C# : collections, énumérations et tuples

Achref El Mouelhi

Docteur de l'université d'Aix-Marseille
Chercheur en programmation par contrainte (IA)
Ingénieur en génie logiciel

elmouelhi.achref@gmail.com

Plan

C#

Collections, énumérations et tuples ?

- des objets.
- permettant de regrouper et gérer plusieurs types de valeurs.

C#

Pourquoi ne pas utiliser les tableaux ?

C#

Pourquoi ne pas utiliser les tableaux ?

Réponse

- Il faut connaitre à l'avance la taille du tableau.
- Si on veut dépasser la taille déclarée, il faut créer un nouveau tableau puis copier l'ancien (certains langages ont proposés l'allocation dynamique mais ça reste difficile à manipuler).
- Il est difficile de supprimer ou d'ajouter un élément au milieu du tableau.
- Il faut parcourir tout le tableau pour localiser un élément (problème d'indexation).
- Les tableaux sont fortement typés : tous les éléments doivent être compatibles avec le type déclaré du tableau.

C#

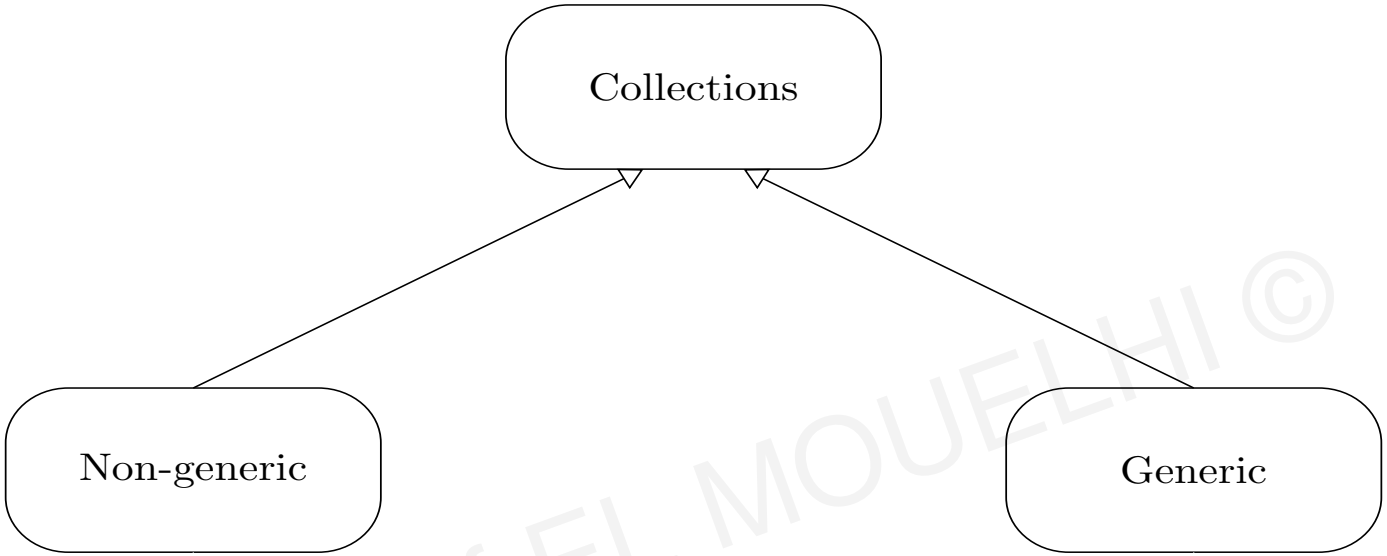
Collections, énumérations et tuples, quelle différence ?

- **Collections** : tableaux de taille dynamique acceptant simultanément des types différents.
- **Énumérations** : comme un tableau de constante.
- **Tuples** : tableau statique acceptant des types différents.

