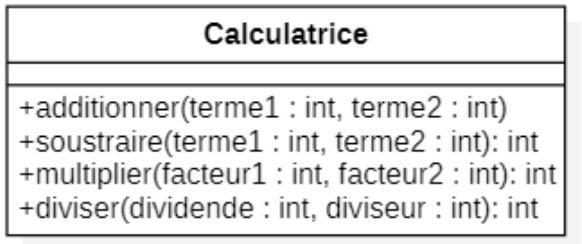
# TP : Premiers tests unitaires avec `JUnit`

## Exercice 1 : La `calculatrice` à la rescousse des premiers tests …

Commencez par créer un projet Java que vous appellerez **`calculatrice`**.
Comme d'habitude n'oubliez pas :

- le ***Don't create module***
- de fermer tous les projets non relatifs à ce projet (***close unrelated project***)

Voici la classe que vous allez implémenter (et tester) dans cet exercice.




***Rappel sur la qualité de code :*** Quand on débute un projet,
il faut choisir si le code devra être écrit en anglais ou si le code devra être écrit en français
car ***attention, écrire le code en franglais est un code smell (mauvaise odeur)***
***qu'il faut absolument éviter !!!***

Pour ce projet, nous vous demandons d'écrire **tout le code métier en français** !
Commencez donc par créer la classe **`Calculatrice`**.

## Focus sur la méthode `additionner`

*(écrire un premier test unitaire qui passe)*

### 1. Implémentation de la méthode `additionner` (code de production)

Conformément au diagramme de classes précédent, la méthode **`additionner`** ne devra manipuler que des ***types primitifs*** entiers (***int***)

Pour cet exercice, l'implémentation de la méthode **`additionner`** sera une implémentation minimale similaire à celle présentée dans le cours sur les tests qui est disponible sur
https://github.com/iblasquez/enseignement-but1-developpement

Une fois, l'implémentation terminée, Il est nécessaire de vérifier le *bon* comportement de la méthode **`additionner`** en implémentant quelque(s) test(s) automatisé(s) .

## 2. Assurer la cohérence du code : Séparation du code de production et du code de tests (dans 2 *sources folders* différents)

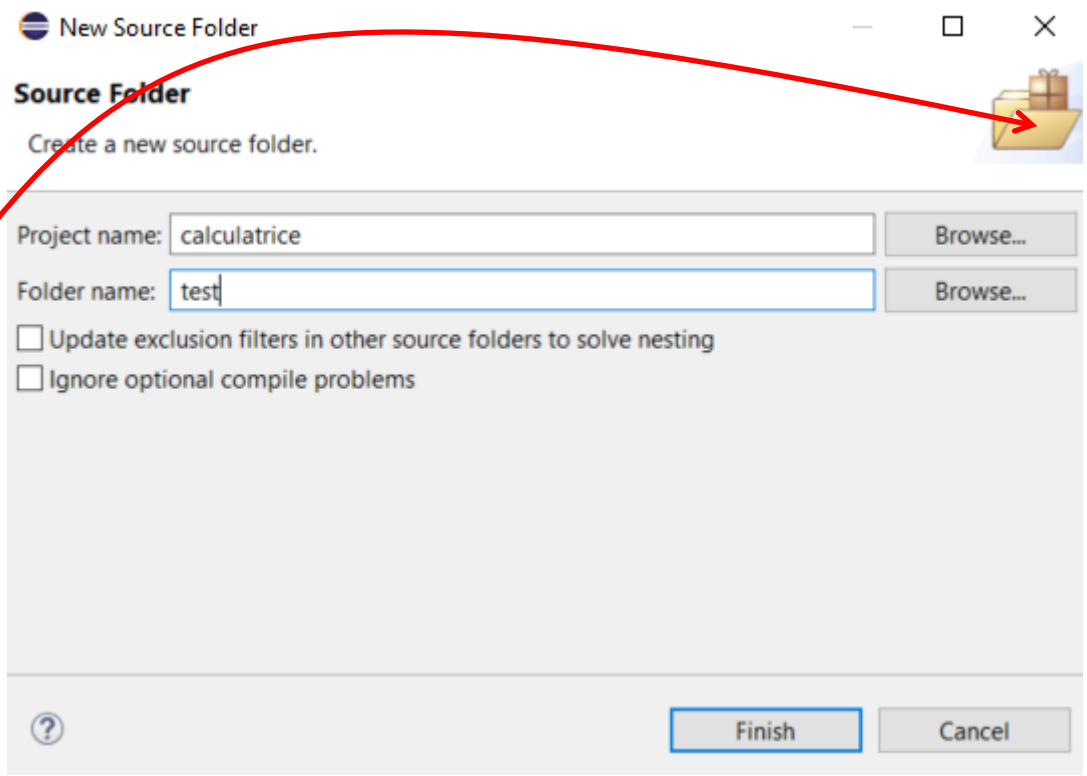
Les bonnes pratiques recommandent les conventions suivantes :
⇨ Le **code de production** doit être écrit dans un répertoire source (source folder) appelé **`src`**
⇨ Le **code de tests** doit être écrit dans un répertoire source appelé **`test`**

**Il faut donc commencer par créer un répertoire source (source folder) : test**

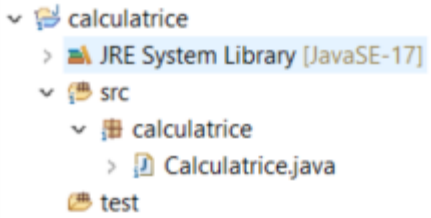
Pour cela, placez-vous sur le projet **`calculatrice`** dans le vue **`Package Explorer`**,

clic droit puis **`New → Source Folder`** (et non simplement **`Folder`**, vérifiez bien que vous ayez l'icône avec le répertoire et le petit cadeau….)

Choisissez **`test`** comme **`Folder name`** et cliquez sur Finish.




