

#### 1. Tests unitaires de l'application « Calculatrice ».

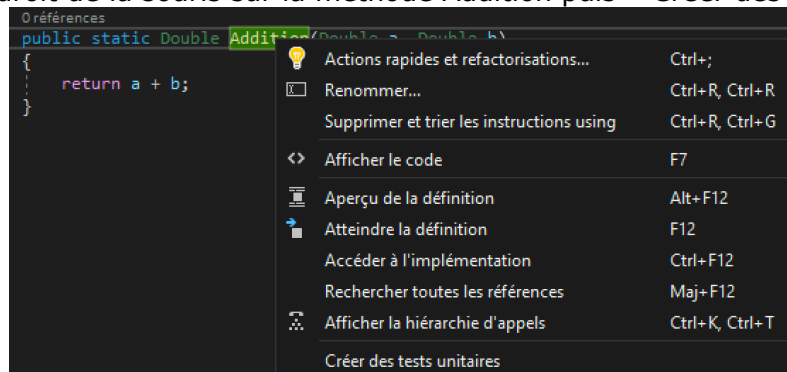
Reprendre votre code de l'application « Calculatrice » (TD1).

##### 1.1. Réalisation d'un premier test unitaire

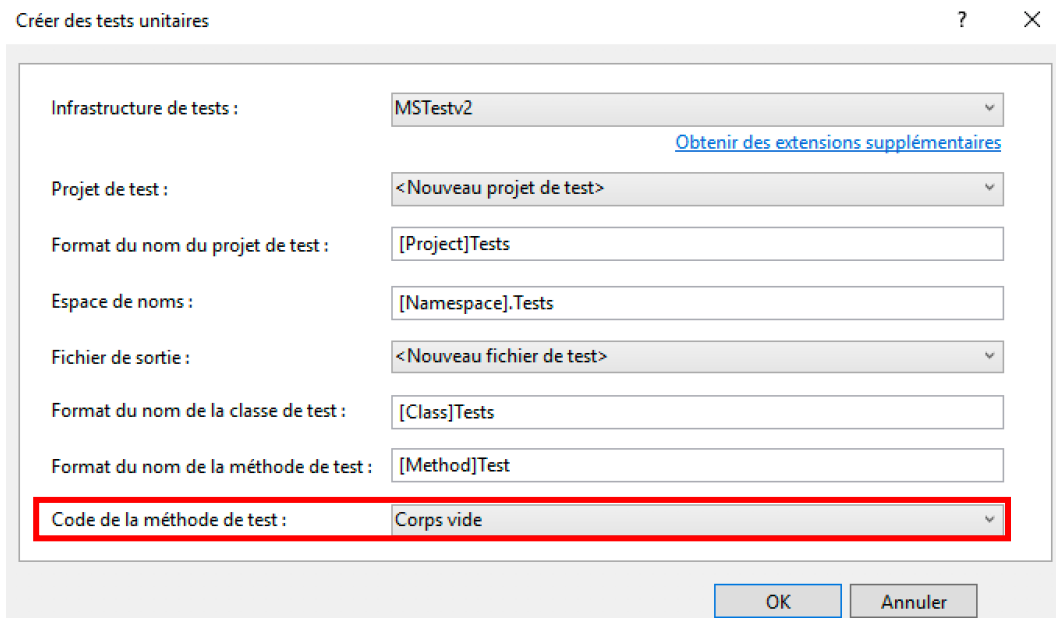
Les tests unitaires sont des classes déroulant une séquence d'appels de méthodes pour les composants d'une application. Les résultats retournés par ces méthodes sont comparés à leurs valeurs théoriques. Plutôt que d'écrire ces tests à la main, Visual Studio fournit un framework de tests automatisés (depuis Visual Studio 2013, il est intégré gratuitement dans toutes les versions). Il fournit au développeur un environnement structuré permettant l'exécution de tests et des méthodes pour aider au développement de ceux-ci.

Il existe plusieurs frameworks de test. Microsoft dispose de son propre framework, *MSTestv2* (il est aussi possible d'utiliser l'ancienne version *MSTest*).

Cliquer avec le bouton droit de la souris sur la méthode Addition puis « Créer des tests unitaires ».

