



## Concepteur Développeur en Informatique

### Développer des composants d'interface

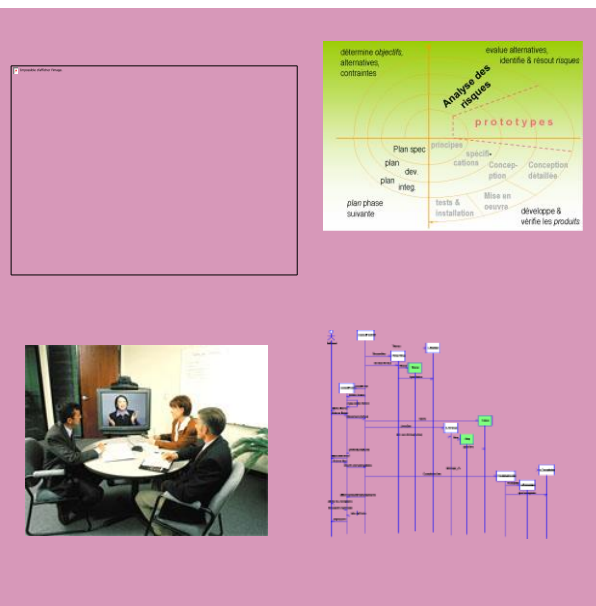
#### POO – Collections

Accueil

Apprentissage

PAE

Evaluation



Localisation : U03-E03-S01

## SOMMAIRE

<b>1. Introduction .....</b>	<b>3</b>
<b>2. Les tableaux dynamiques .....</b>	<b>4</b>
2.1. Les tableaux fixes.....	4
2.2. Les tableaux dynamiques - ArrayList .....	5
2.2.1. Méthodes et propriétés de ArrayList .....	5
<b>3. Déterminer le type d'un objet.....</b>	<b>6</b>
3.1. Méthode GetType et expression typeof .....	6
3.2. L'opérateur is.....	6
3.3. L'opérateur as.....	7
3.4. Traiter les éléments d'un tableau fonction de leur type.....	7
<b>4. Les files d'attente .....</b>	<b>8</b>
4.1. La collection Queue .....	8
4.1.1. Les principales méthodes de Queue .....	8
4.1.2. Exemple illustrant l'usage de Queue .....	9
4.2. La collection Stack.....	10
4.3. Exemple illustrant l'usage de Stack.....	10
<b>5. Les dictionnaires .....</b>	<b>11</b>
5.1.1. La collection HashTable.....	11
5.2. La collection SortedList .....	12
<b>6. Les collections de génériques .....</b>	<b>13</b>
6.1. Introduction aux génériques.....	13
6.2. Un exemple pour comprendre .....	14
<b>7. Les listes chaînées LinkedList&lt;T&gt; .....</b>	<b>15</b>

## 1. Introduction

Ce support est la suite des deux premiers documents consacrés à l'initiation à la programmation objet et traite plus particulièrement de la création et de la manipulation de collections d'objets.

La notion de collection est un concept qui n'est pas sans rapport avec les tableaux que vous avez manipulés lors de la précédente phase d'apprentissage.

Vous venez d'apprendre comment définir des types et créer des instances de ceux-ci.

Vous allez maintenant étudier comment assurer la constitution de collections d'objets.

Une collection s'apparente à un ensemble d'objets, le plus souvent de même type.

Nous aurons souvent besoin de recourir aux collections pour gérer des ensembles homogènes d'objets qu'ils soient de type :

- « métier » : Ils représentent alors les concepts propres à une entreprise : Par exemple, au sein d'une entreprise de distribution, nous allons manipuler des collections de clients et de commandes, chaque client étant une instance de la classe Client et chaque commande une instance de la classe Commande.
- « techniques » : Comme par exemple au sein d'une application Windows : Nous manipulerons des collections de contrôles graphiques, chaque contrôle étant une instance d'une classe de contrôle graphique (formulaire, boîte de texte, liste déroulante, ...).

Vous apprendrez ensuite dans le quatrième document comment préserver la vie de ces objets au-delà de celle de l'application en les rendant persistants.

Nous aborderons ainsi la persistance et la sérialisation des objets.

Nous reviendrons dans un premier temps sur les tableaux statiques puis aborderons les tableaux dynamiques et collections, l'objectif étant de retenir parmi celles-ci, le type de collection adapté à notre besoin :

- Tableaux statiques
- ArrayList
- Queue
- Stack
- Hashtable
- SortedList
- Génériques

Nous reviendrons dans cette étape sur les mécanismes de détermination et de conversion de types.

## 2. Les tableaux dynamiques

### 2.1. Les tableaux fixes

Nous les avons déjà utilisés lors des précédents exercices. Ils sont intéressants mais néanmoins limités. Cette technique nous impose de définir la taille initiale du tableau et celui-ci ne peut être étendu dynamiquement sans réserve.

