Laboratoire GEN *No II*

JUNIT

HES-SO - *Christophe Greppin/ Eric Lefrançois*

Février 2018

Contenu

1 Introduction

Dans la programmation, le test unitaire permet de s'assurer du fonctionnement correct d'une partie d'un programme. C'est pour le programmeur un moyen efficace de tester, indépendamment du programme complet, un bout de code.

Ainsi, cela permet de contrôler que ce dernier correspond aux spécifications et qu'il fonctionne en toutes circonstances.

JUnit est une librairie pour le langage de programmation Java qui permet d'effectuer ces tests. De plus, elle est très simple à utiliser et elle est intégrée par défaut dans les environnements de développement IntelliJ et Eclipse.

L'utilisation de JUnit est ici illustrée avec Intelllj .

2 Exemple de classe à tester

A titre d'exemple nous allons utiliser une classe Java qui fournit diverses méthodes. Elle permettra de tester la majorité des tests unitaires utiles. Cette classe se veut la plus simple possible afin de ne pas perdre de temps sur la compréhension du code.

Note POO : *Cet exemple utilise des méthodes d'instance (factoriel, min, etc..). Du strict point de vue méthodologie POO, il serait préférable de déclarer ces dernières en tant que méthodes de classe. Du point de vue JUnit, cela n'a aucune importance, on peut tester des méthodes de classe comme on peut tester des méthodes d'instance.*

```
public class Exemple {

        private int result;
        private String nom;
        private int valeur;
        private Object object1 = new Object();
        private Object object2 = new Object();

        public Exemple(){}

        public void incVal() {valeur++;}
        public int getVal() {return valeur;}

        public int getResult(){
              return result;
        }

        public String getNom(){
              return nom;
        }

        public Object getObject1(){
              return object1;
```

```
	}

	public Object getObject2(){
		return object2;
	}

	public void factoriel(int n){
		int prov;
		prov = 1;
		for(int i = 1; i <= n; i++)
			prov = prov*i;
		result = prov;
	}

	public void min(int a, int b){
		if(a>b)
			result = b;
		else
			result = a;
	}

	public void plusGrand(){
		nom = "Toto";
	}

	public void plusPetit(){
		nom = "Jean";
	}

	public Object renvoiNull(){
		return null;
	}
}
```

3 Utilisation de JUnit 5

3.1 Création de la classe de test

Afin de créer un test sur une classe existante, il faut tout simplement créer une nouvelle classe que l'on nommera du même nom que la classe testée (dans notre cas : `Exemple`), suivi de `Test`.

```
public class ExempleTest {…}
```

3.2 Création des méthodes de test unitaire

Par convention, les différentes méthodes de tests commenceront par le mot « `test` » (en minuscule), suivi du nom que l'on désire.

 Cette règle n'est plus obligatoire depuis la version 4 de JUnit : Une méthode de test peut avoir n'importe quel nom !

En revanche, le tag « **`@Test`** » doit être présent en dessus de la méthode.

Dans tous les cas, la méthode doit être déclarée `public` et ne doit rien renvoyer (`void`). Voici un exemple permettant de tester la méthode `factoriel` de la classe « `Exemple` ».

```
public class ExempleTest etc..

private Exemple exemple = new Exemple() ;

@Test
public void testFactoriel() {
    exemple.factoriel(4);
    assertEquals(24, exemple.getResult());  // assertEquals: voir ci-dessous
}
}
```

3.3 La classe `Assertions`

Comme vous pouvez le voir dans l'exemple ci-dessus, la méthode **`assertEquals`** a été utilisée afin de créer le test unitaire opérant la comparaison des 2 nombres. Il existe diverses méthodes dans la classe «`org.junit.jupiter.api.Assertions`» qui vont être décrites ci-dessous.

http://junit.org/junit5/docs/current/api/org/junit/jupiter/api/Assertions.html

- **`assertEquals`**
  Permet de tester si deux types primitifs sont égaux (boolean, byte, char, double, float, int, long, short). L'égalité de deux objets peut être testée également (attention, ce n'est pas un test sur la référence). Pour les double et les float, il est possible de spécifier un delta, pour lequel le test d'égalité passera quand même.
- **`assertFalse`** et **`assertTrue`**
  Permet de tester une condition booléenne.
- **`assertNotNull`** et **`assertNull`**
  Permet de tester si une référence est (non) nulle.
- **`assertNotSame`** et **`assertSame`**
  Permet de tester si deux « Object » (Java) se réfèrent ou non au même objet.
- **`fail`**
  Permet de faire échouer sans condition. En cas d'utilisation de **`fail`**, il est conseillé de faire figurer un message indiquant pourquoi le test à échouer.

