

# Structures de données et programmation orientée objet

---

LIST<T>

# Introduction

---

Une structure de données est, comme son nom l'indique, une façon d'organiser logiquement des données pour en faciliter l'utilisation. Elle permet donc à la fois d'**entreposer** des données pendant l'exécution d'un programme puis de les accéder.

Chaque structure de données a des caractéristiques propres lui conférant des avantages et des inconvénients. On choisira donc une structure de données en fonction de la nature du traitement à effectuer.

# Introduction

---

Il existe de nombreuses formes structure de données qui peuvent être catégorisées en fonction de plusieurs critères tels que :

- la quantité d'information pouvant être contenue (fixe ou variable)
- les méthodes d'entreposage et d'accès à l'information
- le type d'organisation de l'information au sein de la structure
- le type d'information pouvant être contenu (homogène ou hétérogène)

# Introduction

---

Nous avons déjà abordé une structure de données primitive, **le tableau**, qui organise physiquement son contenu sous la forme d'une suite d'espaces de mémoire successifs contenant des éléments de même type, et qui organise logiquement son contenu comme un ensemble de valeurs qu'on peut atteindre par un indice.

Un tableau répond aux critères énoncés ci-dessus :

- il est un conteneur dont la taille est fixée à la création
- il permet d'entreposer l'ensemble de ses données en mémoire
- il permet d'accéder à un élément à l'aide d'un indice entier
- il organise ses éléments sous la forme d'une suite de « cases » successives de mémoire
- il contient des données dont le type est homogène

# La liste

---

La première nouvelle structure que nous allons voir est la liste.

Nous allons approcher la liste comme un tableau **très polyvalent**. Il est un choix de structure populaire pour entreposer des données, car il présente les même avantage d'un tableau dont la taille **peut croître selon les besoins durant l'exécution**.

À l'opposé du tableau, on peut ajouter des éléments sans trop se soucier de l'espace initialement alloué puisque cette structure ajuste sa taille automatiquement en fonction du nombre d'éléments qu'on y ajoute.

# La liste

---

Opérations de base sur la liste :

- Initialiser la liste
- Ajouter un élément dans la liste (*.Add*, *.Insert*)
- Accéder à un élément précis de la liste par son indice (*[...]*)
- Retirer un élément de la liste (*.Remove*, *.RemoveAt*, *.RemoveAll*)
- Obtenir le nombre d'éléments de la liste (*.Count*)
- Chercher un élément comportant certaines caractéristiques dans la liste (*.Find*)
- ...

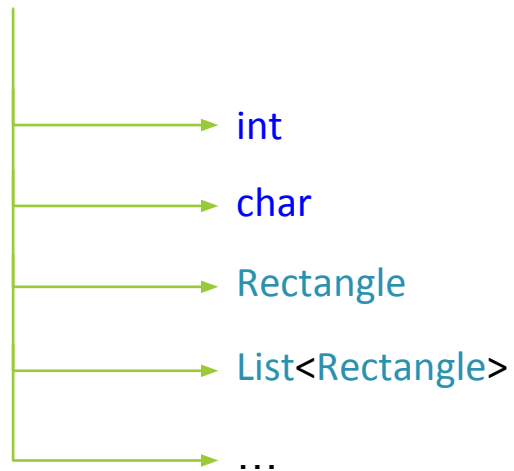
# La classe List

---

Pour représenter une telle liste, nous avons à notre disposition la classe **List**.

Pour déclarer une référence à une liste :

- `List<typeContenuDansLaListe> liste1;`



`List<int> liste1;`

`List<char> liste1;`

`List<Rectangle> liste1;`

`List<List<Rectangle>> liste1;`

# La classe List

---

Pour créer une instance de **List** :

- ... = `new List<typeContenuDansLaListe>();`



```
List<int> liste1 = new List<int>();  
List<Rectangle> liste1 = new List<Rectangle>();  
...
```

Ceci instancie une liste vide d'un type donné, ayant une capacité de 0.

La capacité de la liste (nombre de donnée qu'elle peut contenir), augmente automatiquement lorsque des données y sont ajouté, contrairement à un tableau qui a une capacité fixe tout au long de sa vie.



