1 Module A.3 : Maîtrise des données

C'est bien de pouvoir représenter des nombres et des caractères alphanumériques, mais on reste assez limité pour représenter le monde. Le COBOL fournit une équivalence de représentation pour tous les types précédents. On peut créer des types plus complexes qui sont adaptés à des besoins particuliers.

1.1 Types composés

1.1.1 La structure des données

Une manière simple de déclarer un type de données composé est d'utiliser le mot clé **struct**. Ce type déclaré est composé de types déjà définis, simples ou personnalisés. On peut même utiliser des structures de données dans des structures de données. Une fois celle-ci déclarée, on peut l'utiliser comme n'importe lequel des types vus précédemment.

Exemple : Point cartésien

```
1  // Déclaration de la structure Point.
2  struct Point
3  {
4    public double X; // Le public est important, à utiliser pour qu'il
5    public double Y; // soit accessible par les fonctions utilisant
6                     // la structure.
7  }
8  ...
9  // Utilisation de la structure Point.
10 Point point;
11 point.X = 10;
12 point.Y = 20;
13 Console.WriteLine(point.X);//10
```

On peut créer une fonction particulière qui permet d'initialiser tous ces champs (dans ce cas, c'est une obligation de tous les définir). On ne peut pas les initialiser dans la déclaration du type (sauf exception, comme la partie B de cette formation le montre).

Exemple : Point cartésien 2

```
1  // Déclaration de la structure Point.
2  struct Point
3  {
4    public double X; // Le public est important, à utiliser pour qu'il
5    public double Y; // soit accessible par les fonctions utilisant
6                     // la structure.
7
8    // Fonction d'initialisation.
9    public Point(double x, double y)
10   {
11     X = x;
12     Y = y;
```

```
  }
}
...
// Utilisation de la structure Point.
Point point = new Point(10,20); // Utilisation du mot clé new.
Console.WriteLine(point.X);//10
```

1.1.2 Enumérations

Toute une série de concepts de la réalité n'est pas exprimable telle quelle à l'aide de valeurs numériques ou des chaînes de caractères, car ces concepts sont plus limités que les types précédemment cités. Pour pallier à cette problématique, le type **enum** permet d'établir une concordance entre une énumération de termes et des valeurs numériques.

Exemple : Énumération des couleurs

```
// Déclaration du type Couleur.
enum Couleur
{
  Rouge,
  Marron,
  Vert,
  Bleu
}
```

Par défaut, chaque terme de l'énumération est associé à un type entier (**int** si non explicité). Si la valeur numérique du terme n'est pas explicitée, elle est obtenue en incrémentant de 1 celle du terme précédent.

Exemple : Énumération des jours de la semaine

```
// Déclaration du type Jour.
enum Jour : byte
{
  Lundi,          // 0 : valeur par défaut du type byte.
  Mardi = 15,
  Mercredi,       // 16.
  Jeudi,          // 17.
  Vendredi = 40,
  Samedi,         // 41.
  Dimanche = 100
}
```

1.2 Tableaux

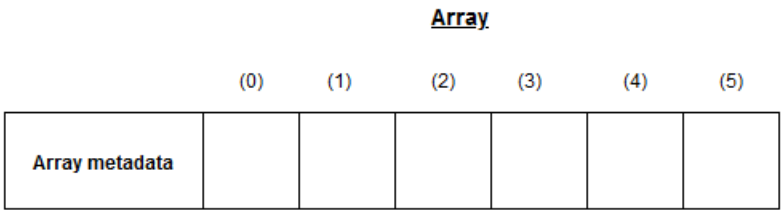
Après avoir vu des types de données plus complexes correspondant à des besoins spécifiques, il peut être intéressant de se demander comme représenter un certain nombre de variables de même type. Par exemple, on souhaite représenter les soldes des comptes bancaires de dix clients. On peut définir 10 entiers signés qui représenteront ces soldes. Cela fonctionne, mais s'il y a 100, 1000 voir 10 millions de clients, il est irréaliste d'utiliser cette solution. Il existe une fonction simple pour résoudre ce problème : les tableaux.