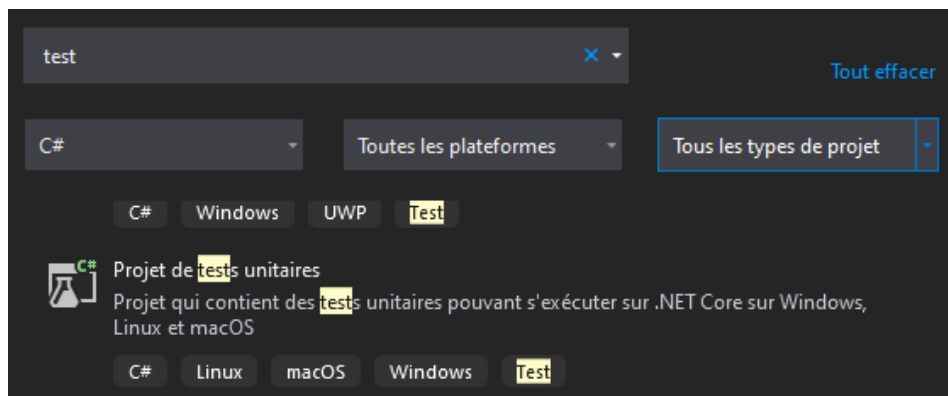


Par défaut, quand vous créez un projet de test unitaire, c'est le framework MSTest Version 2 qui est utilisé. Choisir « Corps vide » comme code de la méthode de test et valider.



Un nouveau projet `CalculatriceTests` est créé.

*Remarque : On peut aussi ajouter le projet manuellement à la solution (bien choisir .NET Core) :*



Dans ce cas, ne pas oublier de rajouter la référence au projet à tester depuis le projet de test.

L'écriture de tests unitaires automatisés nécessite la création d'un second projet car les tests sont toujours déroulés dans un processus distinct de l'application.

La classe de test consiste en une série de méthodes vérifiant le fonctionnement de chaque méthode ciblée de la classe « fonctionnelle » (i.e. à tester). Le programmeur doit renseigner pour chacune d'elles les valeurs des paramètres, les valeurs attendues, le type de comparaison et éventuellement le message d'erreur.

Dans notre cas, nous allons utiliser la méthode `MSTest.Assert.AreEqual()`. Elle permet de vérifier qu'une valeur est égale à une autre attendue. Elle contient en général 2 ou 3 paramètres :

- 1<sup>er</sup> paramètre : valeur attendue
- 2<sup>ème</sup> paramètre : valeur retournée
- 3<sup>ème</sup> paramètre (optionnel) : message d'erreur


Renommer la méthode de test ainsi et saisir le code suivant. Si cela n'est pas fait, ne pas oublier de rajouter la référence au projet à tester depuis le projet de test.

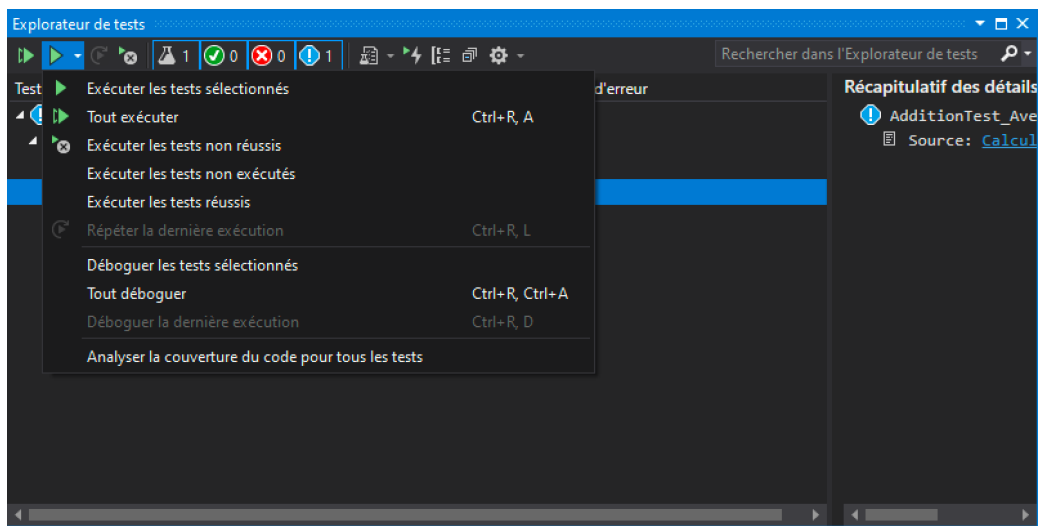
```
[TestMethod()]
public void AdditionTest_AvecValeur_1_2_Retourne3()
{
    //Arrange
    Double a = 1.0;
    Double b = 2.0;
    //Act
    Double resultat = Calcul.Addition(a, b);
    //Assert
    Assert.AreEqual(3.0, resultat, "Test non OK. La valeur doit être égale à 3");
}
```

