisateur non trouvé"));  
    Assert.assertEquals((Long)2L, r2.getIdentifiant());  
    Assert.assertEquals("Jon", r2.getFirstName());
```

```
}
```



2 réponses possibles en fonction de l'identifiant

Vérifier les appels

- Vérifier les paramètres passés au méthode

```
verify(this.userJdbcDao).addUser(eq(u1));
```

- Vérifier le nombre de fois qu'une méthode est appelée

```
verify(this.userJdbcDao, atLeastOnce()).addUser(eq(u1));  
verify(this.userJdbcDao, times(3)).addUser(eq(u1));
```

- Et les variantes : never(), atMost()

Capturer les paramètres internes

- « ArgumentCaptor » permet d'intercepter les objets créés en interne et utilisés pour appeler des méthodes « mockées »

```
@Test
public void should_find_user_by_first_name() {
    //Appel du service
    this.userService.findUserBy("Daenerys", null);

    //Capture du dao appelé
    ArgumentCaptor<UserCriteria> captor = ArgumentCaptor.forClass(UserCriteria.class);
    verify(this.userJdbcDao).findByCriteria(captor.capture());

    //Vérification du passage de criteres
    UserCriteria userCriteria = captor.getValue();

    //Vérification des données
    Assert.assertNull(userCriteria.getName());
    Assert.assertEquals("Daenerys", userCriteria.getFirstName());
}
```

Mocker partiellement un objet

- « spy » : permet de mocker partiellement un objet

```
@Test
public void should_modify_array_size() {
    List<User> users = new ArrayList<>();
    users.add(new User(1L, "Daenerys", "Targaryen", "0611111111"));
    users.add(new User(2L, "Jon", "Snow", "0711111111"));

    assertEquals(2, users.size());

    List<User> spy = spy(users);
    when(spy.size()).thenReturn(10);

    assertEquals(10, spy.size());

    User r1 = spy.get(0);
    assertEquals(r1.getFirstName(), "Daenerys");
}
```

Limitation

- Mockito ne sait pas « mocker » des méthodes statiques

Solution : PowerMock

Les objets de type mock

- En POO, différents type de doublures :
 - Stub (bouchon) : classes qui renvoient une valeur **codée en dur** à l'invocation d'une méthode
 - Spy (espion) : classe qui **vérifie l'utilisation** qui en est faite après l'exécution
 - Mock (simulacre) : classes qui agissent comme **un stub et un spy**
- Un objet de type doublure permet de :
 - **Simuler le comportement** d'un autre objet concret de façon maîtrisée
 - **Remplacer un objet** qui n'est **pas encore écrit**.

Les objets mock dans les tests unitaires

- Réaliser un mock permet de tester le bon comportement de son code sans l'avoir réalisé totalement.
- Par exemple dans le cas des DAOs,
 - Si la base de données n'est pas encore opérationnelle
 - Si elle est vide ou pas encore construite
 - Si l'on a pas sous la main la totalité des DAOs
 - Si l'on veut écrire ses tests avant d'écrire son code

Mockito en pratique

- Dans vos classe de tests, quand vous souhaitez utiliser un Mock :
 - Il faut **créer le mock**
 - Il faut **expliquer son comportement** en fonction de contraintes
 - Les valeur retournées
 - Les exceptions levées

Mock Mockito

- Exemple nous avons réalisé un objet Calculatrice représenté par son interface ICalculette

```
1 package com.cal;
2
3
4+ * Bean calculatrice.
5
6 public class Calculatrice implements ICalculette {
7
8- /**
9-  * Constructeur par défaut.
10-  */
11- public Calculatrice() {
12-     super();
13- }
14
15
16+ public double add(double a, double b) {}
17
18
19
20
21+ public double sub(double a, double b) {}
22
23
24
25
26+ public double div(double a, double b) throws IllegalArgumentException {}
27
28
29
30
31
32
33
34+ public double mul(double a, double b) {}
35
36
37
38 }
```

```
1 package com.cal;
2
3
4+ * Interface qui represente la calculatrice distante.
5
6 public interface ICalculette {
7
8-
9+ * Ajoute deux chiffres.
10
11 public double add(double a, double b);
12
13
14
15
16
17
18
19
20+ * Soustrait deux chiffres.
21
22 public double sub(double a, double b);
23
24
25
26
27
28
29
30
31+ * Divise deux chiffres.
32
33 public double div(double a, double b) throws IllegalArgumentException;
34
35
36
37
38
39
40
41
42
43
44+ * Multiplie deux chiffres.
45
46 public double mul(double a, double b);
47
48
49
50
51
52
53 }
```

Mock Mockito

- Pour tester notre code nous pouvons faire usage de Mockito :