1.2.1 Tableau simple

Un tableau est un ensemble de données de même type stockées au même endroit dans la mémoire avec une petite enveloppe (overhead de 12 ou 16 bytes) fournissant des informations sur le tableau notamment sa taille. Lors de la déclaration d'un tableau, on doit lui donner une taille, si on n'initialise pas les valeurs, c'est la valeur par défaut du type qui est utilisé.

Exemple : Initialisation de tableaux

```
1 int[] tableau1 = new int[5]; // Tableau de taille 5, valeurs des éléments à 0.
2 int[] tableau2 = new int[] { 1, 3, 5, 7, 9}; // Initialisation des éléments.
3 int[] tableau3 = { 1, 3, 5, 7, 9, 11, 13};   // Notation alternative.
```



Caractéristiques importantes :

- Un tableau a une taille fixe, qui ne peut pas être modifiée après création. On peut obtenir cette valeur avec la propriété **Length**.
- Un tableau permet d'accéder à n'importe lequel de ces éléments (le stockage des éléments étant continu en mémoire, l'accès a un coût O [1]).
- Les éléments d'un tableau sont indexés de 0 à n-1 où n est la taille du tableau.

Exemple : Utilisation d'un tableau

```
1 int[] entiers = new int[] {1, 3, 5, 7, 9};
2 int somme = 0;
3
4 for (int i = 0; i < entiers.Length; i++) // de 0 à Length - 1.
5 {
6   somme += entiers[i]; // Ajout de l'élément d'indice i.
7 }
```

1.2.2 Tableaux multidimensionnels et en escalier

Un tableau simple n'a qu'une seule dimension. On peut déclarer des tableaux à plusieurs dimensions ainsi que des tableaux de tableaux (tableaux en escalier). Il existe plusieurs manières d'initialiser un tableau multidimensionnel ou en escalier. L'exemple ci-dessous permet de les mettre en lumière.

Exemple : Initialisation de tableaux

```
1  // Tableaux multi-dimensionnels.
2  // Tableau de dimensions (5,5), valeurs des éléments à 0.
3  int[,] tableauMd1 = new int[5,5];
4  // Initialisation des éléments.
5  int[,] tableauMd2 = new int[,] { {1, 3}, {5, 7}, {9, 10}};
6
7  // Tableaux en escalier
8  // Tableau en escalier, valeur des éléments à 0.
9  int[][] tableauEsc1 = new int[2][2];
10 // Initialisation des éléments.
11 int[][] tableauEsc2 = new int[][]
12 {
13   new int[] { -1, 0, 1},
14   new int[] { 5, 9, 15},
15   new int[] { 25, 29, 33}
16 }
```

Caractéristiques importantes :

- Les dimensions d'un tableau ne sont pas nécessairement identiques.
- L'accès est de même coût que pour les tableaux simples et se réalise de la même manière.
- Attention aux itérations sur les tableaux, tenter d'accéder à un élément qui n'existe pas provoquera l'arrêt du programme à l'exécution (pas d'erreur à la compilation). Il faut privilégier l'utilisation de **foreach** lorsque cela est possible pour éviter les erreurs.

Exemple : Utilisation d'un tableau multidimensionnel

```
1  int[][] entiers = new int[][]
2  {
3    new int[] { 1, 2, 3},
4    new int[] { 4, 5, 6},
5    new int[] { 7, 8, 9}
6  };
7  int somme = 0;
8
9  for (int i = 0; i < entiers.Length; i++)
10 {
11   for (int j = 0; j < entiers[i].Length; j++)
12   {
13   somme += entiers[i][j]; // Ajout de l'élément d'indice (i,j).
14   }
15 }
```

