

# Tutoriel de C#

---

## Table des matières

Collections de données .....	1
Classes .....	3
Records .....	5
Tuples .....	6
Intervalles .....	8

# Collections de données

Explications :

Les Collections sont l'ensemble des méthodes qui peuvent regrouper une certaine quantité d'objets et de les manipuler. Elles se différencient des tableaux par leur souplesse des groupes d'objets manipulés qui peuvent être augmentés ou réduits de manière dynamique. Il existe deux familles de collections les génériques qui ne possèdent qu'un seul type d'objets et les non-génériques qui peuvent en comporter plusieurs.

## Listes de collections non-génériques :

ArrayList

HashTable

SortedList

Stack

Queue

## Listes de collections génériques :

List<T>

Dictionary<Tkey,Tvalue>

SortedList<T>

Stack<T>

Queue<T>

Chaque élément possède sa particularité :

- Les ArrayList et List<T> n'ont pas de taille spécifique et peuvent contenir tous types d'objets.
- Les HashTable et Dictionary<Tkey,Tvalue> utilise une variable « clé » en plus de la valeur de l'élément.
- Les SortedList et SortedList<T> utilisent également une variable « clé » mais ils se différencient par une organisation de tous les éléments dans un ordre trié.
- Les Stack et Stack<T> sont des collections d'objets dites LIFO (Last In First Out). Ils possèdent des fonctions spécifiques et sortent les éléments les plus anciens. Par exemple si vous ajoutez un élément dans la liste, cela s'appelle « Push » et lorsque vous le supprimez, cela s'appelle « Pop ».
- Les Queue et Queue<T> sont des collections d'objets dites FIFO (First In First Out). Ils possèdent des fonctions spécifiques et sortent les éléments les plus récents. Par exemple si vous ajoutez un élément à la liste, cela s'appelle « Enqueue » et lorsque vous supprimez un élément, cela s'appelle « Dequeue »

Exemples : Chaque collection non-générique possède son homologue générique et inversement :

#### ArrayList non-génériques

```
ArrayList test = new ArrayList();  
  
test.Add(2);      //int  
test.Add(1.3);    //float  
test.Add("Csharp"); //string  
test.Add('a');    //char  
  
foreach (object n in test) {  
    Console.WriteLine("{0} ", n);  
}
```

Output :

2  
1.3  
Csharp  
a

#### List<T> générique

```
List<int> test = new List<int>();  
  
test.Add(10);  
test.Add(20);  
test.Add(30);  
test.Add(40);  
  
foreach (int n in test) {  
    Console.WriteLine(n);  
}
```

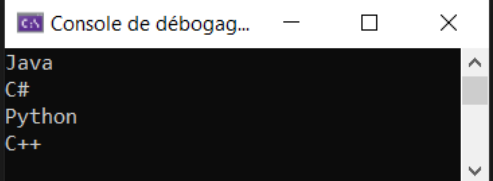
Output :

10  
20  
30  
40

Lorsque l'on lit le contenu d'une Queue le premier objet sorti est le premier objet rentré tandis que pour un Stack le premier objet sorti est le dernier objet rentré :

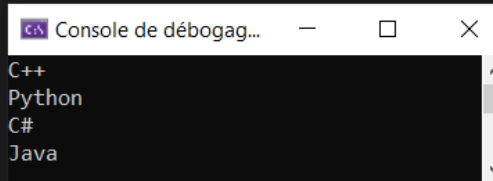
#### Utilisation d'une Queue

```
1 using System.Collections;  
2  
3 Queue languages = new Queue();  
4 languages.Enqueue("Java");  
5 languages.Enqueue("C#");  
6 languages.Enqueue("Python");  
7 languages.Enqueue("C++");  
8 foreach (object o in languages)  
9 {  
10     Console.WriteLine(o);  
11 }
```