# La classe List

---

Si vous savez que la liste doit contenir au moins un certain nombre d'élément, donnez une capacité initiale au constructeur, afin d'éviter de redimensionner (implique le copiage des éléments) constamment la liste.

```
List<int> liste1 = new List<int>(50); //capacité initiale de 50. Nous pouvons mettre 50 éléments avant que la liste doit se  
//redimensionner
```

Pareillement pour une StringBuilder

```
StringBuilder builder = new StringBuilder(50);
```

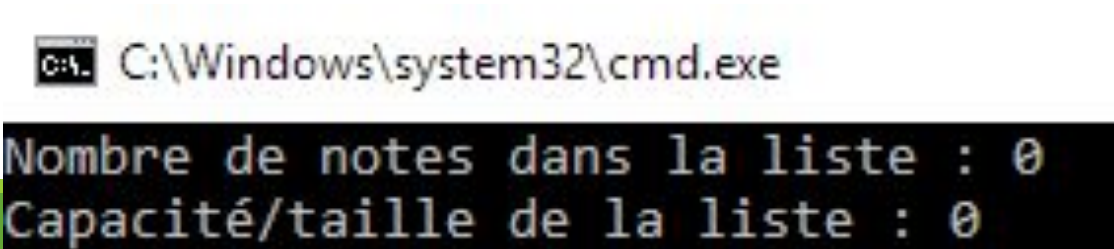
# La classe List

---

Nous pouvons voir le nombre d'item dans une liste, ainsi que la capacité d'une liste, grâce aux propriétés d'instance suivants :

---

1. `//Déclaration et instanciation d'une liste d'entiers`
2. `List<int> notesExamen = new List<int>();`
3. `int nombreNotes = notesExamen.Count;`
4. `int capacitéListe = notesExamen.Capacity;`
5. `Console.WriteLine("Nombre de notes dans la liste : " + nombreNotes);`
6. `Console.WriteLine("Capacité/taille de la liste : " + capacitéListe);`



C:\Windows\system32\cmd.exe

```
Nombre de notes dans la liste : 0  
Capacité/taille de la liste : 0
```

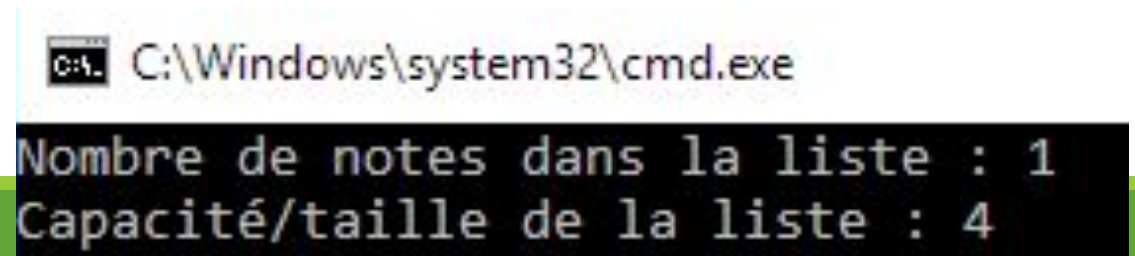
# Ajouter des éléments

---

Ajouter une donnée à la fin d'une liste est simple, en utilisant la méthode **.Add(...)** :

---

1. `//Déclaration et instanciation d'une liste d'entiers`
2. `List<int> notesExamen = new List<int>();`
3. `//Ajout d'un entier dans la liste`
4. `notesExamen.Add(100);`
5. `int nombreNotes = notesExamen.Count;`
6. `int capacitéListe = notesExamen.Capacity;`
7. `Console.WriteLine("Nombre de notes dans la liste : " + nombreNotes);`
8. `Console.WriteLine("Capacité/taille de la liste : " + capacitéListe);`



```
C:\Windows\system32\cmd.exe
Nombre de notes dans la liste : 1
Capacité/taille de la liste : 4
```

# Ajouter des éléments

---

1. //Déclaration et instanciation d'une liste d'entiers
2. `List<int> notesExamen = new List<int>();`
  
