Les **collections** sont une alternative aux tableaux. La différence majeure est que la taille d'un tableau est fixée alors que celle d'une collection peut varier : vous pouvez ajouter et enlever des éléments.

Les collections utilisent la notion de généricité ; je vais donc commencer ce chapitre par un aperçu des génériques. Ensuite, je vous présenterai les listes, les dictionnaires, et les piles. Enfin, je vous parlerai de l'expression foreach ... in.

**Aperçu des génériques**

Certaines méthodes et classes sont **génériques** : elles utilisent des objets dont le type n'est connu qu'à l'exécution. Ce type est par convention représenté par un t majuscule.

**Méthodes génériques**

**Créer une méthode générique**

Il faut rajouter <T> entre le nom de la méthode et la parenthèse ouvrante ; exemple :

static void WriteType<T>(T obj)

{ Console.WriteLine("obj est de type {0}.", typeof(T));}

Il est possible de mettre plusieurs types génériques en les séparant par des virgules ; exemple :

static void WriteType<T, U>(T tObj, U uObj)

{ Console.WriteLine(

"tObj est de type {0} et uObj est de type {1}.",

typeof(T),

typeof(U)); }

**Utiliser une méthode générique**

C'est comme pour un appel normal, mais on rajoute <typeDesParametres> ; exemple :

// Affichera "obj est de type System.Int32."

WriteType<int>(0);

// Affichera "tObj est de type System.Int32 et uObj est de type System.String."

WriteType<int, string>(-1, "fooBarToto");

**Classes génériques**

Vous pouvez avoir des classes génériques. Il faut faire suivre le nom de la classe par <T> ; exemple :

public class GenericList<T>

{ // La classe incluse est aussi générique par rapport à T. // T utilisé pour le type d'un champ.

private class Node

{ private T m\_data;

private Node m\_next;

public Node Next

{

get { return m\_next; }

set { m\_next = value; }

}

// T comme type de retour d'une propriété.

public T Data

{ get { return m\_data; }

et { m\_data = value; } }

// T utilisé dans un constructeur non générique. Une fois la classe créée avec le type T, le constructeur ne peut pas être appelé avec un autre type.

public Node(T t)

{ m\_next = null;

m\_data = t; }

}

private Node m\_head; // Constructeur

public GenericList()

{ m\_head = null; }

public void AddHead(T t) // T comme type de paramètre dans une méthode.

{

Node myNode = new Node(t);

myNode.Next = m\_head;

m\_head = myNode;

}

}

**Les listes**

Je vais vous parler de la classe générique **List<T>** (située dans l'espace de nomsSystem.Collections.Generic). Écrivez juste List<string> dans le Main, faites un clic-droit sur List et cliquez sur "**Go To Definition**" pour voir sa définition. Vous voyez public class List<T> : IList<T>, ICollection<T>, IEnumerable<T>, IList, ICollection, IEnumerable. La classe List implémente notamment l'interfaceICollection<T>, c'est donc une collection.

**Comparaison avec les tableaux**

Une liste est à première vue semblable à un tableau : on y stocke des objets d'un type donné. Il existe cependant des différences significatives entre les deux. D'abord, la taille d'un tableau est fixe alors que la taille d'une liste est variable : vous pouvez ajouter ou enlever des éléments quand vous le souhaitez. De plus, une fois qu'un tableau est créé, vous avez des cases "fixes" ; dans une liste, l'ordre des éléments peut être modifié comme on veut.

**Utiliser une liste**

List est une classe **générique**, donc il faut préciser ce que vous voulez stocker. Vous pouvez aussi préciser au constructeur la taille de la liste ; cela permet d'économiser de l'espace si vous savez combien d'éléments vous voulez stocker. Cela ne vous empêche pas d'en mettre plus, mais agrandir la taille de la liste nécessitera plus de ressources. Voici des exemples d'utilisation d'une liste de chaînes de caractères :

// Crée une liste de chaînes de caractères de taille initiale 3.

List<string> firstNameList = new List<string>(3);

// Affiche la capacité (taille de la liste) ; ici : 3.

Console.WriteLine(firstNameList.Capacity);

// Affiche le nombre d'éléments ; ici : 0.

Console.WriteLine(firstNameList.Count);

// Ajoute des éléments.

firstNameList.Add("Matt");

firstNameList.Add("Tim");

firstNameList.Add("James");

// Supprime des éléments.

// Remove retourne true si l'élément a été supprimé et false sinon.

if (firstNameList.Remove("Tim"))

{

Console.WriteLine("Tim a bien été supprimé de la liste.");

}

else

{

Console.WriteLine("Tim n'a pas été supprimé de la liste.");

}

// Retournera forcément false car Josh n'est pas dans la liste.

if (firstNameList.Remove("Josh"))

{

Console.WriteLine("Josh a bien été supprimé de la liste.");

}

else

{

Console.WriteLine("Josh n'a pas été supprimé de la liste.");

}

// Affichera 2.