Vérifiez, via la vue **`Package Explorer`**, que le source folder **`test`** a bien été ajouté.
Si tel est le cas, vous devriez vous retrouvez avec l'architecture suivante :

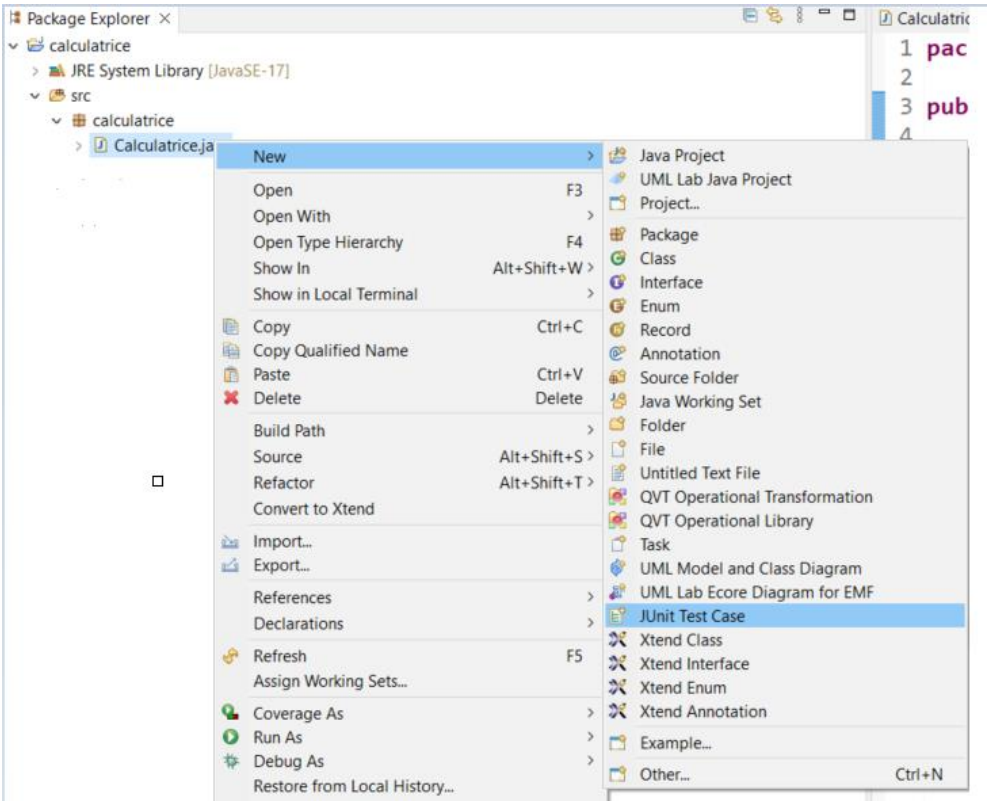



***Remarque :*** Notez la différence d'icône et la différence dans la hiérarchie entre un ***source folder*** et un ***package*** : ce n'est pas le même concept.
A l'intérieur d'un source folder, le code peut être organisé en différents packages !!!

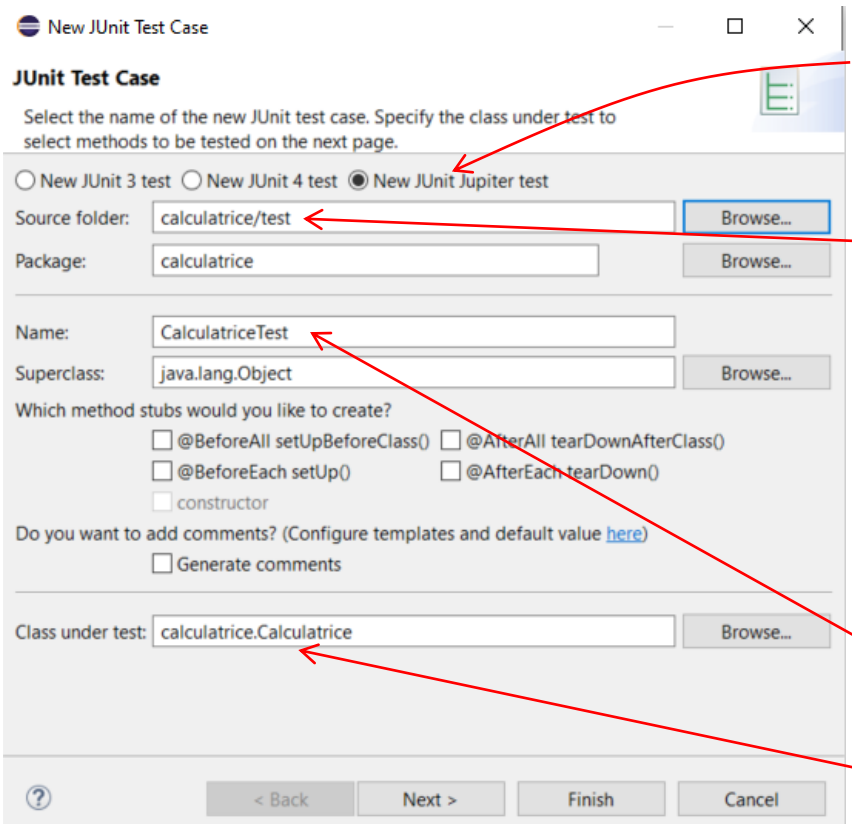
## 3. Vérification du *bon* comportement de la méthode `additionner` (code de test)

### a. Création de la classe de test : `CalculatriceTest`

Dans la vue **`Package Explorer`**, placez-vous sur le fichier **`Calculatrice.java`** puis clic droit afin de pouvoir sélectionner : **`New → JUnitTestCase`**




**La fenêtre de l'assistant New JUnit Test Case devrait s'ouvrir**

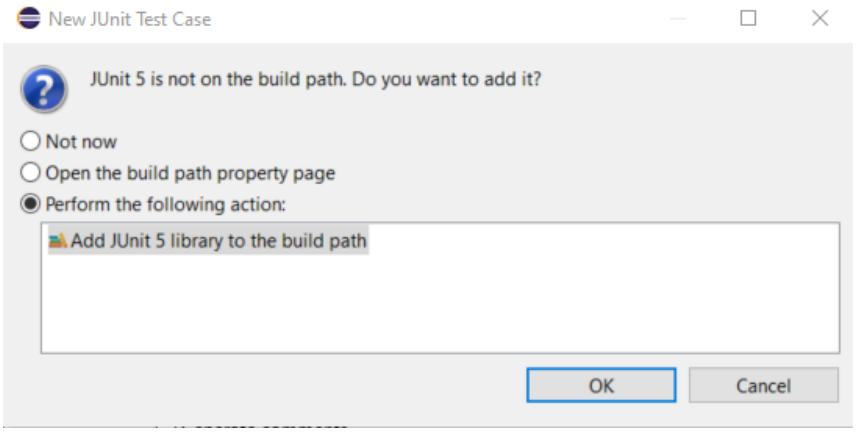



⇨ Pour utiliser **`JUnit5`**, vous devez sélectionner :
**`New JUnit Jupiter Test`**

⇨ Cliquez sur **`Browse`** pour sélectionner le **`Source folder test`** que vous venez de créer.