#### Utilisation d'un Stack

```
1 using System.Collections;  
2  
3 Stack languages = new Stack();  
4 languages.Push("Java");  
5 languages.Push("C#");  
6 languages.Push("Python");  
7 languages.Push("C++");  
8 foreach (object o in languages)  
9 {  
10     Console.WriteLine(o);  
11 }
```



# Classes

Explication :

En programmation orientée objet, les classes sont l'une des premières définitions que l'on voit. Elles permettent de définir le type d'objet que l'on crée avec ses paramètres. On peut y paramétrer toutes sortes de variables et/ou de méthodes. Les variables servent à stocker les données et attributs de l'objet instancié. Quant aux méthodes, elles permettent d'exécuter un programme qui manipule les variables en les modifiant, en les affichant ou en les utilisant à une autre fin. Une classe doit toujours posséder un constructeur, il nous sert à initialiser les variables des méthodes.

Prenons l'exemple suivant qui est la création d'une classe Monster avec 3 variables et 1 méthodes.

Tout d'abord il y a le string du nom du monstre.

Puis son nombre de point de vie avec un integer.

Ensuite le nombre de dégâts que le monstre pourrait infliger.

Le constructeur va nous servir à donner des valeurs aux 3 paramètres ci-dessus, dans l'exemple on dit que son nombre de dégâts est de 5 et ses PV de 50. Le nom sera paramétré, lui, avec le nom contenu dans le string monstername.

La méthode, elle, sert simplement à afficher la mort du monstre lorsque les PV de celui sont inférieur ou égale à 0, s'il reçoit des dégâts d'une potentielle autre class « Héros » par exemple.

```
65 public class Monster
66 {
67     string nom;
68     int hp; //on initialise le nom, les PV et la vitesse du monstre
69     int damage;
70
71     0 références
72     private void monster(string monstername)
73     {
74         this.hp = 50;
75         this.nom = monstername; //le constructeur sert à mettre des valeurs
76         this.damage = 5; //aux variables
77     }
78
79     0 références
80     private void death()
81     {
82         if (this.hp <= 0) //la méthode death affiche le contenu de la ligne ci-dessous
83         {
84             Console.WriteLine("Ce monstre est mort");//lorsque les PV du monstre tombe à 0.
85         }
86     }
87 }
```

Exercice :

Créez la classe Duel qui possède 4 paramètres (le nom des duellistes et 2 booléens pour savoir lequel a gagné) ainsi qu'une méthode permettant d'afficher le gagnant.

Correction : La première chose est de déclarer nos 4 paramètres sans y mettre de valeurs.

```
88 public class Duel
89 {
90     string duellisteUn;
91     string duellisteDeux;
92     bool duellisteUnGagne = false;
93     bool duellisteDeuxGagne = false;
94 }
```

Puis on programme le constructeur :

```
95 private void duel(string nomDuellisteUn, string nomDuellisteDeux)
96 {
97     this.duellisteUn = nomDuellisteUn;
98     this.duellisteDeux = nomDuellisteDeux;
99     bool duellisteUnGagne = false;
100     bool duellisteDeuxGagne = false;
101 }
102
```

Le nom des duellistes sera paramétré avec le nom contenu dans le string nomDuellisteUn et nomDuellisteDeux comme dans l'exemple évoqué plus haut.

Enfin, nous écrivons la méthode afficherLeGagnant qui s'affiche si l'un des booléen prend la valeur true :

```
103 private void afficherLeGagnant()
104 {
105     if (duellisteUnGagne)
106     {
107         Console.WriteLine(duellisteUn + " a gagné contre " + duellisteDeux);
108     }
109     if (duellisteDeuxGagne)
110     {
111         Console.WriteLine(duellisteDeux + " a gagné contre " + duellisteUn);
112     }
113 }
114 }
```

# Records

Explications :

Au même titre que les « Class » ou « Enum », les « Records » sont également un type référence qui permet d'encapsuler les données. La particularité de ces données est qu'ils sont dits immuables. En programmation orientée objet, un objet immuable est fixe et ne peut être modifié après son initialisation.

