Les structures de contrôle

Les structures de contrôle permettent de créer un flux de traitements complexe. Il existe deux types de structures de contrôle :

- Les structures conditionnelles, qui exécutent un traitement en fonction d'une condition.
- Les structures de boucles, qui exécutent plusieurs fois un même traitement.

1. Les structures conditionnelles

Deux structures conditionnelles existent.

a. if ... else

La structure `if ... else` accepte comme paramètre une expression renvoyant une valeur booléenne, comme une expression de comparaison. Si la condition est respectée, le bloc de code correspondant est exécuté.

S'il ne l'est pas, trois possibilités s'offrent à nous :

- Un bloc `else` est défini juste après le `if`, et dans ce cas le code de ce bloc est exécuté.
- Un ou plusieurs blocs `else if` existent après le `if`, et chaque expression de condition est exécutée jusqu'à ce qu'une d'entre elles renvoie `true`. À ce stade, le bloc de l'instruction `else if` (ou `else`, éventuellement) est exécuté.
- Aucun bloc `else` ou `else if` n'existe, et l'exécution du code reprend à la sortie du bloc `if`.

La syntaxe de cette structure est la suivante :

```
if (<expression de condition>)
{
    //Placer ici le code à exécuter si l'expression renvoie true
}
[ else if (<expression de condition 2>)
{
    //Placer ici le code à exécuter si la première expression
    //renvoie false et la seconde expression renvoie true
}]
[else if...]
[ else
{
    //Placer ici le code à exécuter si aucune des expressions
    //des blocs if et else if ne renvoie true
}]
```

L'utilisation de blocs `else if` ou `else` est facultative. Mais, très logiquement, un bloc `else` ou `else if` ne peut exister sans un bloc `if`.

Cette structure peut avoir des dizaines de blocs `else if` associés, mais elle ne peut avoir qu'un seul bloc `else`.

b. switch

La structure `switch` choisit une branche d'exécution en fonction d'un argument et de tests de valeurs de ce paramètre. L'argument passé à cette structure doit être de l'un des types suivants :

- `bool` ou `Nullable<bool>`
- `char` ou `Nullable<char>`
- `string`
- un des types numériques entiers : `byte`, `sbyte`, `short`, `ushort`, `int`, `uint`, `long` ou `ulong` ou un type `Nullable` correspondant (comme `Nullable<int>`).
- énumération (cf. chapitre La programmation orientée objet avec C# - section Les énumérations).

Le corps de cette structure est composé d'étiquettes `case` permettant chacune de tester une valeur. Il est aussi possible d'avoir une étiquette `default` qui gère les cas qui ne sont pas explicitement testés.

La syntaxe de la structure `switch` est la suivante :

```
switch (<argument>)
{
   case <valeur1> :
          //placer ici les traitements à effectuer
          break;
   case <valeur2>:
          //placer ici les traitements à effectuer
          break;
   [ case ... ]
   [ default :
          //placer ici les traitements à effectuer si notre
          //valeur n'a pas de case associée
          break; ]
}
```

En C#, contrairement à d'autres langages, comme C++, il est impossible d'exécuter un traitement dans une étiquette et de continuer en exécutant le code de l'étiquette suivante. Le mot-clé `break` sert ici à rendre ce comportement explicite. Si une étiquette contenant des traitements ne contient pas d'appel à cette instruction, le compilateur retourne une erreur.

En revanche, il est possible d'effectuer le même traitement pour deux étiquettes. Il suffit pour cela de placer deux étiquettes directement l'une au-dessus de l'autre.

Ci-dessous un exemple de structure `switch` montrant l'utilisation de ces différents comportements.

```
int maVariable = 4;

switch (maVariable)
{
    case 1:
        Console.WriteLine("Je suis un 1 !");
        break;
    case 2:
```

```
    case 3:
        Console.WriteLine("Je suis un 2 ou un 3 !");
        break;
    default:
        Console.WriteLine("Je ne suis ni un 1, ni un 2, ni un 3...
Qui suis-je ?");
        break;
}
```

Ce code produira la sortie suivante :

```
Je ne suis ni un 1, ni un 2, ni un 3... Qui suis-je ?
```

2. Les structures d'itération

Il est souvent intéressant de répéter le même traitement sur plusieurs variables, comme les éléments d'un tableau. Pour cela, C# fournit quatre types de structures capables d'exécuter en boucle un traitement.

a. for

La structure `for` est utilisée lorsque le nombre d'itérations à effectuer est connu avant de commencer l'exécution. Cette structure accepte trois arguments, séparés par des points-virgules, qui sont :

- **Une instruction d'initialisation** qui est exécutée avant de commencer les itérations. Cette instruction est en règle générale l'initialisation d'une variable "compteur".
- **Une condition évaluée avant chaque itération**. Si cette condition est évaluée à `false`, la boucle s'arrête. Cette instruction est en règle générale une comparaison de la variable "compteur" avec une valeur telle que la longueur d'un tableau.
- **Une instruction exécutée à la fin de chaque itération** et avant l'évaluation de la condition de sortie. Cette instruction est en règle générale une modification de la variable "compteur" par incrémentation ou décrémentation.

Ces trois instructions sont facultatives, mais attention, une boucle infinie est vite arrivée !