⇨ Si vous créez la classe de test depuis la classe métier, le nom de la classe de test s'est directement inscrit en respectant la convention : **`NomClasseATesterTest`**
Sinon, vous devez écrire le nom de la classe à tester en respectant cette convention.
Dans notre cas, la classe de test doit s'appeler : **`CalculatriceTest`**

⇨La classe sous test (notre **S**ystem **U**nder **T**est appelé plus communément **SUT**) a également été complétée
Vous pouvez alors appuyer sur **`Finish`**




Si la classe de test que vous êtes en train de créer est la première classe de test de votre projet, l'IDE vous demandera d'importer d'ajouter le *framework* (bibliothèque) **JUnit** dans le classpath du projet de la manière suivante.

Cliquez sur **OK** pour voir apparaître la classe **CalculatriceTest** dans l'éditeur de code !

Lors de sa création, une classe de test s'ouvrira toujours en proposant le code suivant c.-à-d. une méthode de test prête à être implémentée avec les imports qui vont bien.

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

class CalculatriceTest {

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

***Astuce :*** Commencez par jeter un petit coup d'œil aux deux instructions **import** pour vérifier qu'elles commencent bien par : `import org.junit.jupiter.api`
Si c'est le cas, vous êtes bien en **JUnit5** et vous pouvez commencer à écrire votre première méthode de test.

### b. Implémentation de votre première méthode de test :

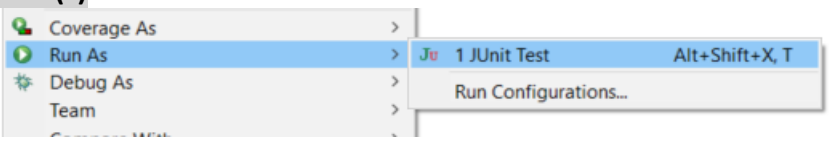
➔ Renommez la méthode **test()** par **doitAdditionerDeuxEntiers()** qui vous indique de manière explicite ce qu'est censé vérifier le test. Pour chaque méthode de test, prenez bien le temps de nommer votre méthode pour illustrer au mieux le comportement à tester.

➔ En vous inspirant du cours, implémentez ce premier test qui permet de tester le comportement d'une addition (de deux nombres positifs par exemple) en respectant le **pattern AAA**

➔ N'oubliez pas de vérifier que votre test se termine bien par une assertion !

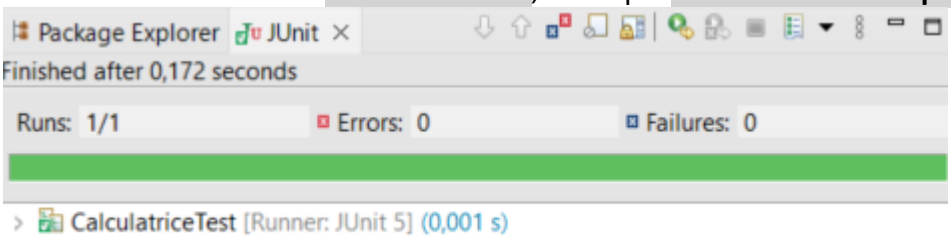
### c. Exécution/Lancement du(des) test(s)

Pour lancer le test, placez-vous sur le fichier implémentant le(s) test(s) (**`CalculatriceTest`**),
puis à l'aide d'un clic droit,




sélectionnez **`Run as → JUnit Test`** (que vous pouvez également retrouvez à partir du menu **`Run`** dans la barre du haut des menus)

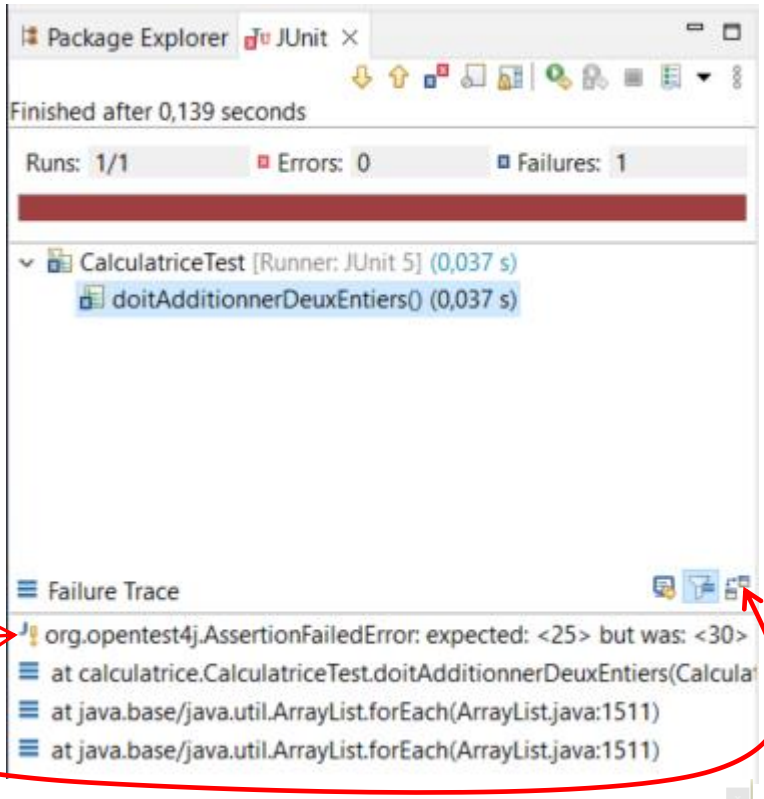
Une vue `JUnit` s'ouvre (normalement à côté de la vue Package Explorer).
Si cette vue vous affiche **une barre verte**, c'est que **votre test est bien passé !**




Modifiez l'étape **`Assert`** (assertion) pour faire échouer le test c.-à-d. faites en sorte que le résultat attendu (**`expected`**) et le résultat calculé (**`actual`**) soient différents.