3.4 Méthode de début et de fin

Une grande partie du code permettant la réalisation d'un test unitaire sert à établir les conditions d'exécution du test. Il se peut qu'au sein d'une même classe de test, les différentes méthodes de tests aient besoin d'une initialisation commune. (Par exemple l'ouverture d'un fichier). Dans la même optique, elles ont peut-être également besoin d'une procédure de fin commune (Par exemple la fermeture du fichier).

Le framework JUnit fournit ces deux méthodes sous les noms **`setUp()`** et **`tearDown()`**. Par défaut, elles sont lancées automatiquement au début et à la fin de chaque test unitaire. Libre à vous de les implémenter selon vos besoins.

Depuis la version 5 de JUnit il faut rajouter le tag **`@BeforeEach`** devant la méthode **`setUp()`** et **`@AfterEach`** devant **`tearDown()`** comme dans l'exemple qui suit :

```
private static int numero = 0;

@BeforeEach
public void setUp() throws Exception {
        numero++;
        System.out.println("Le test numéro "+numero+" a commencé");
}

@AfterEach
public void tearDown() throws Exception {
        System.out.println("Le test numéro "+numero+" est terminé");
}
```

 Les tags **`@BeforeEach`** et **`@AfterEach`** sont obligatoires !

 En revanche, les méthodes de début et de fin n'ont plus besoin de s'appeler `setUp` ou `tearDown`.

3.5 Méthodes pour initialiser/Finaliser une série de tests

Comme on l'a vu précédemment, les méthodes annotées par **`@BeforeEach`** et **`@AfterEach`** permettent d'encadrer **chacun** des tests unitaires.

Si on lance toute une série de tests unitaires, exécutés les uns à la suite des autres, Il est encore possible de prévoir un code d'initialisation qui ne sera opéré qu'une seule fois, avant le début de la série de tests.

- Cette méthode sera annotée par **`@BeforeAll`** et pourra s'appeler comme on le désire
- Il devra s'agir d'une méthode statique (`static void`)

De manière identique, on peut écrire un code qui sera exécuté une seule fois, à la fin de la série des tests unitaires.

- Cette méthode sera annotée par **`@AfterAll`** et pourra s'appeler comme on le désire
- Il devra s'agir d'une méthode statique (`static void`)

4 Mise en place sous IntelliJ

Nous allons créer un **projet Maven**.

Maven est un outil de gestion de projets Java qui offre les avantages suivants :

- La génération d'une arborescence standard du code source.

- La possibilité de configurer les bibliothèques nécessaires au projet, puis d'opérer leur téléchargement et leur mise à jour avec **une gestion automatique des dépendances entre librairies :**

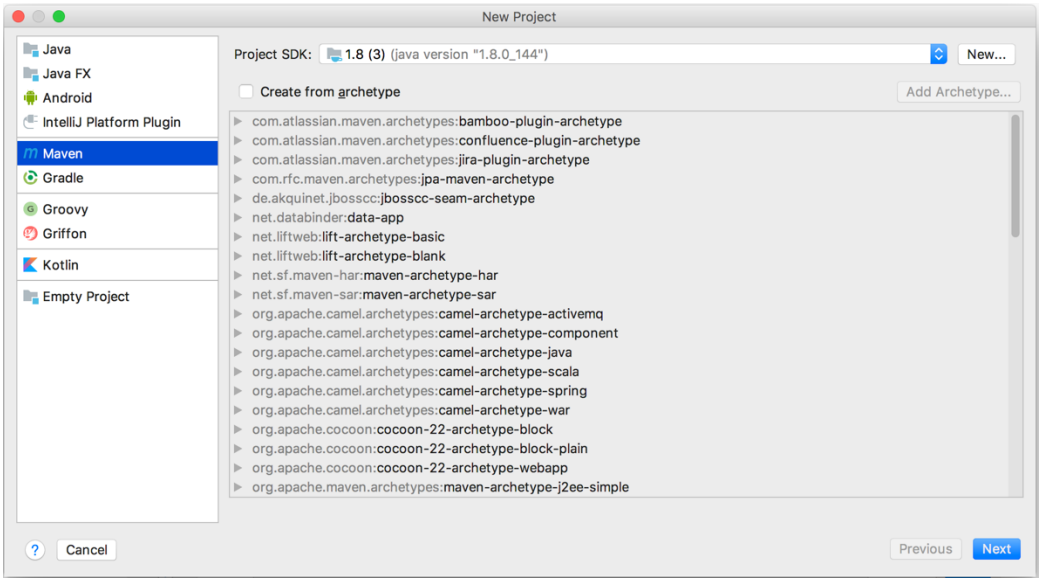
  Si nous désirons utiliser la librairie X et que cette dernière nécessite l'installation d'une librairie Y, cette dernière sera automatiquement installée.

