1 Module A.2 : Structuration fondamentale

L'exécution d'un programme informatique consiste à exécuter une suite de lignes d'instructions de haut en bas. Afin de réaliser des tâches complexes, de nombreuses indications permettent de perturber cette lecture linéaire. La première partie de ce module s'intéresse aux opérateurs qui influencent le résultat de certaines lignes. La seconde traite des mots clés permettant l'exécution conditionnelle de plusieurs blocs de codes. La troisième étudie les exécutions multiples d'un même morceau de code et la dernière de la réutilisation d'ensembles d'instructions.

1.1 Les opérations

C# fournit une série d'opérateurs s'appliquant aux types prédéfinis. Le tableau suivant fournit une liste non exhaustive des opérateurs disponibles.

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

Les opérateurs arithmétiques permettent d'effectuer des opérations arithmétiques en suivant les règles de priorité mathématique. L'opérateur % correspond au reste de la division entière des deux opérandes (le modulo). Pour une division entière de x par y, le quotient et le reste sont égaux à $x \,/\, y$ et $x \,\%\, y$ respectivement.

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
3  if (entier % 2 == 0)
4  {
5    Console.WriteLine ("L'entier est paire");
6  }
7  else
8  {
9    Console.WriteLine ("L'entier est impaire");
10 }
```

1.2.2 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
3  switch (entier)
4  {
5    case 2 :
6      Console.WriteLine ("Entier est égal à 2");
7      break;
8    case 3 :
9    case 4 :
10     Console.WriteLine ("Entier est égal à 3 ou 4");
11     break;
12   default :
13     Console.WriteLine ("Entier n'est pas égal à 2, 3 ou 4");
14     break;
15 }
```

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 :

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

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 :

```
1  int somme = 0;
2  for (int i = 0; i < 100; i++) // somme des entiers impaires de 0 à 10.
3  {
4    if (i % 2 == 0)
5      continue;
6
7    somme += i;
8
9    if (i > 10)
10     break;
11 }
```

Les instructions de saut sont à utiliser avec précaution, car elles rendent la compréhension du déroulement d'un traitement plus compliqué. Dans la plupart des situations, les boucles peuvent être réécrites pour ne plus en avoir.

1.4 Fonctions

On a vu comment exécuter plusieurs fois le même code à l'aide des boucles et comment déterminer les différents comportements à adopter à l'aide des conditions. On va voir comment réutiliser un bloc de code à l'aide des fonctions.

1.4.1 Fondamentaux

Une fonction est un ou plusieurs blocs de code déclarés et pouvant être utilisés dans d'autres blocs de code. Une fonction peut accepter des entrées et donner une sortie. Une fonction a ce que l'on appelle une signature qui permet de la caractériser. Cette signature contient le type de la variable de retour que renvoie la méthode, le nom de la méthode et entre parenthèses la liste des paramètres d'entrée.

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

Pour commencer, présentons un exemple :

```
1  int AdditionEntiers(int x, int y) // signature de la fonction.
2  {
3    int z = x + y;                  // addition des deux paramètres d'entrée.
4    return z;                       // renvoi de la valeur retour.
5  }
```

La première ligne correspond à la signature de la fonction. Le premier terme indique le type de retour de la méthode, ici **int**. Si la fonction ne renvoie pas de valeur, on indique le type **void** (néant). Le second terme correspond au nom de la fonction. Il permet de l'identifier et de l'appeler dans des blocs d'instruction. Les parenthèses contiennent la liste des paramètres utilisés par la méthode. Les parenthèses sont obligatoires même dans le cas où la fonction ne disposerait pas de paramètres.

Le bloc d'instructions consiste ici à additionner les deux paramètres d'entrée et le renvoyer en sortie. Pour faire cela, on utilise le mot clé **return** suivi de la variable. Celui-ci n'est pas nécessaire lorsque le type de retour est **void**.

Pour utiliser une fonction déclarée, il suffit d'utiliser le nom de la méthode suivie de l'affectation de variables à la place des différents paramètres, on parle dans ce cas d'arguments de la fonction. L'affectation suit la règle des différents types déclarés. On peut utiliser des variables déjà définies, des littérales, des résultats d'autres fonctions...

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.
```

Les sous-sections suivantes détaillent plusieurs aspects autour des fonctions. Pour l'instant, il est important de noter un certain nombre de principes de base pour écrire des fonctions.

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 déclarés dans la signature de la fonction. Pour déclarer un paramètre, il faut donner son type puis le nom qui sera utilisé dans le corps de la méthode. Ce nom est indépendant de celui de l'argument de la méthode lors de l'appel de la méthode.

Il est possible de définir une valeur par défaut à un paramètre lors de sa déclaration. Si un paramètre à une valeur par défaut, tous les paramètres à sa droite doivent avoir également défini une valeur par défaut. Dans le cas où une valeur par défaut est définie, lors de l'appel de cette fonction, il n'est pas nécessaire de donner une valeur à ce paramètre. La valeur par défaut sera utilisée si le paramètre ne reçoit pas de valeur explicitement.

Exemple : Valeur par défaut

```
1  int SerieEntiers (int borne = 10)
2  {
3    int somme = 0;
4    for (int i = 0; i <= borne; i++)
5    {
6      somme += i;
7    }
8    return somme;
9  }
10 ...
11 int resultat = SerieEntiers ();// borne = 10.
12 Console.WriteLine (resultat);  // 55.
```

1.4.3 Bonnes pratiques autour des paramètres

Les paramètres devraient être évités le plus possible, ils complexifient la compréhension des fonctions et la réalisation des tests. L'objectif des tests est de vérifier que les différentes combinaisons des valeurs possibles des arguments donnent les bons résultats. Ceci est bien plus difficile quand 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).

Les fonctions niladiques sont à privilégier lorsque c'est possible, les fonctions polyadiques sont à fuir sauf dans de très rares cas.

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.

L'événement est une forme moins fréquente, un argument d'entrée et aucun argument de sortie. Cet argument modifie l'état du système. S'assurer que la fonction de l'événement est claire pour le lecteur. Il faut éviter les fonctions monadiques dans les autres cas. On devrait éviter des paramètres de type booléen car ils indiquent l'existence de plusieurs comportements d'une fonction.

Exemple : Fonctions monadiques

```
1  // Question sur le paramètre.
2  bool EstPositif(int entier)
3  {
4    return entier > 0;
5  }
6
7  // Manipulation de paramètre.
8  int Carre(int entier)
9  {
10   return entier * entier;
11 }
```

Fonction dyadique :

Les fonctions dyadiques sont intéressantes quand les deux arguments sont ordonnés et cohérents entre eux. Par exemple, lorsqu'on calcule la distance à l'origine d'un point, il est cohérent de donner deux paramètres pour les deux axes x et y. Les fonctions dyadiques sont parfois utiles, mais ont un prix et on devrait utiliser les techniques à notre disposition pour les transformer en fonctions monadiques.

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.