# Les fonctions

Les fonctions sont un élément central dans le développement avec C#. En effet, toutes les instructions d'une application écrite en C# doivent être placées dans des fonctions.

Chaque fonction représente une unité de traitement réutilisable pouvant avoir un ou plusieurs paramètres et retournant une valeur.

L'écriture de fonctions permet de structurer le code en découpant de manière logique les fonctionnalités développées. Il est recommandé, pour une bonne lisibilité et une bonne maintenabilité, de limiter la longueur des fonctions.

Nombre de développeurs préconisent ainsi une longueur permettant de voir le code complet d'une fonction dans "un écran". Cette longueur est toute relative, mais peut ainsi convenir à chacun. Cette règle n'est évidemment pas absolue mais elle peut aider, notamment dans le cas d'un travail en équipe, à la relecture et au débogage.

Pour tenir cet objectif, il est nécessaire de limiter les responsabilités des fonctions : chacune effectue un type de tâche uniquement.

## 1. Écriture d'une fonction

La syntaxe générale pour l'écriture d'une fonction est la suivante :

```
<modificateur d'accès> <type de retour> <Nom de méthode>(
[paramètre 1, [paramètre 2]...])
{
    //Traitements
    return <valeur>;
}
```

Le couple nom de méthode/liste de paramètres définit la signature de la fonction. C'est elle qui va permettre au compilateur de reconnaître l'existence d'une fonction lors de son appel.

Le type de retour de la fonction peut être n'importe quel type accessible par le code de la fonction.

Une fonction doit obligatoirement et dans tous les cas retourner une valeur explicitement. Pour cela, on utilise le mot-clé `return` suivi de la valeur à renvoyer.

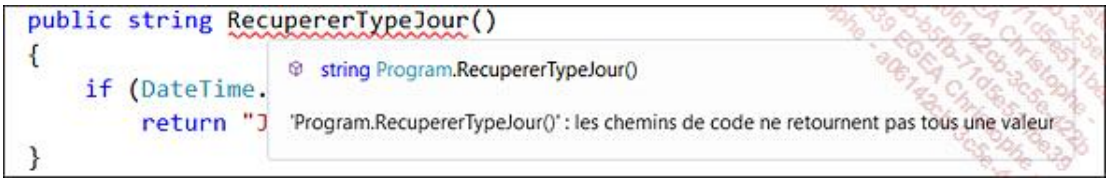
```
public int CalculerAgeDuCapitaine()
{
    int ageDuCapitaine = 0;

    //Traitements...

    return ageDuCapitaine;
}

public string RecupererTypeJour()
{
    if (DateTime.Today.Day % 2 == 0)
           return "JOUR PAIR";
}
```

Ici, la fonction `RecupererTypeJour` n'est pas valide. En effet, elle ne retourne une valeur que si le jour courant est pair. Le compilateur indique d'ailleurs ce problème :




Corrigeons donc cette méthode que le compilateur ne saurait voir :

```
public string RecupererTypeJour()
{
    if (DateTime.Today.Day % 2 == 0)
           return "JOUR PAIR";
    //Ici, on gère tous les cas autres, donc les jours impairs
    return "JOUR IMPAIR";
}
```

Cette fois, le processus de compilation se termine sans encombre.

## 2. Paramètres de fonction

Il est fréquent de devoir effectuer des traitements dépendants de données particulières. Pour cela, les fonctions peuvent avoir des paramètres.

Les paramètres sont des variables locales à une fonction, mais dont la valeur est fournie à partir d'un appel à cette fonction. On déclare un paramètre en précisant son type et son nom.

```
public decimal CalculerPrixTTC(decimal prixHT)
{
    //Application de la TVA à 20%
    return prixHT * 1.2;
}
```

Nous pouvons appeler notre fonction de la manière suivante :

```
decimal prix = CalculerPrixTCC(10.25);
```

### Paramètres optionnels

Il est possible d'ajouter des paramètres optionnels à notre fonction, c'est-à-dire des paramètres que le développeur pourra choisir de ne pas valoriser au moment de l'appel de la fonction. Pour utiliser cette fonctionnalité, il est obligatoire de fournir une valeur par défaut à chacun des paramètres optionnels.

Les paramètres optionnels doivent être les derniers paramètres déclarés dans la signature de la fonction.