Le type référence Record semblent assez similaire au type référence Class dans leurs utilisations. Pourtant, il possède quelques différences notables propres à l'utilisation qu'on veut en faire.

En effet, les Record possède ce qu'on appelle l'égalité des valeurs. Lorsque l'on initialise deux objets d'une Class avec les mêmes paramètres ils seront considérés comme différents alors que dans un type Records il s'agira du même objet. Cela peut être utile dans les cas où l'on a beaucoup d'objet initialisés sur une longue période car cela évite les doublons.

Comme dit plus haut, les objets et les paramètres de type Record ne peuvent pas être modifié après leur création. Cependant, nous pouvons créer une nouvelle instance de Record déjà existant pour pouvoir modifier ses paramètres. Cela s'appelle la mutation non destructive.

Nous savons qu'un type Class peut hériter d'un autre type Class, ainsi les Record peuvent faire de même comme dans l'exemple ci-dessous :

```
public record Personnage(string Nom, int Taille);  
public record Mage(string Nom, int Taille, int Mana): Personnage(Nom, Taille);  
var mage = new Mage("Dumbledore", 183, 12000);
```

Le Record Mage a hérité du Record Personnage avec en plus son niveau de mana qui lui est propre. On a ensuite instancié un nouveau mage que l'on a nommé Dumbledore lui donnant une taille de 183 cm avec un mana d'une valeur de 12000 (valeurs totalement hypothétiques pour l'exemple).

Toutefois, les croisements sont impossibles, un Record ne peut pas hériter d'une Class et inversement.

Il existe un système pour convertir l'instance d'un record en un tuple contenant toutes les propriétés du record et cela s'appelle les déconstructeurs.

En bref, les Records sont utiles dans les cas où on a des valeurs complexes a encapsulé, quand l'héritage semble nécessaire pour l'utilisation qu'on en fait ou lorsque le type de données peut être un type valeur qui contient les données dans sa propre allocation de mémoire et qu'il doit être immuable.

# Tuples

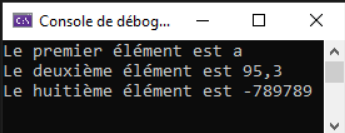
Explications :

La classe Tuple est une structure de données et permet donc de regrouper un certain nombre d'objets pouvant être de types différents sans avoir besoin de le préciser. Pour être précis, un objet de la classe Tuple ne peut pas contenir plus de 8 éléments. En effet, il ne faut pas confondre la classe Tuple et la structure ValueTuple, ces derniers sont une représentation de type valeur du Tuple et peuvent contenir plus que 8 objets.

Exemple :

Ci-dessous, nous définissons un objet test de la classe Tuple ayant 8 objets dont des entiers, des chaînes et des doubles.

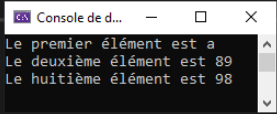
```
117 var test = Tuple.Create("a", 95.3, 3, 1, 1.5, "roger", 98, -789789);
118 Console.WriteLine($"Le premier élément est {test.Item1}");
119 Console.WriteLine($"Le deuxième élément est {test.Item2}");
120 Console.WriteLine($"Le huitième élément est {test.Rest.Item1}");
121
122
123
124
```



Nous remarquons que le huitième élément ne s'appelle pas avec test.Item8 mais avec test.Rest.Item1 ou test.Rest car « Rest » est le nom donné au huitième élément d'un objet de classe Tuple s'il existe.