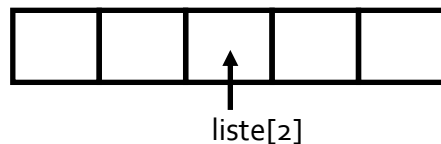
Nous pouvons augmenter l'étendue d'un tableau en redéfinissant sa taille qui sera toutefois toujours constante. Cette approche reste réservée aux tableaux unidimensionnels et n'est pas vraiment optimale comme vous pourrez en juger par le code suivant et l'explication donnée sur les mécanismes sous-jacents mis en œuvre.

Soit un tableau de chaînes stockant 10 mots. La taille de sa première dimension peut être étendue grâce à la méthode statique `Resize` de la classe `Array`.

```
// tableau de 10 postes
string[] tabmots = new string[10];
// nouvelle dimension précisée de longueur du tableau + 10 postes
Array.Resize(ref tabmots, tabmots.Length + 10);
```

Un tableau est déclaré en spécifiant son type, sa dimension (ou rang) et son nom.  
type [ ] nom ;

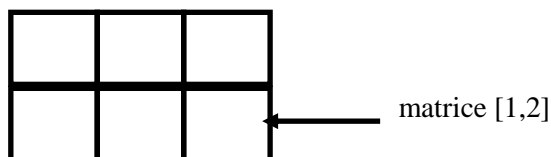
`int [ ] liste;` // tableau à une dimension



`liste[2]` désignera le 3ème élément du tableau `liste`.

Un seul index est nécessaire pour repérer un élément de tableau.

`string [ , ] matrice ;` // tableau à deux dimensions



`matrice [1,2]` désignera l'élément situé sur la 2ème ligne, 3ème colonne du tableau `matrice`.

Deux index sont nécessaires pour repérer un élément de tableau, l'indice des lignes et l'indice des colonnes.

Nous avons même la possibilité de travailler avec des tableaux dont les postes contiennent eux-mêmes des tableaux... Ces tableaux, dits en escaliers, sont peu usités et ne seront donc pas abordés ici.

#### Que fait la méthode `Resize` ?

Un tableau est de taille fixe et la méthode `Resize` cache en fait une technique peu performante. Cette méthode **alloue un nouveau tableau** avec la **taille spécifiée**, copie les éléments de l'ancien tableau dans le nouveau, puis remplace l'ancien tableau par le nouveau. Pas vraiment une affaire donc !

Si la taille spécifiée est inférieure au nombre de valeurs conservées dans le tableau, les éléments de l'ancien tableau sont alors copiés vers le nouveau jusqu'à ce que celui-ci soit rempli. Le reste des éléments du tableau d'origine est donc ignoré !

Quelques précautions d'usage s'imposent donc ...et le recours à d'autres formes de tableaux dans ce cas seraient certainement plus pertinent.

Dans bien des cas, nous solliciterons d'autres structures plus adaptées que les tableaux pour conteneurs

## 2.2. Les tableaux dynamiques - ArrayList

Le type ArrayList permet de créer un tableau dynamique dans lequel il est possible d'ajouter, d'insérer ou de supprimer des éléments.

Cette classe se trouve, comme les autres classes de même nature, dans l'espace de noms System.Collections.

Il faut donc utiliser une directive **using** à **System.Collections** pour faciliter l'accès à ces objets.

Ce type ArrayList ne précise pas la nature des objets qui peuvent y entrer.

Pouvons-nous alors y stocker pêle-mêle des « choux » et des « carottes ».

Pour le vérifier, nous allons déclarer un tableau dynamique auquel nous ajoutons un entier, une chaîne et un double et, soyons fous, un type complexe stagiaire.

Exemple :

```
ArrayList Objets = new ArrayList();
Objets.Add(1); // Ajout d'un entier
Objets.Add("ma chaîne"); // Ajout d'une chaîne
Objets.Add(120.5); // Ajout d'un double

Stagiaire stagiaire = new Stagiaire();
stagiaire.Nom = "boST";
stagiaire.DateNaissance = DateTime.Parse("13/01/1962");
Objets.Add(stagiaire); // Ajout objet complexe
```

Nous avons donc la réponse à notre questionnement précédent : nous pouvons mélanger dans cette liste des instances de type gérés par valeur et par référence.

### 2.2.1. Méthodes et propriétés de ArrayList

Voici les principales méthodes et propriétés d'un tableau dynamique. Vous pouvez préciser la capacité du tableau.

