Ecriture de programmes simples
Les collections C#

François Boisson
francois.boisson@gmail.com

ESILV- Restart

(2018 - 2019)

ESILV LÉONARD DE VINCI | ÉCOLE D'INGÉNIEURS PARIS-LA DÉFENSE

Plan du cours

1. **Les tableaux** (rappel)
   - Les tableaux
   - Les matrices
   - Les tableaux de tableaux
2. **Les collections d'objets**
   - Les ArrayList
   - Les piles (Stack)
   - Les files d'attente (Queue)
3. **Les collections génériques**
   - **les listes <T>**
     - List <T>
     - Stack <T>
     - Queue <T>
   - **les tableaux associatifs <TKey,Tvalue>**
     - Dictionnary <TKey,Tvalue>
     - SortedList <TKey,Tvalue>

Les tableaux

1. Tableaux
2. Matrices
3. Tableaux de tableaux

Les tableaux

Déclaration

- int [ ] tab ; // référence seule
- int [ ] tab = new int[5] ; // avec définition de taille

Propriétés

- int taille = tab.Length ;

Utilisation - accès par l'index

- tab [5] = 7 ;
- Console.Write (tab[5]) ;

Les matrices

Déclaration

- int [,] mat ; // référence seule
- int [,] mat = new int[3,3] ; // avec définition de taille

Propriétés

- pour une matrice int [,] mat = new int [5,2]
- mat.Length vaut 5*2 = 10

Utilisation - accès par l'index

- tab [2,1] = 7 ;
- Console.Write (tab[2,1]) ;

Les tableaux de tableaux (ou tableaux en escalier)

Déclaration

- int [ ] [ ] tab ;
- int [ ] [ ] tab = new int [4] [3] ;

Exercice

Exercice

- définir un tableau de 10 éléments
- insérer dans ce tableau les nombres de 0 à 9
- afficher le contenu du tableau

- insérer dans ce tableau une 11ème valeur (le nombre 10)

Exercice

Comment faire ?

- Déclarer un tableau plus grand (de taille11)
- Recopier les 10 premiers éléments du tableau initial
- Insérer la valeur du 11ème élément

Exercice

Code

```
1  static void Main (string [] args) {
2    int [] tab = new int [10] ;
3    int i ;
4    for ( i = 0 ;    i<= tab.Length-1 ;   i++) {
5      tab [ i ] = i ;
6    }
7  //Creer tableau temporaire allonge
8    int [] tabTemp = new int [tab.Length+1] ;
9  //Recopier le contenu
10   for ( i = 0 ;   i<=tab.Length-1 ; i++) {
11     tabTemp [ i ] = tab[ i ] ;
12   }
13 //Ajouter la 11eme valeur
14   i++ ;
15   tabTemp[ i ] = i ;
16 //Rediriger le lien symbolique tab
17   tab = tabTemp ;
18 }
```

En guise de conclusion sur les tableaux

Les limites

- Des dimensions statiques
- Une organisation limitée malgré
  - matrices - tableaux de tableaux ;

Des extensions souhaitables

- Des dimensions dynamiques
- De nouvelles organisations de données
  - listes - piles - files - dictionnaires - arbres ;

Les collections d'objets

Objectif : disposer de tableaux à dimensionnement dynamique

Les collections d'objets

1. ArrayList ()
2. Stack () - (pile, ou tas)
3. Queue () - (file d'attente)
4. et autres ..... SortedList() .....

IMPORTANT

- Les collections d'objets font partie de System.Collections

[ϟ] il faut vérifier la présence en début du programme (ou l'ajouter) de la ligne de code : using System.Collections ;

Les ArrayList ()

ArrayList *(tableau unidimentionnel dynamique)* ou vecteur

- un ArrayList est un tableau dynamique **d'objets**
- un ArrayList grandit automatiquement pendant l'exécution
- on peut placer n'importe quel type d'objet dans un ArrayList. Pas de déclaration préalable de type comme int [ ]
- les objets contenus dans l'ArrayList peuvent même être de type différents (on parle alors de vecteur hétérogène)

IMPORTANT

- ArrayList (et les autres collections) font partie de System.Collections

[ϟ] il faut vérifier la présence en début du programme (ou l'ajouter) de la ligne de code : using System.Collections ;