Modifions la fonction précédente de manière à avoir deux paramètres optionnels correspondant au taux de TVA et à une réduction à appliquer sur le prix TTC :

```
public decimal CalculerPrixTTC(decimal prixHT, decimal tauxTVA =
0.2, decimal montantReduction = 0)
{
    //Application de la TVA et de la réduction
    return prixHT * (1 + tauxTVA) - montantReduction;
}
```

Il est maintenant possible d'appeler cette fonction des manières suivantes :

```
//Application de la TVA par défaut
decimal prix = CalculerPrixTTC(10.25);

//Application d'une TVA à 10%
decimal prix2 = CalculerPrixTTC(10.25, 0.1);

//Application d'une TVA à 10 % et d'une réduction de 1
decimal prix3 = CalculerPrixTTC(10.25, 0.1, 1);
```

### Paramètres nommés

Pour calculer un prix TTC avec une TVA de 20 %, soit la TVA par défaut, et une remise spécifique, nous avons la possibilité d'appeler la méthode `CalculerPrixTTC` en spécifiant la valeur de tous les paramètres. Mais ceci induit une redondance inutile au niveau du paramètre optionnel `tauxTVA`. Cette redondance deviendrait d'autant plus inutile et difficile à maintenir que le nombre de paramètres optionnels augmenterait.

L'utilisation de paramètres nommés est toute indiquée pour résoudre ce problème.

Il est en effet possible de ne spécifier que les paramètres optionnels dont nous avons besoin en précisant le nom des paramètres en même temps que leur valeur.

Pour la méthode `CalculerPrixTTC`, ceci donnerait l'appel suivant :

```
decimal prix = CalculerPrixTTC(10.25, montantReduction: 1);
```

Il est aussi possible de spécifier le nom de tous les paramètres passés, et de modifier l'ordre des paramètres dans l'appel, comme montré ci-dessous :

```
decimal prix = CalculerPrixTTC(tauxTVA: 0.1, montantReduction:
1, prixHT: 10,25);
```

### Paramètres variables

Certaines fonctions doivent accepter un nombre variable de paramètres. C'est notamment le cas de la fonction `Main` d'une application console.

En effet, dans le cas d'une application console, la fonction `Main` prend en paramètres tous les arguments fournis à l'application. Or, il est impossible de savoir à l'avance quels seront les arguments passés.

Dans un cas comme celui-ci, on définit dans la fonction un paramètre particulier, de type tableau et annoté avec le mot-clé `params`. Lors de l'appel à cette fonction, il est ainsi possible de spécifier un nombre variable de valeurs qui sont regroupées dans un seul paramètre.

```
public int AdditionnerPlusieursEntiers(params int[] entiers)
{
    int resultat = 0;
    for (int compteur = 0; compteur < entiers.Length; compteur++)
    {
           resultat += entiers[compteur];
    }
    return resultat;
}
```

Cette fonction est utilisée des manières suivantes :

```
int resultatAddition = AdditionnerPlusieursEntiers(1, 5, 123, 4, -5);
int resultatAddition2 = AdditionnerPlusieursEntiers(1, 5);

int resultatAddition3 = AdditionnerPlusieursEntiers();
```

Il n'est possible de définir qu'un seul paramètre variable par fonction. Il est toutefois autorisé de définir des paramètres obligatoires avant le paramètre variable.

```
public string AdditionnerPlusieursEntiersAUnNombre(int unNombre,
params int[] entiers)
{
    int resultat = unNombre;
    for (int compteur = 0; compteur < entiers.Length; compteur++)
    {
           resultat += entiers[compteur];
    }
    return resultat;
}
```

Cette fonction est appelée de la même manière que la précédente, mais il est obligatoire de passer au moins un entier en paramètre lors de son appel.

### Paramètres en écriture : out et ref

Certains cas d'utilisations peuvent nécessiter de modifier une des valeurs passées en paramètre à une fonction. Par défaut, un paramètre d'une fonction est en lecture seule lorsque son type est un type valeur. Pour modifier ce comportement, C# introduit les mots-clés `out` et `ref`. Bien qu'ils permettent tous deux de répercuter la modification

d'une valeur de paramètre dans la fonction appelante, ils ne sont pas utilisés dans le même contexte ni de la même manière.

Le mot-clé `ref` devant le type d'un paramètre indique qu'une **variable initialisée** doit être fournie lors de l'appel de la fonction, et il indique également que cette valeur peut être modifiée par le code exécuté dans la fonction. Il est par exemple envisageable que la fonction de calcul de prix TTC écrite plus haut modifie la valeur de la remise appliquée lorsque celle-ci est trop importante.

```
public decimal CalculerPrixTTC(decimal prixHT, decimal tauxTVA,
ref decimal montantReduction)
{
    //Si la réduction est supérieure à 20% du prix HT,
    //on plafonne le montant à 20% de ce prix
    if (montantReduction > prixHT * 0.2)
    {
        montantReduction = prixHT * 0.2;
    }
   //Application de la TVA
   return prixHT * (1 + tauxTVA) - montantReduction;
}
```

Cette fonction est appelée par l'utilisation du code suivant :

```
decimal montantReduction = 4;
decimal prixTTC = CalculerPrixTTC(40, 0.2, ref montantReduction);
```

Le code appelant doit obligatoirement utiliser la fonction en utilisant le mot-clé `ref` devant le nom de la variable qui est passée par référence, et la variable doit obligatoirement avoir été initialisée.