3. //Ajout d'entiers dans la liste
4. `notesExamen.Add(55);`
5. `notesExamen.Add(60);`
6. `notesExamen.Add(47);`
7. `notesExamen.Add(88);`
  
8. `int nombreNotes = notesExamen.Count;`
9. `int capacitéListe = notesExamen.Capacity;`
  
10. `Console.WriteLine("Nombre de notes dans la liste : " + nombreNotes);`
11. `Console.WriteLine("Capacité/taille de la liste : " + capacitéListe);`

C:\Windows\system32\cmd.exe

```
Nombre de notes dans la liste : 4
Capacité/taille de la liste : 4
```

# Ajouter des éléments

---

1. `//Déclaration et instanciation d'une liste d'entiers`
2. `List<int> notesExamen = new List<int>();`
3. `//Ajout d'entiers dans la liste`
4. `notesExamen.Add(55);`
5. `notesExamen.Add(60);`
6. `notesExamen.Add(47);`
7. `notesExamen.Add(88);`
8. `notesExamen.Add(79);`
9. `Console.WriteLine("Nombre de notes dans la liste : " + notesExamen.Count);`
10. `Console.WriteLine("Capacité/taille de la liste : " + notesExamen.Capacity);`

C:\Windows\system32\cmd.exe

```
Nombre de notes dans la liste : 5
Capacité/taille de la liste : 8
```

# Accéder un élément

---

Pour accéder/obtenir une donnée d'une liste, nous pouvons utiliser la syntaxe familière des crochets avec indice, ou bien la méthode **.ElementAt(indice)** :

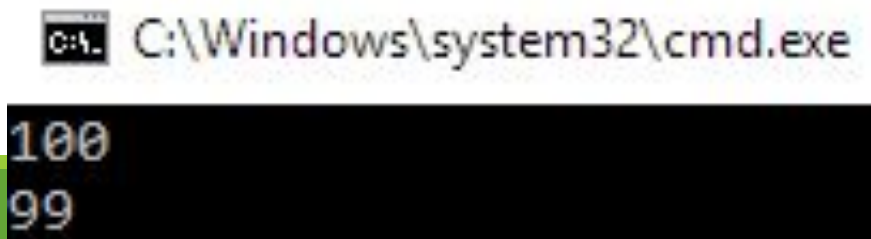
---

```
1. //Déclaration et instanciation d'une liste d'entiers
2. List<int> notesExamen = new List<int>();

3. //Ajout d'entiers dans la liste
4. notesExamen.Add(100);
5. notesExamen.Add(99);

6. int premièreNote = notesExamen[0];
7. int deuxièmeNote = notesExamen.ElementAt(1);

8. Console.WriteLine(premièreNote);
9. Console.WriteLine(deuxièmeNote);
```



```
C:\Windows\system32\cmd.exe
```

```
100
99
```

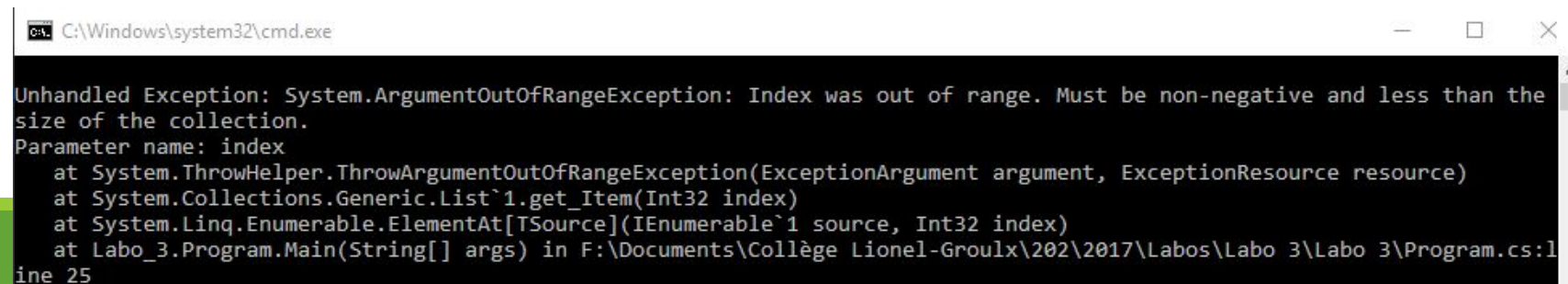
# Accéder un élément

---

Dans les deux cas, il faut toujours faire attention à ne pas faire de tentative d'accès avec un indice hors borne ou à un élément qui n'existe pas :

