C#

Fonctions et paramètres

SOMMAIRE

Les objectifs

- Structuration d'un programme en fonctions élémentaires.
- Syntaxe et présentation des fonctions.
- Passage de paramètres.
- Surcharge des fonctions

Définition

Une fonction est une partie de programme comportant un ensemble d'instructions qui a besoin d'être utilisé plusieurs fois dans un programme ou dans différents programmes.
Parce que C# est un langage orienté objet, les fonctions ne peuvent être déclarées qu'à l'intérieur d'une classe. Il est donc impossible de déclarer des fonctions isolées.

Une fonction peut ou non renvoyer un résultat. Ce résultat n'est pas forcément exploité. Certaines fonctions (déclarée **`void`**) ne renvoient pas de résultat.

Exemple :

```
namespace Fonctions
{
 class Test1
 {

  // définition des fonctions
  static void AfficheMessage()
  {
   Console.WriteLine("Ceci est un message");
  }
  static bool ChercheEtTrouve()
  {
   bool btrouve;
   // ....
   return btrouve;
  }

  // programme principal
  static void Main()
  {
   Console.WriteLine("Programme principal");
   // ....
   AfficheMessage();
   // ....
   AfficheMessage();
   // ....
   bool btrouve = ChercheEtTrouve()
  }
 }
}
```

Dans cet exemple, 4 fonctions ou méthodes sont nommées :

- `Main` est le point d'entrée de l'application
- `WriteLine` est une méthode de la classe System.Console du Framework.
- `AfficheMessage` est une méthode de la classe Fonctions.Test1 qui ne renvoie aucune valeur
- `ChercheEtTrouve` est une méthode de la classe Fonctions.Test1 qui renvoie une valeur booléenne.

Une fonction ou méthode est définie par

- Un nom
- Une liste de paramètres entre parenthèses
- Des instructions codées entre accolades, formant le corps de la méthode,

Exemple :

```
static void AfficheMessage ()
{
 // Corps de la méthode
}
```

Nom

Liste de paramètres

Utilisation des fonctions

Après avoir été définie, une fonction peut être appelée :

- A partir de la même classe

```
static void Main()
{
 Console.WriteLine("Programme principal");
 // ....
 AfficheMessage();
 // ....
 AfficheMessage();
 // ....
 bool btrouve = ChercheEtTrouve()
}
```

La méthode est appelée par son nom.

- A partir d'une autre classe

Le nom de la méthode doit être précédé du nom de la classe qui contient la méthode qui aura été déclarée avec le mot clé **public**.

Une méthode non déclarée **public** est une méthode privée (**private**) pour la classe ; cette méthode ne pourra être appelée que de la classe ou elle est définie.

```
static void AfficheMessage ()
private static void AfficheMessage ()
```

Les 2 notations sont équivalentes.

```
namespace Fonctions
{
 class Test1
 {

  // programme principal
  static void Main()
  {
   Console.WriteLine("Programme principal");
   // ....
   Test2.AfficheMessage();
   // ....
   Test2.AfficheMessage();
  }
 }
 class Test2
 {
  // définition de la fonction
  public static void AfficheMessage()
  {
   Console.WriteLine("Ceci est un message");
  }
 }

}
```

Appel de la méthode de la classe Test2

Déclaration de la méthode **public**

- A partir d'une autre méthode (méthodes imbriquées)

Il est possible également d'appeler une méthode à partir d'une autre méthode.

```
namespace Fonctions
{
 class Test1
 {

  // définition des fonctions
  static void AfficheMessage()
  {
   Console.WriteLine("Ceci est un message");
  }
  static bool ChercheEtTrouve()
  {
   bool btrouve;
   // ....
   AfficheMessage();
   // ....
   return btrouve;
  }

  // programme principal
  static void Main()
  {
   Console.WriteLine("Programme principal");
   // ....
   AfficheMessage();
   // ....
   AfficheMessage();
```

`AfficheMessage` est appelée à partir de `ChercheEtTrouve`

```
    // ....
    bool btrouve = ChercheEtTrouve()
   }
  }
```

Le corps de méthode

Les variables

Des variables peuvent être déclarées dans le corps de méthode : elles sont locales, crées au début de la méthode et détruites à la sortie

```
   static void ChercheEtTrouve ()
   {
    bool btrouve = false;
    // ....
   }
```