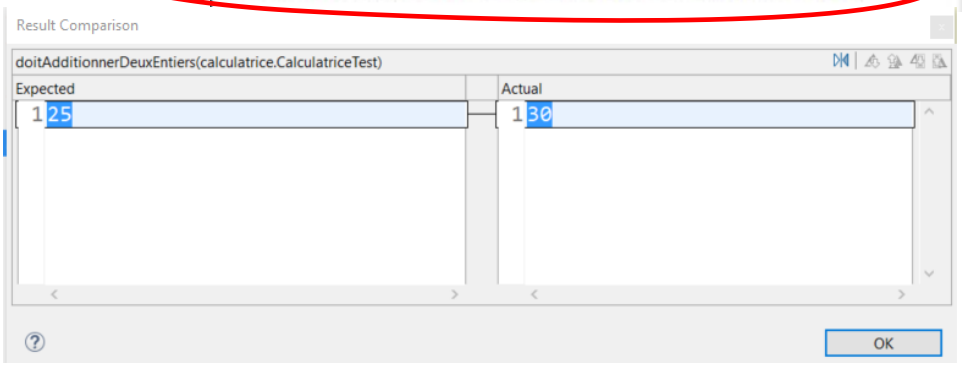
Relancez le test. Cette fois-ci la :

⇨ la vue **`JUnit`** affiche **une barre rouge** :
on dit que le **test échoue !**

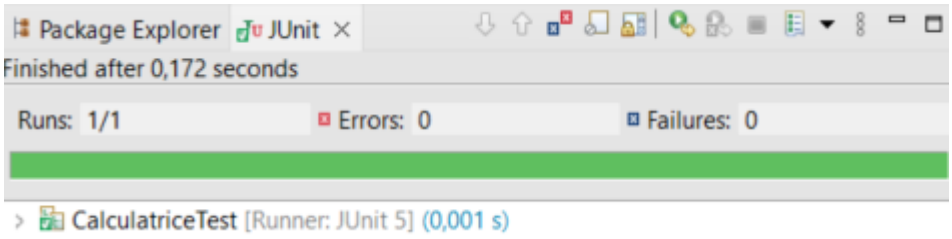



⇨ la rubrique **`Failure Trace`** est complétée :
La première ligne vous indique pourquoi l'assertion a échouée : différence entre la valeur attendue **`expected :<…>`** et la valeur calculée **`but was <…>`**

⇨ En cliquant sur l'icône **`Failure Comparaison`** la plus à droite) une fenêtre, bien plus visuelle, nous permet de comparer plus en détails ces deux valeurs (ce sera par exemple intéressant pour les tests qui échoueront autour des collections).
Cliquez sur **OK** pour fermer cette fenêtre.



**Pour continuer, faites en sorte que votre assertion soit correcte et que votre test repasse AU VERT !!!**




# Focus sur la méthode `soustraire`

*(visualiser la couverture de code par les tests)*

## 4. Implémentation de la méthode `Soustraire` (code de production)

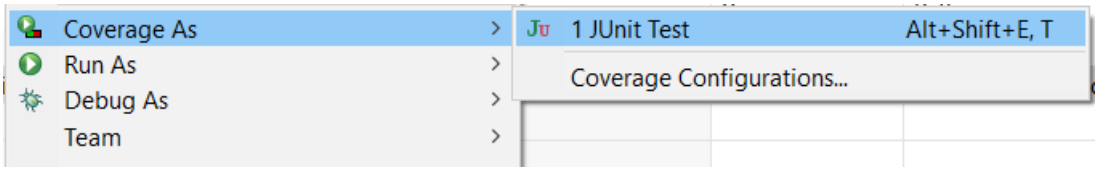
Conformément au diagrammes de classe précédent, implémenter dans le code de production (classe **`Calculatrice`** dans **`src`**), la méthode **`soustraire`** qui manipule des entiers (implémentation minimale similaire à celle d'**`additionner`**)

## 5. S'informer sur la couverture de code par les tests

Avoir des tests c'est bien, connaître la couverture de code par ces tests peut parfois s'avérer intéressant, pour savoir quels comportements sont bien couverts par les tests…

### a. Demande d'exécution de la couverture de code

Pour lancer la couverture de code par les tests, placez-vous sur une classe de tests (**`CalculatriceTest`**), puis à l'aide d'un clic droit, sélectionnez **`Coverage As→ JUnit Test`**




Une fois la couverture lancée, l'IDE va :

- d'une part générer une nouvelle vue **Coverage** disponible en bas de votre fenêtre de travail
- et d'autre part surligner votre code selon un code couleur.

### b. Zoom sur la vue Coverage (Visualisation de la couverture de code)

La vue **Coverage** permet d'obtenir un **taux de couverture** de code par les tests qui peut être plus ou moins détaillé

*Pour détailler, il suffit de **déplier au fur et à mesure chaque élément***

⇨ Visualisation du couverture de couverture **des *source folders* et *packages***

Problems | Javadoc | Declaration | Coverage

| Element | Coverage | Covered Instru... | Missed Instruct... | Total Instructio... |
|---|---|---|---|---|
| calculatrice | 87,1 % | 27 | 4 | 31 |
| src | 63,6 % | 7 | 4 | 11 |
| calculatrice | 63,6 % | 7 | 4 | 11 |
| test | 100,0 % | 20 | 0 | 20 |
| calculatrice | 100,0 % | 20 | 0 | 20 |

⇨ Visualisation du couverture de couverture jusqu'aux **classes**

Problems | Javadoc | Declaration | Coverage

| Element | Coverage | Covered Instru... | Missed Instruct... | Total Instructio... |
|---|---|---|---|---|
| calculatrice | 87,1 % | 27 | 4 | 31 |
| src | 63,6 % | 7 | 4 | 11 |
| calculatrice | 63,6 % | 7 | 4 | 11 |
| Calculatrice.java | 63,6 % | 7 | 4 | 11 |
| test | 100,0 % | 20 | 0 | 20 |
| calculatrice | 100,0 % | 20 | 0 | 20 |
| CalculatriceTest.java | 100,0 % | 20 | 0 | 20 |