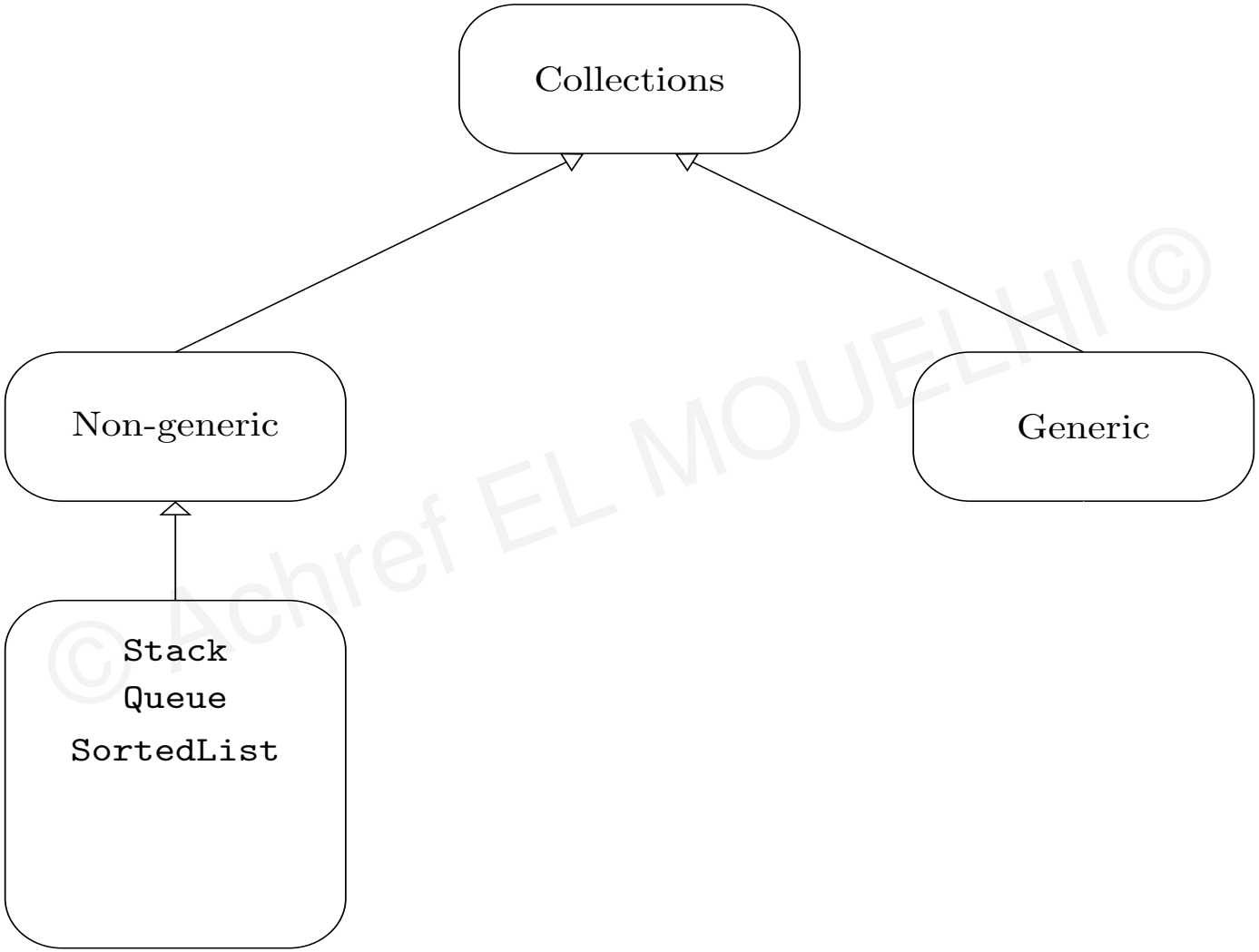
C#

Collections

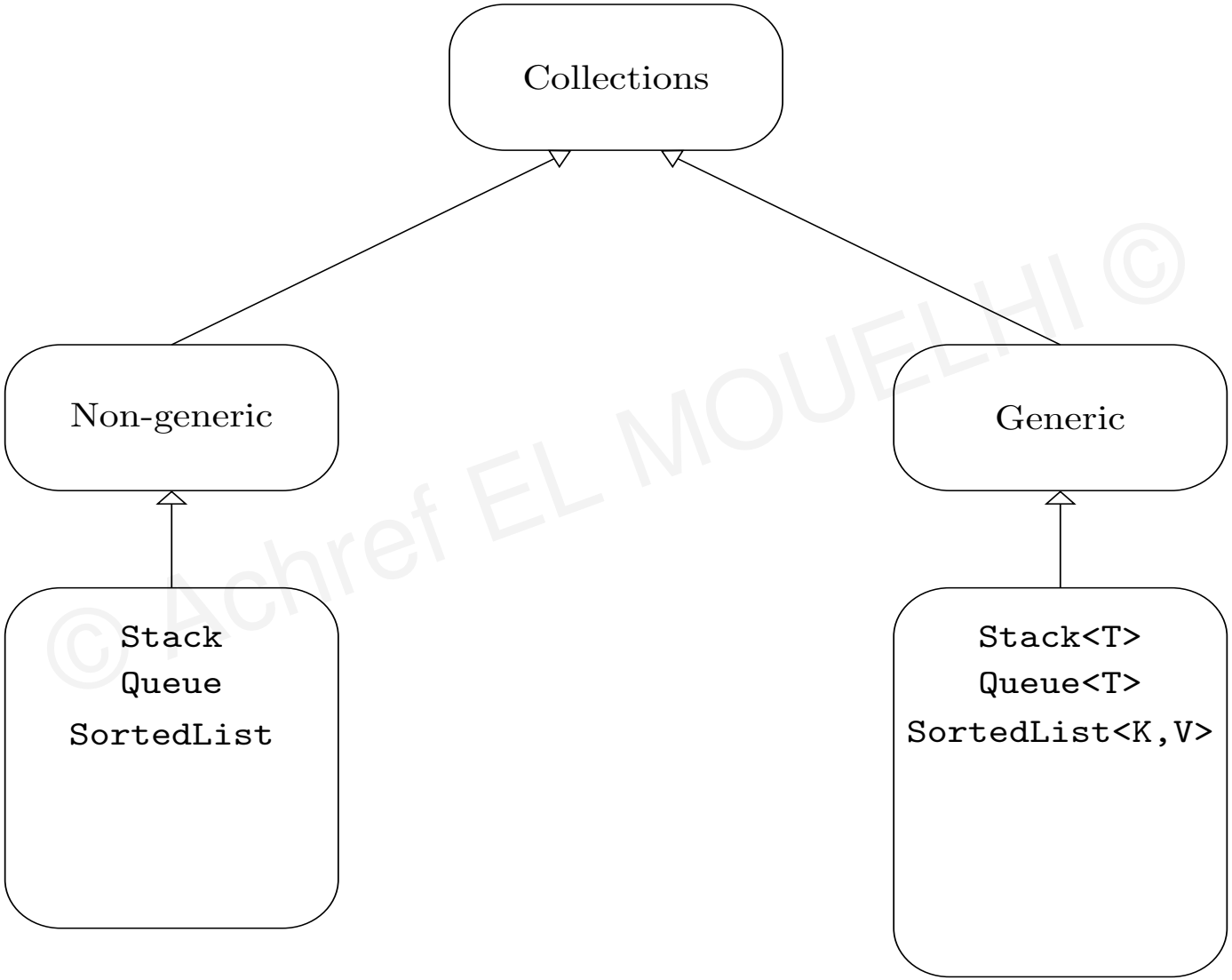
C#