Pour partager une des informations entre méthodes, on peut utiliser une variable de classe.

```
namespace Fonctions
{
 class Test1
 {
  // définition de la variable de classe
  static string strMess;
  // définition les fonctions
  static void EtablitMessage()
  {
   strMess = "Ceci est un message";
  }

  static void AfficheMessage()
  {
   Console.WriteLine(strMess);
  }
  // programme principal
  static void Main()
  {
   Console.WriteLine("Programme principal");
   // ....
   EtablitMessage();
   // ....
   AfficheMessage();
   // ....
  }
 }
```

L'instruction return

L'instruction **return** dans une fonction permet le retour immédiat à l'appelant.

Dans le cas de l'exemple précédent :

```
static void Main()
{
 Console.WriteLine("Programme principal");
 // ....
 EtablitMessage();
 return;
 AfficheMessage();
 // ....
}
```

L'ajout de l'instruction return entre l'appel des 2 fonctions provoque un avertissement du compilateur « impossible d'atteindre le code détecté » ; le programme n'exécute jamais la fonction `AfficheMessage`.
Il est plus courant d'utiliser l'instruction **return** dans une expression conditionnelle : le retour au niveau supérieur se fait que si la condition est vraie.

Dans le cas de méthode renvoyant un résultat, l'instruction **return** suivie d'une expression termine la méthode immédiatement en retournant l'expression comme valeur de retour de la méthode.

```
static bool ChercheEtTrouve ()
{
 bool btrouve = false;
 // ....
 return btrouve;
}
```

L'instruction **return** est obligatoire dans ce cas.

Le passage de parametres

Les paramètres permettent de passer des informations à l'intérieur et à l'extérieur d'une méthode.

. Déclaration des paramètres

Chaque paramètre se caractérise par un type et un nom. Chaque paramètre est séparé de l'autre par une virgule.

```
static void AfficheMessage(string strMess, int nbfois)
{
 for (int i = 1; i <= nbfois; i++)
 {
   Console.WriteLine(strMess);
 }
}
```

. Appel de la méthode

Le code appelant doit fournir les valeurs des paramètres lors de l'appel de la méthode, dans l'ordre de définition.
Dans le cas de l'exemple précédent :

```
static void Main()
{
 Console.WriteLine("Programme principal");
 // ....
 string strMess = "Ceci le premier message affiché 2 fois";
 AfficheMessage(strMess, 2);
 // ....
 strMess = "Ceci est le 2eme message";
 AfficheMessage(strMess, 1);
 // ....
}
```

C# utilise par défaut le mécanisme de passage des paramètres par valeur : les données sont transférées de l'extérieur de la méthode vers l'intérieur : ce sont des **paramètres d'entrée**.

Méthode de passage de paramètres

Il existe trois façons de passer des paramètres :

- **Par valeur** (paramètres **d'entrée**)
  Les données sont transférées de l'extérieur vers l'intérieur de la méthode.

  La valeur du paramètre est copiée : la variable peut être modifiée à l'intérieur de la méthode sans influence sur sa valeur à l'extérieur.

```
namespace Fonctions
{
 class Test2
 {
  // définition de la fonction
  static void Incremente(int n)
  {
   n++ ;
  Console.WriteLine("n= {0}",n); // Affiche 2
  }

  // programme principal
  static void Main()
  {
   Console.WriteLine("Programme principal");
   // ....
   int x = 1;
   Incremente(x);
   Console.WriteLine("x= {0}", x); // Affiche 1

   Console.ReadLine();
  }
 }
```

```
}
```

Le paramètre doit être initialisé avant l'appel de la fonction ; Après l'appel de la fonction, la valeur affichée est toujours 1 : Il n'y a pas de retour de la valeur de la fonction vers l'appelant.

- **Par référence** (paramètres **d'entrée/sortie**)
  Les données peuvent être transférées de l'extérieur vers l'intérieur, puis vers l'extérieur de la méthode.