⇨ Visualisation du couverture de couverture jusqu'aux **méthodes**

Problems | Javadoc | Declaration | Coverage

| Element | Coverage | Covered Instru... | Missed Instruct... | Total Instructio... |
|---|---|---|---|---|
| calculatrice | 87,1 % | 27 | 4 | 31 |
| src | 63,6 % | 7 | 4 | 11 |
| calculatrice | 63,6 % | 7 | 4 | 11 |
| Calculatrice.java | 63,6 % | 7 | 4 | 11 |
| Calculatrice | 63,6 % | 7 | 4 | 11 |
| soustraire(int, int) | 0,0 % | 0 | 4 | 4 |
| additionner(int, int) | 100,0 % | 4 | 0 | 4 |
| test | 100,0 % | 20 | 0 | 20 |
| calculatrice | 100,0 % | 20 | 0 | 20 |
| CalculatriceTest.java | 100,0 % | 20 | 0 | 20 |
| CalculatriceTest | 100,0 % | 20 | 0 | 20 |
| doitAdditionnerDeuxEr | 100,0 % | 17 | 0 | 17 |

Nous constatons que la méthode **`additionner`** est couverte à **100%** (grâce au test que nous avons écrit précédemment), alors que la méthode **`soustraire`** est couverte à **0%** : normal, nous n'avons encore écrit aucun test pour cette méthode 😉

### c. Zoom sur la coloration du code ( Visualisation de la couverture de code)

La couverture est également **visible directement dans le code** source java grâce à **un surlignage du code selon un code couleur**.
Par défaut le code couleur est le suivant :
⇨ un surlignage **VERT** pour montrer une **couverture totale** de la ligne de code par les tests unitaires
⇨ un surlignage **JAUNE** pour montrer une **couverture partielle** de la ligne de code par les tests unitaires
⇨ un surlignage **ROUGE** pour montrer une **absence de couverture** de la ligne de code par les tests unitaires.

***Remarque :*** Des **diamants colorés** pourront également aussi apparaître à gauche de certaines lignes de code, celles qui seront relatives à une instruction de décision (**if, switch, for,…**). Le code couleur des diamants est identique à celui des lignes : vert pour une couverture totale, jaune pour une couverture partielle et rouge pour une absence de couverture.

***Remarque*** *:* Le code couleur peut être personnalisé à partir du menu **`Windows->preferences->General->Editors->Text Editors ->Annotations`** et en modifiant les couleurs de **`Full Coverage`**, **`Partial Coverage`** et **`No Coverage`**.
Nous ne vous conseillons pas de modifie ces couleurs, à moins que vous soyez daltonien, car **rouge et vert sont les couleurs par convention des bonnes pratiques !**

Si vous consultez votre classe **`Calculatrice`**, vous constaterez que la couverture de similaire à la suivante :

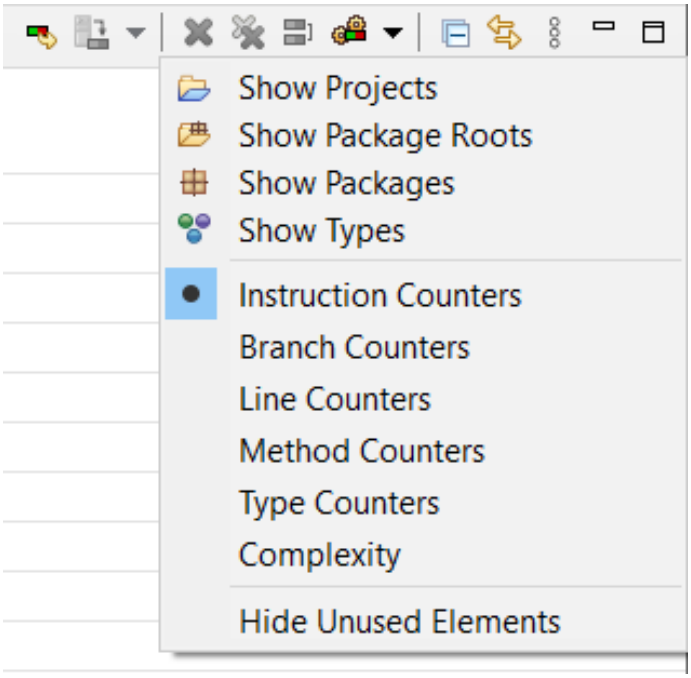
```
package calculatrice;

public class Calculatrice {

    public int additionner(int terme1, int terme2) {
        return terme1 + terme2;
    }

    public int soustraire(int terme1, int terme2) {
        return terme1 - terme2;
    }

}
```

⇨ **En VERT**, les instructions : `public class Calculatrice {`
et `return terme1 + terme2;`

⇨ **En ROUGE**, l'instruction : `return terme1 - terme2;`

**Par défaut, la couverture de code se fait sur la couverture des instructions (`Instruction Counters`)**

***Pour information :***
Il est possible de connaître d'autres couvertures de code (voir cours).
Pour cela la **vue `Coverage`,** vous propose en haut à droite un certains nombre d'options.
Cliquez sur l'icône des **trois petits points verticaux (`View Menu`).**




Par défaut, la couverture de code est **`Instruction Counters`**
Essayez d'autres couvertures de code en cliquant sur `Branch Counters` puis `Line Counters` puis `Methode Counters`….
et visualisez à chaque fois, les pourcentages de couverture de code qui évoluent en conséquence 😉

Pour cette année, nous resterons sur de l'**`Instruction Counters`**.
Vérifiez bien que cette instruction est sélectionnée avant de quitter le menu des trois petits points 😉

### d. Quelques mots sur la couverture de code

Dans un projet de la « vraie » vie, il n'est pas pertinent de viser à tout prix une couverture de 100%, sachant en outre que dans la pratique il est quasiment impossible d'atteindre les 100%.
Une couverture autour de 80% est quant à elle tout à fait raisonnable.

