1 Bronze 3 : Tableaux

1.1 Tableaux

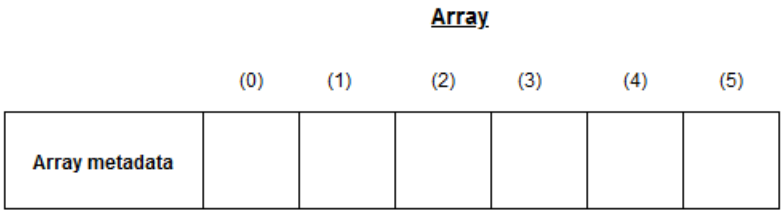
Après avoir vu des types de données plus complexes correspondant à des besoins spécifiques, il peut être intéressant de se demander comme représenter un certain nombre de variables de même type. Par exemple, on souhaite représenter les soldes des comptes bancaires de dix clients. On peut définir 10 entiers 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.

1.1.1 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   }
```

1.1.2 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][];
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.
```

1.1.3 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.
```