Les ArrayList ()

Création

- Instancié à partir de la Classe System.Collections.ArrayList

  $\Rightarrow$ **ArrayList vect = new ArrayList ()**

Comme pour les tableaux :

✎ *Utilisation :*

- Accès en lecture/écriture par l'index : **vect[index]**

✎ *Propriété :*

- vect.**Count** $\Rightarrow$ le nombre d'éléments du vecteur

Mais, en plus des tableaux : des Méthodes

- ajouter un objet à la fin : void vec.**Add** (object value)
- insérer à la position index : void vect.**Insert** (int index, object value)
- supprimer 1ère occurr d'un objet : void vect.**Remove** (object value)
- supprimer l'objet à l'index spécifié : void vect.**RemoveAt** (int index)
- supprimer tous les objets : void vect.**Clear** ( )
- trier les objets d'un vecteur : void vect.**Sort** ( )
- et d'autres ....

Les ArrayList ()

Code (Exemple)

```
1 ArrayList vector = new ArrayList () ;
2 //
3 for ( int i = 0 ; i<=2 ;  i++ )
4   vector.Add ( "valeur" + i ) ;
5 //
6 Console.WriteLine ("taille du vecteur = " + vector.Count) ;
7 //
8 for ( int i = 0 ; i<=vector.Count-1 ;  i++ )
9   Console.WriteLine ("pos["+ i +"]="+ (string)vector[i] );
```

Résultat

Taille du vecteur = 3
pos[0] = valeur0
pos[1] = valeur1
pos[2] = valeur2

[ϟ] ATTENTION : ne pas oublier l'opérateur de cast pour les lectures.

Les collections d'objets

[ϟ] ATTENTION :
Un ArrayList est un tableau d'objets (non typés) : donc
un opérateur de cast est OBLIGATOIRE pour la lecture d'un élément d'un ArrayList

- Quand on écrit dans un ArrayList, on peut y placer n'importe quel type de données (transformé en la forme générale d'objet).
- **donc :**
- Quand on extrait un objet d'un arrayList, le programme ne connait pas le type de l'objet extrait et ne peut donc pas dimensionner d'espace mémoire pour l'objet lu.

  donc :
- Quand on lit, **il faut typer l'objet lu par un cast**.

[ϟ] remarque : c'est la même chose pour toutes les collections d'objets (piles, queues etc....)

Les collections d'objets

- Les ArrayList
- **Les piles (ou stack) LIFO et les files (ou queues) FIFO**

Stack () et Queue ()

Stack et Queue : *2 collections conçues pour conserver des éléments en attendant leur traitement*

- on ajoute des objets sur la pile (ou à la fin de la file)
- on peut placer n'importe quel type d'objet dans une Stack ou une Queue sans déclaration préalable de type
- donc quand on retire les objects de la pile ou de la file, il faudra les "caster" ((int) pile.Pop())
- on retire les objets de la pile suivant en mode LIFO (Last in - First out) et de la file en mode FIFO (First in - First out)

IMPORTANT

- Les Stack et Queue font partie de System.Collections

[ϟ] il faut vérifier la présence en début du programme (ou l'ajouter) de la ligne de code : using System.Collections ;

Stack () et Queue ()

Les Stack et Queue sont des objets créés (instanciés) à partir des classes System.Collections.Stack et System.Collections.Queue.

Création

=> **Stack tas = new Stack ( ) ;**

=> **Queue file = new Queue ( ) ;**

Propriété commune :

- tas.**Count** => le nombre d'éléments contenus
- file.**Count** => le nombre d'éléments contenus

Méthodes pour Stack (Lifo) :

- empiler un objet : void tas.**Push** (object item)
- dépiler et renvoyer l'objet du dessus de la pile : object tas.**Pop** ( )
- renvoit l'objet du dessus sans l'enlever : object tas.**Peek** ( )

Méthodes pour Queue (Fifo) :

- placer un objet à la fin : void file.**Enqueue** (object item)
- retirer et renvoyer l'objet au début : objet file.**Dequeue** ()
- renvoit l'objet du debut sans l'enlever : object file.**Peek** ( )

