Les collections

Il est fréquent qu'une application doive manipuler de grandes quantités de données. Pour ceci, le framework .NET fournit plusieurs structures de données, regroupées sous l'appellation collections. Celles-ci sont adaptées à différents types de situations : le stockage désordonné de données disparates, le stockage de données par type, le stockage de données par nom...

1. Types existants

Les différentes classes permettant la gestion de collections sont regroupées dans deux espaces de noms :

- `System.Collections`
- `System.Collections.Generic`

Le premier comporte les types "classiques", tandis que le second comporte les classes génériques équivalentes permettant de travailler avec des objets fortement typés.

a. Array

La classe `Array` ne se trouve pas dans l'espace de noms `System.Collections`, mais elle peut tout de même être considérée comme une collection. Elle implémente en effet plusieurs interfaces propres aux collections : `IList`, `ICollection` et `IEnumerable`. Cette classe est la classe de base pour tous les tableaux utilisés en C#.

Cette classe n'est néanmoins pas très utilisée directement : on lui préfère dans la majorité des cas la syntaxe C#.

La classe `Array` étant abstraite, il n'est pas possible de l'instancier avec l'opérateur `new`. Pour cela, on utilisera l'une des surcharges de la méthode statique `Array.CreateInstance`.

```
Array tableau = Array.CreateInstance(typeof(int), 5);
```

Cette déclaration de variable est équivalente à celle-ci :

```
int[] tableau = new int[5];
```

b. ArrayList et List<T>

Les classes `ArrayList` et leur contrepartie générique `List<T>` sont des évolutions de la classe `Array`. Elles apportent certaines améliorations par rapport aux tableaux :

- La taille d'un objet `ArrayList` ou `List<T>` est dynamique et s'ajuste en fonction des besoins.
- Ces classes apportent des méthodes pour l'ajout, l'insertion ou la suppression d'un ou plusieurs éléments.

En revanche, les listes n'ont qu'une dimension, ce qui peut compliquer certains traitements.

Les collections de type `ArrayList` peuvent aussi poser des problèmes de performance car elles ne manipulent que des éléments de type `Object`, ce qui implique de nombreux transtypages. On leur préfère souvent

l'utilisation de tableaux fortement typés ou de listes génériques (`List<T>`) qui assurent, elles aussi, un typage fort.

La classe `ArrayList` peut être instanciée à l'aide d'un de ses trois constructeurs. Le premier ne prend aucun paramètre : l'objet créé aura ainsi une capacité initiale de zéro et sera dimensionné automatiquement lors du premier ajout d'un élément. Ce dimensionnement étant coûteux en termes de ressources, il est préférable, quand la taille finale de la liste est connue, d'utiliser le second constructeur qui prend comme paramètre une valeur numérique indiquant la capacité initiale de l'objet. Le troisième et dernier constructeur permet quant à lui d'instancier un objet `ArrayList` en fournissant une collection source, à partir de laquelle il doit être initialement rempli.

En cours d'utilisation, il est possible d'obtenir le nombre d'éléments dans un `ArrayList` en utilisant la propriété `Count`. La propriété `Capacity` renvoie quant à elle le nombre d'éléments que la collection peut contenir dans son état courant. Ces deux valeurs varient en fonction des ajouts et suppressions d'éléments.

Le code suivant illustre le fonctionnement de la classe `ArrayList` :

```
static void Main(string[] args)
{
    ArrayList liste = new ArrayList();
    Console.WriteLine("Nombre d'éléments dans la liste : {0}",
liste.Count);
    Console.WriteLine("Capacité initiale de la liste : {0}",
liste.Capacity);
    Console.WriteLine();

    Console.WriteLine("Ajout de 10 éléments dans la liste");
    for (int i = 0; i < 10; i++)
    {
        string valeur = "chaine " + i;
        liste.Add(valeur);
    }

    Console.WriteLine("Le nombre d'éléments dans la liste est
maintenant {0}", liste.Count);
    Console.WriteLine("Sa capacité est maintenant {0}",
liste.Capacity);

    Console.WriteLine("Les éléments présents dans la liste sont :");
    foreach (string chaine in liste)
    {
        //Le caractère \t est une tabulation, permettant
        //d'indenter notre liste d'éléments
        Console.WriteLine("\t{0}", chaine);
    }
    Console.WriteLine();

    Console.WriteLine("Ajout de 10 nouveaux éléments dans la liste");
    for (int i = 10; i < 20; i++)
    {
        string valeur = "chaine " + i;
        liste.Add(valeur);
    }
```

```
    Console.WriteLine("Le nombre d'éléments dans la liste est
maintenant {0}", liste.Count);
    Console.WriteLine("Sa capacité est maintenant {0}",
liste.Capacity);
    Console.WriteLine();

    string chaineIndex3 = (string)liste[3];
    string chaineIndex4 = (string)liste[4];
    string chaineIndex5 = (string)liste[5];
    string chaineIndex6 = (string)liste[6];
    Console.WriteLine("La chaine à l'index 3 de la liste est
'{0}'", chaineIndex3);
    Console.WriteLine("La chaine à l'index 4 de la liste est
'{0}'", chaineIndex4);
    Console.WriteLine("La chaine à l'index 5 de la liste est
'{0}'", chaineIndex5);
    Console.WriteLine("La chaine à l'index 6 de la liste est
'{0}'", chaineIndex6);
    Console.WriteLine();

    Console.WriteLine("Suppression des éléments aux index 3 et 4
    liste.RemoveRange(3, 4);

    chaineIndex3 = (string)liste[3];
    chaineIndex4 = (string)liste[4];
    chaineIndex5 = (string)liste[5];
    chaineIndex6 = (string)liste[6];
    Console.WriteLine("Les chaînes aux index 3, 4, 5 et 6 sont
maintenant '{0}', '{1}', '{2}' et '{3}'", chaineIndex3,
chaineIndex4, chaineIndex5, chaineIndex6);
    Console.WriteLine();

    Console.WriteLine("Insertion de la chaine 'élément inséré' à
l'index 3");
    liste.Insert(3, "élément inséré");

    chaineIndex3 = (string)liste[3];
    chaineIndex4 = (string)liste[4];
    chaineIndex5 = (string)liste[5];
    chaineIndex6 = (string)liste[6];
    Console.WriteLine("Les chaînes aux index 3, 4, 5 et 6 sont
maintenant '{0}', '{1}', '{2}' et '{3}'", chaineIndex3,
chaineIndex4, chaineIndex5, chaineIndex6);
    Console.WriteLine();

    Console.WriteLine("Suppression de tous les éléments de la liste");
    liste.Clear();

    Console.WriteLine("Le nombre d'éléments dans la liste est
maintenant {0}", liste.Count);
    Console.WriteLine("Sa capacité est maintenant {0}",
liste.Capacity);

    Console.ReadLine();
}
```