Méthodes ou propriétés	But
Capacity	Nombre d'éléments que le tableau peut contenir
Count	Nombre d'éléments actuellement dans le tableau
Add(object)	Ajoute un élément au tableau
Remove(object)	Enlève un élément du tableau
RemoveAt(int)	Enlève un élément à l'indice fourni
Insert(int, object)	Insère un élément à l'indice fourni
Clear()	Vide le tableau
Contains(object)	Renvoie un booléen vrai si l'objet fourni est présent dans le tableau
al[index] <sup>1</sup>	Fournit l'objet situé à la valeur de l'index

<sup>1</sup> où al prend le nom de la variable ArrayList ex : object elt = tabElements[0] affecte l'objet en position 0

Vous pouvez donc utiliser une ArrayList pour y stocker des valeurs de différentes natures. Il s'agit d'une collection faiblement typée et vous ne serez pas assuré que les valeurs qui y sont stockées soient correctement ordonnées.

Sauf à mettre en œuvre des mécanismes complexes de tris...

La limite de cette collection, mais aussi son intérêt dans une certaine mesure, réside dans le fait que l'on manipule des objets de toute nature voire des types valeur comme int ou double et des types référence.

Comment dès lors savoir à qui nous avons affaire lorsque nous consultons un objet extrait de la liste ? Nous devrons en effet connaître son type pour pouvoir invoquer les bonnes méthodes et consulter ou affecter les bonnes propriétés.

Ce besoin va être satisfait grâce aux méthodes d'interrogation de type.

Ce mécanisme repose sur un principe essentiel de l'architecture .Net, la réflexion et l'auto-description des types.

Introduisons donc les techniques de détermination de type.

### 3. Déterminer le type d'un objet

Nous avons pour satisfaire cet objectif plusieurs mécanismes à notre disposition.

Nous mobiliserons ensemble ou alternativement :

- L'opérateur **Is**
- L'opérateur **As**
- La méthode **GetType** disponible sur chaque objet
- L'expression **typeof**

#### 3.1.Méthode GetType et expression typeof

La méthode **GetType()** permet d'extraire le type d'un objet et comparer celui-ci au type passé à l'expression **typeof(Type)** qui permet d'obtenir l'objet **System.Type** d'un type et donc de comparer ces deux valeurs.

```
Type type = stagiaire.GetType();
```

L'expression suivante pourrait alors être :

```
if (type == typeof(Stagiaire))
```

#### 3.2.L'opérateur is

Une expression **is** prend la valeur **true** si l'expression fournie n'est pas **null**, et que l'objet fourni peut être converti en type fourni sans entraîner la levée d'une exception.

Dans l'exemple qui suit, si l'objet est de type Salarie alors nous modifions sa propriété Nom.

```
if (!(salarie is Salarie)) return false; else ((Salarie)salarie).Nom = "Bost";
```

**Notez** que l'opérateur **is** considère uniquement les conversions de référence, les conversions boxing et les conversions unboxing (voir point suivant).

D'autres conversions, comme les conversions définies par l'utilisateur, ne sont pas prises en compte.

### 3.3.L'opérateur as

Assez proche de l'opérateur **is**, **as** peut être plus performant. Il fournit l'objet converti dans son type si c'est possible, sinon une absence de référence. Il y a une seule conversion ici alors qu'il y en a deux dans l'exemple précédent avec **is**.

```
Salarie salaireConverti = salaire as Salarie;
if (salarieConverti == null) return false;
else salaireConverti.Nom = "Bost";
```

### 3.4.Traiter les éléments d'un tableau fonction de leur type

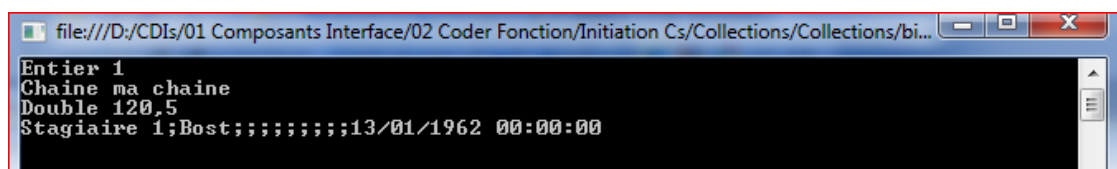
Regardons dans l'exemple quels sont les types stockés dans notre collection.

Comme toutes les collections, et les tableaux dont la définition s'appuie aussi sur le type collection, nous pouvons élaborer une liste énumérée des objets stockés dans notre collection de type ArrayList.

```
foreach (object element in Objets)
{
    if (element.GetType() == typeof(int))
    {
        Console.WriteLine("Entier {0}", element.ToString());
    }
    if (element.GetType() == typeof(string))
    {
        Console.WriteLine("Chaine {0}", element.ToString());
    }
    if (element is double)
    {
        Console.WriteLine("Double {0}", element.ToString());
    }
    if (element.GetType() == typeof(Stagiaire))
    {
        Console.WriteLine("Stagiaire {0}", element.ToString());
    }
}
```

Si nous devons faire appel à une méthode spécifique de notre objet, il nous faudra alors le convertir, comme dans les deux exemples précédents.

Vous remarquerez l'invocation de la méthode substituée ToString() du type Stagiaire :



## 4. Les files d'attente

Nous aurons parfois besoin de manipuler des éléments en précisant la manière dont ils seront insérés et sortis de la collection.