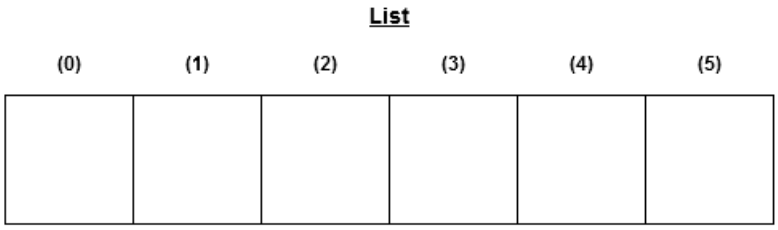
Params :
Le mot clé **params** permet de définir un nombre variable d'arguments de même type. Cette déclaration doit être définie en dernier. L'ensemble de ces paramètres est décrit dans la signature comme un tableau et il est utilisable dans le corps de la fonction de la même façon qu'un tableau. Étant donné le type de paramètre, on peut le considérer comme un seul paramètre.

Exemple : Utilisation de params

```
1  // Fonction monadique.
2  int SommeEntiers (params int[] tableau)
3  {
4    int somme = 0;
5    foreach(element in tableau)
6    {
7      somme += element;
8    }
9    return somme;
10 }
11 ...
12 int resultat = SommeEntiers (3, 5, 7); // resultat = 15.
```

1.2.3 Tableaux de taille dynamique

Les tableaux vus dans la section précédente ont une taille fixée à l'initialisation. Dans de nombreux contextes, le nombre d'éléments à stocker n'est pas connu à l'avance. Une solution simple consiste à choisir une taille largement supérieure aux besoins usuels, mais elle entraîne un gaspillage de ressources mémoires. Il existe une alternative plus élégante pour pallier à ce problème à savoir les listes dont le type associé est **List<T>**. Cette structure de données permet de représenter une série d'éléments d'un même type **T** accessible par son indice, de taille *dynamique*.



Le tableau suivant fournit une série de méthodes accessibles permettant d'interagir avec une liste.

Méthodes	Explications
void Add (**T** item)	Ajout d'un élément à la fin.
void AddRange (IEnumerable<**T**> collection)	Ajout d'une collection d'éléments à la fin.
void Insert (int index, **T** item)	Insertion d'un élément à l'indice index.
void InsertRange (int index, IEnumerable<**T**> collection)	Insertion d'une collection d'éléments à l'indice index.
bool Remove (**T** item)	Suppression de l'élément item.
void RemoveAt (int index)	Suppression de l'élément à l'indice index.
void RemoveRange (int index, int count)	Suppression des éléments des indices index à index + count.
List<**T**> GetRange (int index, int count)	Obtention des éléments des indices index à index + count.
void CopyTo (**T**[] array)	Copie l'ensemble des éléments dans array.
void CopyTo (**T**[] array, int arrayIndex)	Copie l'ensemble des éléments en partant de l'indice arrayIndex dans array.
void Clear ()	Suppression de tous les éléments de la liste.

List<**T**> encapsule un tableau de taille fixe qui est redimensionné lorsque la taille est insuffisante. L'ajout de nouveaux éléments est efficace (O(1) amorti, O(n) dans le pire des cas) mais l'insertion d'éléments est lente (O(n) car les éléments doivent être changés de place dans le tableau encapsulé). La suppression du dernier élément est efficace mais celle des autres est lente. Cette collection de données est appropriée lorsque la position des éléments dans la structure n'est pas importante, avec beaucoup d'ajouts, mais peu d'insertions ou de suppressions.

Exemple :

```
1  List<string> mots = new List<string>(); // Création d'une liste vide
2
3  mots.Add ("melon");                          // mots : melon.
4  mots.Add ("avocat");                         // mots : melon, avocat.
5  mots.AddRange (new[] { "banane", "prune"); // Insertion à la fin.
6  mots.Insert (0, "citron");                   // Insertion au début.
7  mots.InsertRange (0, new[] { "pêche", "fraise" }); // Insertion au début.
8
9  // mots : pêche, fraise, citron, melon, avocat, banane, prune.
10
11 mots.Remove ("melon");
12 mots.RemoveAt (3);          // Suppression du quatrième élément.
13 mots.RemoveRange (0, 2);    // Suppression des deux premiers éléments.
14
15 // mots : citron, banane, prune.
16
17 Console.WriteLine (mots[0]);                          // citron.
18 Console.WriteLine (mots[mots.Count - 1]);             // prune.
19 foreach (string s in words) Console.Write (s + " "); // citron banane prune.
20
21 List<string> sousEnsemble = mots.GetRange (1, 2);    // banane, prune.
```