```
1 package com.cal.mock.compte;
2
3 import org.junit.Assert;
10
12 * Test sur la classe calculatrice avec Mockito.
14 public class TestCalculatrice {
15
16     private ICalculette calculatrice;
17
18     @Before
19     public void init() throws Exception {
20         // Creation du mock
21         this.calculatrice = Mockito.mock(ICalculette.class);
22
23         // Ajout du comportement du mock
24         double a = 50;
25         double b = 80;
26         Mockito.when(this.calculatrice.add(a, b)).thenReturn(a + b);
27         Mockito.when(this.calculatrice.sub(a, b)).thenReturn(a - b);
28         Mockito.when(this.calculatrice.div(a, b)).thenReturn(a / b);
29         Mockito.when(this.calculatrice.mul(a, b)).thenReturn(a * b);
30
31         // Comportement des exceptions
32         Mockito.doThrow(new IllegalArgumentException("Pas de division par zero")).when(this.calculatrice)
33             .div(ArgumentMatchers.anyDouble(), ArgumentMatchers.eq(0D));
34     }
```

Mock Mockito

- Nos tests sont totalement standards (au sens Junit)
- Ils doivent rester dans les bornes du paramétrage

```
37 * Test l'addition.
39 @Test
40 public void testAdd() {
41     Assert.assertNotNull("Le bean doit etre present", this.calculatrice);
42     double a = 50;
43     double b = 80;
44     double suppr = a + b;
45     Assert.assertTrue("a+b=" + suppr, this.calculatrice.add(a, b) == suppr);
46 }
47
49 * Test la soustraction.
51 @Test
52 public void testSub() {
53     Assert.assertNotNull("Le bean doit etre present", this.calculatrice);
54     double a = 50;
55     double b = 80;
56     double suppr = a - b;
57     Assert.assertTrue("a-b=" + suppr, this.calculatrice.sub(a, b) == suppr);
58 }
59
61 * Test la division.
63 @Test
64 public void testDiv() {
65     Assert.assertNotNull("Le bean doit etre present", this.calculatrice);
66     double a = 50;
67     double b = 80;
68     double suppr = a / b;
69     Assert.assertTrue("a/b=" + suppr, this.calculatrice.div(a, b) == suppr);
70 }
71
73 * Test la division avec b=0.
78 @Test(expected = IllegalArgumentException.class)
79 public void testDiv2() {
80     Assert.assertNotNull("Le bean doit etre present", this.calculatrice);
81     double a = 50;
82     double b = 00;
83     this.calculatrice.div(a, b);
84 }
85
87 * Test la multiplication.
89 @Test
90 public void testMult() {
91     Assert.assertNotNull("Le bean doit etre present", this.calculatrice);
92     double a = 50;
93     double b = 80;
94     double suppr = a * b;
95     Assert.assertTrue("a*b=" + suppr, this.calculatrice.mul(a, b) == suppr);
96 }
```

Mock Mockito

- Mockito sait faire plus que juste faire des mock
- Il peut aussi être utilisé sur des objets concrets pour ne pas avoir à tout recoder
- Il suffit d'utiliser la méthode **spy** de la classe Mockito

```
18 @Before
19 public void init() throws Exception {
20     // Creation du mock sur un objet existant (et non plus une interface)
21     this.calculatrice = Mockito.spy(Calculatrice.class);
22
23     // Ajout du comportement du mock uniquement pour add
24     double a = 5D;
25     double b = 8D;
26     Mockito.when(this.calculatrice.add(a, b)).thenReturn(a + b + 3);
27
28     // Retourner -1 pour mul
29     Mockito.doReturn(-1D).when(this.calculatrice).mul(a, b);
30
31     // On ne touche pas à div ni à sub
32 }
```


Mock Mockito

- On valide dans nos tests
- On a changé add et mul uniquement.

```
34 * Test l'addition.
36 @Test
37 public void testAdd() {
38     Assert.assertNotNull("Le bean doit etre present", this.calculatrice);
39     double a = 50;
40     double b = 80;
41     double suppr = a + b + 3;
42     Assert.assertTrue("a+b=" + suppr, this.calculatrice.add(a, b) == suppr);
43     // Ok
44     Mockito.verify(this.calculatrice).add(a, b);
45 }
46
47 * Test la soustraction.
48 @Test
49 public void testSub() {
50     Assert.assertNotNull("Le bean doit etre present", this.calculatrice);
51     double a = 50;
52     double b = 80;
53     double suppr = a - b;
54     Assert.assertTrue("a-b=" + suppr, this.calculatrice.sub(a, b) == suppr);
55     // Ok
56     Mockito.verify(this.calculatrice).sub(a, b);
57 }
58
59 * Test la division.
60 @Test
61 public void testDiv() {
62     Assert.assertNotNull("Le bean doit etre present", this.calculatrice);
63     double a = 50;
64     double b = 80;
65     double suppr = a / b;
66     Assert.assertTrue("a/b=" + suppr, this.calculatrice.div(a, b) == suppr);
67     // Ok
68     Mockito.verify(this.calculatrice).div(a, b);
69 }
70
71 * Test la division avec b=0.
72 @Test(expected = IllegalArgumentException.class)
73 public void testDiv2() {
74     Assert.assertNotNull("Le bean doit etre present", this.calculatrice);
75     double a = 50;
76     double b = 0;
77     this.calculatrice.div(a, b);
78 }
79
80 * Test la multiplication.
81 @Test
82 public void testMult() {
83     Assert.assertNotNull("Le bean doit etre present", this.calculatrice);
84     double a = 50;
85     double b = 80;
86     Assert.assertTrue("a*b=-1", this.calculatrice.mul(a, b) == -1);
87     // Ok
88     Mockito.verify(this.calculatrice).mul(a, b);
89 }
90
91 }
```