En effet, une couverture élevée ne signifie pas nécessairement une bonne utilisation des tests comme le montre l'image ci-contre extraite du compte twitter de @francesc :

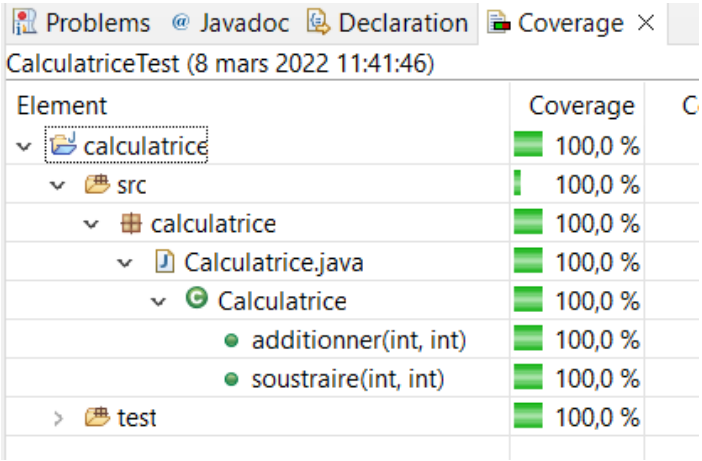
100% code coverage
"@bloerwald: SUCCESS: 26/26 (100%) Tests passed "

**En pratique, il est donc plus important de se focaliser sur la qualité des tests plutôt que sur le taux de couverture à atteindre.**
Le taux de couverture doit juste être considéré comme un moyen d'améliorer la qualité globale du code et des tests pour le valider.
Si vous souhaitez aller plus loin dans cette réflexion, vous pourrez lire tranquillement chez vous les articles suivants :

- https://blog.engineering.publicissapient.fr/2008/02/07/lanalyse-de-couverture-de-code-en-java/
- Un article de Martin Fowler d'où est extraite l'image suivante : https://martinfowler.com/bliki/TestCoverage.html

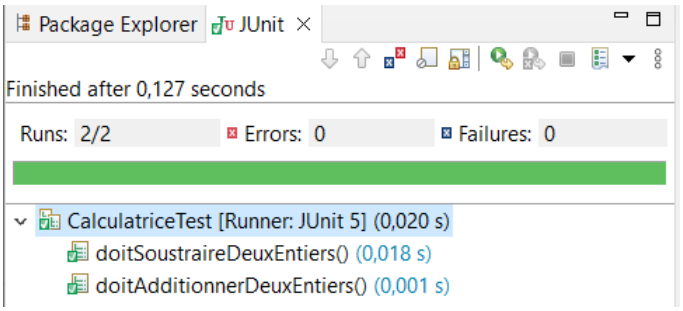

## 6. Vérification du *bon* comportement de la méthode `soustraire` (code de test)

Pour vérifier le *bon* comportement de la méthode **`soustraire`** de la classe **`Calculatrice`**, il suffit d'enrichir la classe **`CalculatriceTest`** autant de test(s) unitaire(s) pertinent(s) et judicieusement nommé(s) qui vous seront nécessaire pour atteindre une couverture de code de 100% sur la méthode soustraire.
Bien sûr, vu la simplicité de l'implémentation de la méthode soustraire, un seul test devrait suffire 😉




A vous de jouer !
N'oubliez pas de vérifier que votre test se termine bien par une assertion !

Notez bien que lorsque vous relancez les tests, tous les tests sont rejoués !!!




D'où l'utilité d'avoir des tests bien nommés pour pouvoir identifier rapidement d'où provient le problème, si vous aviez modifié malencontreusement votre code et un test venait à passer au ROUGE 😉

# Focus sur la méthode `multiplier`

*(mettre en place Infinitest)*

## 7. Implémentation de la méthode `multiplier` (code de production)

## 8. Vérification du *bon* comportement de la méthode `multiplier`(code de test)

En vous inspirant de ce que vous venez de faire précédemment :

- Implémentez de manière minimale la méthode **`multiplier`** (code de production)
- Ajoutez dans la classe **`CalculatriceTest`**, le(s) test(s) nécessaire(s) pour assurer une couverture de 100%

## 9. Mise en place du plug-in Infinest pour tester « en continu »


Les tests permettent de garantir le « bon » comportement du programme. Il est donc recommandé d'exécuter le plus fréquemment possible les tests unitaires afin de vérifier que les changements apportés au code ne *cassent* pas le comportement existant et de pouvoir ainsi modifier son code en toute confiance.

Infinitest est un plugin très pratique s'intégrant au sein de l'IDE (Eclipse ou IntelliJ) qui permet de lancer automatiquement les tests quand un changement est détecté dans le code source c'est-à-dire à chaque fois que vous sauvegardez un fichier modifié.

Avec Infinitest, lorsque vous sauvegardez votre code, les tests sont automatiquement lancés.
Plus besoin de passer par le menu Run As → JUnit pour savoir si les tests continuent à passer AU VERT **un simple CTRL + S** permet de lancer automatiquement tous les tests unitaires du projet, et donc de tester de manière quasi-continue le code...

Pour installer Infinitest, rien de plus simple, rendez-vous sur la page web du projet, tout est expliqué : https://infinitest.github.io/

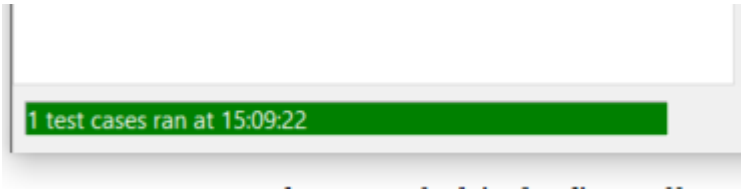
⇨ Installer le plug-in **Infinitest** à partir de l'**Eclipse Market (Help → Eclipse Market)**

⇨Une fois l'IDE redémarré, la barre Infinitest apparaît en bas à gauche de votre IDE.
Si cela n'est pas déjà fait fermé tous les projets autre que **`calculatrice`** (`close Unrelated Projects`).

⇨Faire une petite modification dans la classe de tests **`CalculatriceTest`**, par exemple en choisissant le menu **`Source → Format`** (pour reformatter correctement votre code…pensez à la faire de temps en temps 😉). Si le fichier avait besoin d'être reformaté, une * est apparue, si le fichier n'a pas été modifié à l'issue de ce formatage (pas d'* devant le nom du fichier), mettez un petit espace pour simuler une modification).

Sauvegardez le fichier à l'aide de **CTRL+S.**

**La barre Infinitest relance les tests devrait passer AU VERT !!!**



**Remarques :**

- **Infinitest permet d'avoir un retour « immédiat » pour savoir si tous les tests PASSENT (VERT) ou pas (ROUGE).** Par contre, Infinitest met juste à jour la barre en bas de votre IDE.
  Donc pour savoir avec précision quel(s) test(s) échoue(nt), il faudra relancer « classiquement » les tests avec le menu **`Run As -> JUnit Test`** pour que la **vue `JUnit`** se mette à jour !