Nous avons deux collections spécialisées qui nous permettent de mettre en œuvre des mécanismes courants de gestion de file d'attente de type FIFO (First In First Out) et LIFO (Last In First Out).

Ces approches sont utiles en matière de gestion et valorisation de stocks, et plus généralement dans la mise en place de traitements asynchrones d'objets.

### 4.1. La collection Queue

La classe **Queue** implémente un mécanisme de type FIFO (First In First Out).  
Un élément est inséré à la fin de la file et il est supprimé au début de la file.

La collection Queue implémente certaines des méthodes et propriétés communes aux collections et deux méthodes particulières pour insérer et supprimer des éléments dans la file **Enqueue** et **Dequeue**.

Compte tenu de son fonctionnement, elle n'implémente pas de méthodes qui permettent d'ajouter, d'extraire ou de supprimer un élément à une position donnée.

#### 4.1.1. Les principales méthodes de Queue

Méthodes	But
Clear	Supprime tous les objets de Queue.
Clone	Crée une copie superficielle de Queue.
Contains	Détermine si un élément est dans Queue.
CopyTo	Copie les éléments Queue dans un Array existant à une dimension commençant au niveau de l'index de tableau spécifié.
Dequeue	Supprime et retourne l'objet au début de Queue.
Enqueue	Ajoute un objet à la fin de Queue.
Equals	Détermine si l'objet Object spécifié est égal à l'objet Object en cours. (Hérité de Object.)
Finalize	Autorise Object à tenter de libérer des ressources et d'exécuter d'autres opérations de nettoyage avant qu'Object soit récupéré par l'opération garbage collection. (Hérité d'Object.)
GetEnumerator	Retourne un énumérateur qui parcourt Queue.
Peek	Retourne l'objet situé au début de Queue sans le supprimer.
Synchronized	Retourne un wrapper Queue qui est synchronisé (thread-safe).
ToArray	Copie les éléments Queue vers un nouveau tableau.
TrimToSize	Définit la capacité au nombre réel d'éléments dans Queue.



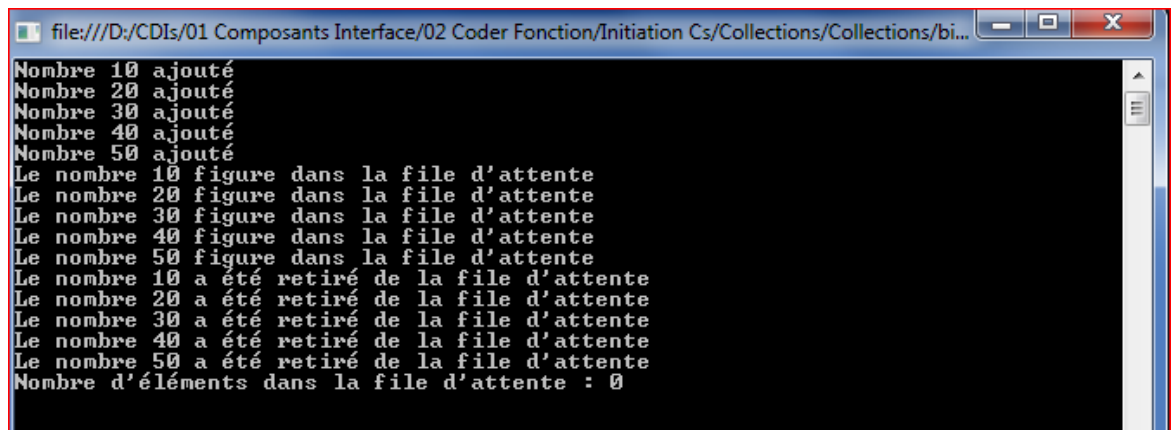
#### 4.1.2. Exemple illustrant l'usage de Queue

```
private static void fileFIFO()
{
    Queue listeFIFO = new Queue();
    int[] tabEntiers = new int[5] { 10, 20, 30, 40, 50 };
    // Remplissage de la file d'attente avec un tableau d'entiers
    foreach (int nombre in tabEntiers)
    {
        listeFIFO.Enqueue(nombre);
        Console.WriteLine("Nombre {0} ajouté", nombre);
    }
    // Parcours de la file d'attente
    foreach (int nombre in listeFIFO)
    {
        Console.WriteLine("Le nombre {0} figure dans la file d'attente", nombre);
    }

    // Vidage de la file d'attente

    while (listeFIFO.Count > 0)
    {
        int nombre = (int)listeFIFO.Dequeue(); // unbox
        Console.WriteLine("Le nombre {0} a été retiré de la file d'attente", nombre);
    }
    Console.WriteLine("Nombre d'éléments dans la file d'attente : {0}", listeFIFO.Count);
    Console.ReadKey();
}
```

