# 1 Bronze 2 : Opérations, Conditions et Boucles

De manière habituelle, l'exécution d'un programme informatique consiste en l'enchaînement linéaire d'une série d'instructions. Toutefois, pour accomplir des tâches plus complexes, il devient impératif de rompre cette linéarité. Ce module est structuré en quatre parties distinctes, chacune présentant un moyen d'y parvenir. La première partie se penche sur les opérateurs qui exercent une influence sur le résultat des instructions. La deuxième aborde les mots clés qui autorisent l'exécution conditionnelle de divers blocs de code. La troisième se consacre aux itérations multiples d'un même fragment de code, tandis que la dernière traite de la réutilisation de blocs de code.

## 1.1 Les opérations

C# fournit une série d'opérateurs s'appliquant aux types prédéfinis. Le tableau qui suit présente une liste non exhaustive des opérateurs à disposition.

| Opérateurs | Explications |
|---|---|
| $+, -, *, /, \%$ | Opérations arithmétiques |
| $==, !=$ | Opérations d'égalité |
| $<, >, <=, >=$ | Opérations de comparaison |
| $=, +=, -=, *=, /=, ++, --$ | Opérations d'affectation et d'incrément |
| &, && | ET, ET conditionnel |
| \|, \|\| | OU, OU conditionnel |
| ! | NON conditionnel |

Les opérateurs arithmétiques offrent la capacité d'effectuer des calculs mathématiques en respectant les règles de priorité conventionnelles. En particulier, l'opérateur % représente le reste issu de la division entière de deux opérandes (également appelé modulo). Pour une division entière de $x$ par $y$, le quotient résulte en $x \,/\, y$, tandis que le reste est donné par $x \,\%\, y$.

Exemple :

```
1  int x = 13; int y = 5;
2  // Division entière.
3  int quotient = x / y; // 2.
4  int reste = x % y;    // 3.
```

Les opérateurs d'affectation différents de = permettent d'affecter le résultat de l'opération arithmétique avec l'opérande de droite. Les opérateurs d'incrément permettent d'ajouter ou de soustraire 1 à la valeur de l'opérande associé. La place de l'opérateur est importante. Pour une valeur $x$ donnée, la valeur de $x++$ est $x$ alors que celle de $++x$ est $++x$.

Exemple :

```
1  int x = 13;
2  int y = 5;
3  // Opérateurs d'affectation :
4  x *= y; //  65, équivalent à x = x * y;
5  y -= x; // -60, équivalent à y = y - x;
6  // Opérateurs d'incrément
7  Console.WriteLine (x++); // x++ =  66.
8  Console.WriteLine (--y); // --y = -61.
```

Les opérateurs d'égalité et de comparaison produisent des booléens. Ils sont utiles notamment dans les structures présentées dans ce chapitre. Les opérateurs booléens jouent le même rôle, mais prennent comme opérandes des booléens.

Exemple :

```
1  int x = 13; int y = 2; int z = 3;
2  bool xSupy = x > y;                 // true.
3  bool ySupz = y > z;                 // false.
4  bool relation = xSupy && xSupz;     // false.
```

## 1.2 Les conditions

### 1.2.1 If-Else

Les conditions permettent d'obtenir un comportement différent en fonction des cas rencontrés. Les conditions sont des valeurs booléennes. Si la valeur est **true**, le bloc d'instruction est effectué, sinon il ne l'est pas. L'utilisation des conditions fonctionne comme en COBOL avec les mêmes mots clés **if**, **else**.

```
if (condition 1)
{
// bloc d'instructions 1.
}
...
else if (condition k)
{
// bloc d'instructions k.
}
else
{
// bloc d'instructions par défaut.
}
```

Exemple :

```
1  int entier = 255;
2  if (entier % 2 == 0)
3  {   Console.WriteLine ("L'entier est paire");   }
4  else
5  {   Console.WriteLine ("L'entier est impaire"); }
```

### 1.2.2 Opération ternaire

Lorsqu'un ensemble conditionnel **if-else** a pour unique objectif d'affecter une valeur ou une référence différente à une variable en fonction de la condition du **if**, il est possible de le simplifier à l'aide de l'opération ternaire. Cette forme plus concise permet d'assigner à la variable une valeur issue de l'expression de gauche si la condition est vraie, ou une autre issue de l'expression de droite dans le cas contraire.

```
variable = condition ? expr si true : expr si false;
```

Dans le cas suivant, les deux implémentations sont équivalentes.

