Les fonctions

- Qu'est-ce que c'est?
- Définition.
- Invocation.
- Objectif d'une fonction.
- Saviez-vous que.
- Principe de responsabilité unique.

Qu'est-ce qu'une fonctions

- Les programmes informatiques peuvent facilement contenir des centaines, voir des milliers, de lignes de code.
- Pour faciliter la réutilisation, l'encapsulation et la lisibilité, du code, le concept des fonctions a été inventé.
- Une fonctions est un petit bout de code indépendant pouvant être appelé dans une autre partie du programme.

Fonction ou méthode ?

- Vous allez régulièrement entendre les termes *fonction* et *méthode* pour décrire ce qui semble être la même chose. Quel est la différence?
- En *Programmation orientée objet* on utilise le terme méthode pour désigner une fonction qui est associé à un objet.
    - Par exemple, lorsqu'on utilise une liste, la commande `maListe.Add("Pomme");`, `maListe` est l'objet et `Add()` la méthode de l'objet `maListe`.
    - C'est la terminologie officielle en C# puisque celui-ci est considéré comme un langage orientée objet.
- Une fonction représente à peu près la même chose, mais dans un contexte qui n'est pas associé à un objet. Comme si j'avais juste `Add("Pomme");`
    - *Puisque, pour le moment nous n'utilisons pas les objets, j'utiliserai, personnellement, le terme fonction.*

Définition d'une fonction

- La définition de la fonction possède :
  - Le nom de la fonction (standard => "UpperCamelCase")
    - Utiliser idéalement un verbe **significatif** à l'infinitif pour définir le nom: Afficher, Display, Calculer, Calculate, Effacer, Delete, etc.
    - Alternativement, si la fonction retourne un booléen, vous pouvez utilisez *Est/Is*:EstPositif / IsPositive, EstValide, IsValid, etc.
  - Les paramètres de la méthode sont les données entrantes. Pour chaque paramètre, nous devons définir le type de donnée.
  - La méthode retourne une valeur d'un type précis avec le mot-clé return.

Définition d'une fonction

Voici un exemple d'une fonction.

```
string GetPassFailMessage(int note) {
    if(note > 60) {
        return "Réussi"
    }
    else {
        return "échoué";
    }
}
```

Fonction sans une valeur de retour

- Le type `void` signifie nul ou vide.
- Le type `void` est utilisé pour les fonctions sans valeur de retour.
- Il n'y a pas le mot `return` dans la fonction

```
void AfficherMenu()
{
    Console.WriteLine("==== Menu ====");
    Console.WriteLine("1. Afficher");
    Console.WriteLine("2. Quitter");
}
```

- *On remarque au passage que les fonctions peuvent aussi ne pas avoir de paramètre.*

Fonction avec une valeur de retour

- Le mot clé `return` sert à retourner une valeur. Celle-ci peut être une donnée, une variable, une constante, un calcul, etc.
- En autant qu'elle respecte le type de retour de la fonction.

```
int RetournerUneDonnée() {
    return 10; //Retour d'un entier
}
double RetournerUneVariable() {
    double valeur = 10.10;
    return valeur; //Retour d'une variable
}
double RetournerCalcul() {
    return (18.0 / 2) + (1.0 / 2); //Retour d'un calcul (résultat : 9.5)
}
```

Invocquer une fonction

- Un fois notre fonction définie, nous pouvons l'invocquer n'importe où dans notre code.
- Si la fonction n'a pas de paramètre:
  - Écrire le nom de la fonction avec des parenthèses vides.
- Si la fonction a des paramètres:
  - Écrire le nom de la fonction avec des parenthèses et écrire les valeurs à envoyer en paramètre entre les parenthèses.
- Si la fonction a une valeur de retour, assurez vous de la stocker quelque part.

```
double RetournerResultat() { // Déclaration de la fonction
    return (18.0 / 2) + (1.0 / 2);
}
double reponse = RetournerResultat(); // Invocation de la fonction
Console.WriteLine(reponse);
```

Objectif d'une fonction

- Une fonction vous permet de découper votre programme en petit algorithme réutilisable.
- Tout comme les algorithmes qu'on fait depuis le début, une fonction possède:
  - Des entrées: les paramètres
  - Une sortie: le return
  - Une suite d'opération pour partir de l'entrée et retourner la sortie: Tout ce qui est dans le corps de la fonction.

Saviez vous que

- L'environnement dotnet met à notre disposition un ensemble de méthode tout faites pour nous simplifier la vie.
- Vous utilisez déjà courament ceux-ci depuis le début de la session.
    - `Console.WriteLine("allo");`
        - Ici, Console est l'objet.
        - Ici, WriteLine est le nom de la fonction.
        - Les () pour l'invocquer.
        - "allo" est l'argument
        - Si nous allions voir le code source de cette fonction, sont type serait void car elle ne retourne pas de valeur.

Saviez vous que (suite)

- `int chiffre = Convert.ToInt32("45");`
  - Ici, Convert est l'objet.
  - Ici, ToInt32 est la fonction
  - Les () pour l'invocquer
  - "45" est la valeur en argument.
  - 45 est la valeur de retour de la fonction.
- À partir d'ici, soyez attentif aux fonctions que vous utilisez. Cela vous aidera à savoir quelle syntaxe utiliser. Par exemple, lorsque vous utilisez une commande pour effectuer une opération, il y a de bonne chance que ce soit une fonction. C'est pourquoi on envoit une valeur en argument entre les parenthèses et on affecte à une variable la valeur retourné par la fonction à la fin de son invocation.