  On utilise le mot clé **ref** par paramètre, pour spécifier un appel par référence, dans la méthode et lors de son appel.

```
namespace Fonctions
{
 class Test2
 {
  // définition de la fonction
  static void Incremente (ref int n)
  {
   n++ ;
  Console.WriteLine("n= {0}",n); // Affiche 2
  }

  // programme principal
  static void Main()
  {
   Console.WriteLine("Programme principal");
   // ....
   int x = 1;
   Incremente (ref x);
   Console.WriteLine("x= {0}", x); // Affiche 2

   Console.ReadLine();
  }
 }
}
```

Le paramètre doit être initialisé avant l'appel de la fonction ; Après l'appel de la fonction, la valeur affichée est maintenant 2 : Il y a retour de la valeur de la fonction vers l'appelant.

- **Paramètres de sortie**
  Les données ne peuvent être transférées que de l'intérieur vers l'extérieur de la méthode.

  On utilise le mot clé **out** par paramètre, pour spécifier un appel avec paramètre de sortie, dans la méthode et lors de son appel.

```
namespace Fonctions
{
 class Test3
 {
  // définition de la fonction
  static void Incremente (out int n)
  {
   n=2 ;
   n++ ;
  Console.WriteLine("n= {0}",n); // Affiche 3
  }

  // programme principal
  static void Main()
  {
   Console.WriteLine("Programme principal");
   // ....
   int x ;
   Incremente (out x);
   Console.WriteLine("x= {0}", x); // Affiche 3

   Console.ReadLine();
  }
 }
}
```

Dans la fonction, le paramètre doit être initialisé avant d'être utilisé ; Après l'appel de la fonction, la valeur affichée est 3 : Il y a retour de la valeur de la fonction vers l'appelant.

La récursivité

Le langage C# permet les fonctions récursives. C'est-à-dire que l'une des instructions d'une fonction peut être un appel à la fonction elle-même. C'est très pratique pour coder certains algorithmes comme la factorielle :

factorielle(n) = n * factorielle(n - 1)

ou l'algorithme d'Euclide :

si n2 > n1, pgcd(n1, n2) = pgcd(n1, n2 - n1)

Ce principe est basé sur une notion mathématique : la *récurrence* :
Pour démontrer qu'une propriété est vraie quelle que soit la valeur de n, on démontre que :

- La propriété est vraie pour n=1.
- Si la propriété est vraie pour n-1, elle est vraie pour n.

Ainsi, si les deux théorèmes précédents sont démontrés, on saura que la propriété est vraie pour n=1 (1er théorème). Si elle est vraie pour n=1 elle est vraie pour n=2 (2ème théorème). Si elle est vraie pour n=2, elle est vraie pour n=3... Et ainsi de suite.

La création d'une fonction récursive risque d'engendrer un phénomène sans fin. C'est pourquoi, on prévoira toujours une fin de récursivité. Cette fin correspond en fait au codage du premier théorème :

```
static long SommeDesNombres(long n)
{
    if (n == 1)
    {
        return 1;                    //  1er théorème
    }
    else
    {
        return n + SommeDesNombres(n - 1); // 2ème théorème
    }
}
```

Les méthodes surchargées

Le langage C# permet que deux fonctions différentes aient le même nom à condition que les paramètres de celles-ci soient différents soit dans leur nombre, soit dans leur type, soit dans le mode de passage du paramètre (out ou ref).

Par exemple, les trois fonctions prototypées ci-dessous sont trois fonctions différentes. Cette propriété du langage C# s'appelle la *surcharge*.

```
double maFonction(double par1);
double maFonction(double par1, double par2);
double maFonction(int par1);
```

Contrairement aux autres langages de programmation, ce qui permet au compilateur d'identifier et de distinguer les sous-programmes, ce n'est pas le nom seul de la fonction, mais c'est la *signature*. Deux fonctions sont identiques si elles ont la même signature, c'est-à-dire le même **nom**, le même **nombre de paramètres** de même **type** et de même **mode** de passage.

Le nom du paramètre et le type de retour de la fonction n'affectent pas la signature par exemple, ces 2 fonctions sont identiques, et ne pourront donc pas être déclarées dans la même classe.

```
double maFonction(double par1);
int maFonction(double parx);
```

Exemple :

```
class Exemple
{
 static int Add(int a, int b)
 {
  return a + b;
 }
 static int Add(int a, int b, int c)
```

```
    {
     return Add(a,b)+ c;
    }
    static void Main()
    {
     Console.WriteLine(Add(1, 2) + Add(1, 2, 3));
    }
  }
```

La méthode **Add** a été surchargée pour pouvoir traiter l'addition de 3 entiers. On notera que dans le corps de la méthode surchargée, on utilise la méthode initiale, pour éviter la redondance de code.