- Une mise en œuvre très aisée des tests unitaires JUnit

L'utilisation de Maven nécessite une connexion internet.

Créer le projet Maven

`New project >`
.. Choisir Maven..

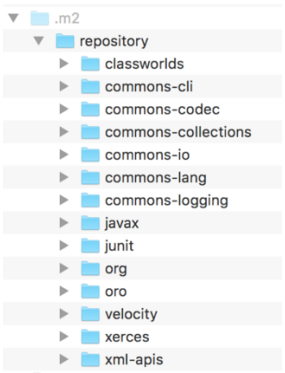


A la création du projet, Maven se connecte au dépôt central http://repo.maven.apache.org/ et commence à télécharger un grand nombre de fichiers **POM** (**P**roject **O**bject **M**odel) et JAR (Java ARchiver) qui lui seront utiles de manière générale.

Par défaut, tous ces fichiers sont stockés par défaut dans votre dossier « `Mes Documents` » dans un répertoire nommé « **`.m2`** ».

Par la suite, à la prochaine utilisation, Maven n'aura plus besoin de télécharger ces fichiers.

Vous pouvez consulter le dossier **`.m2`** :



Vous allez créer votre projet Maven sans vous appuyer sur un quelconque **archétype**, à savoir sans vous appuyer sur un squelette de programme pré-fabriqué.

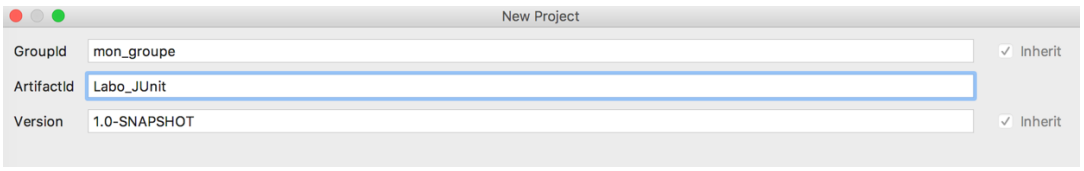
- Ne pas cocher la case « Create from archetype »

- Appuyer simplement sur Next

L'interface suivante vous propose :

- **De spécifier l'identifiant de votre groupe.**
  Saisissez ce que vous voulez.
  Si on s'appuie sur certains archétypes comme l'archétype quickstart d'Apache, cela aura pour effet de créer votre code source dans un paquetage qui portera le nom du groupe que vous aurez saisi.

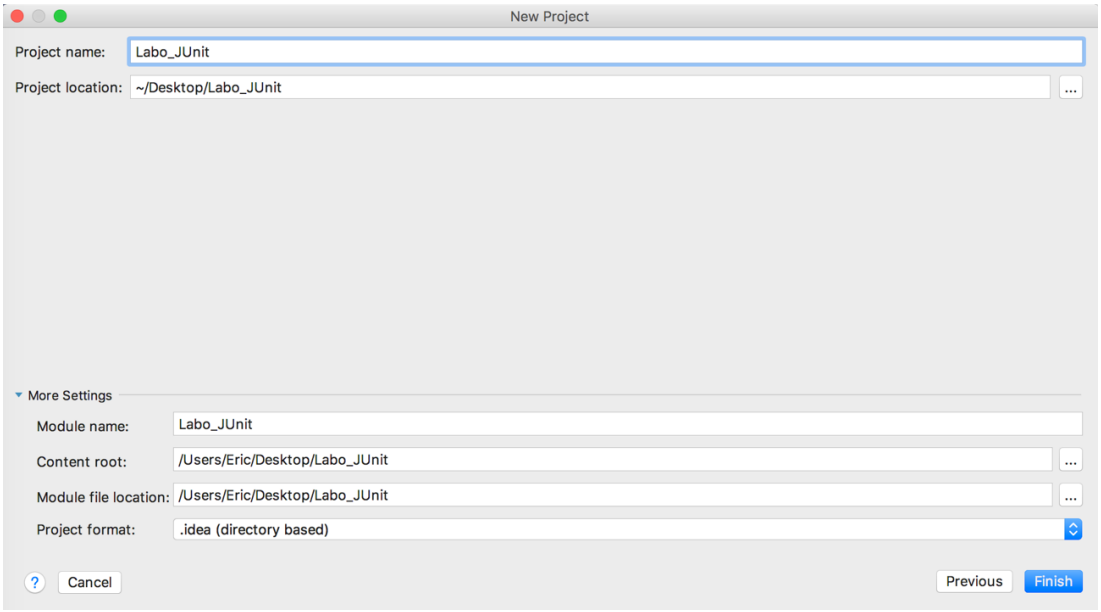
  C'est ignoré dans notre cas.
- **L'identifiant de votre projet (ArtifactId)**
  Ce que vous saisissez sera proposé par la suite comme nom donné à votre projet.
- **La version de votre projet**
  Par défaut, à chaque fois que l'on régénère son projet, le no de version est incrémenté automatiquement d'une unité.
  En phase de développement, on préfère évidemment que le no de version soit stable et ne soit pas incrémenté. A cette fin, il suffit de donner à cette version la valeur "**`1.0-SNAPSHOT`**".