Résultat obtenu :



```
file:///D:/CDIs/01 Composants Interface/02 Coder Fonction/Initiation Cs/Collections/Collections/bi...
Nombre 10 ajouté
Nombre 20 ajouté
Nombre 30 ajouté
Nombre 40 ajouté
Nombre 50 ajouté
Le nombre 10 figure dans la file d'attente
Le nombre 20 figure dans la file d'attente
Le nombre 30 figure dans la file d'attente
Le nombre 40 figure dans la file d'attente
Le nombre 50 figure dans la file d'attente
Le nombre 10 a été retiré de la file d'attente
Le nombre 20 a été retiré de la file d'attente
Le nombre 30 a été retiré de la file d'attente
Le nombre 40 a été retiré de la file d'attente
Le nombre 50 a été retiré de la file d'attente
Nombre d'éléments dans la file d'attente : 0
```

## 4.2. La collection Stack

La classe **Stack** implémente un mécanisme de type LIFO (Last In First Out).  
Un élément rejoint le haut de la pile et quitte le haut de la pile.

La collection Stack implémente certaines des méthodes et propriétés communes aux collections et deux méthodes particulières pour insérer et supprimer des éléments dans la file **Push** et **Pop**.

Compte tenu de son fonctionnement, elle n'implémente pas de méthodes qui permettent d'ajouter, d'extraire ou de supprimer un élément à une position donnée.

## 4.3. Exemple illustrant l'usage de Stack

```
private static void FileLIFO()
{
    Stack listeLIFO = new Stack();
    int[] tabEntiers = new int[5] { 10, 20, 30, 40, 50 };

    // Remplissage de la file d'attente avec un tableau d'entiers

    foreach (int nombre in tabEntiers)
    {
        listeLIFO.Push(nombre);
        Console.WriteLine("Le nombre {0} a été empilé sur la file d'attente", nombre);
    }

    // Parcours de la file d'attente

    foreach (int nombre in listeLIFO)
    {
        Console.WriteLine("Le nombre {0} figure dans la file d'attente", nombre);
    }
    // Vidage de la file d'attente
    while (listeLIFO.Count > 0)
    {
        int nombre = (int)listeLIFO.Pop(); // unbox
        Console.WriteLine("Le nombre {0} a été retiré de la pile", nombre);
    }
    Console.WriteLine("Nombre d'éléments dans la file d'attente : {0}", listeLIFO.Count);
    Console.ReadKey();
}
```

```
file:///D:/CDIs/01 Composants Interface/02 Coder Fonction/Initiation Cs/Collections/Collections/bi...
Le nombre 10 a été empilé sur la file d'attente
Le nombre 20 a été empilé sur la file d'attente
Le nombre 30 a été empilé sur la file d'attente
Le nombre 40 a été empilé sur la file d'attente
Le nombre 50 a été empilé sur la file d'attente
Le nombre 50 figure dans la file d'attente
Le nombre 40 figure dans la file d'attente
Le nombre 30 figure dans la file d'attente
Le nombre 20 figure dans la file d'attente
Le nombre 10 figure dans la file d'attente
Le nombre 50 a été retiré de la pile
Le nombre 40 a été retiré de la pile
Le nombre 30 a été retiré de la pile
Le nombre 20 a été retiré de la pile
Le nombre 10 a été retiré de la pile
Nombre d'éléments dans la file d'attente : 0
```

## 5. Les dictionnaires

Vous trouverez à votre disposition dans le Framework un ensemble de collections organisées sous la forme de clés-valeurs désignées sous le vocable de dictionnaires (dictionary).

Ces collections offrent donc un mécanisme d'identification des éléments figurant dans la liste et cette approche n'est pas sans rappeler les mécanismes mis en œuvre dans les bases de données au niveau des tables : chaque ligne de la table possède en effet un identifiant désigné comme clé primaire de la table (Primary Key). Nous aborderons ce point lors de la mise en œuvre des bases de données.

### 5.1.1. La collection HashTable

Les tables HashTable sont aussi connues dans certains langages sous le terme de tableaux associatifs.

Jusqu'à maintenant, nous avons vu que les éléments de nos collections étaient mappés par un entier qui représentait la position de l'élément dans la collection.

Cette fonctionnalité s'avère maintes fois insuffisante et nous préférons mapper les éléments du tableau selon un identifiant qui peut être numérique ou alpha comme dans l'exemple du Salarié (Matricule).

La classe HashTable, et plus généralement toutes les collections de type dictionnaire, vont nous fournir cette fonctionnalité.

Comment ? En fait, la classe HashTable conserve en interne deux tableaux synchronisés : un tableau pour les **clés** à partir desquelles sont mappés les éléments de l'autre tableau constitué des **valeurs**.

Ainsi, nous avons une organisation de nos éléments sous la forme de **paires clé/valeur**.