Il est évident que l'abus de cette possibilité peut amener à écrire des programmes difficiles à maintenir.

Cependant, cela permet de donner le même nom à des fonctions qui jouent le même rôle mais dont le nombre et le type de paramètres doivent être différents. C'est le cas, par exemple, de la fonction **`IndexOf`** de la classe **`String`** qui peut recevoir 1, 2 voire 3 paramètres. Dans tous les cas, le premier paramètre est la portion de chaîne (ou le caractère) que l'on veut repérer dans la chaîne globale (voir paragraphe II.3.b du présent support).

Exercices

Table de multiplication
Ecrivez une fonction qui affiche une table de multiplication.
Votre fonction doit prendre un paramètre qui permet d'indiquer quelle table afficher.

```
TableMultiplication(7);
```

Doit afficher :

```
1 x 7 = 7
2 x 7 = 14
3 x 7 = 21
…
```

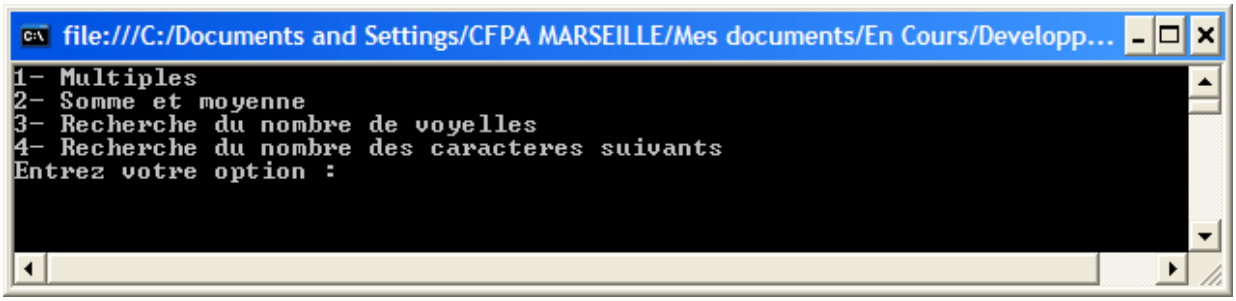
Compter le nombre de lettres
Ecrivez une fonction qui prend deux paramètres :
phrase de type string
lettre de type string
La fonction compte le nombre de fois ou **lettre** apparait dans **phrase**

La fonction pourrait s'utiliser de la façon suivante :

```
Console.Write(CompteLettre("combien de fois", "o"));
```

Cet exemple devrait afficher 2.

Menu
A partir du menu affiché à l'écran



Vous exécuterez, par les 3 premières options, les exercices réalisés, appelés sous forme de fonction (transformation de vos TP).

L'option 4 est une généralisation de la recherche du nombre de voyelles dans un mot : elle permet de rechercher la présence de n'importe quel caractère dans une chaîne.

La recherche de voyelles dans une chaîne constitue une surcharge de cette fonction, dans la mesure où les caractères à rechercher seront fournis sous forme de chaîne.

Créer 2 fonctions supplémentaires pour automatiser la lecture de données standard au clavier, que vous appellerez systématiquement pour toute lecture clavier (modification de vos TP):

- une fonction **GetString** pour lire et restituer une chaîne de caractères au clavier,
- une fonction **GetInteger** pour lire et restituer un entier au clavier,

Calcul de factorielles
Ecrire une fonction qui implémente le calcul de factorielles.

n ! = n * (n-1) !
0 ! = 1

4 ! = 4 * 3 * 2 * 1

String Token

Concevez la function strtok qui prend trois paramètres en entrée et renvoie uen chaîne de caractères.

```
strtok (string str1, string str2, int n)
```

On suppose que str1 est composée d'une liste de mots séparés par le caractère str2. strtok sert à extraire le nième mot de str1.

Exemple

str1 = « robert ;dupont ;amiens ;80000 »

strtok (str1, « ; », 3) -> « amiens »