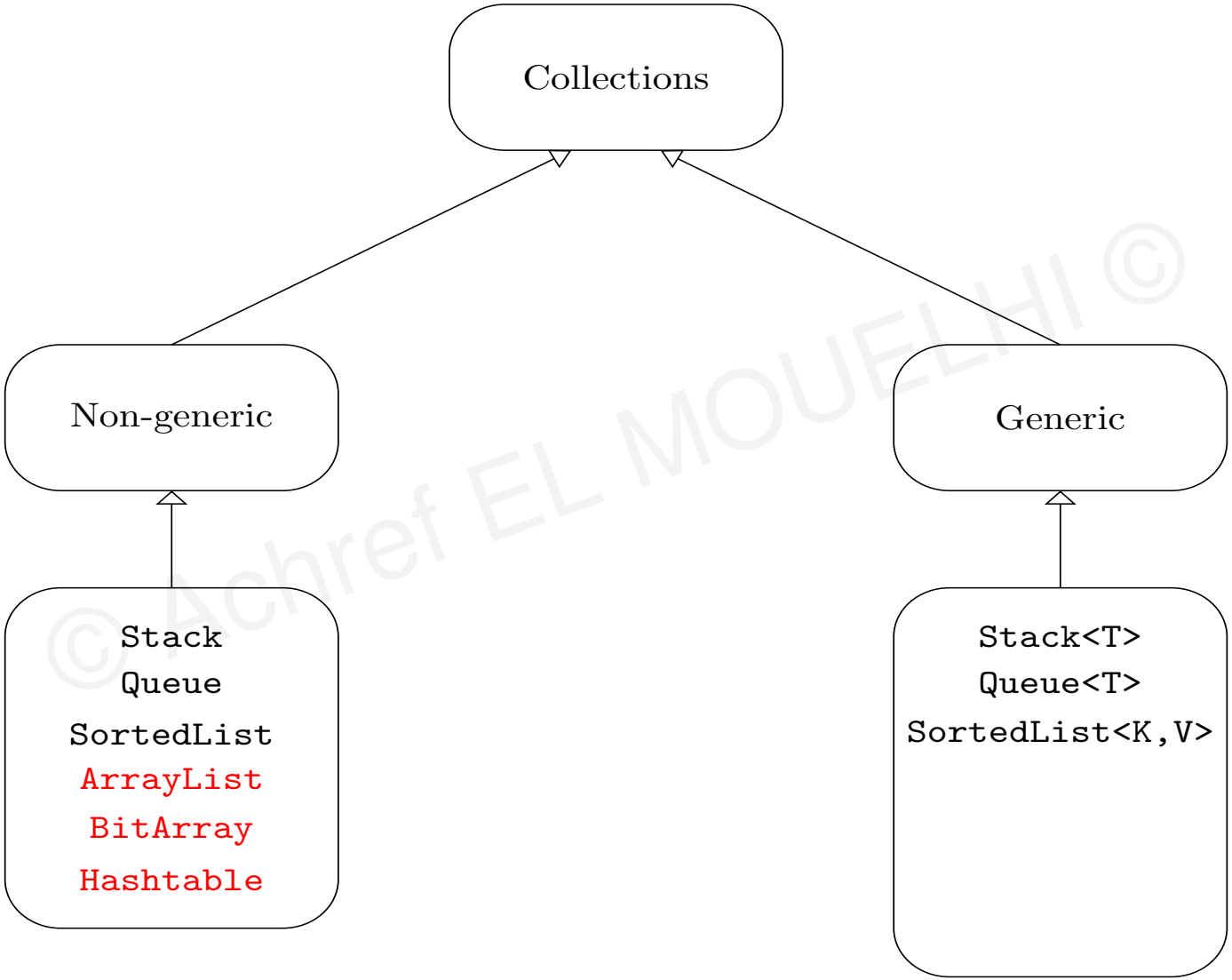
C#



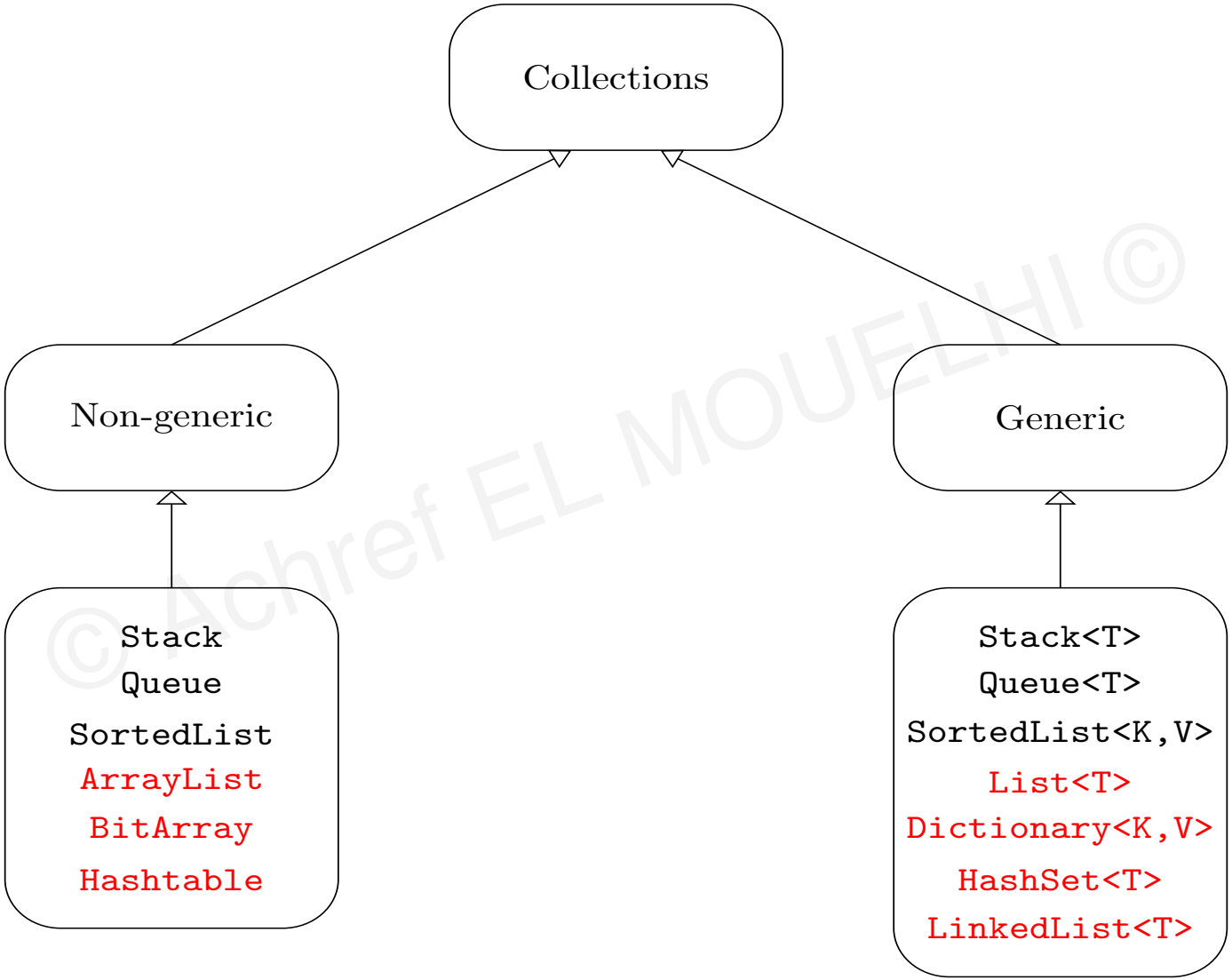
C#



C#



C#



C#

Collections ?