Stack () et Queue ()

Code (Exemple)

```
1  Stack pile = new Stack ( ) ;
2  Queue file = new Queue ( );
3  //
4  for ( int i = 0 ; i<=5 ;  i++ ) {
5     pile.Push ( "valeur" + i ) ;
6     file.Enqueue ( "valeur" + i ) ;
7  }
8  //
9  Console.WriteLine("1er element extrait de la pile (Lifo) = "
10             + (string) pile.Pop() ) ;
11 Console.WriteLine("1er element extrait de la file (Fifo) = "
12             + (string) file.Dequeue() ) ;
```

Résultat

1er element extrait de la pile (Lifo) = valeur5
1er element extrait de la file (Fifo) = valeur0

[ϟ] ATTENTION : ne pas oublier l'opérateur de cast pour les lectures

Les collections génériques

Collections d'objets pour un environnement fortement typé

Les collections génériques

Elles présentent la particularité d'être génériques : elles utilisent des objets dont le type n'est connu qu'à l'exécution.
Ce type est par convention représenté par un t majuscule : <T>

- List <T>
- Stack <T>
- Queue <T>
- Dictionnary <TKey,TValue>
- SortedList <TKey,TValue>
- Hashtable <TKey,TValue>
- LinkedList <T>
- HashSet<T>
- et d'autres ....

IMPORTANT

- Ces collections font partie de System.Collections.Generic

[ϟ] il faut vérifier la présence en début du programme de la ligne de code : using System.Collections.Generic ;

Les collections génériques versus collections d'objets

- Les collections génériques (avec un <T>) , plus récentes ("plus modernes") que les "anciennes" collections d'objets, sont les collections les mieux adaptées à un environnement à typage fort (tel que C#).
- Elles se différencient des collections d'objets en ce que le type des élements composants la collection est ici précisé par le T
- Il est recommandé que les applications utilisant .NET Framework 2.0 et suivant utilisent ces collections génériques plutôt que les non génériques plus anciennes
- Ces méthodes ont en commun la notion de généricité. Elles utilisent des objets dont le type T n'est connu qu'à l'exécution et il n'est donc plus nécessaire de caster les objets au moment des lectures
- Pour le reste, leur emploi est similaire à celui des collections d'objets correspondantes

Les List <T>

Création (avec ici T = string)

- List<string> list = new List<string> () ;
- List<string> list = new List<string> (10) ;

Utilisation : (comme un tableau)

- Accès en lecture/écriture par l'index : **list [index]**

Propriété :

- list.**Capacity** => la capacité de la liste (taille de la liste)
- list.**Count** => le nombre d'éléments de la liste

Méthodes :

- ajouter un objet à la fin : void list.**Add** (string item)
- insérer à la position index : void list.**Insert** (int index, string item)
- supprimer à la position index spécifiée : void list.**RemoveAt** (int index)
- supprimer 1ère occurrè d'un élément : bool list.**Remove** (string item)
- supprimer tous les objets contenus dans la liste : void list.**Clear** ( )
- trier les objets de la lister : void list.**Sort** ( )

Les List <T>

Lecture par foreach... :
Les collections peuvent être parcourues par une boucle foreach (elles implémentent l'interface IEnumerable... Notion d'interface qui sera précisée dan un cours ultérieur)

foreach permet de parcourir les objets des les collections génériques

Code (Exemple)

```
1 List<string> list = new List<string> () ;
2 foreach (string name in list)
3 {
4     Console.WriteLine(name);
5 }
```

Les List <T>

Code (Exemple)

```
1  List<string> list = new List<string> (10) ;
2  //
3  for ( int i = 0 ; i<=2 ;  i++ )
4    list.Add ( "valeur" + i ) ;
5  //
6  Console.WriteLine ("taille de la liste = " + list.Capacity)
       ;
7  Console.WriteLine ("nombre d'elements = " + list.Count) ;
8  //
9  for ( int i = 0 ; i <= list.Count-1 ;  i++ )
10   Console.WriteLine ("pos["+ i +"]="+ list[i] );
```

Résultat

taille de la liste = 10

nombre d'éléments = 3

list[0] = valeur0

list[1] = valeur1

list[2] = valeur2

Deux cas particuliers

Les Stack (avec ici T = char)