Pour contrer la contrainte du nombre d'élément, il existe une parade que l'on appelle les Tuple imbriqués. Cela consiste à créer un autre Tuple dans notre objet test contenant lui aussi entre 1 et 8 éléments. À noter que cela marche également avec des structures ValueTuple comme le montre l'exemple ci-dessous.

```
116 var b = ("zef", 89, 3.6);
117 var test = Tuple.Create("a", b, 3, 1, 1.5, "roger", 98);
118 Console.WriteLine($"Le premier élément est {test.Item1}");
119 Console.WriteLine($"Le deuxième élément est {test.Item2.Item1}");
120 Console.WriteLine($"Le huitième élément est {test.Item7}");
121
122
123
124
125
```



Ainsi l'usage du Tuple est utile :

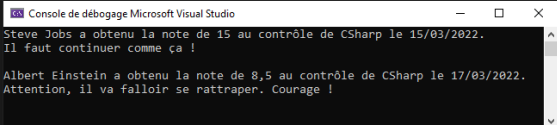
- Si l'on souhaite conserver un enregistrement de base de données ou certaines valeurs sans créer une classe distincte.
- Si l'on veut transmettre plusieurs valeurs à une méthode grâce à un seul paramètre.
- Si l'on veut renvoyer plusieurs valeurs à partir d'une méthode sans utiliser les mots clés ref ou out.

Exercice :

Créer une méthode permettant d'afficher un message sur la note qu'un élève a reçu. Il faudra utiliser un Tuple comportant 4 éléments : la note de l'élève, son nom, son prénom et la date de l'examen. Comme un message automatique après avoir renseigné les données sur l'élève.

Proposition de correction :

```
136 var eleve1 = Tuple.Create(15.0, "Steve", "Jobs", "15/03/2022");
137 var eleve2 = Tuple.Create(8.5, "Albert", "Einstein", "17/03/2022");
138 AfficherNoteEleve(eleve1);
139 AfficherNoteEleve(eleve2);
140
141
142 static void AfficherNoteEleve(Tuple<double, string, string, string > person)
143 {
144     Console.WriteLine($"{person.Item2} {person.Item3} a obtenu la note de {person.Item1} au contrôle de CSharp le {person.Item4}.");
145     if (person.Item1 > 10)
146     {
147         Console.WriteLine("Il faut continuer comme ça !\n");
148     }
149     if (person.Item1 <= 10)
150     {
151         Console.WriteLine("Attention, il va falloir se rattraper. Courage !\n");
152     }
153 }
```





