TP2 - Les bases des tests unitaires

Objectifs

- Ecrire vos premiers tests unitaires avec **xUnit**
- Comprendre le pattern **Arrange / Act / Assert**
- Utiliser les assertions de base : `Assert.Equal`, `Assert.True`, `Assert.Throws`
- Decouvrir les tests parametres avec `[Theory]` et `[InlineData]`

Mise en place

Creez un nouveau projet de test xUnit dans votre solution

Exercice 1 - StringCalculator

Contexte

Un collegue a ecrit une classe `StringCalculator`. La methode `Add` prend une chaine de caracteres contenant des nombres separes par des virgules et retourne leur somme.

Exemples attendus :

- `""` → `0`
- `"1"` → `1`
- `"1,2"` → `3`
- `"1,2,3,4,5"` → `15`

Voici son implementation :

```
namespace TestingTP2;

public class StringCalculator
{
    public int Add(string numbers)
    {
        if (numbers == null)
            return 0;

        if (numbers.Length == 0)
            return 0;

        var parts = numbers.Split(',');

        int sum = 0;
        for (int i = 1; i <= parts.Length; i++)
        {
            sum += int.Parse(parts[i]);
        }

        return sum;
    }
}
```

Objectif

Votre mission est d'ecrire des tests unitaires pour valider le comportement de cette methode. Creez la classe de test suivante et completez-la :

```
using TestingTP2;

namespace TestingTP2.Tests;
```

```
public class StringCalculatorTests
{
    [Fact]
    public void Add_EmptyString_ReturnsZero()
    {
        // Arrange
        var calculator = new StringCalculator();

        // Act
        int result = calculator.Add("");

        // Assert
        Assert.Equal(0, result);
    }

    // A vous d'ajouter d'autres tests !
}
```

Etape 1 - Tester les cas de base

Ecrivez un test pour chacun des cas suivants :

1. Une chaine vide `""` doit retourner `0`
2. Un seul nombre `"5"` doit retourner `5`
3. Deux nombres `"1,2"` doivent retourner `3`
4. Plusieurs nombres `"1,2,3,4,5"` doivent retourner `15`

Lancez vos tests. Que constatez-vous ?

Etape 2 - Analyser les echecs

Certains de vos tests echouent. Pour chaque echec :

1. Lisez le message d'erreur
2. Identifiez la ligne du code source qui pose probleme
3. Expliquez le bug en une phrase

Etape 3 - Corriger le code

Corrigez l'implementation de `StringCalculator` pour que tous vos tests passent.

Etape 4 - Cas limites

Ecrivez des tests supplementaires pour ces cas limites :

Cas	Entree	Resultat attendu
Espaces autour des nombres	`"1, 2, 3"`	`6`
`null` en entree	`null`	`0`
Un seul nombre avec virgule	`"7,"`	`7`

Vos tests passent-ils ? Si non, adaptez l'implementation.

Etape 5 - Refactoring avec `[Theory]`

Vous remarquez que vos tests se ressemblent beaucoup. Refactorisez-les en un seul test parametre :

```
[Theory]
[InlineData("", 0)]
[InlineData("1", 1)]
[InlineData("1,2", 3)]
[InlineData("1,2,3", 6)]
[InlineData("1,2,3,4,5", 15)]
public void Add_VariousInputs_ReturnsExpectedSum(string input, int
expected)
{
    var calculator = new StringCalculator();

    int result = calculator.Add(input);

    Assert.Equal(expected, result);
}
```

Lancez les tests. Chaque `[InlineData]` genere un test distinct dans l'explorateur de tests.

Etape 6 - Nombres negatifs interdits

La calculatrice doit lever une exception `ArgumentException` si l'entree contient un nombre negatif.

Ecrivez un test qui verifie ce comportement :

```
[Fact]
public void Add_NegativeNumber_ThrowsArgumentException()
{
    var calculator = new StringCalculator();

    var exception = Assert.Throws<ArgumentException>(() =>
calculator.Add("1,-2,3"));

    Assert.Contains("-2", exception.Message);
}
```

Implementez cette regle dans `StringCalculator`.

Etape 7 - Delimiteur personnalise

Ajoutez le support d'un delimiteur personnalise. Le format est : `"//[delimiteur]\n[nombres]"`

Exemples :

- `"//;\n1;2"` → `3`
- `"//|\n4|5|6"` → `15`

Ecrivez les tests d'abord, puis l'implementation.