**A noter :** Les dictionnaires clés/valeurs où clés et valeurs sont des chaînes sont très utilisés dans le contexte du développement Web (ASP, Ajax, JavaScript, JSON Java Script On Notation, ...).

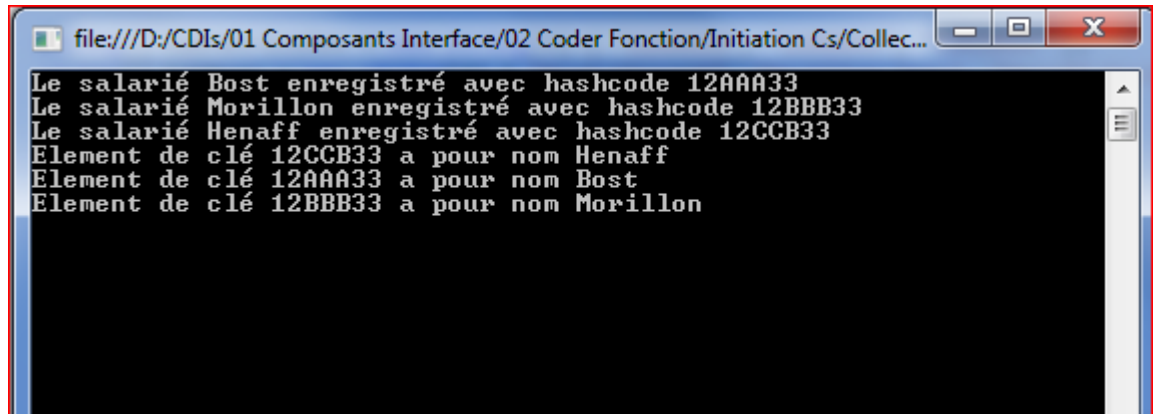
```
private static void TableHachage()
{
    Hashtable tableHach = new Hashtable();

    string[] tabMatricules = new string[3] { "12AAA33", "12BBB33", "12CCB33" };
    string[] tabNoms = new string[3] { "Bost;Vincent", "Morillon;Jean", "Henaff;Marion" };

    // Entrée dans le dictionnaire
    int i = 0;
    |
    foreach (string matricule in tabMatricules)
    {
        string[] nomprenom = tabNoms[i].Split(';');
        Salarie oSalarie = new Salarie(nomprenom[0], nomprenom[1], matricule);
        tableHach.Add(matricule, oSalarie);
        i++;
        Console.WriteLine("Le stagiaire {0} enregistré avec hashcode {1}", oSalarie.Nom, matricule);
    }
    // Parcours du dictionnaire

    // Accès aux clés par le biais de la propriété Key
    // Accès à la valeur par le biais de la propriété Value
    foreach (DictionaryEntry element in tableHach)
    {
        Console.WriteLine("Element de clé {0} a pour nom {1}",
            element.Key, ((Salarie)element.Value).Nom);
    }
    Console.ReadKey();
}
```

Le résultat en image :



```

file:///D:/CDIs/01 Composants Interface/02 Coder Fonction/Initiation Cs/Collec...
Le salarié Bost enregistré avec hashcode 12AAA33
Le salarié Morillon enregistré avec hashcode 12BBB33
Le salarié Henaff enregistré avec hashcode 12CCB33
Element de clé 12CCB33 a pour nom Henaff
Element de clé 12AAA33 a pour nom Bost
Element de clé 12BBB33 a pour nom Morillon
    
```

Quelques particularités :

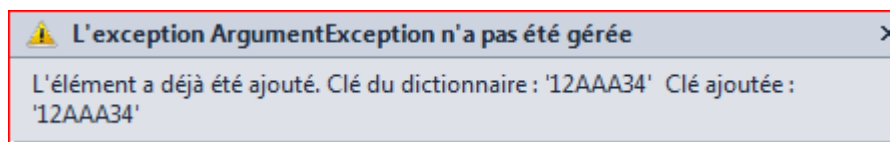
- Lorsque vous utilisez une instruction foreach pour parcourir une HashTable, vous obtenez un élément de la collection de type **DictionaryEntry** qui vous permet d'extraire la valeur **Value** et la clé **Key**.
- Une hashtable ne peut contenir de clés en double et c'est une bonne chose ! Vous devez donc veiller à ce qu'aucun élément disposant d'une clé identique à celle que vous souhaitez associer au nouvel élément ne figure déjà dans la table de hash. Vous avez à votre disposition la méthode **ContainsKey()** pour vérifier cette règle.

Ajout d'un nouvel élément

```

if (!tableHach.ContainsKey("12AAA34"))
    tableHach.Add("12AAA34", new Salarie("Goddaert", "Elisabeth", "12AAA34"));
    
```

Si vous ajoutez un élément dont la clé est dupliquée vous obtiendrez l'exception suivante :



## 5.2.La collection SortedList

Cette collection a un fonctionnement très similaire à celui de la collection de type HashTable.