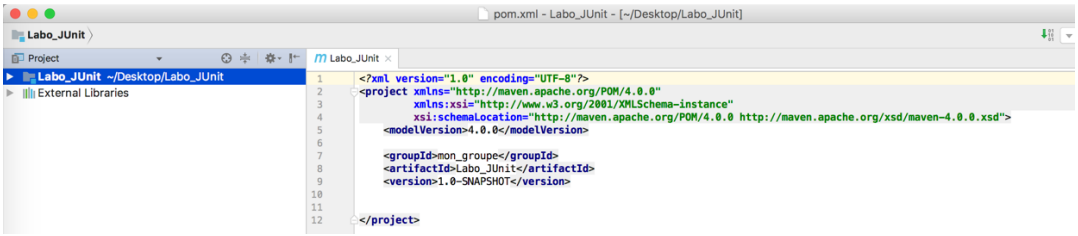
Continuez avec Next

Pour finir, la dernière interface vous propose de donner un nom à votre projet et de spécifier l'endroit où il sera stocké.

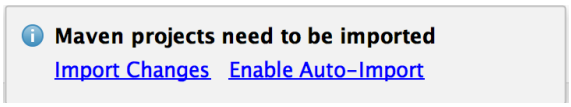
- **Nom du projet** : Intelllj vous propose par défaut l'ArtifactId que vous avez déjà spécifié, c'est une bonne idée
- **Localisation** : Intelllj vous propose par défaut de créer le nouveau projet sur le bureau. A vous de choisir



Ca prend un peu de temps.. Puis vous entrez dans votre projet Maven.



Le cas échéant, si Intelllj vous le propose, opérez une mise à jour de certaines informations (Import changes):

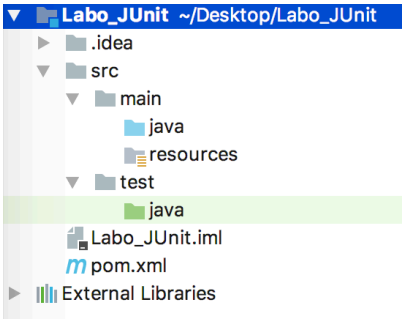


S'affiche à droite le contenu du fichier **pom.xml** :

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">

    <modelVersion>4.0.0</modelVersion>
    <groupId>mon_groupe</groupId>
    <artifactId>Labo_JUnit</artifactId>
    <version>1.0-SNAPSHOT</version>