- Attention, si le chemin de vote projet contient des espaces, Infinitest pourrait ne pas fonctionner correctement 😉

# Focus sur la méthode `diviser`

*(Cas Limite & Exception)*

## 10. Implémentation de la méthode `diviser` (code de production)

Pour cet exercice, vous vous contenterez pour le moment d'implémenter la méthode diviser de la classe **`Calculatrice`** de la manière minimale suivante :

```
public int diviser(int dividende, int diviseur) {
     return dividende/diviseur;
     }
```

## 11. Vérification du *bon* comportement de la méthode `diviser` (code de test)

Ajoutez dans la classe **CalculatriceTest**, la méthode de test suivante :

```
@Test
public void doitDiviserDeuxEntiers() {

     Calculatrice calculatrice = new Calculatrice();
     int division = calculatrice.diviser(42, 5);
     assertEquals(8, division);
}
```

⇨ Sauvegardez le fichier (CTRL+S) et vérifiez grâce à Infinitest que **tous les tests passent (AU VERT).**
⇨ Lancez la couverture de code et constatez que **la classe `Calculatrice` est à 100% de couverture .**

Et pourtant…. Tous les cas n'ont pas été testé et le programme n'est pas sûr
et bugger à l'exécution en cas de division par zéro !!!

## 12. SandBox autour de la division par zéro

**⇨ Scénario de test illustrant un exemple « classique » :**
Pour expérimenter ce qui se passe dans le cas d'une division par zéro, nous allons créer, une classe **SandBox** avec une méthode **`main`** implémentée pour commencer de la manière suivante :

```
public class SandBox {

    public static void main(String[] args) {

        Calculatrice calculatrice = new Calculatrice();

        int dividende = 42;
        int diviseur = 2;
        int quotient = calculatrice.diviser(dividende, diviseur);
        System.out.println("Le résultat de la division de " +
            dividende + " par " + diviseur + " est : " +
        quotient);
    }

}
```

Exécutez ce code et vérifiez que tout se passe bien et que le message suivant s'affiche dans la console :

**`Le quotient de la division de 42 par 2 est : 21`**

**⇨ Scénario de test illustrant un « cas limite » :**
A la suite du code précédent, écrire l'instruction suivante :

```
resultat = calculatrice.diviser(42, 0);
```

Que se passe-t-il à l'exécution ?

- **Quelle exception est levée** (écrite en bleue dans la console) : ……
- **La pile des erreurs (en rouge dans la console), vous indique les lignes où l'exception est levée** que ce soit **dans le classe `Calculatrice`** ou **dans la classe Sandox**. En réalité l'exception part de la classe **`Calculatrice`** (première ligne) et remonte ensuite dans la classe **Sandox**. Cliquez sur les liens bleus pour vous retrouver sur les lignes d'où part et où passe l'exception.

Pour éviter que le programme « bugge » et s'arrête intempestivement avec une pile d'erreur, il faut capturer l'exception à moment donnée. Le seul endroit où on peut capturer l'exception ici est dans la méthode main.

Pour attraper une exception, il faut décorer votre instruction avec un `try…catch` de la manière suivante :

```
try {
     resultat = calculatrice.diviser(42, 0);
     } catch (Exception e) {
     System.out.println("La division par zéro est impossible");
     }
```

Après avoir implémenté ce code, relancez la classe **SandBox.**
Cette fois-ci, votre programme fonctionne correctement, il n'y a pas d'erreur (pas de ligne rouge dans la console), mais bien l'affichage des deux messages : celui du premier scénario de test et celui du second.

```
Le résultat de la division de 42 par 2 est : 21
La division par zéro est impossible
```

**⇨ Jouons un peu plus avec notre exception**

Quand on capture une exception (catch), il est possible d'en savoir plus sur cette exception.
Par exemple, ajoutez une ligne dans le **`catch`** de manière à récupérer le message porté éventuellement par l'exception :

```
 catch (Exception e) {
 System.out.println("La division par zéro est impossible");
 System.out.println("L'exception capturée est : "+e.getMessage());
}
```

Exécutez le code…

**⇨ Soyons plus précis sur l'exception à capturer**

A la page précédente, vous avez surement noté que l'exception qui s'est déclenchée était une `ArithmeticException.` Vous pouvez donc écrire votre catch de manière plus spécifique en ne capturant que les `ArithmeticException`

```
catch (ArithmeticException e) {
 System.out.println("La division par zéro est impossible");
 System.out.println("L'exception capturée est : "+e.getMessage());
}
```

Exécutez le code, vous obtenez les mêmes messages que précédemment, donc vous capturez bien l'exception.

***Remarque :*** Nous remanipulerons très prochainement les exceptions et créerons même nos propres exceptions (*Checked Exception*) dans un TP à venir. Pour cette introduction aux tests unitaires, seules

ces petites manipulations autour des (*Unchecked Exception*) suffisent, quittez donc votre bac à sable pour revenir à vos tests unitaires …

## 13. Vérifier (avec JUnit 5) la levée d'une exception durant un test

Pour savoir comment tester les exceptions, rien de tel qu'un petit tour dans la javadoc du framework JUnit.


Commencez par vous rendre sur **la page officielle du framework JUnit 5** : **https://junit.org/junit5/**
Cliquez sur le bouton **`User Guide.`**

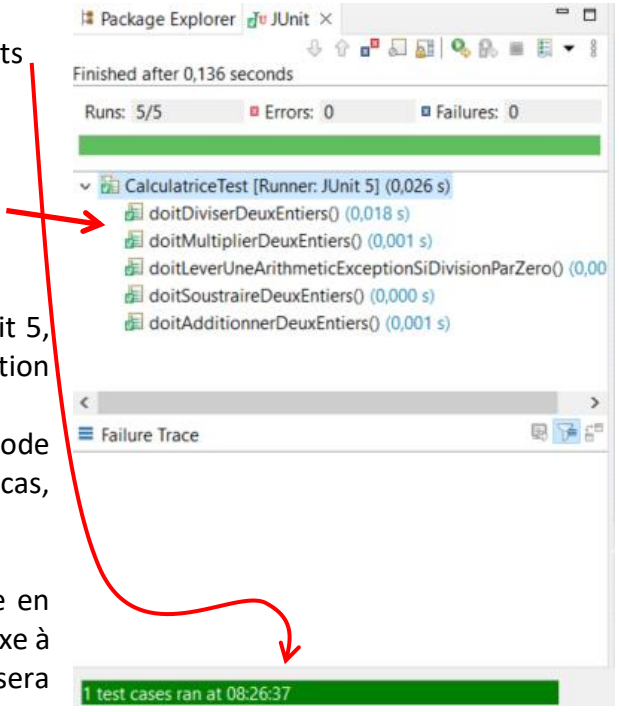

Rendez-vous ensuite à partir de la **`Table of Content`** dans la partie **2.4 Assertions**

Recherchez la méthode **`exceptionTesting`** qui illustre comment tester une exception.
Recopiez ce bout de code et collez-le dans la classe **`CalculatriceTest`**.