Il affiche le résultat suivant :

```
Ajout de 10 éléments dans la liste
Le nombre d'éléments dans la liste est maintenant 10
Sa capacité est maintenant 16
Les éléments présents dans la liste sont :
        chaine 0
        chaine 1
        chaine 2
        chaine 3
        chaine 4
        chaine 5
        chaine 6
        chaine 7
        chaine 8
        chaine 9

Ajout de 10 nouveaux éléments dans la liste
Le nombre d'éléments dans la liste est maintenant 20
Sa capacité est maintenant 32

La chaine à l'index 3 de la liste est 'chaine 3'
La chaine à l'index 4 de la liste est 'chaine 4'
La chaine à l'index 5 de la liste est 'chaine 5'
La chaine à l'index 6 de la liste est 'chaine 6'

Suppression des éléments aux index 3, 4, 5 et 6
Les chaînes aux index 3, 4, 5 et 6 sont maintenant 'chaine 7',
'chaine 8', 'chaine 9' et 'chaine 10'

Insertion de la chaine 'élément inséré' à l'index 3
Les chaînes aux index 3, 4, 5 et 6 sont maintenant 'élément
inséré', 'chaine 7', 'chaine 8' et 'chaine 9'

Suppression de tous les éléments de la liste
Le nombre d'éléments dans la liste est maintenant 0
Sa capacité est maintenant 32
```

Le code ci-dessus fonctionne exactement de la même manière en remplaçant la ligne :

```
ArrayList liste = new ArrayList();
```

par la ligne suivante utilisant la collection générique `List<T>` :

```
List<string> liste = new List<string>();
```

Afin de profiter au maximum de cette liste générique, il est recommandé d'éliminer tous les transtypages permettant de lire les valeurs dans la liste. Ils ne sont en effet plus nécessaires, l'objet `List<string>`

renvoyant déjà des objets de type `string`.

```
string chaineIndex3 = (string)liste[3];
```

devient alors :

```
string chaineIndex3 = liste[3];
```

c. Hashtable et Dictionary<TKey, TValue>

Les classes `Hashtable` et leur équivalent générique `Dictionary<TKey, TValue>` permettent de stocker des objets en fournissant pour chacun un identifiant unique. Celui-ci peut être une chaîne de caractères, un numérique, un objet `Type` ou tout autre objet. Les données peuvent par la suite être récupérées à l'aide de leur identifiant grâce aux indexeurs implémentés dans ces deux collections.

Ces types implémentent aussi la méthode `ContainsKey`, permettant de savoir si un identifiant existe dans la collection. Cette méthode peut s'avérer très pratique car tout accès à une clé qui n'existe pas génère une exception.