Il est préférable d'ajouter un message d'erreur en 3<sup>ème</sup> paramètre :

```
Assert.AreEqual(3.0, resultat, "Test non OK. La valeur doit être égale à 3");
```

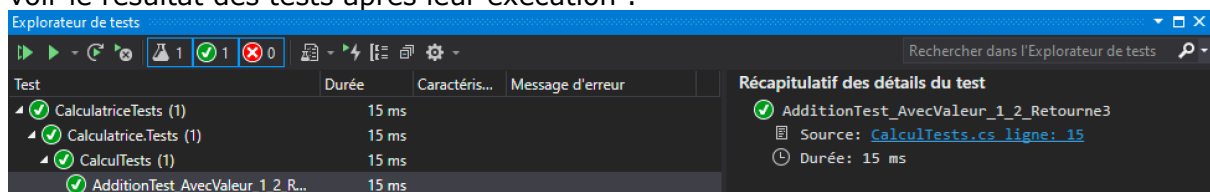
Une fois les méthodes de test renseignées, le projet de test doit être démarré. Il va instancier chaque composant cible et lui appliquer la séquence de tests. Un rapport est ensuite élaboré à l'attention du programmeur. Pour cela, aller dans le menu *Test > Explorateur de tests*.

On peut alors exécuter tous les tests  ou choisir d'autres options (*Exécuter les tests sélectionnés*, etc.) en utilisant le 2<sup>nd</sup> bouton.



Remarque : On peut déboguer les tests, car il s'agit toujours de code ! Il faut d'abord positionner un point d'arrêt.

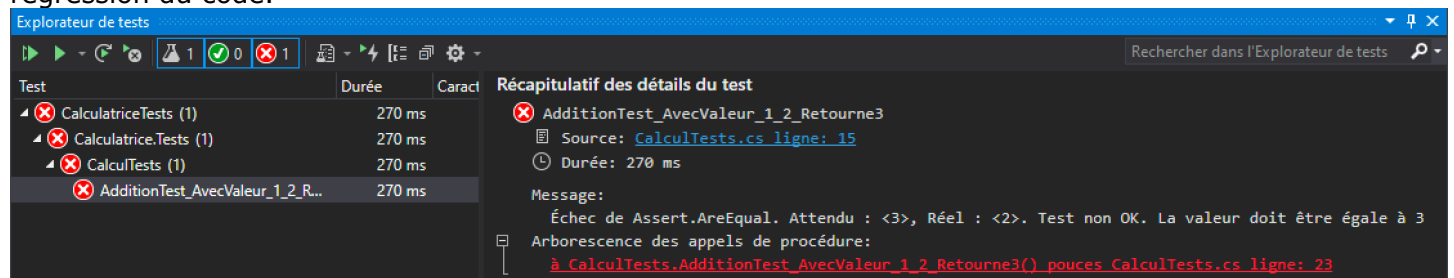
On peut voir le résultat des tests après leur exécution :



Avec une méthode si simple, il y a peu de chance que l'on voit l'échec du test. Pour cela, modifier la méthode Addition de la classe Calcul (normalement, on ne retouche plus au test une fois celui-ci créé), par exemple :

```
return a - b;
```

Maintenant, vous devriez voir l'échec en réexécutant les tests. Ici, nous avons - volontairement - créer une regression du code.



Un test, même après modification de la méthode testée, doit toujours fonctionner (sauf si on l'a bien sûr mal codé !). Il est important de lancer systématiquement les tests après des modifications réalisées sur la méthode testée.