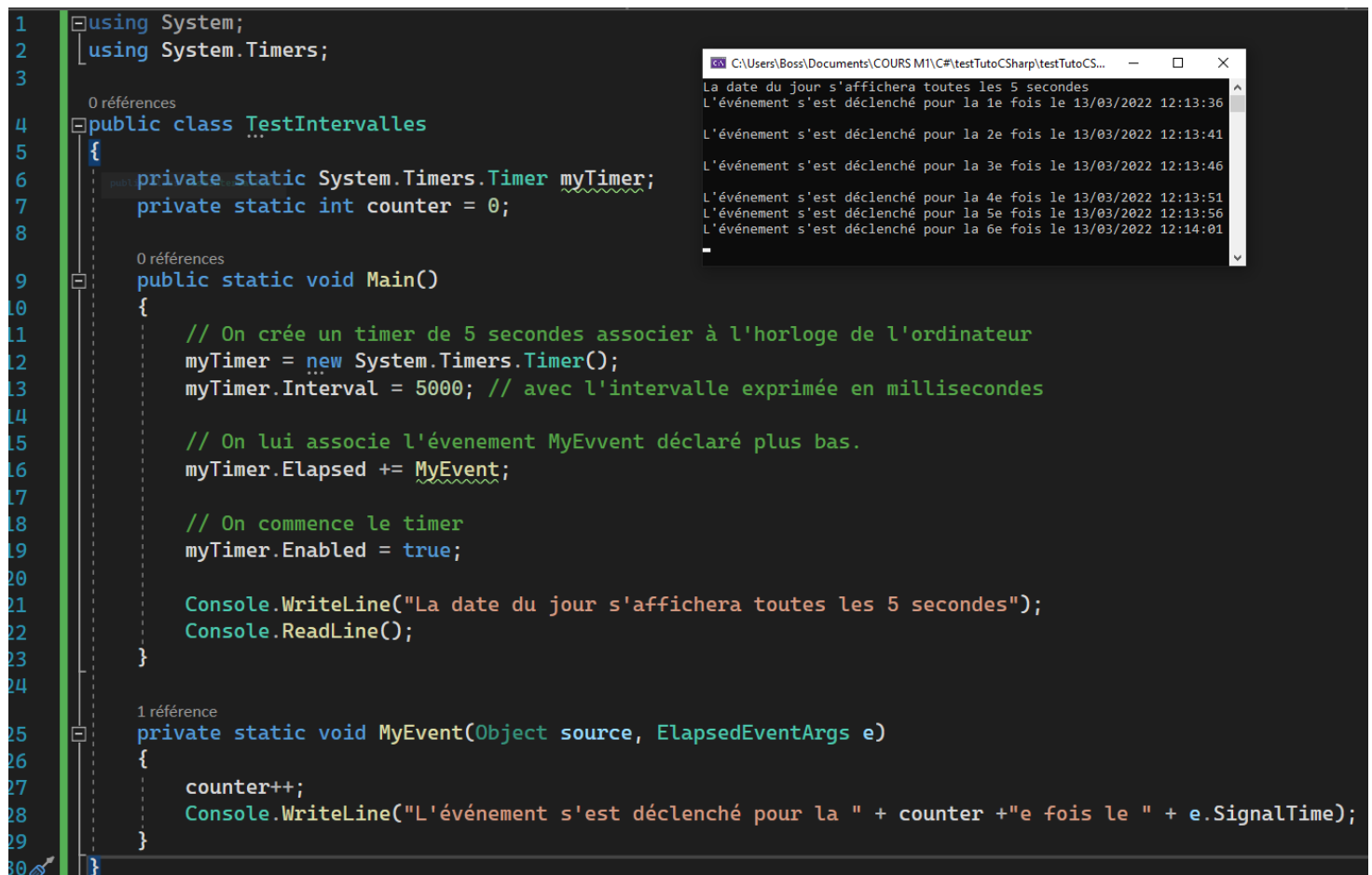
# Intervalles

Explications :

Les intervalles forment une propriété de la classe Timer. Elles sont définies en millisecondes auxquelles un événement ou une action, que l'on appelle Elapsed, doit être déclenché à la fin du compteur. Sa valeur doit supérieure à 0 et inférieure à MaxValue (soit 2 147 483 647) qui est la valeur maximale que peut prendre un Int32. Ainsi cela nous permet de créer soit des événements répétés à intervalle régulier ou encore créer un minuteur ou un chronomètre.

Prenons un exemple simple pour illustrer :

On initialise un timer de 5 secondes et on lui associe l'événement suivant : afficher le nombre de fois qu'il a été déclenché et l'heure exacte de son déclenchement.



```
1 using System;
2 using System.Timers;
3
4 public class TestIntervalles
5 {
6     private static System.Timers.Timer myTimer;
7     private static int counter = 0;
8
9     public static void Main()
10    {
11        // On crée un timer de 5 secondes associer à l'horloge de l'ordinateur
12        myTimer = new System.Timers.Timer();
13        myTimer.Interval = 5000; // avec l'intervalle exprimée en millisecondes
14
15        // On lui associe l'événement MyEvent déclaré plus bas.
16        myTimer.Elapsed += MyEvent;
17
18        // On commence le timer
19        myTimer.Enabled = true;
20
21        Console.WriteLine("La date du jour s'affichera toutes les 5 secondes");
22        Console.ReadLine();
23    }
24
25    private static void MyEvent(Object source, ElapsedEventArgs e)
26    {
27        counter++;
28        Console.WriteLine("L'événement s'est déclenché pour la " + counter + "e fois le " + e.SignalTime);
29    }
30 }
```