</project>
```

Le fichier **`pom.xml`** constitue le cœur de la configuration Maven des différentes librairies qui seront nécessaires à votre projet. Nous verrons un peu plus loin comment installer la librairie JUnit 5.

Petit coup d'œil sur l'arborescence de votre projet

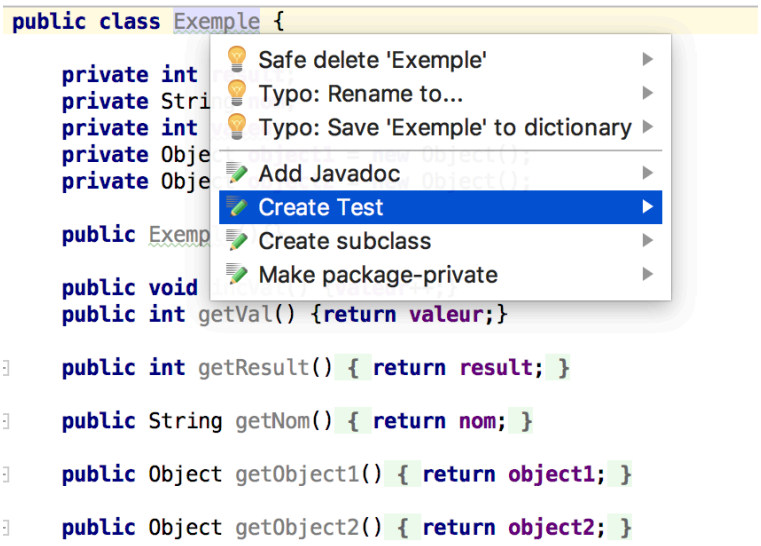


- Les fichiers source Java seront placés dans le dossier `src/main/java`
- Les tests unitaires JUnit seront placés dans le dossier `src/test/java`
- Les fichiers annexes de votre application (images, données XML, etc…) seront placés dans le dossier `src/main/resources`

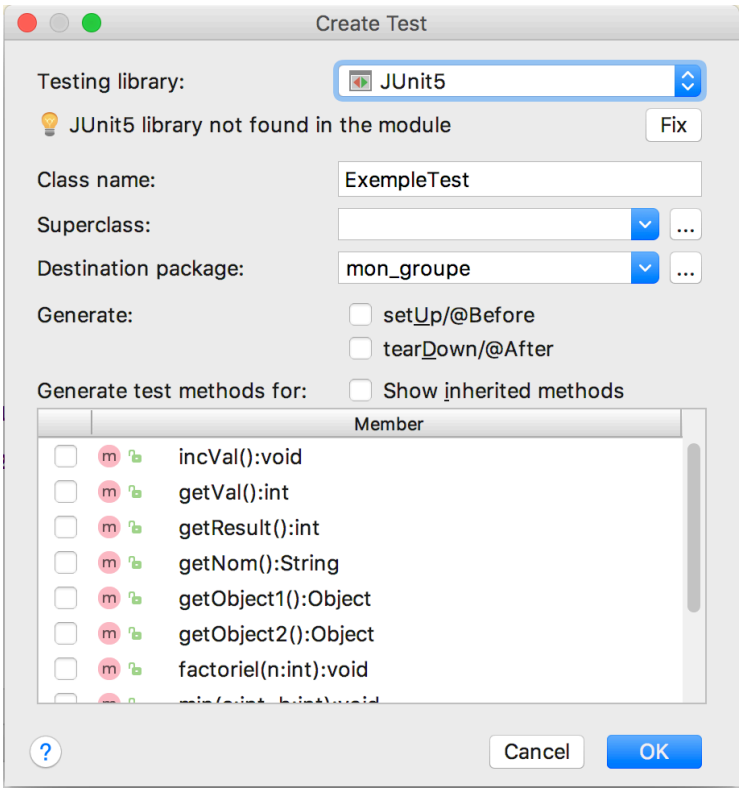
Placer le fichier `Exemple.java` (confié par le professeur) dans le dossier `src/main/java`

Créer le test unitaire pour la classe `Exemple`

1. Glisser la souris sur le nom de la classe
2. Presser ALT-Enter



⚠ Choisissez la librairie de test JUnit5 !



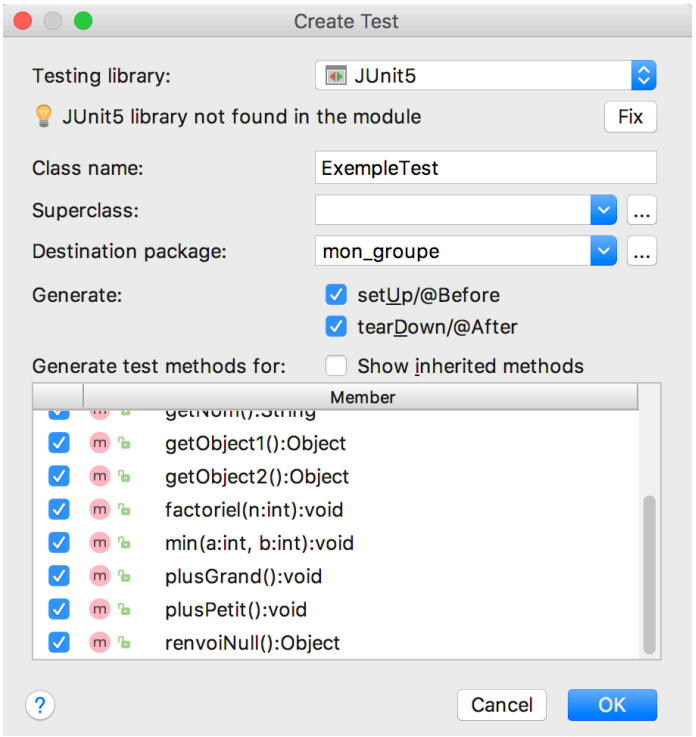
Vous remarquerez dans l'image ci-dessus l'avertissement suivant « JUnit 5 Library not found »

⚠ N'appuyez pas sur le bouton **Fix** !!

Ce qui aurait pour effet de régler le problème.. 

Dans le cadre de ce tutorial, nous allons plutôt opérer en agissant par le biais du fichier de configuration pom.xml, **voir plus loin.**

Puis configurez votre classe de test, comme ci-dessous, puis appuyez sur Ok.



Ok

Le système génère automatiquement votre classe de test `ExempleTest` :

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

class ExempleTest {
    @org.junit.jupiter.api.BeforeEach
    void setUp() {
    }

    @org.junit.jupiter.api.AfterEach
    void tearDown() {
    }

    @org.junit.jupiter.api.Test
    void incVal() {
    }

    @org.junit.jupiter.api.Test
    void getVal() {
    }

    @org.junit.jupiter.api.Test
    void getResult() {
    }

    @org.junit.jupiter.api.Test
    void getNom() {
    }
```

```
    @org.junit.jupiter.api.Test
    void getObject1() {
    }

    @org.junit.jupiter.api.Test
    void getObject2() {
    }

    @org.junit.jupiter.api.Test
    void factoriel() {
    }

    @org.junit.jupiter.api.Test
    void min() {
    }

    @org.junit.jupiter.api.Test
    void plusGrand() {
    }

    @org.junit.jupiter.api.Test
    void plusPetit() {
    }

    @org.junit.jupiter.api.Test
    void renvoiNull() {
    }

}
```