Dans le menu *Test > Live Unit Testing*, cliquer sur « Démarrer ». Le live unit testing, nouvelle fonctionnalité disponible depuis la version 2017, permet d'avoir des indicateurs en temps réel sur la couverture de code ainsi que son statut. Ces indicateurs sont représentés par des petits sigles sur le côté gauche du code. Ainsi, au fût et à mesure que vous allez écrire votre code de test, celui-ci sera vérifié et exécuté.

**Pour que le *Live Unit Testing* fonctionne, votre projet doit être local, par exemple stocké sur le bureau (et non sur P:).**

```
[TestMethod()]
0  | 0 références
1  public void AdditionTest_AvecValeur_1_2_Retourne3()
2  {
3      //Arrange
4      Double a = 1.0;
5      Double b = 2.0;
6      //Act
7      Double resultat = Calcul.Addition(a, b);
8      //Assert
9      Assert.AreEqual(3.0, resultat, "Test non OK. La valeur doit être égale à 3");
10 }
11
```

Couverts par 1 test.

```
7
8  Test
9  ✓ AdditionTest_AvecValeur_1_2_Retourne3
10 Exécuter tout | Déboguer tout
11 2 références | ✓ 1/1 ayant réussi
12 public static Double Addition(Double a, Double b)
13 {
14     return a + b;
15 }
16
```

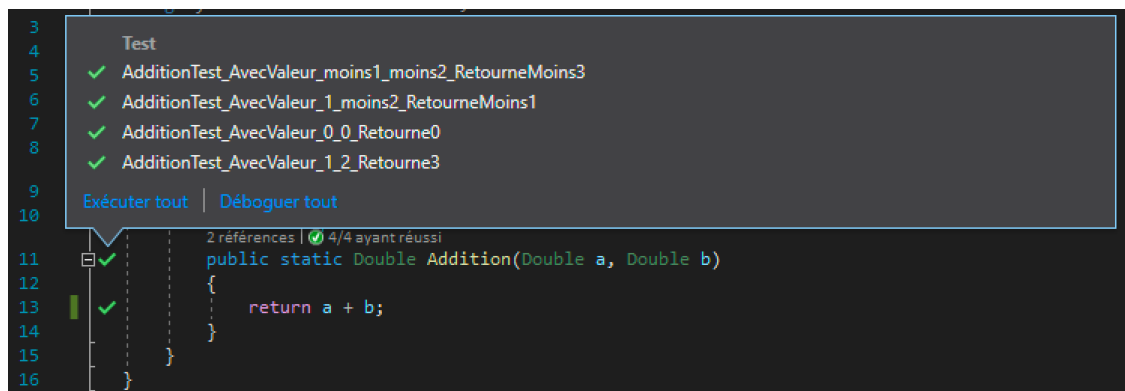
Si vous modifiez votre méthode ou la méthode de test, on peut voir en temps réel si le test réussit ou est en échec.

```
7  namespace Calculatrice
8  {
9      2 références
10     public class Calcul
11     {
12         2 références | ✗ 0/1 ayant réussi
13         public static Double Addition(Double a, Double b)
14         {
15             return a - b;
16         }
17     }
18 }
```

```
11
12 Test
13 ✗ AdditionTest_AvecValeur_1_2_Retourne3
14 Exécuter tout | Déboguer tout
15 0 références
16 public void AdditionTest_AvecValeur_1_2_Retourne3()
17 {
18     //Arrange
19     Double a = 1.0;
20     Double b = 2.0;
21     //Act
22     Double resultat = Calcul.Addition(a, b);
23     //Assert
24     Assert.AreEqual(3.0, resultat, "Test non OK. La valeur doit être égale à 3");
25 }
```

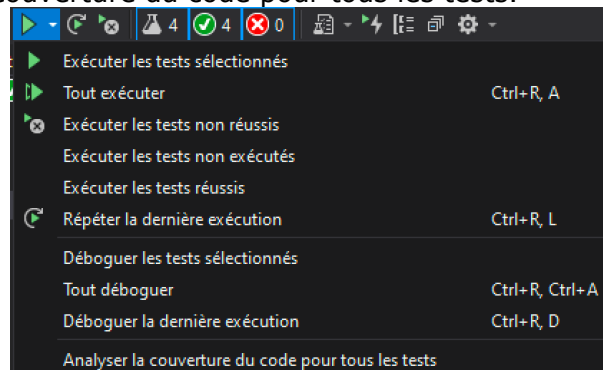
On peut alors, si nécessaire, déboguer le test.