---

1. //Déclaration et instanciation d'une liste d'entiers
2. `List<int> notesExamen = new List<int>();`
3. //Ajout d'un entier dans la liste
4. `notesExamen.Add(100);`
5. `int premièreNote = notesExamen[0];`
6. //Il n'y a pas une donnée initialisée à la position 1 de la liste, malgré la capacité de 4.
7. `int deuxièmeNote = notesExamen.ElementAt(1);`



```
C:\Windows\system32\cmd.exe

Unhandled Exception: System.ArgumentOutOfRangeException: Index was out of range. Must be non-negative and less than the size of the collection.
Parameter name: index
   at System.ThrowHelper.ThrowArgumentOutOfRangeException(ExceptionArgument argument, ExceptionResource resource)
   at System.Collections.Generic.List`1.get_Item(Int32 index)
   at System.Linq.Enumerable.ElementAt[TSource](IEnumerable`1 source, Int32 index)
   at Labo_3.Program.Main(String[] args) in F:\Documents\Collège Lionel-Groulx\202\2017\Labos\Labo 3\Labo 3\Program.cs:1
line 25
```

# Accéder tous les éléments

---

Classique :

```
1. //Déclaration et instanciation d'une liste de rectangles
2. List<Rectangle> hitboxes = new List<Rectangle>();

3. //Ajout de rectangles dans la liste
4. hitboxes.Add(new Rectangle(10, 6, '*'));
5. hitboxes.Add(new Rectangle(2, 4, '@'));
6. hitboxes.Add(new Rectangle(4, 5, '%'));

7. for (int i = 0; i < hitboxes.Count; i++)
8. {
9.     Rectangle hitbox = hitboxes.ElementAt(i);
10.    Console.WriteLine(hitbox.Largeur);
11. }
```



# Accéder tous les éléments (foreach)

---

```
1. //Déclaration et instanciation d'une liste de rectangles
2. List<Rectangle> hitboxes = new List<Rectangle>();

3. //Ajout de rectangles dans la liste
4. hitboxes.Add(new Rectangle(10, 6, '*'));
5. hitboxes.Add(new Rectangle(2, 4, '@'));
6. hitboxes.Add(new Rectangle(4, 5, '%'));
```

```
7. foreach (Rectangle r in hitboxes)
8. {
9.     Console.WriteLine(r.Largeur);
10. }
```

 **foreach** (**T** **n** **in** **x**) { ... }

Pour chaque donnée  $n$  d'un type primitif ou complexe  $T$  dans la liste  $x$ , exécuter les traitements entre les  $\{ \}$ .

*Ex : Pour chaque **Rectangle**  $r$  dans  $hitboxes$ , afficher la largeur de  $r$*

Nous faisons référence à l'élément actuel avec un alias donné.  
Ici, c'est «  $r$  ».

# Retirer un élément

Pour enlever un élément à une position précise d'une liste, nous pouvons utiliser **.RemoveAt(indice)** :

```
1. //Déclaration et instanciation d'une liste d'entiers
2. List<int> notesExamen = new List<int>();

3. //Ajout d'entiers dans la liste
4. notesExamen.Add(55);
5. notesExamen.Add(60);
6. notesExamen.Add(47);
7. notesExamen.Add(88);
8. notesExamen.Add(79);
9. //Retirement d'entiers dans la liste : retirer trois fois l'élément à la tête de la liste
10. notesExamen.RemoveAt(0);
11. notesExamen.RemoveAt(0);
12. notesExamen.RemoveAt(0);

13. foreach (int note in notesExamen)
14. {
15.     Console.WriteLine(note);
16. }
17. Console.WriteLine("Nombre de notes dans la liste : " + notesExamen.Count);
18. Console.WriteLine("Capacité/taille de la liste : " + notesExamen.Capacity);
```

C:\Windows\system32\cmd.exe

```
88
79
Nombre de notes dans la liste : 2
Capacité/taille de la liste : 8
```

# Retirer un élément

---

Il est possible de retirer la première occurrence d'un élément dans une liste, à l'aide de **.Remove(élémentÀSupprimer)**. Cette méthode d'une instance de liste fait une recherche pour trouver le premier élément dans la liste équivalent à l'élément envoyé en paramètre. Si trouvé, l'élément est retiré de la liste et la fonction retourne **true**. Sinon, la liste n'est pas modifiée et la fonction retourne **false**.