Après voir renommé cette méthode **`doitLeverUneArithmeticExceptionSiDivisionParZero`** adaptez ce code de test au code de votre projet **`calculatrice`**.

⇨ Sauvegardez de manière à vérifier via **Infinitest** si tous vos tests passent AU VERT !

⇨ Lancez également les tests avec le menu **`Run As-> JUnit`** pour visualiser tous vos tests AU VERT dans le vue JUnit




***Remarque :*** Comme indiquée dans la documentation, avec JUnit 5, c'est **assertThrows** qui permet de vérifier la levée d'une exception durant le test.
L'assertion **`assertThrows`** vérifie que l'exécution de la méthode passée en paramètre lève l'exception précisée : si ce n'est pas le cas, elle lève une exception pour faire échouer le test.
**`assertThrows`** utilise une notation fonctionnelle avec **`->`**
Pour l'instant, nous vous demandons de vous positionner juste en tant qu'utilisateur de **`assertThrows`** et de juste retenir la syntaxe à utiliser avec **`assertThrows`**. Le paradigme fonctionnel vous sera enseigné plus tard…

***Bonne pratique*** : Quand vous souhaitez couvrir par les tests le comportement d'un bout de code :

❑ Il faut veiller à couvrir **tous les scénarios de tests « normaux »** avec éventuellement plusieurs méthodes de tests (**exemples**) => *la couverture de code peut vous aider…*

❑ Il faut aussi penser à **aller chercher et tester les cas limites** qui pourraient déclencher des **comportements alternatifs voire des exceptions** !!! => c'est à vous à bien identifier que ces cas-là ont bien été traité pour éviter tout comportement anormal et/ou bug dans votre programme.
La couverture de code ne peut rien pour vous, comme disait Disjkstra « *Tester peut seulement montrer la présence d'erreur mais pas leur absence* » 😉

# Supprimer un peu de duplication de code …

Relisez attentivement le code de la classe CalculatriceTest.

Ne voyez-vous pas un peu de duplication de code dans cette classe ?

```
Calculatrice calculatrice = new Calculatrice();
```
est une instruction qui est en effet réalisée avant chaque méthode de test.

Rendez-vous dans le cours Quid du Test dans un développement logiciel ? dans le dépôt https://github.com/iblasquez/enseignement-but1-developpement et recherchez la diapositive **Annotations relatives au cycle de vie du test : Eviter la duplication (DRY)**

Quelle annotation permet d'exécuter un bout de code avant chaque méthode de test ? ……
Quelle annotation permet d'exécuter un bout de code une fois avant l'exécution du premier test de la classe ? ……

Quelle annotation pourriez-vous utiliser pour factoriser votre code et éviter de retrouver l'instruction de l'étape Arrange (`Calculatrice calculatrice = new Calculatrice();`) dupliquée dans chaque méthode de test.

Commencez par supprimer l'instruction précédente de toutes les méthodes de tests et à la place ajoutez une méthode correctement annotée qui permet de factoriser l'étape Arrange.

**Relancez vos tests qui doivent tous passer AU VERT et**
**appelez votre enseignant de TP pour vérifier votre factorisation de l'étape Arrange**
**et pouvoir continuer …**

# Exercice 2: Pour les plus rapides …
# Transformez les jeux d'essais en tests automatisés

Revenez sur le projet **computerpioneers.**

1. Créez un source folder **test** dans lequel vous ajouterez une classe **DeviceTest**
   Créez une première méthode test que vous appellerez **create_device**
   Dans le premier TP, vous aviez écrit dans une méthode **main** le jeu d'essai suivant à lancer dans la console :

```
Device babbageMachine = new Device("Babbage Analytical Machine", 1837);
System.out.println(babbageMachine.toString());
```

qui donnait comme affichage : **The Babbage Analytical Machine was invented in 1837.**

⇨ Transformez ce jeu d'essai en test unitaire automatisé dans la méthode **create_device.**

***A retenir*** : Pas de `System.out.println` dans un test automatisé !!!

2. Dans le source folder **test,** ajoutez une classe **ComputerPioneerTest**

   a. Créez une première méthode test que vous appellerez **create_computerPioneer_with_device**

   Dans le premier TP, vous aviez écrit dans une méthode **main** le jeu d'essai suivant à lancer dans la console :

```
Device babbageMachine = new Device("Babbage Analytical Machine", 1837);
ComputerPioneer adaLovelace = new ComputerPioneer("Ada", "Lovelace",
                                                  babbageMachine);
System.out.println(adaLovelace.toString());
```

   ⇨ Après avoir visualiser l'affichage obtenu, transformez ce jeu d'essai en test unitaire automatisé dans la méthode **create_computerPionner_with_device**

   a. Créez une autre méthode test que vous appellerez **computerPioneer_should_worksOn_device**

   Dans le premier TP, le dernier jeu d'essai que vous aviez écrit dans la méthode **main** est le jeu d'essai suivant :

```
Device babbageMachine = new Device("Babbage Analytical Machine", 1837);
ComputerPioneer adaLovelace = new ComputerPioneer("Ada", "Lovelace",
                                                  babbageMachine);

Device turingEngine = new Device("Turing Engine", 1936);
ComputerPioneer alanTuring = new ComputerPioneer("Alan", "Turing",
                                                 turingEngine);
```

```
Device babbage = new Device ("Babbage Analytical Machine",1837);
Device turing = new Device ("Turing Engine",1936);

System.out.println(adaLovelace.worksOn(babbage));
System.out.println(adaLovelace.worksOn(turing));
System.out.println(alanTuring.worksOn(babbage));
System.out.println(alanTuring.worksOn(turing));
```

Qui donnait comme affichage console :

```
true
false
false
true
```

⇨ Transformez ce jeu d'essai en test unitaire automatisé dans la méthode **`computerPioneer_should_worksOn_device`**

… De la même manière, Vous pouvez vous amuser à écrire des tests automatisés sur tous les projets déjà écrits …