**Il est important qu'une méthode de test ne s'occupe de tester qu'un seul cas d'une unique fonctionnalité (méthode),** comme nous venons de le faire. La première méthode teste la méthode `Addition` pour le cas où les opérandes sont 1 et 2. **Rajouter les 3 autres méthodes (cf. écran page suivante) au test de la méthode `Addition` (ne pas oublier l'attribut `[TestMethod]`).**



Remarque : on peut suspendre, arrêter ou redémarrer le live testing à tout moment.

On peut également calculer la couverture du code (menu Test > Résultats de la couverture du code). Il faut ensuite lancer l'analyse de la couverture du code pour tous les tests.



Résultats de la couverture du code				
Etudiant_DESKTOP-3LADFE1 2019-09-22 20_...				
Hierarchie	Non couverts (blocs)	Non couverts (% blocs)	Couverts (blocs)	Couverts (% blocs)
Etudiant_DESKTOP-3LADFE1 2019-0...	0	0,00 %	14	100,00 %
calculatrice.dll	0	0,00 %	2	100,00 %
calculatricetests.dll	0	0,00 %	12	100,00 %

Il faut bien sûr essayer de tendre vers 100% de couverture (ce n'est pas toujours très simple !) pour les classes fonctionnelles (calculatrice.dll ici).

**Bilan : La méthode `Addition()` est par définition fonctionnelle, mais il est important de prendre le réflexe de tester des fonctionnalités qui sont déterminantes pour l'application. Il faut écrire au moins 1 test qui réussit et 1 test par exception (si une ou plusieurs exceptions sont gérées dans le code). Mais comme dans le cas de l'addition, on souhaite également additionner des nombres nuls, positifs et/ou négatifs, nous avons créé plusieurs tests qui "réussissent".**

## 1.2. Réalisation des tests unitaires de la division

Jeux de tests :

Paramètre 1	Paramètre 2	Résultat
1	2	0.5
1	-2	-0.5
0	1	0
1	0	Exception : DivideByZeroException("Erreur division par zéro")

Coder les tests de la méthode `Division` (les 4 cas possibles du tableau ci-dessus).

Deux possibilités pour coder le test d'une exception :

- 1<sup>ère</sup> possibilité :
 

```
// Arrange
...
Exception expectedException = null;

// Act
```

```

try
{
    ...
}
catch (Exception ex)
{
    expectedException = ex;
}

// Assert
Assert.IsNotNull(expectedException, "Test non OK. Erreur exception.");

```

Assert.IsNotNull() vérifie si l'objet en paramètre est bien instancié et donc, ici, si une instance de l'exception a bien été créée. On peut ajouter en 2<sup>nd</sup> paramètre le message, comme pour les tests précédents.

Il est cependant préférable d'être précis et de gérer une exception spécifique `DivideByZeroException` plutôt qu'`Exception`. Modifier ainsi le code précédent (aux 2 endroits !).

- 2<sup>ème</sup> possibilité (préférable car plus concis), Cf. CM. :

```

[TestMethod()]
[ExpectedException(typeof(MonException))]
public void MaMethodeTest_AvecValeur_X_Y_RetourneMonException()
{
    // Arrange
    ...
    // Act levant l'exception attendue de type MonException
    ...
    // PAS d'Assert
}

```

Coder ces 2 possibilités pour tester la division par zéro.

Exécuter les tests.

### 1.3. Réalisation des tests unitaires de la factorielle

Réaliser les tests de la méthode `Factorielle`.

Tests à coder : 0 (donne 1), 1 (donne 1), 10 (donne 3628800) et nombre < 0 (gérer une exception de type `ArgumentException`). On peut aussi vérifier si le nombre est bien un entier (sinon exception de type `ArgumentException`).

Le code de la factorielle n'étant pas optimal, le remplacer par une fonction récursive.

Ré-exécuter les tests après cette modification majeure afin de s'assurer qu'il n'y a pas de régression de code.

### 1.4. Autres méthodes à tester

Coder les tests unitaires des autres méthodes (soustraction, multiplication).