- sont des objets
- permettent de regrouper et gérer plusieurs objets de taille et/ou type dynamiques
- (des tableaux multi-types extensibles)

C#

Collections

- `ArrayList` : tableau dynamique ordonné.
- `BitArray` : tableau statique de booléens.
- `Queue` : file appliquant le principe FIFO (First In First Out).
- `Hashtable` : collection de couple clé/valeur.
- `SortedList` : collection de couple clé/valeur ordonnée selon la clé.
- `Stack` : collection appliquant le principe LIFO (Last In First Out)

C#

Commençons par créer un nouveau projet dans `MaSolution`

- Dans l'`Explorateur de solutions`, faire clic droit sur `MaSolution`
- Aller à `Ajouter > Nouveau projet`
- Sélectionner `Application console`
- Cliquer sur `Suivant`
- Remplir les champs
  - `Nom` avec `CoursCollection`
  - `Solution` avec `MaSolution`
- Valider

C#

Pour utiliser une liste, il faut importer l'espace de nom suivant

```
using System.Collections;
```

C#

Pour utiliser une liste, il faut importer l'espace de nom suivant

```
using System.Collections;
```

Pour créer un `BitArray`

```
BitArray bitArray = new BitArray(2);
```

C#

Pour utiliser une liste, il faut importer l'espace de nom suivant

```
using System.Collections;
```

Pour créer un `BitArray`

```
BitArray bitArray = new BitArray(2);
```


Pour ajouter des éléments dans un `BitArray`

```
bitArray.Set(0, true);
bitArray.Set(1, false);
```

C#

Pour parcourir un `BitArray`

```
foreach (bool b in bitArray)
{
    Console.WriteLine(b);
}
```

C#

Pour parcourir un `BitArray`

```
foreach (bool b in bitArray)
{
    Console.WriteLine(b);
}
```

Ça affiche

```
True
False
```

C#

Pour parcourir un `BitArray`

```
foreach (bool b in bitArray)
{
    Console.WriteLine(b);
}
```

Ça affiche

```
True
False
```

La dernière instruction déclenche une exception car on ne peut dépasser la taille d'un `BitArray`

```
bitArray.Set(2, true);
```

C#

Les Files

- Collection appliquant l'algorithme **FIFO** (**F**irst **I**n **F**irst **O**ut)
- Impossible donc de supprimer/modifier/lire un élément au milieu de la liste

C#

Exemple avec Queue

```
Queue q = new Queue();
q.Enqueue(2);
q.Enqueue(3);
q.Enqueue("bonjour");
q.Enqueue(5);
q.Dequeue();

foreach (var o in q)
{
    Console.WriteLine(o);
}
```

C#

Exemple avec Queue

```
Queue q = new Queue();
q.Enqueue(2);
q.Enqueue(3);
q.Enqueue("bonjour");
q.Enqueue(5);
q.Dequeue();

foreach (var o in q)
{
    Console.WriteLine(o);
}
```

Ça affiche

```
3
bonjour
5
```

C#

Les piles

- Collection appliquant l'algorithme **LIFO** (**L**ast **I**n **F**irst **O**ut)
- Impossible donc de supprimer/modifier/lire un élément au milieu de la liste

C#

Déclaration d'une pile

```
Stack stack = new Stack();
```

C#

Déclaration d'une pile

```
Stack stack = new Stack();
```

Empiler : ajouter un élément dans une pile

```
stack.Push(2);
stack.Push(5);
stack.Push("Bonjour");
stack.Push('c');
```

C#

Dépiler : supprimer l'élément au sommet de la pile

```
pile.Pop();
```

C#

Dépiler : supprimer l'élément au sommet de la pile

```
pile.Pop();
```

Supprimer tous les éléments de la pile

```
pile.Clear();
```

C#

Parcourir une pile

```
foreach (var elt in stack)
{
    Console.WriteLine(elt);
}
```

C#

Parcourir une pile

```
foreach (var elt in stack)
{
    Console.WriteLine(elt);
}
```

Ça affiche

```
Bonjour
5
2
```

C#

Consulter l'élément au sommet de la pile

```
pile.Peek();
```

C#

Pour créer un `ArrayList`

```
ArrayList arrayList = new ArrayList();
```

C#

Pour créer un `ArrayList`

```
ArrayList arrayList = new ArrayList();
```

Ajouter des nouveaux éléments dans un `ArrayList`

```
arrayList.Add("Bonjour");
arrayList.Add('c');
arrayList.Add(8);
arrayList.Add(true);
```

C#

Pour supprimer une valeur d'un `ArrayList`

```
arrayList.Remove('c');
```

C#

Pour supprimer une valeur d'un `ArrayList`

```
arrayList.Remove('c');
```

Pour supprimer un élément d'un `ArrayList` selon sa position

```
arrayList.RemoveAt(1);
```

C#

Pour parcourir les éléments d'un `ArrayList`

```
foreach (var elt in arrayList)
{
    Console.WriteLine(elt);
}
```

C#

Pour parcourir les éléments d'un `ArrayList`

```
foreach (var elt in arrayList)
{
    Console.WriteLine(elt);
}
```

Ça affiche

```
Bonjour
True
```

C#

Pour créer un `ArrayList` et l'initialiser avec les valeurs d'un tableau

```
int[] tab = new int[] { 2, 3, 5, 8, 2, 4, 2 };
ArrayList arrayList = new ArrayList(tab);
```

C#

Pour créer un `ArrayList` et l'initialiser avec les valeurs d'un tableau

```
int[] tab = new int[] { 2, 3, 5, 8, 2, 4, 2 };
ArrayList arrayList = new ArrayList(tab);
```

Ou en plus simple

```
ArrayList arrayList = new ArrayList(new int[] { 2, 3, 5, 8, 2, 4, 2 });
```

C#

Pour créer un `ArrayList` et l'initialiser avec les valeurs d'un tableau

```
int[] tab = new int[] { 2, 3, 5, 8, 2, 4, 2 };
ArrayList arrayList = new ArrayList(tab);
```