La principale différence est que le tableau de clés est toujours trié d'où le nom SortedList. Lorsque vous insérez une paire clé/valeur, la clé est insérée à l'indice correct permettant d'obtenir un tableau de clés toujours ordonné.

Comme pour la HashTable, le tableau ne peut contenir de clés dupliquées.

**A noter :**

Sauf à mettre en place votre propre logique applicative, les dictionnaires et collections de paires clés/valeurs ne sont pas sérialisables avec les classes `Serializer` de base fournies par le framework. Nous reviendrons sur ce point lorsque nous aborderons la persistance des objets.

## 6. Les collections de génériques

Les génériques offrent la solution à une limitation des versions antérieures du CLR et du langage C# dans lesquels la généralisation s'effectue par casting des types vers et depuis le type de base universel `Object`.

En créant une classe générique, vous pouvez créer une collection qui est de type sécurisé au moment de la compilation et fortement typée.

Les limitations de l'utilisation de classes de collection non génériques ont été démontrées précédemment lors de l'utilisation de l'`Arraylist` et de la mise en évidence des mécanismes de boxing / unboxing.

Les collections non typées comme `ArrayList` peuvent être d'une utilisation très commode dès lors que vous souhaitez stocker tout type référence ou valeur et donc de disposer de collections hétérogènes.

Mais attention car à l'origine hétéroclite, du grec **ετερόκλητος**, signifie « qui s'écarte des règles de l'art »,)

Principaux reproches faits aux classes non génériques :

- Les performances sont affectées par les mécanismes de conversion valeur-type-valeur ou boxing unboxing. L'effet des conversions boxing et unboxing peut être très significatif lorsque vous parcourez des collections très volumineuses.
- Manque de vérification à la compilation du type stocké dans la collection. Les dégâts peuvent être importants si vous ne testez pas les types avant de les manipuler et si vous autorisez d'y stocker des types non prévus.

### 6.1.Introduction aux génériques

Les classes génériques et les méthodes combinent un niveau de réutilisabilité, de sécurité de type et d'efficacité sans commune mesure avec leurs homologues non génériques.

Vous trouvez un espace de noms, **`System.Collections.Generic`**, qui contient les classes de collection basées sur le type générique.

Les applications doivent utiliser les nouvelles classes de collections génériques plutôt que des équivalents non génériques.

Les génériques introduisent dans le .NET Framework le concept de paramètres de type. Cette technique offre des possibilités très intéressantes en matière de programmation objet. Nous y reviendrons dans le dernier module de la formation lors de la conception d'applications en couches.

Nous allons ici juste nous initier à cette technique en définissant des collections pour nos propres objets qui utilisent un **paramètre T** de type générique et en le remplaçant au moment de la conception par notre type spécifique.

Avec les génériques, vous pourrez garantir le type de l'élément introduit dans une collection et ce, dès la compilation.

Avec les génériques fini le casting permanent et les erreurs liées aux conversions non explicitement codées.

## 6.2. Un exemple pour comprendre

Nous allons implémenter une nouvelle version de l'exemple précédent en remplaçant la collection de type **HashTable** par une collection de type **SortedDictionary** générique avec le type Salarie.

Nous éviterons ainsi les mécanismes de conversion depuis ou vers Object.

### Les particularités :

- **Paramètres de type** sont déclarés au niveau de la définition de la liste ordonnée entre <>. On remplace les valeurs <**TKey**, **TValue**> par les types qui seront respectivement utilisés pour les clés et les éléments rangés dans la collection.
- Lorsque vous utilisez une instruction foreach pour parcourir cette liste ordonnée, vous obtenez un élément de type **KeyValuePair**<Tkey,TValue>.

Vous pouvez tout de suite vérifier l'intérêt de l'utilisation des génériques lors de l'utilisation de clealeur.Key, clealeur.Value.Nom sans conversion de type.

```
private static void TableGeneriqueTrie()
{
    SortedDictionary<String, Salarie> salaries = new SortedDictionary<String, Salarie>();

    string[] tabMatricules = new string[3] { "12AAA33", "12BBB33", "12CCB33" };
    string[] tabNoms = new string[3] { "Bost;Vincent", "Morillon;Jean", "Henaff;Marion" };

    int i = 0;

    foreach (string matricule in tabMatricules)
    {
        string[] nomprenom = tabNoms[i].Split(';');
        salaries.Add(matricule, new Salarie(nomprenom[0], nomprenom[1], matricule));
        i++;
    }

    foreach (KeyValuePair<String, Salarie> element in salaries)
    {
        Console.WriteLine("Element de clé {0} a pour nom {1}",
            element.Key, element.Value.Nom);
    }
    Console.ReadKey();
}
```