Exercice 2 - PasswordValidator

Contexte

Vous allez creer un validateur de mot de passe et le tester rigoureusement. Le but est de pratiquer les tests parametres et la gestion de cas limites.

Regles de validation

Un mot de passe est valide si **toutes** les conditions suivantes sont remplies :

1. Au moins **8 caracteres**
2. Contient au moins **une lettre majuscule**
3. Contient au moins **une lettre minuscule**
4. Contient au moins **un chiffre**

Etape 1 - La classe de base

Creez la classe `PasswordValidator` :

```
namespace TestingTP2;

public class PasswordValidator
{
    public bool IsValid(string password)
    {
        throw new NotImplementedException();
    }
}
```

Etape 2 - Tester les mots de passe valides

Commencez par verifier qu'un mot de passe respectant toutes les regles est accepte :

```
using TestingTP2;

namespace TestingTP2.Tests;

public class PasswordValidatorTests
{
    [Theory]
    [InlineData("Abcdefg1")]
    [InlineData("MonMotDePasse9")]
    [InlineData("P4ssw0rdOk")]
    public void IsValid_ValidPassword_ReturnsTrue(string password)
    {
        var validator = new PasswordValidator();
```

```
        bool result = validator.IsValid(password);

        Assert.True(result);
    }
}
```

Implementez `IsValid` pour faire passer ces tests.

Etape 3 - Tester chaque regle individuellement

C'est a vous d'ecrire les tests. Pour chaque regle violee, verifiez que `IsValid` retourne `false` :

Trop court (< 8 caracteres) :

```
[Theory]
[InlineData("Ab1")]
[InlineData("Abcdef1")]
// A completer...
public void IsValid_TooShort_ReturnsFalse(string password)
{
    // A vous !
}
```

Pas de majuscule :

```
[Theory]
[InlineData("abcdefg1")]
// A completer...
public void IsValid_NoUppercase_ReturnsFalse(string password)
{
    // A vous !
}
```

Pas de minuscule :

```
// A vous d'ecrire ce test !
```

Pas de chiffre :

```
// A vous d'ecrire ce test !
```

Etape 4 - Cas limites

Ecrivez des tests pour les cas suivants. Reflechissez au resultat attendu **avant** d'ecrire le test :

Cas	Mot de passe	Valide ?
Chaine vide	`""`	?
Exactement 8 caracteres	`"Abcdefg1"`	?
Que des chiffres	`"12345678"`	?
Que des majuscules + chiffre	`"ABCDEFG1"`	?
Mot de passe tres long	`"Aaaaaaaaaaaaaaa1"`	?

Etape 5 - Retourner des messages d'erreur

Refactorisez `PasswordValidator` pour retourner la **liste des regles violees** au lieu d'un simple booleen :

```
public class PasswordValidator
{
    public ValidationResult Validate(string password)
    {
        // A implementer
    }
}

public class ValidationResult
{
    public bool IsValid => !Errors.Any();
    public List<string> Errors { get; init; } = new();
}
```

Ecrivez les tests correspondants. Par exemple :

```
[Fact]
public void Validate_TooShortAndNoDigit_ReturnsBothErrors()
{
    var validator = new PasswordValidator();

    var result = validator.Validate("Abcde");

    Assert.False(result.IsValid);
    Assert.Equal(2, result.Errors.Count);
    Assert.Contains(result.Errors, e => e.Contains("8 caracteres"));
    Assert.Contains(result.Errors, e => e.Contains("chiffre"));
}
```

Etape 6 - A vous de jouer

Ajoutez une nouvelle regle : le mot de passe doit contenir au moins **un caractere special** parmi `!@#$%^&*`.

1. Ecrivez d'abord les tests

2. Verifiez qu'ils echouent
3. Implementez la regle
4. Verifiez que tous les tests passent

N'oubliez pas de mettre a jour vos tests existants pour qu'ils restent valides avec cette nouvelle regle !

Recapitulatif

Concept	**Ou l'avez-vous pratique ?**
`[Fact]`	Exercice 1, etapes 1 a 4
`[Theory]` + `[InlineData]`	Exercice 1 etape 5, Exercice 2
`Assert.Equal`	Exercice 1
`Assert.True` / `Assert.False`	Exercice 2
`Assert.Throws`	Exercice 1 etape 6
`Assert.Contains`	Exercice 2 etape 5
Pattern Arrange / Act / Assert	Partout
Tester du code existant	Exercice 1, etapes 1 a 3
Tester les cas limites	Exercices 1 et 2
Ecrire le test avant le code	Exercice 1 etape 7, Exercice 2