1.3 Grands principes autour des types de données

Nous avons vu toute une série de types de données (et nous en verrons encore d'autres types plus tard dans cette formation). Il est important de présenter un certain nombre de principes sur les types de données.

1.3.1 Portée des variables

La portée des variables définit l'ensemble des blocs de code sur lesquels une variable est accessible. La portée d'une variable est en général celle du bloc de code dans laquelle celle-ci a été déclarée (des exceptions existent notamment lorsque des variables sont capturées, mais nous ne l'aborderons pas ici). Une erreur de portée entraînera une erreur de compilation.

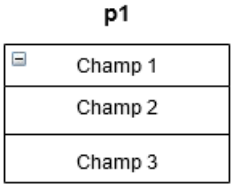
Exemple :

```
1  int somme = 0;
2
3  for (int i = 0; i <= 100; i++) // Portée de i : la boucle.
4  {
5    int entier = i; // entier a pour portée une itération de la boucle.
6    somme += entier;
7  }
8  // entier et i ne sont pas définis ici.
9  Console.WriteLine ("La somme des entiers de 0 à 100 : " + somme);
```

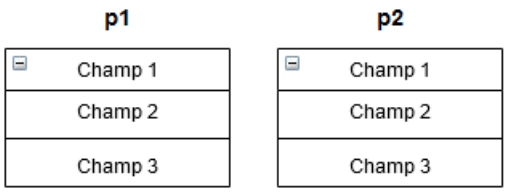
1.3.2 Types par valeur et par référence

En C#, il existe deux catégories de types à savoir les types par valeur et les types par référence. La différence se situe au niveau du contenu de la variable et de la gestion de la mémoire. Les types par valeur comprennent les types numériques entiers, décimaux, les booléens, les caractères ainsi que les structures et les énumérations. Les types par référence comprennent tous les tableaux, les collections (voir sous-section correspondante) et les classes.

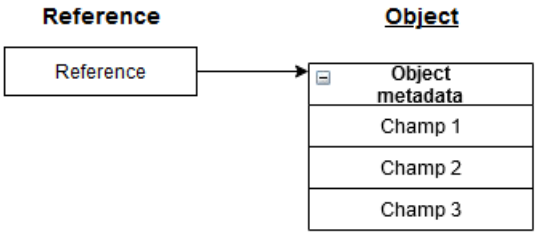
Types par valeur :
Une variable de type par valeur contient une instance de type, à savoir une valeur directement. Lors du passage en argument d'une fonction ou en valeur retour, la valeur de l'instance d'un type par valeur est copiée. Les variables de types par valeur sont gérées par la pile (Stack) interne à .NET. Le Stack est un des deux mécanismes de gestion de la mémoire. Le Stack (FIFO) ajoute les variables appartenant à une nouvelle portée lorsqu'il la rencontre et les élimine lorsqu'il la quitte.



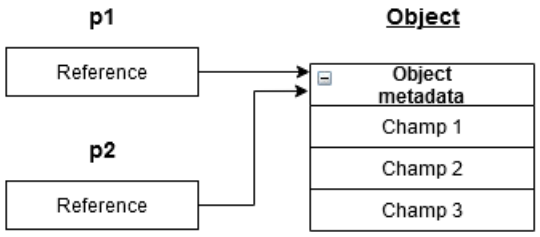
Par exemple pour illustrer la copie d'un type par valeur, prenons une structure Liste contenant trois champs, Champ1, Champ2 et Champ3. Si on copie une variable de ce type, on obtient ce que l'on voit dans le diagramme suivant.