Exemple :

```
1  int entier;
2  int entierA = 4;
3  int entierB = 8;
4
5  // bloc if-else
6  if (entierA > entierB)
7  { entier = 1; }
8  else
9  { entier = -1; }
10
11 // opération ternaire
12 entier = entierA > entierB ? 1 : -1;
```

L'opération ternaire fonctionne dans tous les cas où les expressions renvoient une valeur. Elle peut donc être utilisée dans une *assignation* (comme vu précédemment), comme *argument* d'un appel de fonction, ou dans une autre expression. Le type des valeurs retournées par ces expressions doivent être mutuellement convertibles.

Exemple :

```
1  // Exemple du bloc if-else revisité
2  int entier = 255;
3  Console.WriteLine (entier % 2 == 0 ? "L'entier est paire" :
4                                       "L'entier est impaire");
5  // l'opération ternaire renvoie un string qui est passé en argument
6  // de Console.WriteLine().
```

Les opérations ternaires peuvent être imbriquées, mais cela entrave la lisibilité du code produit.

### 1.2.3 Switch

Le **switch** est équivalent au **EVALUATE** présent en COBOL. Il prend en entrée une variable et teste sa valeur. Le mot clé **case** est suivi d'une valeur possible et **default** indique le cas par défaut (valeurs non testées précédemment).

```
switch (variable) // Test sur la valeur de la variable.
{
  case valeur 1 :
    // bloc d'instructions 1.
    break;
  ...
  case valeur k :
  case valeur k + 1 :
    // bloc d'instructions k + 1.
    break;
  ...
  default :
    // bloc d'instructions par défaut.
    break;
}
```

Exemple :

```
1  int entier = 4;
2  switch (entier)
3  {
4    case 2 :
5      Console.WriteLine ("Entier est égal à 2");
6      break;
7    case 3 :
8    case 4 :
9      Console.WriteLine ("Entier est égal à 3 ou 4");
10     break;
11   default :
12     Console.WriteLine ("Entier n'est pas égal à 2, 3 ou 4");
13     break;
14 }
```

Depuis le C#7.0, il est possible d'écrire des conditions multiples ou plus complexes dans un bloc **switch** en combinant le mot clé **case** avec une clause conditionnelle **when** (plus proche du fonctionnement présent en COBOL). Ainsi, au lieu de se limiter à comparer une simple valeur, on peut exécuter un bloc en fonction d'une expression booléenne. Lors de l'évaluation d'un **switch**, le premier case dont la condition est satisfaite provoquera l'exécution du bloc d'instructions qui lui est associé.

```
case type nom when condition
```

Entre le **case** et le **when**, il est possible de déclarer une variable locale qui peut servir dans la condition **when**. Cette variable reçoit la valeur passée au **switch**, si elle est compatible.

Exemple :

```
int entier = -8;
switch (entier)
{
    // match de entier dans n
    case int n when n > 0 && n % 2 == 0:
        Console.WriteLine("Entier est positif et pair");
        break;
    case int n when n > 0 && n % 2 != 0:
        Console.WriteLine("Entier est positif et impair");
        break;
    default:
        Console.WriteLine("Entier est nul ou négatif");
        break;
}
```

Pour le moment, on exécute une seule fois un code donné. Si on veut exécuter plusieurs fois le même bloc de code, il existe les boucles et les itérations.

## 1.3 Les boucles

### 1.3.1 Boucles à condition

L'exécution d'un bloc de code est répétée tant que la condition de la boucle est vraie. Dès qu'elle n'est plus vraie, on passe au bloc suivant. Il y a un risque de boucle infinie si la condition reste toujours vraie. Il existe deux types de boucles. La boucle où la condition est vérifiée avant l'exécution du bloc de code. Elle est obtenue avec le mot clé **while**. Elle n'est pas exécutée si la condition n'est jamais vraie.

```
while (condition)
{
  // bloc d'instructions
}
```

Exemple :

```
int n = 0;
while (n < 5)
{
  Console.WriteLine (n); // Affiche sur chaque ligne : 0, 1, 2, 3, 4.
  n++;
}
```

Le second type de boucle teste la condition après le bloc d'instructions qui est donc réalisé au moins une fois. On le construit avec les mots clés **do** et **while**.

```
do
{
  // bloc d'instructions
} while (condition);
```

Exemple :

```
1  int n = 0;
2  do
3  {
4    Console.WriteLine (n);
5    n++;
6  } while (n < 5); // Affiche sur chaque ligne : 0, 1, 2, 3, 4 et 5.
```

### 1.3.2 Boucles pas-à-pas

Les boucles pas à pas sont des boucles qui permettent de définir leur exécution dans sa déclaration. Il existe deux sortes de boucles pas à pas en C#. Elles sont définies à l'aide des mots clés **for** et **foreach**.