Le mot-clé `out` est, quant à lui, utilisé lorsque plusieurs données doivent être retournées par une fonction. Ici, on peut imaginer que la fonction `CalculerPrixTTC` puisse retourner de manière classique le prix TTC d'un article, et qu'elle renvoie le pourcentage de réduction accordé relativement au montant HT à l'aide d'un paramètre marqué avec le mot-clé `out`.

```
public decimal CalculerPrixTTC(decimal prixHT, decimal tauxTVA,
decimal montantReduction, out decimal pourcentageReduction)
{
    //Calcul du pourcentage de réduction sur le montant HT
    pourcentageReduction = montantReduction * 100 / prixHT;

    //Application de la TVA
    return prixHT * (1 + tauxTVA) - montantReduction;
}
```

Dans ce cas, la fonction est appelée de la manière suivante :

```
decimal pourcentage;
decimal prixTTC = CalculerPrixTTC(40, 0.2, 5, out pourcentage);
```

Après exécution, la variable pourcentage contient la valeur calculée à l'intérieur de la fonction. On peut remarquer qu'elle n'a pas été initialisée avant l'exécution du calcul, comme aurait dû l'être une variable passée à l'aide du mot-clé `ref`.

## 3. Procédures

Il est possible d'écrire des fonctions ne renvoyant pas de valeur. Ces fonctions sont appelées procédures.

En pratique, on définit que leur type de retour est `void`. L'utilisation de ce mot-clé permet au compilateur de savoir qu'il n'est pas possible de placer un appel à une procédure à un endroit où une valeur est attendue (assignation ou calcul, par exemple).

```
public void UneProcedure()
{
    //Traitements

    //Mot-clé facultatif dans une procédure
    return;
}
```

Dans une procédure, le mot-clé `return` est totalement facultatif. Il peut néanmoins être intéressant d'utiliser ce mot-clé dans l'optique de raccourcir un traitement, notamment en fonction du contexte.

```
public void AllumerLaLumiere()
{
    bool lumiereEstAllumee = false;
    if (lumiereEstAllumee)
           return;

    //Si on arrive ici, la lumière est forcément éteinte
    AppuyerSurInterrupteur();
    lumiereEstAllumee = true;
}
```

## 4. Surcharges

Le langage C# permet de définir plusieurs fonctions ou procédures ayant le même nom mais des signatures différentes, c'est-à-dire que les paramètres qu'elles peuvent accepter doivent différer de manière significative : le nombre ou le type des paramètres doit être différent. Ces fonctions sont appelées **surcharges**.

```
public int Additionner(int nombre1, int nombre2)
{
    return nombre1 + nombre2;
}

public int Additionner(int nombre1, int nombre2, int nombre3)
{
    return nombre1 + nombre2 + nombre3;
```

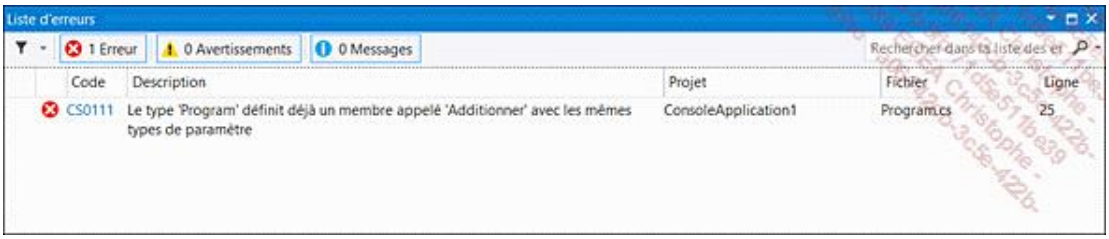
```
}
```

Ces deux fonctions peuvent tout à fait coexister : le compilateur C# est capable de savoir quelle fonction exécuter en fonction du nombre de paramètres passés dans l'appel.

```
public int Additionner(int nombre1, int nombre2, int nombre3 = 3)
{
    return nombre1 + nombre2 + nombre3;
}

public int Additionner(int nombre1, int nombre2, int nombre3)
{
    return nombre1 + nombre2;
}
```

Ici, on obtient une erreur de compilation. En effet, si l'on effectuait l'appel suivant, le compilateur serait dans l'incapacité de savoir quelle surcharge appeler.




Il est en revanche tout à fait possible d'écrire le code suivant :

```
public int Additionner(int nombre1, int nombre2)
{
    return nombre1 + nombre2;
}

public decimal Additionner(decimal nombre1, decimal nombre2)
{
    return nombre1 + nombre2;
}
```

Le compilateur est capable de savoir que la première fonction doit être exécutée lorsque l'on additionne deux entiers, et que la seconde fonction est adaptée pour les sommes de nombres décimaux.