Le résultat en image :

```

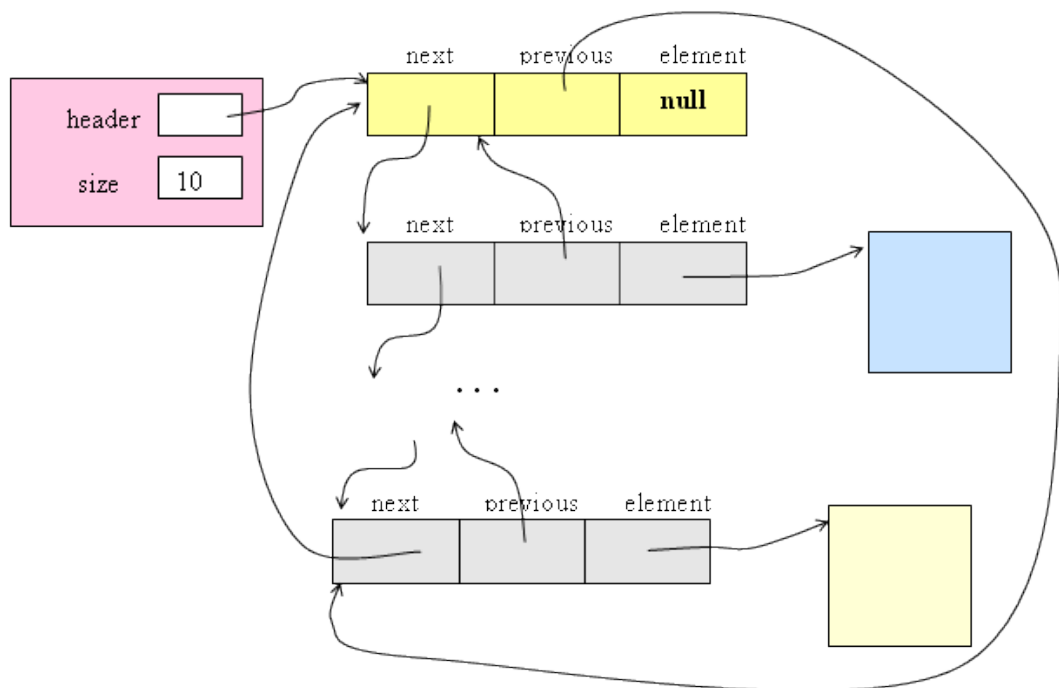
file:///D:/CDIs/01 Composants Interface/02 Coder Fonction/Initiation ...
Element de clé 12AAA33 a pour nom Bost
Element de clé 12BBB33 a pour nom Morillon
Element de clé 12CCB33 a pour nom Henaff
  
```

## 7. Les listes chaînées `LinkedList<T>`

Les listes chaînées sont des listes d'éléments qui, en plus de leurs données, possèdent une référence vers un élément contigu dans la liste. Ces listes sont intéressantes lorsque l'ordre des éléments est important et que ces listes font l'objet de nombreuses insertions ou suppressions. Ces listes existent dans deux formes, simplement chaînées ou doublement chaînées.

En C#, la liste `LinkedList<T>` est celle d'une liste doublement chaînée. Disposant de plus de fonctionnalités, elle requiert toutefois plus de ressources qu'une liste avec chaînage simple. Une liste doublement chaînée permet, à partir d'un élément, de connaître son prédécesseur et son successeur.

Le schéma ci-dessous expose ce principe :



Une liste doublement chaînée dispose de méthodes particulières permettant d'ajouter un élément avant, après, en premier, en dernier, ...

Chaque élément est repéré par un nœud (node).

Petit exemple ci-dessous pour illustrer son usage avec les nombres de la semaine  
La liste est initialisée à partir d'un tableau.

```
static void Main(string[] args)
{
    string[] jours = new string[]{"lundi", "mercredi", "jeudi", "samedi"};

    // La liste est initialisée avec un tableau de mots
    LinkedList<string> listeChaine = new LinkedList<string>(jours);

    // Je souhaite compléter la liste avec les jours manquants
    // Je récupère le nœud qui représente le mercredi pour ajouter le jour mardi manquant
    LinkedListNode<string> courant = listeChaine.Find("mercredi");

    // J'ajoute le jour avant le nœud courant
    listeChaine.AddBefore(courant, "mardi");

    // J'ajoute le dimanche en dernier
    listeChaine.AddLast("dimanche");

    // je vérifie quel jour suit le jour lundi
    Console.WriteLine("Le jour qui suit le lundi est {0}", listeChaine.Find("lundi").Next.Value);
    Console.ReadLine();

    // je vérifie le premier et le dernier jour de la semaine
    Console.WriteLine("Le premier jour est {0} et le dernier {1}", listeChaine.First(), listeChaine.Last());
    Console.ReadLine();
}
```

Le résultat produit est conforme à mon attente.



Si les insertions et suppressions sont très efficaces avec cette forme de liste, les opérations de lecture sont séquentielles (pas de possibilité de se rendre à une position donnée directement) et donc plus longues.