0 références

0 références

1 référence

La date du jour s'affichera toutes les 5 secondes  
L'événement s'est déclenché pour la 1e fois le 13/03/2022 12:13:36  
L'événement s'est déclenché pour la 2e fois le 13/03/2022 12:13:41  
L'événement s'est déclenché pour la 3e fois le 13/03/2022 12:13:46  
L'événement s'est déclenché pour la 4e fois le 13/03/2022 12:13:51  
L'événement s'est déclenché pour la 5e fois le 13/03/2022 12:13:56  
L'événement s'est déclenché pour la 6e fois le 13/03/2022 12:14:01

On peut constater que le gestionnaire de tâches affiche bien la date toutes les cinq secondes à partir du moment où j'ai lancé mon programme.

On peut comparer cela à la fonction void Update() que l'on utilise à Unity ou les Coroutines et la fonction WaitForSeconds(), tout en pouvant modifier l'intervalle de temps entre les événements.

Une fois que l'on sait cela l'utilisation de cette propriété dépend de notre imagination et du contexte dans le lequel on se trouve.

Prenons l'exemple d'un personnage s'approchant trop près d'un feu. Ses points de vie ne baisseront pas immédiatement mais décroîtront d'un nombre de PV choisi au préalable toutes les x secondes. Fixons une baisse de 10 PV toutes les 1,5 secondes.

Tout d'abord, on crée les caractéristiques du personnage, les dégâts du feu et le timer comme tel :

```
1  using System;
2  using System.Timers;
3
4  0 références
5  public class PersonnageEnFeu
6  {
7      private static System.Timers.Timer myTimer;
8      private static int hp = 100;
9      private static int damageOfFire = 10;
10     private static bool alive = true;
```

Avec un booléen « alive » pour voir s'il est en vie.

Puis nous définissons le Main avec les caractéristiques du timer :

```
11 public static void Main()
12 {
13     // On crée un timer de 5 secondes associer à l'horloge de l'ordinateur
14     myTimer = new System.Timers.Timer();
15     myTimer.Interval = 1500; // avec l'intervalle exprimée en millisecondes
16     // On lui associe l'événement MyEvvent déclaré plus bas.
17     myTimer.Elapsed += TakingDamageInFire;
18     // On commence le timer
19     myTimer.Enabled = true;
20
21     Console.WriteLine("Les PV initiales du personnage sont fixés à " + hp);
22     Console.ReadLine();
23 }
24
```

La fonction TakingDamageInFire à la ligne 17 est le nom de notre troisième étape qui est l'événement lancé à chaque intervalle. Pour ce faire on peut dire que le texte affiché sera les PV restants du personnage si celui-ci est encore en vie sinon on affiche qui est mort. La correction est ci-dessous.

```

25 private static void TakingDamageInFire(Object source, ElapsedEventArgs e)
26 {
27     if (alive)
28     {
29         hp = hp - damageOfFire;
30         Console.WriteLine("Le personnage est en feu, voici ses PV restants : " + hp);
31         if (hp <= 0)
32         {
33             alive = false;
34         }
35     }
36     else
37     {
38         Console.WriteLine("Le personnage est mort");
39     }
40 }
41

```

Et voilà ce qu'on obtient :

```

C:\Users\Boss\Documents\COURS M1\C#\testTutoCSharp\testTutoCSharp\bin\Debug\net6.0\testTutoCSharp.exe
Les PV initiales du personnage sont fixés à 100
Le personnage est en feu, voici ses PV restants : 90
Le personnage est en feu, voici ses PV restants : 80
Le personnage est en feu, voici ses PV restants : 70
Le personnage est en feu, voici ses PV restants : 60
Le personnage est en feu, voici ses PV restants : 50
Le personnage est en feu, voici ses PV restants : 40
Le personnage est en feu, voici ses PV restants : 30
Le personnage est en feu, voici ses PV restants : 20
Le personnage est en feu, voici ses PV restants : 10
Le personnage est en feu, voici ses PV restants : 0
Le personnage est mort
Le personnage est mort

```