Création

- Stack<char> tas = new Stack<char> ( ) ;
- Stack<char> tas = new Stack<char> (10) ;
- Stack<char> tas = new Stack<char> (<char>collection) ;

Propriété

- tas.**Count** => le nombre d'éléments contenus

Méthodes spécifiques des stack (Lifo) :

- empiler un élément : void tas.**Push** (char item)
- dépiler et renvoyer l'élément du dessus de la pile : char tas.**Pop** ( )
- renvoit l'élément du dessus sans l'enlever : char tas.**Peek** ( )
- true ou false si item est dans tas : bool tas.**Contains** (char item)

Les Queue (avec ici T = int)

Création

- Queue<int> file = new Queue<int> ( ) ;
- Queue<int> file = new Queue<int> (10) ;
- Queue<int> file = new Queue<int> (<int>collection) ;

Propriété

- file.**Count** => le nombre d'éléments contenus

Méthodes spécifiques des Queue (Fifo) :

- placer un élément à la fin : void file.**Enqueue** (int item)
- retirer et renvoyer l'élément au début : int file.**Dequeue** ()
- renvoit l'élément du debut sans l'enlever : int file.**Peek** ( )
- true ou false si item est dans file : bool file.**Contains** (int item)

Les Stack <T> et Queue <T>

Code (Exemple)

```
1  Stack pile<string> = new Stack<string> ( ) ;
2  Queue queue<string> = new Queue<string> ( );
3  //
4  for ( int i = 0 ; i<=5 ;  i++ ) {
5     pile.Push ( "valeur" + i ) ;
6     queue.Enqueue ( "valeur" + i ) ;
7  }
8  //
9  Console.WriteLine("1er element extrait de la pile (Lifo) = "
10             + pile.Pop() ) ;
11 Console.WriteLine("1er element extrait de la file (Fifo) = "
12             +  queue.Dequeue() ) ;
```

Résultat

1er element extrait de la pile (Lifo) = valeur5

1er element extrait de la file (Fifo) = valeur0

Les tableaux associatifs

Les SortedList<TKey,TValue>

SortedList

- On place dans la liste des couples (clé d'accès, valeur) : Les KeyValuePair
- Ces KeyValuePair sont accessibles par des index (int) comme dans un tableau
- Dans un KeyValuePair, on peut accéder à la clé ou à la valeur
- Les valeurs sont aussi accessibles directement par les clés d'accès
- Les clés et valeurs dans la liste sont de n'importe quel type.

Utilisation : on accède aux valeurs placées dans la liste par leur clés associés en utilisant les méthodes des SortedList.

IMPORTANT

- Les SortedList font partie de System.Collections.Generic

[↯] il faut vérifier la présence en début du programme de la ligne de code : using System.Collections.Generic ;

SortedList<TKey,TValue>

(avec en exemple ici TKey = int et TValue = string)

Création

- SortedList<int,string> liste = new SortedList<int,string> ( ) ;
- SortedList<int,string> liste = new SortedList<int,string> (10) ;

Propriétés :

- int liste.**Count** => le nombre d'éléments dans la liste

  int nombre = liste.Count() ;
- IList<TKey> liste.**Keys** => La liste des Keys

  List<int> listeDesCles = liste.Keys.ToList<int>() ;
- IList <TValue> liste.**Values** => La liste des Values

  List<string> listeDesValeurs = liste.Values.ToList<string>() ;

SortedList<TKey,TValue>

(avec en exemple ici TKey = int et TValue = string)

Méthodes :

- ajouter un objet : void liste.**Add** (TKey key, TValue value)
- obtenir la valeur correspondant à la clé spécifiée :
  string liste.**TryGetValue** (TKey key)

- sList contient ? :
  la valeur value ? : bool liste.**ContainsValue** (TValue value)
  la clé key ? : bool liste.**ContainsKey** (TKey key)
- l'index correspondant
  à la clé : int liste.**IndexOfKey** (TKey key)
  à la valeur : int liste.**IndexOfValue** (TValue value)
- retirer la paire (cle,valeur)
  à l'index spécifié : void liste.**RemoveAt** (int index)
  pour la clé spécifiée : bool liste.**Remove** (TKey key)