Ou en plus simple

```
ArrayList arrayList = new ArrayList(new int[] { 2, 3, 5, 8, 2, 4, 2 });
```

Ou encore

```
ArrayList arrayList = new ArrayList(new[] { 2, 3, 5, 8, 2, 4, 2 });
```

C#

On peut aussi initialiser un ArrayList ainsi

```
ArrayList arrayList = new ArrayList{ 2, 3, 5, 8, 2, 4, 2 };
```

C#

On peut aussi initialiser un ArrayList ainsi

```
ArrayList arrayList = new ArrayList{ 2, 3, 5, 8, 2, 4, 2 };
```

Ou plus simple depuis C# 6

```
ArrayList arrayList = [2, 3, 5, 8, 2, 4, 2];
```

C#

Considérons la liste suivante

```
ArrayList arrayList = [2, 3, 5, 8, 2, 4, 2];
```

C#

Considérons la liste suivante

```
ArrayList arrayList = [2, 3, 5, 8, 2, 4, 2];
```

Exercice 1

Écrire un code **C#** qui permet d'afficher la position de la deuxième occurrence de 2 (la solution proposée doit fonctionner quelle que soit la liste).

C#

Considérons la liste suivante

```
ArrayList arrayList = [ 2, 3, 5, 8, 2, 4, 2 ];
ArrayList positions = new ArrayList();
```

C#

Considérons la liste suivante

```
ArrayList arrayList = [ 2, 3, 5, 8, 2, 4, 2 ];
ArrayList positions = new ArrayList();
```

Exercice 2

Écrire un code **C#** qui permet de parcourir `arrayList` et stocker la position de toutes les occurrences de 2 dans `positions`.

C#

Considérons les listes suivantes

```
ArrayList arrayList = [ 2, 3, 5, 8, 2, 4, 2 ];
ArrayList pairs = new ArrayList();
```

C#

Considérons les listes suivantes

```
ArrayList arrayList = [ 2, 3, 5, 8, 2, 4, 2 ];
ArrayList pairs = new ArrayList();
```

Exercice 3

Écrire un code **C#** qui permet de stocker tous les nombres pairs de `arrayList` dans `pairs`.

C#

Les collections génériques ?

- sont des collections
- avec précision de type, mais toujours de taille dynamique
- (des tableaux extensibles)

C#

Exemples de collection génériques

- `List`
- `LinkedList`
- `SortedList`
- `Queue`
- `Stack`
- `HashSet`
- `Dictionary`
- ...

C#

Pourquoi utiliser les collections génériques ?

Pour

- imposer un type pour tous les éléments de la collection
- éviter d'avoir des exceptions si le type attendu ne correspond pas au type d'un élément
- éviter de faire de conversions inutiles
- ...

C#

Pour utiliser une liste, il faut importer le namespace

```
using System.Collections.Generic;
```

C#

Pour utiliser une liste, il faut importer le namespace

```
using System.Collections.Generic;
```

Déclaration

```
List <type> nomListe = new List<type>();
```

C#

Pour utiliser une liste, il faut importer le namespace

```
using System.Collections.Generic;
```

Déclaration

```
List <type> nomListe = new List<type>();
```

Exemple

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

C#

Déclaration + initialisation

```
List <string> voitures = new List<string> {"Citroen", "Ford", "Peugeot"
   , "Mercedes"};
```

C#

Déclaration + initialisation

```
List <string> voitures = new List<string> {"Citroen", "Ford", "Peugeot"
   , "Mercedes"};
```

Ou le raccourci

```
List<string> voitures = new(){ "Citroen", "Ford", "Peugeot", "Mercedes"
    };
```

C#

Déclaration + initialisation

```
List <string> voitures = new List<string> {"Citroen", "Ford", "Peugeot"
   , "Mercedes"};
```

Ou le raccourci

```
List<string> voitures = new(){ "Citroen", "Ford", "Peugeot", "Mercedes"
    };
```

Ou [Depuis C# 6.0]

```
List<string> voitures = ["Citroen", "Ford", "Peugeot", "Mercedes"];
```

Ajout d'un élément (à la fin de la liste)

```
voitures.Add("Renault");
```

C#

Ajout d'un élément (à la fin de la liste)

```
voitures.Add("Renault");
```

Accès à un élément d'une liste en lecture ou en écriture

```
voitures[3] = "Volkswagen";
// correct

Console.WriteLine(voitures[2]);
// imprime peugeot

voitures[5] = "Fiat";
// déclenche une exception car il n'existe aucun élément d'indice 5
```

C#

Pour ajouter un élément à une position précise

```
voitures.Insert(2, "Jeep");
```

Pour ajouter un élément à une position précise

```
voitures.Insert(2, "Jeep");
```

Pour ajouter plusieurs éléments à une position précise

```
liste.InsertRange(2, ["BMW", "Seat"]);
```

C#

Parcourir une liste avec un `for`

```
for (int i = 0; i < voitures.Count; i++)
{
    Console.WriteLine(voitures[i]);
}
// Count retourne la taille exacte du tableau
// Atttention à l'utilisation de Capacity qui retourne le plus
   petit multiple de 4 supérieur au égal à la taille réelle de
   la liste
```

C#

Parcourir une liste avec un `for`

```
for (int i = 0; i < voitures.Count; i++)
{
    Console.WriteLine(voitures[i]);
}
// Count retourne la taille exacte du tableau
// Atttention à l'utilisation de Capacity qui retourne le plus
   petit multiple de 4 supérieur au égal à la taille réelle de
   la liste
```


Parcourir une liste avec un `foreach`

```
foreach (var voiture in voitures)
{
    Console.WriteLine(voiture);
}
```

Parcourir une liste avec un `.ForEach`

```
voitures.ForEach(elt => Console.WriteLine(elt));
```

Parcourir une liste avec un `.ForEach`

```
voitures.ForEach(elt => Console.WriteLine(elt));
```

Ou en plus simple

```
voitures.ForEach(Console.WriteLine);
```

Parcourir une liste avec un `.ForEach`

```
voitures.ForEach(elt => Console.WriteLine(elt));
```

Ou en plus simple

```
voitures.ForEach(Console.WriteLine);
```

On peut aussi définir une méthode d'impression personnalisée

```
static void Print(string s)
{
    Console.WriteLine(s);
}
```

Parcourir une liste avec un `.ForEach`

```
voitures.ForEach(elt => Console.WriteLine(elt));
```