### 1.5. Remarques

*Autres tests possibles :*

```

bool b = true;
Assert.IsTrue(b);

bool b = false;
Assert.IsFalse(b);

String s = null;
Assert.IsNull(s);

```

```
String s = "Bonjour";
Assert.IsNotNull(S);
```

Et bien d'autres car les méthodes sont nombreuses :

<https://msdn.microsoft.com/fr-fr/library/microsoft.visualstudio.testtools.unittesting.assert.aspx>

*Autres méthodes de tests possibles :*

Il existe deux attributs supplémentaires : les attributs `[TestInitialize]` et `[TestCleanup]`.

Ils permettent de décorer des méthodes qui seront appelées respectivement avant chaque test et après chaque test. Elles sont utilisées pour factoriser des initialisations ou des nettoyages dont dépendent tous les tests.

```
[TestClass]
public class MesTests
{
    [TestInitialize]
    public void InitialisationDesTests()
    {
        // Rajouter les initialisations
        // exécutées avant chaque test
    }

    [TestMethod()]
    public void Factorielle_AvecValeur1_Retourne1()
    {
        // test à faire
    }

    [TestCleanup]
    public void NettoyageDesTests()
    {
        // Nettoyer les variables, ...
        // après chaque test
    }
}
```

## **2. Tests unitaires de l'application « Comptes bancaires ».**

Reprendre votre code de l'application « Comptes bancaires » (TD4).

Les tests ne concerneront que la couche métier (`BusinessLayer`).

### **2.1. Codage des tests de `GetAllComptes`**

La méthode `GetAllComptes` exécute une instruction `SELECT` sur la vue `vComptes` de la base de données et crée une liste de `Compte`. Il faut donc qu'il y ait toujours les mêmes comptes (mêmes id & solde) dans la BD de façon que les tests fonctionnent toujours, même si l'on ajoute des comptes et/ou modifie les soldes. Il paraît donc important avant d'effectuer tout test :

- De supprimer tous les comptes existants.
- D'insérer des comptes qui seront attendus dans les tests de la méthode.

Ainsi, coder la méthode `InitialisationDesTests` permettant de supprimer les lignes de la table `Compte` (`delete...`) et d'ajouter 2 comptes : (1234567, 1000) et (2345678, 2000). Cette méthode utilisera à chaque fois (pour la suppression et les 2 insertions) la méthode `SetData` de la classe `DataAccess`.

```
/// <summary>
/// Cette méthode appelle la méthode SetData sur un objet de type DataAccess pour
supprimer les données de la table Compte
/// et insérer 2 comptes bancaires : 1234567 / 1000 et 2345678 / 2000.
/// </summary>
[TestInitialize()]
```

```

public void InitialisationDesTests()
{
    ...
}

```

Rappel : la méthode `TestInitialize` sera exécutée lors de chaque test.

Remarque : penser à ajouter une référence à `DataLayer` dans le projet de test.

Coder ensuite 2 méthodes permettant de tester que le 1<sup>er</sup> compte de la liste retournée par `GetAllComptes` est bien le compte (1234567, 1000) attendu. De même pour le second (2345678, 2000).

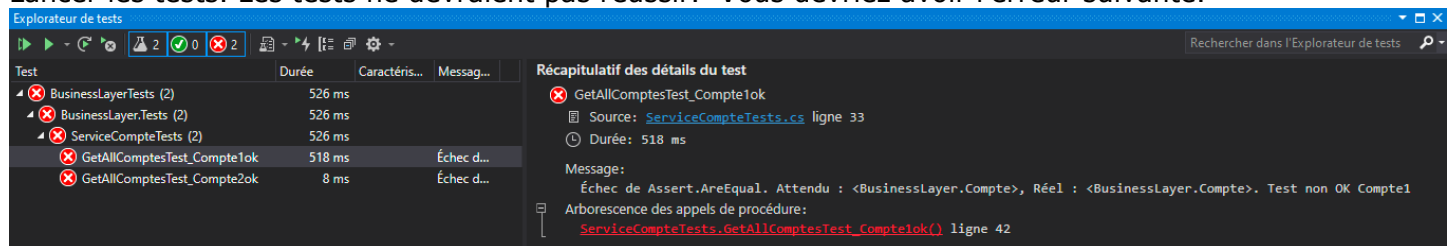
```

[TestMethod()]
public void GetAllComptesTest_Compte1()
{
    ...
}

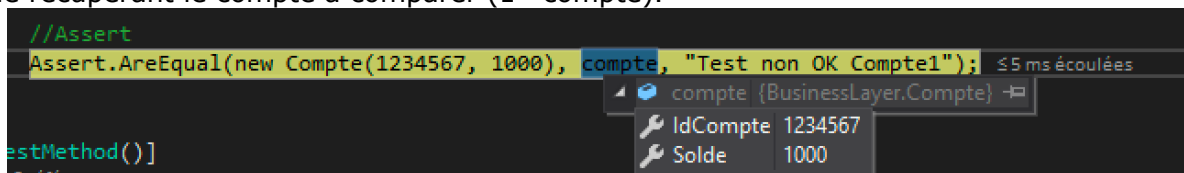
[TestMethod()]
public void GetAllComptesTest_Compte2()
{
    ...
}