Types par référence :
Une variable de type par référence contient une référence à une instance de type. Lors du passage en argument d'une fonction ou en valeur retour, seule la référence est copiée pour un type par référence (pour un type par valeur il faut utiliser le mot clé ref pour cela). Le point précédent permet de mettre en lumière qu'une instance de type par référence peut être référencée par plusieurs variables. Une modification de l'une de ces variables modifie toutes les autres. Attention aux effets de bords. Les variables de types par référence sont gérées par le tas (Heap). Le Heap est le second mécanisme de gestion de la mémoire. Le Heap gère la mémoire dynamiquement, il alloue de la mémoire lorsque des variables sont créées. Même quand une variable est hors de portée et qu'on ne peut pas l'utiliser, elle peut exister dans le tas. On ne peut pas prévoir le moment où l'espace mémoire utilisé sera libéré par le ramasse-miettes (on peut depuis les dernières versions donner des instructions au GC, mais on ne le contrôle pas).



Par exemple pour illustrer la copie d'un type par référence, prenons une instance de la classe Liste contenant trois champs, Champ1, Champ2 et Champ3. Si on copie une variable de ce type, on obtient ce que l'on voit dans le diagramme suivant.



1.3.3 Fonctions et effets de bords

Effets de bords :
La manière de représenter les types par valeur et par référence a une implication dans la manière de gérer les passages en paramètres des variables. Lorsqu'on une variable est passée en paramètre, le programme réalise une copie de cette variable et travaille avec cette copie. Cela implique que les modifications faites à la copie n'impacteront pas l'instance d'un type par valeur mais affectera celle d'un type par référence (comme la copie ne consiste qu'à copier la référence, les deux instances se partagent la même valeur). On parle d'effet de bord.

Exemple : Effets de bords

```
1  int MethodeParValeur (int x)
2  {
3    int carre = x * x;
4    int x = 0;            // Pourquoi pas ?
5    return carre;
6  }
7
8  int MethodeParReference (int[] x)
9  {
10   int somme = 0;
11   for (int i = 0; i < x.Length; i++)
12   {
13     somme += x[i];
14     x[i] = 0;           // Pourquoi pas ?
15   }
16 }
17 ...
18 int[] tableau = {1, 2, 3, 4};
19 int entier = 3;
20
21 int carre = MethodeParValeur (entier);
22 int somme = MethodeParReference (tableau);
23
24 // entier = 3;
25 // tableau = {0, 0, 0, 0};
```

Les effets de bords sont à proscrire dans une large majorité de cas, car les fonctions mentent dans ces situations. Elles modifient des éléments qu'elles ne devraient pas.

Passage par référence :
Il existe une série de mots clés pour indiquer que l'on veut passer par référence un argument à savoir **ref**, **in** et **out**. Ce mot clé doit être placé avant la déclaration du type de paramètre et avant le nom de la variable dans l'appel de la fonction. Le mot clé **ref** indique un passage par référence. Il est conseillé de marquer également les paramètres de type par référence pour clairement indiquer son intention de modifier cette variable. **in** indique un passage par référence où la variable ne peut pas être modifiée dans le corps de la fonction et **out** un passage par référence où la variable doit être forcément modifiée.

Exemple : Passage par référence

```
int MethodeParValeur (ref int x)
{
  int carre = x * x;
  int x = 0;            // Pourquoi pas ?
  return carre;
}
int entier = 3;

int carre = MethodeParValeur (entier);

// entier = 0;
```

Les paramètres de sortie heurtent la logique de fonctionnement des fonctions. Une fonction devrait fournir un résultat en valeur de retour ou modifier l'état de l'instance. Il faudrait les éviter dans la plupart des situations. La seule exception se produit dans le cadre des fonctions fournies pour convertir un type en un autre.

Exemple : Contre-exemple out

```
string str = "Hello world";
int entier;
// Tentative de conversion de str vers entier.
// Si conversion réussie, entier reçoit la valeur en sortie.
bool etatConversion = Int32.TryParse (str, out entier);
```