**Note :** à cause que cette méthode doit faire une recherche pour trouver l'élément à retirer dans la liste, elle est beaucoup moins efficace (en terme de temps d'exécution) que *.RemoveAt(indice)*.

# Retirer un élément

---

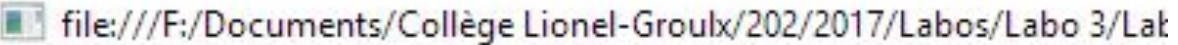
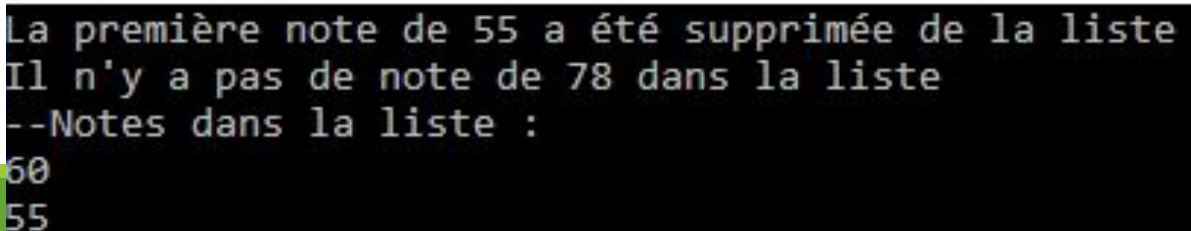
```
1. //Déclaration et instanciation d'une liste d'entiers
2. List<int> notesExamen = new List<int>();

3. //Ajout d'entiers dans la liste
4. notesExamen.Add(55);
5. notesExamen.Add(60);
6. notesExamen.Add(55);

7. if (notesExamen.Remove(55))
8.     Console.WriteLine("La première note de 55 a été supprimée de la liste");
9. else
10.    Console.WriteLine("Il n'y a pas de note de 55 dans la liste");

11. if (notesExamen.Remove(78))
12.    Console.WriteLine("La première note de 78 a été supprimée de la liste");
13. else
14.    Console.WriteLine("Il n'y a pas de note de 78 dans la liste");

15. Console.WriteLine("--Notes dans la liste : ");
16. foreach (int note in notesExamen)
17.     Console.WriteLine(note);
```

# Autres méthodes

---

- **Clear** : supprime tous les éléments
- **Contains** : détermine si un élément est dans la liste
- **Find** : retourne la première occurrence d'un élément contenu dans la liste qui correspond à un certain critère
- **Insert** : insère un élément dans la liste à l'indice indiqué
- **IndexOf** : retourne l'indice de la première occurrence d'un élément dont on fournit la valeur
- **Sort** : *voir prochaine page*
- Etc...

# Trier une liste

---

Afin de trier une liste, il est possible d'utiliser la méthode **.Sort(...)** :

---

## Ordre croissant

```
1. notesExamen.Sort(delegate (int x, int y)
2. {
3.     return x.CompareTo(y);
4. });
```

## Ordre décroissant

```
1. notesExamen.Sort(delegate (int x, int y)
2. {
3.     return y.CompareTo(x);
4. });
```

## Ordre croissant par largeur de rectangle

```
1. hitboxes.Sort(delegate(Rectangle x, Rectangle y)
2. {
3.     return x.Largeur.CompareTo(y.Largeur);
4. });
```

## Ordre décroissant par largeur de rectangle

```
1. hitboxes.Sort(delegate(Rectangle x, Rectangle y)
2. {
3.     return y.Largeur.CompareTo(x.Largeur);
4. });
```