Ou en plus simple

```
voitures.ForEach(Console.WriteLine);
```

On peut aussi définir une méthode d'impression personnalisée

```
static void Print(string s)
{
    Console.WriteLine(s);
}
```

et ensuite la référencer

```
voitures.ForEach(Print);
```

C#

Autres méthodes sur les listes

- `RemoveAt(n)` : supprime d'une liste l'élément d'indice `n` (il existe aussi `Remove` et `RemoveAll`).
- `IndexOf(n)` : retourne l'indice de la première apparition de la valeur `n` dans une liste.
- `Contains(n)` : retourne `true` si `n` appartient à la liste.
- `Find(elt => condition)` : retourne le premier élément de la liste qui respecte `condition` (`Exists` fonctionne d'une manière similaire mais elle retourne un booléen).
- `Sort()` : trie une liste d'entiers.
- `ToArray()` : retourne un tableau statique résultat de la conversion de la liste.
- ...

C#

Considérons la liste suivante

```
List<string> voitures = ["Citroen", "Ford", "Peugeot", "
   Mercedes"];
```

C#

Considérons la liste suivante

```
List<string> voitures = ["Citroen", "Ford", "Peugeot", "
   Mercedes"];
```

Exercice

Écrire un code **C#** qui permet d'afficher la marque qui contient le plus petit nombre de caractères.

C#

Tableaux vs listes

- Un tableau peut être multidimensionnel mais il est de taille fixe.
- Une liste est unidimensionnel mais elle est de taille variable.
- On ne peut supprimer un élément situé au début ou au milieu d'un tableau.

C#

Les dictionnaires

- une collection de couple clé/valeur
- une clé est un indice personnalisé (unique)

C#

Déclaration

```
Dictionary<type1, type2> dic = new Dictionary<type1, type2>();
```

C#

Déclaration

```
Dictionary<type1, type2> dic = new Dictionary<type1, type2>();
```

Exemple

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

C#

Déclaration

```
Dictionary<type1, type2> dic = new Dictionary<type1, type2>();
```

Exemple

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

Exemple

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

C#

Remplir le dictionnaire

```
fcb.Add(10, "Messi");
fcb.Add(23, "Umtiti");
fcb.Add(4, "Rakitic");
fcb.Add(9, "Suarez");
```

C#

Remplir le dictionnaire

```
fcb.Add(10, "Messi");
fcb.Add(23, "Umtiti");
fcb.Add(4, "Rakitic");
fcb.Add(9, "Suarez");
```


`Add()` lève une exception si la clé existe déjà.

C#

Création + ajout

```
var joueurs = new Dictionary<int, string>
{
    [10] = "Messi",
    [23] = "Umtiti",
    [4]  = "Rakitic",
    [9]  = "Suarez"
};
```

C#

Vérifier l'appartenance d'une valeur et/ou d'une clé à un dictionnaire

```
Console.WriteLine(fcb.ContainsValue("iniesta"));
Console.WriteLine(fcb.ContainsKey(10));
```

Parcourir le dictionnaire et afficher le couple (clé,valeur)

```
foreach (KeyValuePair<int,string> elt in fcb)
{
    Console.WriteLine(elt.Key + elt.Value);
}
```

Parcourir le dictionnaire et afficher le couple (clé,valeur)

```
foreach (KeyValuePair<int,string> elt in fcb)
{
    Console.WriteLine(elt.Key + elt.Value);
}
```

Parcourir le dictionnaire pour afficher les valeurs

```
foreach (string elt in fcb.Values)
{
    Console.WriteLine(elt);
}
```

Parcourir le dictionnaire et afficher le couple (clé,valeur)

```
foreach (KeyValuePair<int,string> elt in fcb)
{
    Console.WriteLine(elt.Key + elt.Value);
}
```

Parcourir le dictionnaire pour afficher les valeurs

```
foreach (string elt in fcb.Values)
{
    Console.WriteLine(elt);
}
```

Pour récupérer la liste des clés, il faut déclarer un `KeyCollection`

```
Dictionary<int,string>.KeyCollection clefs = fcb.Keys;
foreach (int elt in clefs)
{
    Console.WriteLine(elt);
}
```

C#

Pour récupérer la liste des valeurs, il faut déclarer un `ValueCollection`

```
Dictionary<int,string>.ValueCollection vals = fcb.Values;
foreach (string elt in vals)
{
    Console.WriteLine(elt);
}
```

C#

Pour récupérer la liste des valeurs, il faut déclarer un `ValueCollection`

```
Dictionary<int,string>.ValueCollection vals = fcb.Values;
foreach (string elt in vals)
{
    Console.WriteLine(elt);
}
```

Pour modifier la valeur d'un élément selon la clé

```
fcb[10] = "Rivaldo";
```

C#

Pour récupérer la liste des valeurs, il faut déclarer un `ValueCollection`

```
Dictionary<int,string>.ValueCollection vals = fcb.Values;
foreach (string elt in vals)
{
    Console.WriteLine(elt);
}
```

Pour modifier la valeur d'un élément selon la clé

```
fcb[10] = "Rivaldo";
```

L'indexeur ajoute l'élément si la clé n'existe pas et modifie si elle existe

```
fcb[6] = "Xavi";
```

C#

Pour récupérer la valeur associée à une clé

```
Console.WriteLine(fcb[10]);
// messi
```

C#

Pour récupérer la valeur associée à une clé

```
Console.WriteLine(fcb[10]);
// messi
```

Si la clé n'existe pas ⇒ `KeyNotFoundException` **levée**

```
Console.WriteLine(fcb[15]);
```

C#

Pour récupérer la valeur associée à une clé

```
Console.WriteLine(fcb[10]);
// messi
```

Si la clé n'existe pas ⇒ `KeyNotFoundException` **levée**

```
Console.WriteLine(fcb[15]);
```


Utiliser `GetValueOrDefault` **pour spécifier une valeur par défaut si la clé n'existe pas**

```
Console.WriteLine(fcb.GetValueOrDefault(15, "Numéro non affecté
    "));
```

C#

Une autre alternative avec `GetValueOrDefault` pour spécifier une valeur par défaut si la clé n'existe pas

```
fcb.TryGetValue(10, out string joueur);
Console.WriteLine(joueur ?? "Numéro non affecté");
// Messi

fcb.TryGetValue(15, out string joueur2);
Console.WriteLine(joueur2 ?? "Numéro non affecté");
// Numéro non affecté
```

C#

Exercice 1

Écrire un code **C#** qui permet de

- demander à l'utilisateur de saisir le numéro d'un joueur,
- afficher le nom du joueur ayant ce numéro s'il existe, une erreur sinon.