Hello World avec JUnit

Complétez les 3 premières méthodes avec les sorties suivantes :

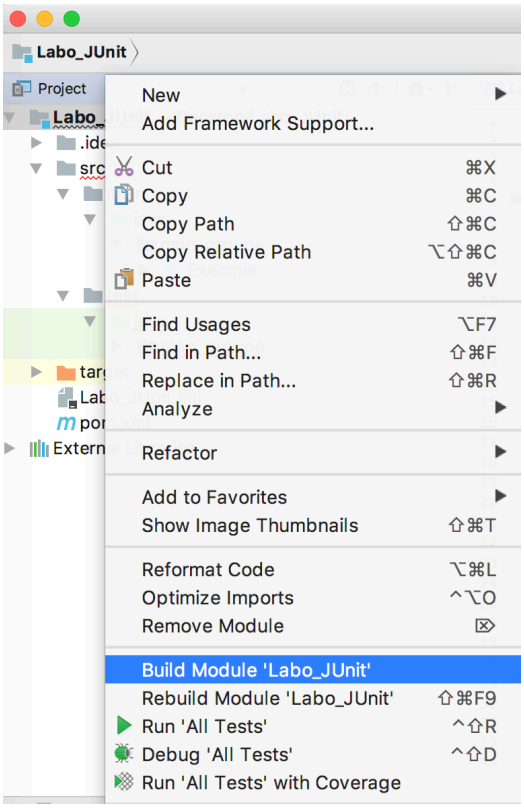
```
class ExempleTest {

    @ org.junit.jupiter.api.BeforeEach
    void setUp() {
        System.out.println("Exécution de la méthode 'setUp'");
    }

    @org.junit.jupiter.api.AfterEach
    void tearDown() {
        System.out.println("Exécution de la méthode 'tearDown'");
    }

    @org.junit.jupiter.api.Test
    void incVal() {
        System.out.println("Test de la méthode 'incVal'");
    }

    @org.junit.jupiter.api.Test
    void getVal() {
    }
```

:

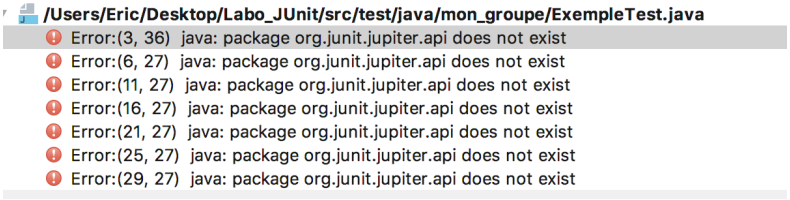
Contrôlez si tout est en place : Construisez le module de votre projet

Dans la fenêtre de gauche (Project Window tool),

- Click droit sur votre projet
- puis « Build Module nomDeVotreProjet »



Des erreurs affichées… Il manque la librairie JUnit 5..