```

Lancer les tests. Les tests ne devraient pas réussir. Vous devriez avoir l'erreur suivante.



La méthode `GetAllComptesTest_Compte1` compare 2 comptes à priori identiques (1, 1000) mais la méthode `AreEqual()` renvoie `false`. Vous pouvez le vérifier en mettant un point d'arrêt sur la ligne `Assert...` de la méthode `GetAllComptesTest_Compte1` puis menu `Test > Déboguer > Tous les tests`. Vérifier le contenu de la variable récupérant le compte à comparer (1<sup>er</sup> compte).



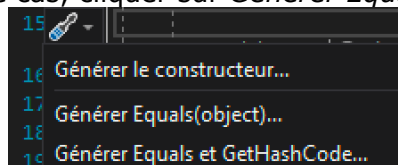
Cela est dû au fait que les 2 objets résidant dans des espaces mémoires différents ne sont pas égaux. Il est nécessaire de coder la méthode `Equals` (comme l'an dernier !) dans la classe `Compte`, de façon que l'on puisse comparer les 2 objets.

```

public override bool Equals(object obj)
{
    ...
}

```

Vous pouvez aussi la générer (dans ce cas, cliquer sur `Générer Equals et GetHashCode`) :



**Relancer les tests. Cette fois, ils devraient réussir.**

### Améliorer les tests

Plutôt que de tester chaque objet, MSTest permet de comparer une collection d'objets (par exemple, une liste) à une collection espérée (ici, "rentrée" en dur), en utilisant la classe `CollectionAssert`.

Créer une nouvelle procédure de test nommée `GetAllComptesTest_TousLesComptes()` utilisant `CollectionAssert.AreEqual()` (<https://msdn.microsoft.com/fr-fr/library/ms243721.aspx>).

Remarques :

- Conserver les 2 tests précédemment créés.



- Nous n'avons pas testé l'exception retournée par la méthode `GetAllComptes`. Pour la lever, il faut supprimer soit la vue `vComptes`, soit la table `Compte`, ou que le chaîne de connexion ne soit pas valide. Dans tous les cas, les 3 tests réalisés précédemment seront en échec. Il n'est donc pas vraiment utile de tester l'exception (même si on peut le faire).

## 2.2. Codage des tests de `SetDebitCredit`

Créer un premier test nommé `SetDebitCreditTest_DebitOK` testant le débit sur un compte existant. Vous testerez la valeur retournée par le booléen.

Créer un 2<sup>ème</sup> test nommé `SetDebitCreditTest_CreditOK` testant le crédit sur un compte existant.

Créer un 3<sup>ème</sup> test nommé `SetDebitCreditTest_CompteInconnuNonOK` testant un débit ou un crédit sur un compte bancaire inconnu.

Exécuter les tests.

### Améliorer les tests

Ici, nos tests ne sont pas suffisants, car ils ne permettent pas de vérifier que le solde a bien été mis à jour dans la base de données. Modifier les tests pour qu'ils vérifient le nouveau solde du compte passé en paramètre. **Indication : il est possible de faire plusieurs Assert dans la même méthode de test.**

Si vous avez codé l'interdiction de découvert, vos tests devront prendre en compte cette fonctionnalité.

## 2.3. Amélioration

Ajouter la méthode suivante, s'exécutant après les tests, permettant de supprimer toutes les données et insérer les données initiales contenues dans le script SQL

```
[TestCleanup()]
public void NettoyageDesTests()
{
}
```

## 2.4. Codage des tests de `Virement`

A vous de jouer...