C#

Exercice 2

Écrire un code **C#** qui permet de

- demander à l'utilisateur de saisir un numéro et un nom,
- mettre à jour le nom si le numéro existe, ajouter la paire sinon,
- afficher le contenu du dictionnaire.

C#

Exercice 3

Écrire un code **C#** qui permet de

- demander à l'utilisateur de saisir le nom d'un joueur,
- afficher le numéro du joueur ayant ce nom s'il existe, une erreur sinon.

C#

Exercice : étant donnée la liste suivante :

```
ArrayList arrayList = new ArrayList { 2, 5, "Bonjour
   ", true, 'c', "3", "b", false, 10 };
```

Écrire un programme C# qui permet de stocker dans un dictionnaire les types contenus dans la liste `arrayList` ainsi que le nombre d'éléments de cette liste appartenant à chaque type.

Résultat attendu :

```
Int32 : 3
String : 3
Boolean : 2
Char : 1
```

C#

HashSet

- Collection générique non ordonnée qui stocke des éléments uniques
- Basée sur une table de hachage
- Pas d'accès aux éléments via l'indice
- Les opérations principales sont très rapides

C#

Construction à partir d'une liste + suppression de doublons

```
var liste = new List<int> { 2, 3, 5, 3, 2, 8 };
var uniques = new HashSet<int>(liste);

foreach (var x in uniques)
{
    Console.WriteLine(x);
}
```

C#

Construction à partir d'une liste + suppression de doublons

```
var liste = new List<int> { 2, 3, 5, 3, 2, 8 };
var uniques = new HashSet<int>(liste);

foreach (var x in uniques)
{
    Console.WriteLine(x);
}
```

Pour tester l'appartenance d'un élément

```
var autorises = new HashSet<string> { "admin", "manager", "user" };

if (autorises.Contains("admin"))
{
    Console.WriteLine("Accès autorisé");
}
```

C#

Création et initialisation

```
var set1 = new HashSet<int>();

var set2 = new HashSet<int> { 1, 2, 3 };

var set3 = new HashSet<int>(new[] { 1, 2, 2, 3 });

HashSet<int> set4 = [1, 2, 3, 4];
```

C#

Pour ajouter un élément, utiliser `Add` qui retourne `False` si l'élément existe

```
var liste = new List<int> { 2, 3, 5, 3, 2, 8 };
var vus = new HashSet<int>();

foreach (var x in liste)
{
    if (!vus.Add(x))
    {
        Console.WriteLine($"Doublon détecté : {x}");
    }
}
```

C#

Pour supprimer un ou plusieurs éléments

```
set.Remove(3);
// true si supprimé

set.Clear();
// supprime tout
```

C#

Pour supprimer un ou plusieurs éléments

```
set.Remove(3);
// true si supprimé

set.Clear();
// supprime tout
```


Pour compter le nombre d'éléments

```
Console.WriteLine(set.Count);
```

C#

Opérations ensemblistes

```
var a = new HashSet<int> { 1, 2, 3, 4 };
var b = new HashSet<int> { 3, 4, 5, 6 };

a.IntersectWith(b);
// {3, 4}

a.UnionWith(b);
// {1,2,3,4,5,6}

a.ExceptWith(b);
// {1,2}
```

C#

Syntaxe de déclaration d'une pile acceptant un type générique

```
Stack<type> nomPile = new Stack<type>();
```

Exemple

```
Stack<int> pile = new Stack<int>();
```

C#

Une énumération ?

- un ensemble de constantes nommées
- ne pouvant pas être déclaré dans le `Main`

Déclaration d'une énumération

```
enum Sports { Foot, Hand, Hockey, Tennis, Basket };
```

Par défaut, le premier élément a la valeur 0.
L'élément successif a une valeur augmenté de 1.

Déclaration d'une énumération

```
enum Sports { Foot, Hand, Hockey, Tennis, Basket };
```

Par défaut, le premier élément a la valeur 0.
L'élément successif a une valeur augmenté de 1.

Exemple

```
Console.WriteLine(Sports.Tennis); // affiche Tennis
```

Déclaration d'une énumération

```
enum Sports { Foot, Hand, Hockey, Tennis, Basket };
```

Par défaut, le premier élément a la valeur 0.
L'élément successif a une valeur augmenté de 1.

Exemple

```
Console.WriteLine(Sports.Tennis); // affiche Tennis
```

On peut utiliser une énumération comme un type

```
Sports sport = Sports.Foot;
Console.WriteLine(sport); // affiche Foot
```

Déclaration d'une énumération

```
enum Sports { Foot, Hand, Hockey, Tennis, Basket };
```

Par défaut, le premier élément a la valeur 0.
L'élément successif a une valeur augmenté de 1.

Exemple

```
Console.WriteLine(Sports.Tennis); // affiche Tennis
```

On peut utiliser une énumération comme un type

```
Sports sport = Sports.Foot;
Console.WriteLine(sport); // affiche Foot
```

Récupérer la valeur

```
Console.WriteLine((int)sport); // affiche 0
Console.WriteLine((int)Sports.Tennis); // affiche 3
```

C#

Déclaration d'une énumération + modification des constantes par défaut

```
enum Sports { Foot=2, Hand, Hockey=5, Tennis, Basket };
```

Exemple

```
Console.WriteLine((int)Sports.Hand); // affiche 3
Console.WriteLine((int)Sports.Hockey); // affiche 5
Console.WriteLine((int)Sports.Tennis); // affiche 6
```

C#

Un tuple ?

- un objet acceptant plusieurs valeurs de type différent.
- permettant à une méthode de retourner plusieurs valeurs sans créer de classe ou de structure.
- ses éléments pouvant être modifiés.
- impossible d'ajouter un nouvel élément, après création, ou d'en supprimer un existant.