Nous allons installer la librairie JUnit5

Rajouter dans **pom.xml** le contenu suivant:

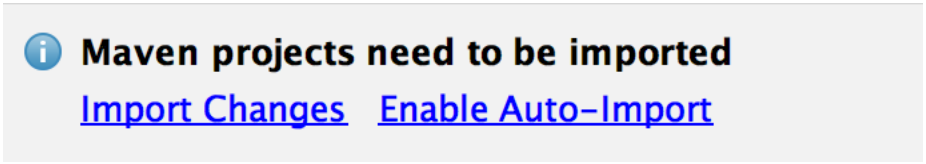
```
<dependencies>
<dependencies>
  <dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-engine</artifactId>
    <version>5.0.0</version>
    <scope>test</scope>
  </dependency>
</dependencies>
```


Une librairie est désignée par un artifact, qui est le nom qui permet de désigner la librairie indépendamment de sa version.

Nous allons utiliser la librairie **`junit-jupiter-engine`** (librairie JUnit 5)

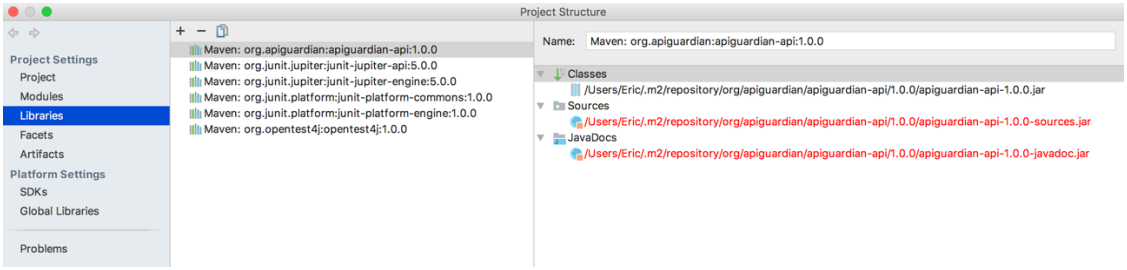
Le tag `<scope>test</scope>` est une indication pour Maven : Les fichiers de test JUnit sont placés dans le dossier `test`.

Puis demandez à Maven d'importer la librairie (**`import changes`**)



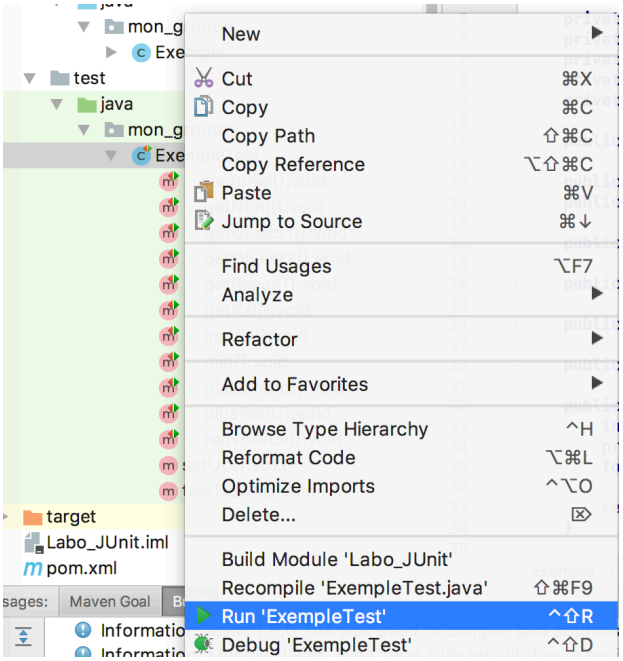

Vous pouvez contrôler la liste de toutes les librairies dépendantes que Maven a dû importer au passage :