La boucle **for** est divisée en trois parties, l'initialisation, la condition et l'itération. Ces trois parties sont facultatives. La partie initialisation n'est exécutée qu'une seule fois. Elle sert généralement à déclarer une variable qui sera accessible dans le bloc d'instructions de la boucle et partager entre chaque exécution. Le bloc condition est testé à chaque exécution de la boucle, il s'agit d'une expression booléenne. La boucle continue si la condition est vraie. Le bloc itération est exécuté à la fin de l'exécution du bloc d'instructions de la boucle. Il permet en général de mettre à jour la variable définie dans le bloc initialisation.

```
for (initialisation; condition; itération)
{
  // bloc d'instructions
}
```

Exemple :

```
1  int somme = 0;
2  for (int i = 0; i < 100; i++) // Itération de 0 à 99.
3  {
4    somme += i;
5  }
```

La boucle **foreach** s'exécute sur chaque élément d'une collection (voir section Collections). L'élément est accessible en lecture dans l'exécution du bloc d'instructions.

```
foreach (var element in Enumerables)
{
  // bloc d'instructions
}
```

### 1.3.3 Instructions de saut

Les instructions de saut permettent de court-circuiter le déroulement normal d'une boucle. L'instruction **continue** permet de passer à l'itération suivante. L'instruction **break** permet de quitter la boucle.

Exemple :

```
int somme = 0;
for (int i = 0; i < 100; i++) // somme des entiers impaires de 0 à 10.
{
  if (i % 2 == 0)
    continue;

  somme += i;
  if (i > 10)
    break;
}
```

L'usage des instructions de saut requiert une approche prudente, car elles peuvent complexifier la compréhension du déroulement d'un traitement. En général, il est possible de remanier la structure des boucles de manière à éliminer leur usage.

## 1.4 Fonctions

La répétition de l'exécution d'un code grâce à l'utilisation de boucles, ainsi que l'adaptation du code en fonction de différentes conditions, ont été abordées précédemment. Cette section se penche maintenant sur la manière efficace de réutiliser des blocs de code en utilisant des fonctions.

### 1.4.1 Fondamentaux

Une fonction est constituée d'un ou plusieurs blocs de code qui sont déclarés et peuvent être utilisés au sein d'autres parties du code. Les fonctions ont la capacité de recevoir des données en entrée et de produire une sortie. Chaque fonction possède ce qu'on appelle une signature, qui sert à la définir. Cette signature comprend trois éléments : le **type** de la valeur de retour que renvoie la fonction, le **nom** de la fonction, et entre parenthèses, la liste des **paramètres** d'entrée.

```
typeRetour nomFonction (type1 nom1, type2 nom2, ..., typeN nomN)
{
  // Blocs d'instruction.
}
```

Le code ci-dessous est un exemple de déclaration de fonction :

```
int AdditionEntiers(int x, int y) // signature de la fonction.
{
  int z = x + y;                  // addition des deux paramètres d'entrée.
  return z;                       // renvoi de la valeur retour.
}
```

La première ligne correspond à la signature de la fonction. Le premier élément spécifie le type de retour de la fonction, dans l'exemple **int**. Lorsqu'une fonction ne renvoie aucune valeur, le type **void** est utilisé pour l'indiquer. Le deuxième élément est le nom de la fonction, ici **AdditionEntiers**, qui permet de l'identifier et de l'appeler dans le code. Les parenthèses contiennent la liste des paramètres,

troisième élément, utilisés par la méthode. Les parenthèses sont obligatoires même dans le cas où la fonction ne dispose pas de paramètres. La fonction ci-dessus contient deux paramètres, **x** et **y**, de type **int**.