C#

Considérons les deux méthodes qui permettent de retourner la valeur `min` ou `max`

```
public static int FindMax(int i, int j)
{
    return i > j ? i : j;
}

public static int FindMin(int i, int j)
{
    return i < j ? i : j;
}
```

C#

Considérons les deux méthodes qui permettent de retourner la valeur `min` ou `max`

```
public static int FindMax(int i, int j)
{
    return i > j ? i : j;
}

public static int FindMin(int i, int j)
{
    return i < j ? i : j;
}
```

Question

Comment faire pour fusionner les deux méthodes sans tout en gardant le même nombre de paramètres ?

On peut utiliser les tuples

```
public static Tuple<int, int> FindMinMax(int i, int j)
{
    int max = i > j ? i : j;
    int min = i < j ? i : j;
    return new Tuple<int, int>(min, max);
}
```

On peut utiliser les tuples

```
public static Tuple<int, int> FindMinMax(int i, int j)
{
    int max = i > j ? i : j;
    int min = i < j ? i : j;
    return new Tuple<int, int>(min, max);
}
```

Une deuxième écriture

```
public static Tuple<int, int> FindMinMax(int i, int j)
{
    int max = i > j ? i : j;
    int min = i < j ? i : j;
    return Tuple.Create(min, max);
}
```

On peut utiliser les tuples

```
public static Tuple<int, int> FindMinMax(int i, int j)
{
    int max = i > j ? i : j;
    int min = i < j ? i : j;
    return new Tuple<int, int>(min, max);
}
```

Une deuxième écriture

```
public static Tuple<int, int> FindMinMax(int i, int j)
{
    int max = i > j ? i : j;
    int min = i < j ? i : j;
    return Tuple.Create(min, max);
}
```

Question

Comment faire pour accéder aux valeurs du tuple retourné ?

C#

Pour accéder aux valeurs d'un tuple

```
var tuple = FindMinMax(2, 3);
Console.WriteLine($"Le min de 2 et 3 est : { tuple.Item1 }");
// affiche 2

Console.WriteLine($"Le max de 2 et 3 est : { tuple.Item2 }");
// affiche 3
```

C#

La déclaration du tuple peut être rendu implicite ainsi [C# 7]

```
static (int, int) FindMinMax(int i, int j)
{
    int max = i > j ? i : j;
    int min = i < j ? i : j;
    return (min, max);
}
```

C#

La déclaration du tuple peut être rendu implicite ainsi [C# 7]

```
static (int, int) FindMinMax(int i, int j)
{
    int max = i > j ? i : j;
    int min = i < j ? i : j;
    return (min, max);
}
```

Pour récupérer le résultat

```
var tuple = FindMinMax(2, 3);
Console.WriteLine($"Le min de 2 et 3 est : { tuple.Item1 }");
// affiche 2

Console.WriteLine($"Le max de 2 et 3 est : { tuple.Item2 }");
// affiche 3
```

C#

Remarques

- Un tuple accepte au maximum 8 éléments.
- Les éléments d'un tuple sont accessibles avec les `Item1`, `Item2`, ... `Item7`.
- Le dernier élément (le 8 ème élément) sera renvoyé à l'aide de la propriété `Rest`.
- Le dernier élément est souvent utilisé pour imbriquer un autre tuple.
- Un tuple peut imbriquer d'autres tuples. Cependant, il est recommandé de le placer dans la dernière position.

C#

Considérons le tuple suivant

```
var t = Tuple.Create(1, 2, 3, 4, 5, 6, 7, 8);
```

C#

Considérons le tuple suivant

```
var t = Tuple.Create(1, 2, 3, 4, 5, 6, 7, 8);
```

`Rest` retourne un tuple

```
Console.WriteLine(t.Rest);
// affiche (8)
```

C#

Considérons le tuple suivant

```
var t = Tuple.Create(1, 2, 3, 4, 5, 6, 7, 8);
```

`Rest` **retourne un tuple**

```
Console.WriteLine(t.Rest);
// affiche (8)
```

Pour récupérer la dernière valeur d'un tuple

```
Console.WriteLine(t.Rest.Item1);
// affiche 8
```

C#

Tableaux vs tuples

- Un tableau peut être multidimensionnel mais il est de taille fixe et type unique.
- Un tuple est un regroupement de valeurs non nommés pas forcément de même type.
- On ne peut supprimer un élément ni d'un tableau ni d'un tuple.

C#

ValueTuples ?

- un tuple acceptant un nombre variable de valeurs de type différent (même plus que 8).
- ses éléments peuvent être nommés.

C#

Considérons le `ValueTuple` suivant

```
var t = (1, 2, 3, 4, 5, 6, 7, 8, 9);
```

C#

Considérons le `ValueTuple` suivant

```
var t = (1, 2, 3, 4, 5, 6, 7, 8, 9);
```

Pour accéder au dernier élément

```
Console.WriteLine(t.Item9);
// affiche 9
```

C#

La propriété `Rest` permet de récupérer les éléments situés à partir de la position 8

```
Console.WriteLine(t.Rest);
// affiche (8, 9)
```

C#

La propriété `Rest` permet de récupérer les éléments situés à partir de la position 8

```
Console.WriteLine(t.Rest);
// affiche (8, 9)
```

L'élément à la position 8 en utilisant la propriété `Rest`

```
Console.WriteLine(t.Rest.Item1);
// affiche 8
```

C#

La propriété `Rest` permet de récupérer les éléments situés à partir de la position 8

```
Console.WriteLine(t.Rest);
// affiche (8, 9)
```

L'élément à la position 8 en utilisant la propriété `Rest`

```
Console.WriteLine(t.Rest.Item1);
// affiche 8
```

L'élément à la position 9 en utilisant la propriété `Rest`

```
Console.WriteLine(t.Rest.Item2);
// affiche 9
```

C#

Il est possible de nommer les membres d'un tuple

```
(int note1, int note2, int note3) notes = (10, 20, 13);
```

C#

Il est possible de nommer les membres d'un tuple

```
(int note1, int note2, int note3) notes = (10, 20, 13);
```

Pour accéder à un élément, on utilise son nom

```
Console.WriteLine(notes.note1);
// affiche 10
```

C#

Il est possible de nommer les membres d'un tuple

```
(int note1, int note2, int note3) notes = (10, 20, 13);
```

Pour accéder à un élément, on utilise son nom

```
Console.WriteLine(notes.note1);
// affiche 10
```

Ou la propriété `Item`

```
Console.WriteLine(notes.Item1);
// affiche 10
```