Menu **`File > Project Structure`**

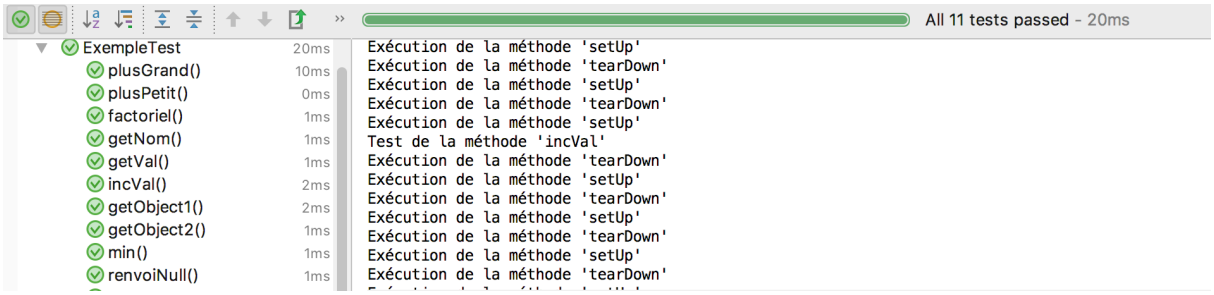



Vous pouvez maintenant contrôler que la construction de votre projet compile normalement !

Exécuter la série de tests avec un clic droit sur la classe `ExempleTest`



Vous obtenez quelque chose ressemblant à :



5 Complétez la classe ExempleTest

La classe ci-dessous permet de donner un aperçu des différents tests unitaires décrits plus haut. Complétez la classe générée par l'environnement JUnit et testez le résultat.

```
import org.junit.jupiter.api.AfterEach;
import org.junit.jupiter.api.BeforeAll;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
```

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

public class ExempleTest {

    private static int numero = 0;

    private Exemple exemple = new Exemple();

    private static boolean erreur;

    @BeforeAll
    public static void setUpBeforeClass() throws Exception {
        erreur = true;
    }
    @BeforeEach
    public void setUp() throws Exception {
        numero++;
        System.out.println("Le test numéro "+numero+" a commencé");
    }
    @AfterEach
    public void tearDown() throws Exception {
        System.out.println("Le test numéro "+numero+" est terminé");
        System.out.println("Etat de la valeur: "+ exemple.getVal());
        exemple.incVal();
    }
    @Test
    public void testFactoriel() {
        exemple.factoriel(4);
        assertEquals(24, exemple.getResult());
    }
    @Test
    public void testMin() {
        exemple.min(5, 6);
        assert(exemple.getResult()==5);
    }

    @Test
    public void testPlusGrand() {
        exemple.plusGrand();
        assertTrue(exemple.getNom().equals("Toto"));
    }
    @Test
    public void testPlusPetit() {
        exemple.plusPetit();
        assertFalse(exemple.getNom().equals("Toto"));
    }
    @Test
    public void testRenvoiNull() {
        assertNull(exemple.renvoiNull());

    }
    @Test
    public void testRenvoiNonNull() {
        assertNotNull(exemple.getObject1());
    }
    @Test
    public void testMemeObject() {
        assertSame(exemple.getObject1(),exemple.getObject1());
    }
    @Test
    public void testObjectDifferent() {
        assertNotSame(exemple.getObject1(),exemple.getObject2());
    }
    @Test
    public void testEchec() {
        if (erreur) fail("Echec");
```

```
    }
}
```

Essayez !

Et vous constaterez (manipulation de la variable « `valeur` » de l'objet « `exemple` ») que :

Les tests JUNIT sont **stateless**! (sans état)

A chaque test **une nouvelle instance de la classe `Test` est créée**!

- ⇨ L'état des variables d'instance éventuelles déclarées dans la classe de test n'est donc pas sauvegardé d'un test à l'autre !!
- ⇨ Les informations générales à tous les tests doivent être enregistrés dans des variables de classe

En conséquence :

 Les tests sont tous indépendants les uns des autres...

- ⇨ Peuvent s'exécuter dans n'importe quel ordre
- ⇨ Peuvent même être exécutés en parallèle!!

6 But du laboratoire

En annexe de la donnée, vous trouverez les classes « `Personne.java` » et « `Annuaire.java` ».

La première classe permet de représenter de manière très simple une personne. La personne est caractérisée par un nom, un prénom, une année de naissance et une adresse e-mail. Diverses méthodes existent afin de modifier ou récupérer ces informations.

La deuxième classe regroupe un ensemble de personnes. Elle permet l'ajout et la suppression d'une personne à l'annuaire. Elle fournit également les méthodes permettant de récupérer les données relatives à une personne.

L'adresse e-mail permet de différencier les différentes personnes étant donné qu'elle est unique.

Nous vous demandons de créer une ou deux classes de tests unitaires sur ces deux classes. Vous devez utiliser au moins une fois chacune des méthodes de la classe « `Assert` » vu plus haut ainsi que les méthodes de début et de fin de classe et de test (« `setUp()` » et « `tearDown()` ») .