Le bloc d'instructions dans une fonction permet d'effectuer une tâche qui renverra éventuellement une valeur en sortie. Par exemple, pour la fonction d'addition, les deux paramètres d'entrée sont additionnés, et le résultat est renvoyé en sortie. Pour cela, le mot-clé **return** est utilisé suivi de la variable contenant le résultat. Cependant, lorsque le type de retour est **void**, l'utilisation de **return** n'est pas nécessaire.

Pour utiliser une fonction déjà déclarée, il suffit d'utiliser son nom, suivi de la substitution des différents paramètres de la fonction par des variables. Ces variables sont appelées "arguments" de la fonction. L'affectation des arguments suit les règles de typage définies pour les paramètres. On peut utiliser des variables existantes, des valeurs littérales, ou même les résultats d'autres fonctions comme arguments.

Exemple : Utilisation de fonction

```
1 // Affectation de la valeur de retour à somme.
2 int somme = AdditionEntiers (5, 10); // arguments : 5 et 10.
3 Console.WriteLine (somme);           // 15.
```

Règles de base :

— Écrire des fonctions courtes.
  - Traditionnellement ne pas dépasser la taille d'un écran (ex : VT100, 25 lignes de 80 colonnes).
  - Ne pas dépasser 150 caractères par ligne et 100 lignes (rarement plus de 20 lignes).
  - Les blocs d'instructions de type **if**, **else**, **while**... doivent tenir sur une ligne, une bonne façon de faire est d'utiliser une fonction descriptive retournant un booléen.

— Une fonction doit faire une seule chose. Elle doit le faire bien et ne faire que cela.
  - Les blocs d'instructions d'une fonction doivent être au même niveau d'abstraction. S'il y a plusieurs niveaux d'abstractions, découper la fonction en question. Ne pas mélanger les niveaux d'abstraction.
  - Si une fonction peut être sectionnée, elle fait plusieurs choses.
  - La description d'une fonction doit pouvoir être décrite comme un paragraphe POUR.

— Une fonction de niveau d'abstraction élevée doit se trouver plus haute qu'une fonction de niveau plus bas. Règle de décroissance. Ensembles de paragraphes POUR qui se lisent de haut en bas.

### 1.4.2 Paramètres

Les paramètres sont spécifiés dans la signature de la fonction. Pour déclarer un paramètre, il faut indiquer son type, suivi du nom qui sera utilisé dans le corps de la méthode. Il est important de noter que le nom du paramètre est *indépendant* de celui de l'argument lors de l'appel de la méthode.

Il est également possible de définir une valeur par défaut pour un paramètre lors de sa déclaration. Si un paramètre possède une valeur par défaut, tous les paramètres à sa droite doivent également avoir une valeur par défaut définie. Lors de l'appel de la fonction, si un paramètre avec une valeur par défaut n'est pas explicitement fourni, la valeur par défaut sera utilisée à la place.

Exemple : Valeur par défaut

```
int SerieEntiers (int borne = 10)
{
  int somme = 0;
  for (int i = 0; i <= borne; i++)
  {
    somme += i;
  }
  return somme;
}
...
int resultat = SerieEntiers ();// borne = 10.
Console.WriteLine (resultat);  // 55.
```

### 1.4.3 Fonction récursive

Une fonction a la capacité d'appeler d'autres fonctions au sein de son corps et d'utiliser leurs valeurs de retour. Lorsqu'une fonction s'appelle elle-même à l'intérieur de sa propre définition, elle est qualifiée de *fonction récursive*. Chaque fois que la fonction est appelée, un nouvel espace de stockage est alloué aux paramètres, garantissant que les valeurs des appels précédents ne sont pas écrasées. Les paramètres sont accessibles uniquement au sein de l'instance actuelle de la fonction dans laquelle ils sont créés. Les instances précédentes de la fonction ne peuvent pas accéder directement aux paramètres des instances suivantes. L'un des principaux risques inhérent aux fonctions récursives est la création d'une boucle infinie d'appels récursifs. Pour éviter cette situation, il est essentiel de définir une condition qui permet de mettre fin à la récursivité, ce que l'on appelle une *condition de terminaison*.