Les principes S.O.L.I.D

- Les principes SOLID sont un ensemble de bonne pratique qui favorise la conception de logiciels modulaires et facile à maintenir.
- Le principe de responsabilité unique, aussi appelé SRP (Single Responsability Principle), est le premier des principes **S**OLID.
- Le SRP est le premier des deux principes que nous verrons dans ce cours. Vous verrez les 3 autres plus tard dans la technique.

Principe de responsabilité unique

- Le SRP stipule qu'une fonction (ou une méthode) doit se concentrer sur une et une seule tâche. Cela signifie:
  - Une tâche par fonction : Assurez-vous que chaque fonction fait une seule chose clairement définie.
  - Nommez précisément : Le nom de la fonction doit refléter précisément sa responsabilité unique.
  - Divisez les tâches complexes : Si une fonction devient trop longue ou gère plusieurs responsabilités, divisez-la en plusieurs fonctions plus petites.
  - Limitez les effets de bord : Une fonction doit éviter d'affecter l'état global ou de modifier des variables en dehors de son *scope*.
  - Gardez les paramètres simples : Utilisez un nombre réduit de paramètres et évitez de passer des structures de données complexes si la tâche ne le justifie pas.

Principe de responsabilité unique (suite)

- Les objectifs de l'application sur SRP sont:
  - La clarté : Une fonction qui respecte le SRP est plus facile à comprendre, car elle se concentre sur une tâche unique.
  - La réutilisabilité : En ayant une seule responsabilité, une fonction peut être utilisée dans différents contextes sans introduire de comportements indésirables.
  - La restabilité : Une fonction ayant une seule responsabilité est plus simple à tester, car elle a moins de dépendances et d'effets de bord.
  - La maintenance : Lorsqu'une fonction est modifiée, seules les parties du code concernées par cette tâche spécifique sont affectées, réduisant ainsi le risque d'introduire des bugs.

Voyont quelques do/don't pour mieux comprendre le principe.

1. Une tâche par fonction

dont

```
void ProcessAndDisplayData(int[] numbers)
{
    int sum = 0;
    for(int i = 0; i < numbers.Length; i++)
    {
        sum += numbers[i];
    }
    Console.WriteLine("Sum: " + sum);
}
```

Cette fonction fait deux tâches : elle calcule la somme et l'affiche.

1. Une tâche par fonction

do

```
int CalculateSum(int[] numbers)
{
    int sum = 0;
    for(int i = 0; i < numbers.Length; i++)
    {
        sum += numbers[i];
    }
    return sum;
}

void DisplaySum(int sum)
{
    Console.WriteLine("Sum: " + sum);
}
```

Chaque fonction fait une seule tâche : l'une calcule la somme, l'autre l'affiche.

2. Nommez précisément

dont

```
void DoStuff(int[] numbers)
{
    // Calcul de la moyenne
}
```

Le nom DoStuff ne décrit pas la tâche accomplie.

2. Nommez précisément

do

```
double CalculateAverage(int[] numbers)
{
    // Calcul de la moyenne
}
```

Le nom CalculateAverage décrit clairement ce que fait la fonction.

3. Divisez les tâches complexes

dont

```
void ProcessArray(int[] numbers)
{
    int sum = 0;
    for(int i = 0; i < numbers.Length; i++)
    {
        sum += numbers[i];
    }
    double average = sum / numbers.Length;
    Console.WriteLine("Sum: " + sum);
    Console.WriteLine("Average: " + average);
}
```

La fonction fait trop de choses : somme, moyenne et affichage.

3. Divisez les tâches complexes

do

```
int CalculateSum(int[] numbers) {
    int sum = 0;
    for(int i = 0; i < numbers.Length; i++)     {
        sum += numbers[i];
    }
    return sum;
}

double CalculateAverage(int sum, int count) {
    return (double)sum / count;
}

void DisplayResults(int sum, double average) {
    Console.WriteLine("Sum: " + sum);
    Console.WriteLine("Average: " + average);
}
```

La tâche complexe est divisée en trois fonctions distinctes.

4. Limitez les effets de bord

dont

```
int[] numbers = [1,2,3,45,5,6,7,8,9,67,5,54,3524,243,463,654,7654];
int globalSum = 0;

void CalculateAndStoreSum()
{
    for(int i = 0; i < numbers.Length; i++)
    {
        globalSum += numbers[i];
    }
}
```

Cette fonction modifie des variables globales, ce qui crée des effets de bord.

4. Limitez les effets de bord

do

```
int CalculateSum(int[] numbers)
{
    int sum = 0;
    for(int i = 0; i < numbers.Length; i++)
    {
        sum += numbers[i];
    }
    return sum;
}
int[] numbers = [1,2,3,45,5,6,7,8,9,67,5,54,3524,243,463,654,7654];
int globalSum = CalculateSum(numbers);
```

La fonction reçoit le tableau en paramètre et renvoie la somme sans utiliser ou modifier d'état global.

5. Gardez les paramètres simples

dont

```
void ProcessData(int[] numbers, int sum, double average, int count)
{
    // Traitement des données
}
```

Trop de paramètres, ce qui rend la fonction complexe.

5. Gardez les paramètres simples

do

```
void ProcessData(int[] numbers)
{
    int sum = CalculateSum(numbers);
    double average = CalculateAverage(sum, numbers.Length);
    DisplayResults(sum, average);
}
```

Les paramètres sont simplifiés en passant uniquement le tableau et en calculant les résultats à l'intérieur.