La syntaxe pour l'écriture de boucles `for` est la suivante :

```
for ([<instruction d'initialisation>]; [<condition de début
d'itération>]; [<instruction de fin d'itération>])
{
    //Code à exécuter à chaque itération
}
```

En règle générale, l'utilisation d'une boucle `for` ressemble à ceci :

```
int nombreIterations = 7;
for (int compteur = 0; compteur < nombreIterations; compteur++)
{
    //Code à exécuter à chaque itération
```

```
}
```

L'instruction de fin d'itération peut paraître étrange au premier abord. L'opérateur ++ est l'opérateur d'incrémentation. L'utilisation de l'instruction `compteur++` est équivalente à l'instruction :

```
compteur = compteur + 1;
```

b. while

La structure `while` est utilisée principalement lorsque le nombre d'itérations est complètement inconnu avant le début de l'exécution de la boucle. Elle accepte comme argument une expression booléenne. La boucle terminera son exécution lorsque cette condition renverra la valeur `false`.

Une boucle `while` s'écrit de la manière suivante :

```
while (<condition>)
{
   //Traitement à exécuter
}
```

c. do ... while

Les structures `do ... while` sont utilisées lorsqu'une itération au minimum doit être exécutée et que le nombre total d'itérations est inconnu avant le début de l'exécution.

Tout comme une boucle `while`, elle accepte une expression booléenne comme argument. Sa syntaxe est la suivante :

```
do
{
    //Traitement à exécuter au moins une fois
} while (<condition>)
```

d. foreach

Les boucles `foreach` sont des structures un peu particulières. Elles permettent d'itérer sur les éléments d'une collection. La collection utilisée doit respecter une condition : implémenter l'interface `IEnumerable`. Cette interface est notamment implémentée par les tableaux.


Le concept d'interfaces est traité au chapitre La programmation orientée objet avec C# - section Les interfaces.

La syntaxe de cette structure est la suivante :

```
foreach (<type> <nom de variable> in <collection>)
{
   //Traitements
```

```
}
```

L'écriture de boucle `foreach` n'étant pas forcément évidente, un exemple concret sera probablement plus parlant.

```
string[] arcEnCiel = {
"rouge", "orange", "jaune", "vert", "bleu", "indigo", "violet" };

foreach (string couleur in arcEnCiel)
{
    Console.WriteLine(couleur);
}
```

e. Contrôler l'exécution d'une boucle

Comme évoqué plus haut, il est très simple d'écrire une boucle infinie en C#. Pour en sortir, il suffit d'utiliser le mot-clé `break` que nous avons déjà vu avec la structure `switch`.

```
int i = 0;
//Création d'un boucle infinie
while (true)
{
    //Condition de sortie
    if (i == 5)
           break;
    i++;
}
```

Pour stopper un traitement dans une boucle et passer directement à l'itération suivante, il faut utiliser le mot-clé `continue`.

```
for (int i = 0; i < 20; i++)
{
   if (i == 5)
          continue;
   //Traitements spécifiques aux nombres différents de 5
}
```

3. Autres structures

Il existe deux autres structures en C#, la première étant nettement plus utilisée que la seconde.

a. using

La structure `using` est une aide syntaxique destinée à l'usage d'objets implémentant l'interface `IDisposable`. Les classes qui implémentent cette interface font généralement usage d'une ou plusieurs ressources externes, comme des fichiers. Ces ressources sont libérées par la méthode `Dispose`.

Le type `FileStream` représente un flux de données lié à un fichier. Il implémente l'interface `IDisposable` de manière à pouvoir libérer les ressources utilisées lorsque le traitement est terminé.

Sans l'instruction `using`, le code permettant son utilisation ressemble à celui-ci :

```
FileStream stream = new FileStream(@"C:\monfichier.txt",
FileMode.Open);

// Traitements...

stream.Dispose();
```

Ce type de code est sujet à problèmes car il ne donne pas l'assurance que la méthode `Dispose` sera toujours appelée. Le flux d'exécution peut en effet être interrompu par une erreur, ou l'appel peut même être omis par le développeur.

La réécriture de ce code à l'aide d'un bloc `using` donne le résultat suivant :

```
using (FileStream stream = new FileStream(@"C:\monfichier.txt",
FileMode.Open))
{
    //Traitements
}
```

L'appel à la méthode `Dispose` est généré par le compilateur de façon à être systématiquement effectué, même dans le cas d'une erreur d'exécution. De plus, il est impossible pour le développeur d'omettre l'appel à cette méthode puisque c'est le compilateur qui gère cet appel.

Pour ces raisons, l'utilisation de structures `using` est considérée comme une des bonnes pratiques de C#.

b. goto

La structure `goto` est un héritage de langages tels que le BASIC. Elle permet d'effectuer des sauts dans l'exécution du code, du point d'exécution courant vers une étiquette prédéfinie. Du fait qu'elle complique la lecture du code, cette structure est rarement utilisée dans du code C#, mais elle peut parfois se révéler très pratique, notamment pour sortir de traitements très complexes.

On l'utilise de la manière suivante :

```
<nom d'étiquette>:
//Traitements
goto <nom d'étiquette>;
```

L'instruction `goto` peut également se trouver avant la définition d'étiquette.

```
Console.WriteLine("Exécutons du code...");
```

```
goto etiquette;
Console.WriteLine("entre le goto et l’étiquette");

etiquette:
Console.WriteLine("après la définition de l’étiquette");
```

Ce code produit la sortie suivante :

```
Exécutons du code...  après la définition de l’étiquette
```