Exemple : Fonction récursive de n!

```
uint Factorielle(uint n)
{
    // Condition de terminaison de la récurrence
    if (n >= 1)
    {
        return 1;
    }
    // Appel récursif
    return n * Factorielle(n - 1);
}
...
Console.WriteLine (Factorielle(3));  // 6.
```

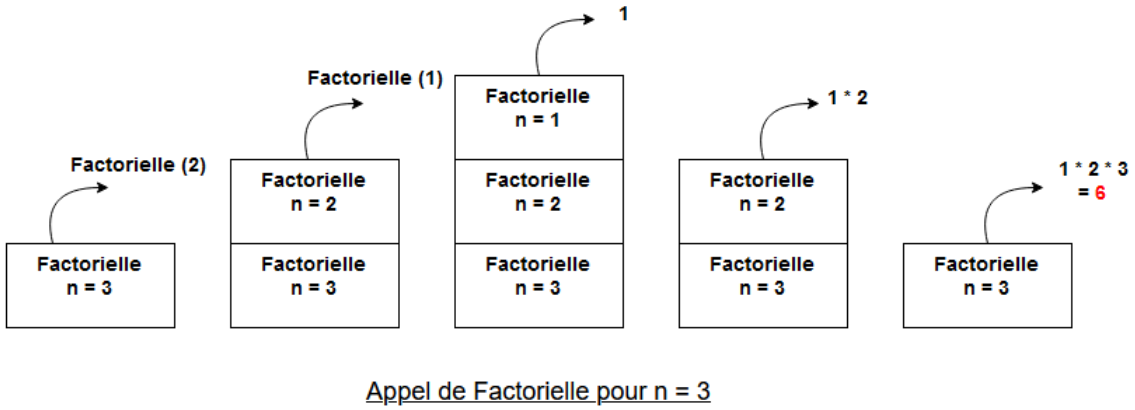


FIGURE 1 – Visualisation de la récursion de l'exemple

### 1.4.4 Bonnes pratiques autour des paramètres

Il est préférable de limiter autant que possible l'utilisation de paramètres, car ils peuvent rendre la compréhension des fonctions plus complexe et compliquer les tests. L'objectif des tests est de vérifier que les différentes combinaisons de valeurs possibles des arguments produisent les résultats attendus. Cela devient nettement plus ardu lorsque le nombre de paramètres augmente.

Nombre d'arguments d'une fonction :

- 0 argument (fonction niladique).
- 1 argument (monadique).
- 2 arguments (dyadique).
- 3 arguments (triadique).
- Plus d'arguments (polyadique).

Il est préférable de privilégier les fonctions *niladiques* chaque fois que cela est possible, tandis que les fonctions *polyadiques* devraient être évitées, sauf dans des situations exceptionnelles et rares.

Fonction monadique : Deux cas classiques d'utilisation de ces types de fonctions.

- Question sur le paramètre.
- Manipulation de l'argument, transformation en autre chose qui est retourné.

Les événements sont moins courants, avec un seul argument d'entrée et aucun argument de sortie. Cet argument a pour but de modifier l'état du système. Il est essentiel de s'assurer que la fonction de l'événement soit claire pour le lecteur. Dans les autres cas, il est préférable d'éviter les fonctions *monadiques*. Il est également recommandé d'éviter l'utilisation de paramètres de type *booléen*, car ils peuvent indiquer la présence de plusieurs comportements au sein d'une fonction.

Exemple : Fonctions monadiques

```
1  bool EstPositif(int entier) // Question sur le paramètre.
2  {
3    return entier > 0;
4  }
5
6  int Carre(int entier) // Manipulation de paramètre.
7  {
8    return entier * entier;
9  }
```

Fonction dyadique :

Les fonctions *dyadiques* présentent de l'intérêt lorsque les deux arguments sont ordonnés et interagissent de manière cohérente entre eux. Par exemple, lors du calcul de la distance à l'origine d'un point, il est cohérent d'utiliser deux paramètres pour les deux axes, x et y. Bien que les fonctions *dyadiques* puissent s'avérer utiles dans certaines situations, elles comportent un coût. Dans l'idéal, il est recommandé d'exploiter les techniques à notre disposition pour les transformer en fonctions *monadiques* lorsque cela est possible.

Exemple : Fonctions dyadiques

```
1  // Aisé à comprendre.
2  bool DistanceOrigine(int x, int y)
3  {
4    return Math.Sqrt (Math.Pow (x, 2) + Math.Pow (y, 2));
5  }
```

Fonction triadique :

Il s'agit du même problème en plus prononcé. A éviter dans l'extrême majorité des cas.