Le code ci-dessous montre le fonctionnement de la classe `Hashtable` :

```
static void Main(string[] args)
{
    Hashtable collection = new Hashtable();
    Console.WriteLine("Le nombre d'éléments initial est {0}",
collection.Count);
    Console.WriteLine();

    Console.WriteLine("Ajout de 10 éléments");
    for (int i = 0; i < 10; i++)
    {
        collection.Add(i, "chaîne " + i);
    }

    Console.WriteLine("Le nombre d'éléments est {0}",
collection.Count);
    Console.WriteLine();

    Console.WriteLine("Les éléments dans la collection sont :");
    foreach (int identifiant in collection.Keys)
    {
        string valeur = (string)collection[identifiant];
        Console.WriteLine("\tClé : {0} - Valeur : {1}",
identifiant, valeur);
    }
    Console.WriteLine();

    Console.WriteLine("Suppression de l'élément dont la clé est 3");
    collection.Remove(3);
```

```
    string cleExiste;
    if (collection.ContainsKey(3))
    {
        cleExiste = "OUI";
    }
    else
    {
        cleExiste = "NON";
    }
    Console.WriteLine("La clé 3 existe-t-elle ? {0}", cleExiste);
    Console.WriteLine("Le nombre d'éléments est {0}", collection.Count);
    Console.WriteLine();

    Console.WriteLine("Suppression de tous les éléments");
    collection.Clear();
    Console.WriteLine("Le nombre d'éléments est {0}", collection.Count);

    Console.ReadLine();
}
```

Ce code produit le résultat suivant :

```
Le nombre d'éléments initial est 0

Ajout de 10 éléments
Le nombre d'éléments est 10

Les éléments dans la collection sont :
        Clé : 9 - Valeur : chaîne 9
        Clé : 8 - Valeur : chaîne 8
        Clé : 7 - Valeur : chaîne 7
        Clé : 6 - Valeur : chaîne 6
        Clé : 5 - Valeur : chaîne 5
        Clé : 4 - Valeur : chaîne 4
        Clé : 3 - Valeur : chaîne 3
        Clé : 2 - Valeur : chaîne 2
        Clé : 1 - Valeur : chaîne 1
        Clé : 0 - Valeur : chaîne 0

Suppression de l'élément dont la clé est 3
La clé 3 existe-t-elle ? NON
Le nombre d'éléments est 9

Suppression de tous les éléments
Le nombre d'éléments est 0
```

Ce code peut être adapté très simplement pour utiliser un objet de type `Dictionary` en lieu et place de l'objet `HashTable`.

Il suffit en effet de remplacer la ligne :

```
Hashtable collection = new Hashtable();
```

par la ligne suivante :

```
Dictionary<int, string> collection = new Dictionary<int, string>();
```

Le type `Dictionary` étant fortement typé, il n'est plus nécessaire d'effectuer de transtypage pour récupérer une valeur de type `string`. La ligne :

```
string valeur = (string)collection[identifiant];
```

devient donc :

```
string valeur = collection[identifiant];
```

d. Stack et Stack<T>

Les piles (*stack* en anglais) sont des collections suivant le principe LIFO (*Last In, First Out*) : il n'est possible d'accéder qu'au dernier élément de la collection, et lorsque l'on ajoute un élément, celui-ci est automatiquement placé en dernière position de la collection.

Une analogie courante pour représenter ces objets est une pile d'assiettes. Chaque assiette ajoutée est posée en haut de la ligne, et lorsque l'on souhaite enlever une assiette de la pile, on prend celle qui se trouve au-dessus des autres. La dernière posée est la première à sortir de la pile.

Les trois principales opérations disponibles pour ces deux types sont :

- `Push`, pour ajouter un élément au sommet de la pile.
- `Peek`, pour récupérer l'élément au sommet de la pile sans l'ôter de la collection.
- `Pop`, pour récupérer le dernier élément de la pile en le supprimant de la collection.

e. Queue et Queue<T>

Les files (*queue* en anglais) suivent le même principe que les piles, mais un point majeur les différencie : les files suivent le principe FIFO (*First In*, *First Out*), ce qui signifie que le premier élément ajouté sera le premier à sortir de la file.

L'analogie évidente que l'on peut utiliser pour illustrer ce principe est la file d'attente : la première personne arrivée est la première à sortir de la file d'attente pour être accueillie à un guichet ou une caisse.

Comme les classes `Stack` et `Stack<T>`, les types `Queue` et `Queue<T>` implémentent trois méthodes principales :

- `Enqueue`, pour ajouter un élément à la fin de la fin.
- `Peek`, pour récupérer le premier élément de la file sans le supprimer de la collection.
- `Dequeue`, pour récupérer le premier élément de la file et l'enlever de la collection.

2. Choisir un type de collection

Il est très important d'identifier les besoins relatifs à une collection d'objets avant d'en choisir le type. En effet, la structure d'une classe entière, voire plus, peut dépendre d'un type de collection.

Voici une liste de points à prendre en considération pour choisir le type adapté à votre besoin :

- S'il est nécessaire de pouvoir accéder par index à un élément de la collection, optez pour un tableau, un `ArrayList` ou une `List<T>`.
- Si vous devez stocker des couples "clé unique & élément", utilisez une `Hashtable` ou un `Dictionary<TKey, TValue>`.
- Si vous devez accéder aux éléments dans l'ordre dans lequel ils ont été placés dans la collection, utiliser une `Queue` ou sa contrepartie générique. Si vous devez accéder aux éléments dans l'ordre inverse, utilisez une `Stack` ou une `Stack<T>`.
- Si vous devez stocker des éléments triés, utilisez les classes `ArrayList` ou `Hashtable`, ou éventuellement les collections plus avancées `SortedSet`, `SortedList` ou `SortedList<Tkey, TValue>`.