SortedList<TKey,TValue>

avec les KeyValuePair = <int,string>
(avec en exemple ici TKey = int et TValue = string)

Méthodes :

- obtenir l'élément kv (clé,valeur) situé à la position index :
  KeyValuePair kv = liste.**ElementAt** (index) :

Propiétés (en utilisant les KeyValuePair) :

- accéder aux composants d'un élément KeyValuePair
  (cle,valeur) de kv => kv.**Key** et kv.**Value**
  et donc à la position index
  * clé : int liste.**ElementAt** (index).**Key**
  * valeur : string liste.**ElementAt** (index).**Value**

Et encore d'autres méthodes... encore plus puissantes :

- Union, Except, Intersect, Clear ...
  consulter la doc Microsoft https ://msdn.microsoft.com

Code (Exemple)

```
1  SortedList < int , string> liste
2       = new SortedList<int , string >();
3 liste.Add(110,"Luc");
4 liste.Add(130,"Jean");
5 liste.Add(100,"Marcel");
6 Console.WriteLine("Element a l'index 1 => "
7             + liste.ElementAt(1));
8 Console.WriteLine("Valeur a l'index 1 => "
9             + liste.ElementAt(1).Value);
10 //=> Suppression de l'element de cle = 100
11 liste.Remove(100);
12 Console.WriteLine("Element a l'index 1 => "
13            + liste.ElementAt(1));
```

Résultat

Element à l'index 1 => [110, Luc]
Valeur à l'index 1 => Luc
//=> Suppression de l'élément de clé = 100
Element à l'index 1 => [130, Jean]

Code (Exemple)

```
1  SortedList < int , string> liste
2        = new SortedList<int , string >()
3  liste.Add(110,"Luc");
4  liste.Add(130,"Jean");
5  liste.Add(100,"Marcel");
6  Console.WriteLine("Index de la cle 130 => "
7      + liste.IndexOfKey(130));
8  Console.WriteLine("Element place a la cle 130 => "
9      + liste.ElementAt(liste.IndexOfKey(130)));
10 Console.WriteLine("valeur associee a la cle 130 => "
11     + liste.ElementAt(liste.IndexOfKey(130)).Value );
12 //=> ou bien ...
13 liste.TryGetValue(130, out sring nom);
14 Console.WriteLine("valeur associee a la cle 130 =>" + nom);
```

Résultat

Index de la clé 130 => 1
Element placé à la clé 130 => [130, Jean]
Valeur associée à la clé 130 => Jean
// => ou bien ...
Valeur associée à la clé 130 =>Jean

d'autres collections

Dictionary <TKey,TValue> **vs** SortedList<TKey,TValue>

Dictionary <TKey,TValue> et **SortedDictionary** <TKey,TValue> :

- même signature
- même domaine d'utilisation
- même mode de création
- des propriétés similaires
- mêmes méthodes (presque...)
- mais pas d' IndexOfKey ni d IndexOfValue pour les Dictionary

Mais des différences :

	SortedDictionnary	SortedList
Organisation	arbre binaire	liste
Mémoire occupée	+	-
Unsorted data	+rapide	-rapide
Sorted data	-rapide	+rapide

Dictionary <TKey,TValue> **vs** SortedList<TKey,TValue>

Operation	Dictionary<K,V>	SortedDictionary<K,V>	SortedList<K,V>
this[key]	*O(1)*	*O(log n)*	*O(log n) or O(n)*
Add(key,value)	*O(1) or O(n)*	*O(log n)*	*O(n)*
Remove(key)	*O(1)*	*O(log n)*	*O(n)*
ContainsKey(key)	*O(1)*	*O(log n)*	*O(log n)*
ContainsValue(value)	*O(n)*	*O(n)*	*O(n)*

et encore d'autres structures de données

- Les listes chainées
- Les tables de hachage
- Les arbres
- Les set
- et d'autres ....

Objectif : Disposer d'une structure de données adaptée au besoin

Se documenter :
https ://msdn.microsoft.com/fr-fr/library/0x6a29h6(v=vs.90).aspx
L'autocomplétion et l'aide contextuelle de VisualStudio