Console.WriteLine(firstNameList.Count);

// Supprime tous les éléments.

firstNameList.Clear();

**Les dictionnaires**

Parlons un peu de la classe **Dictionary<TKey, TValue>** située dans l'espace de nomsSystem.Collections.Generic. Un dictionnaire est une collection de paires clef/valeur, chaque clef étant unique. C'est une classe **doublement générique** : il faut préciser le type des clefs et le type des valeurs. Nous allons ici travailler avec des clefs et des valeurs qui seront des chaînes de caractères, mais vous pouvez tout aussi bien choisir d'autres types.

**Utiliser un dictionnaire**

L'utilisation est proche de celle d'une liste, puisque les deux sont des collections. Étudions un exemple de dictionnaire faisant l'association entre une extension de fichiers et le programme utilisé par défaut pour ouvrir les fichiers portant cette extension :

// Création du dictionnaire.

Dictionary<string, string> openWith = new Dictionary<string, string>();

// Ajout de quelques éléments. Il ne peut pas y avoir

// deux clefs identiques mais les valeurs peuvent l'être.

openWith.Add("txt", "notepad.exe");

openWith.Add("bmp", "paint.exe");

openWith.Add("dib", "paint.exe");

openWith.Add("rtf", "wordpad.exe");

// La méthode Add lance une exception si

// la clef existe déjà.

try

{

openWith.Add("txt", "winword.exe");

}

catch (ArgumentException)

{

Console.WriteLine("Un élément possède déjà la clef \"txt\".");

}

// Comme pour un tableau, vous pouvez utiliser un indexeur (avec les crochets : '[' et ']').

// Avec un tableau l'indexeur est un entier positif tandis

// que pour un dictionnaire ce doit être une clef

// (donc ici de type string).

Console.WriteLine(

"La valeur associée à la clef \"rtf\" est {0}.",

openWith["rtf"]);

// L'indexeur peut aussi être utilisé en écriture.

openWith["rtf"] = "winword.exe";

Console.WriteLine(

"La nouvelle valeur associée à la clef \"rtf\" est {0}.",

openWith["rtf"]);

// Si la clef n'existe pas déjà, l'utilisation de l'indexeur

// crée un nouveau couple clef/valeur.

openWith["doc"] = "winword.exe";

// L'indexeur lance une exception si la clef

// n'est pas définie dans le dictionnaire.

try

{

Console.WriteLine(

"La valeur associée à la clef \"tif\" est {0}.",

openWith["tif"]);

}

catch (KeyNotFoundException)

{

Console.WriteLine("La clef \"tif\" est introuvable.");

}

// Si vous voulez accéder à plusieurs clefs, sans savoir si elles existent,

// le plus rapide est d'utiliser la méthode TryGetValue plutôt

// que d'utiliser des try ... catch.

// Si la valeur a pu être récupérée, elle est mise dans le paramètre value passé avec out

// et la méthode retourne true ; sinon elle retourne false.

string value = string.Empty;

if (openWith.TryGetValue("tif", out value))

{

Console.WriteLine(

"La valeur associée à la clef \"tif\" est {0}.",

value);

}

else

{

Console.WriteLine("La clef \"tif\" est introuvable.");

}

// La méthode ContainsKey permet de savoir si la clef existe déjà ou non.

// Elle retourne true si la clef existe déjà et false sinon.

if (!openWith.ContainsKey("ht"))

{

openWith.Add("ht", "hypertrm.exe");

Console.WriteLine(

"Ajout de la clef \"ht\" avec la valeur {0}.",

openWith["ht"]);

}

// La méthode Remove permet de supprimer une paire clef/valeur.

Console.WriteLine("Suppression de la clef \"doc\" et de la valeur associée.");

openWith.Remove("doc");

if (!openWith.ContainsKey("doc"))

{

Console.WriteLine("La clef \"doc\" est introuvable.");

}

// Count indique le nombre de paires stockées.

Console.WriteLine(

"Ce dictionnaire contient {0} paires.",

openWith.Count);

// La méthode Clear permet de supprimer toutes les paires clef/valeur.

openWith.Clear();

Console.WriteLine(

"Après suppression de toutes les paires, ce dictionnaire contient {0} paires.",

openWith.Count);

**Les piles**

Intéressons-nous maintenant à la classe générique **System.Collections.Generic.Stack<T>**. Une **pile**("**stack**" en anglais) est une collection de type dernier entré premier sorti ("LIFO" en anlgais).

**Fonctionnement**

Concrètement, il s'agit d'une sorte de liste que vous remplissez au fur et à mesure et dont seul le dernier élément inséré est accessible. Il n'y a donc pas d'indexeur ni de méthode Remove. Vous pouvez par contre utiliser la méthode Clear ou la propriété Count. À chaque fois que vous ajoutez un élément, il se place avant tous les autres. Certes, vous ne pouvez accéder qu'à l'élément se trouvant au sommet de la pile, c'est-à-dire à celui qui a été ajouté en dernier ; toutefois, vous pouvez le supprimer pour accéder à l'élément du dessous.

**En pratique**

Voici un exemple dans lequel j'utilise une pile d'entiers :

// Création d'une pile d'entiers.

Stack<int> numbers = new Stack<int>();

// La méthode Push sert à ajouter un élément au sommet de la pile.

numbers.Push(0);

numbers.Push(1);

numbers.Push(2);

numbers.Push(3);

numbers.Push(4);

Console.WriteLine(

"{0} éléments ont été ajoutés.",

numbers.Count);

// La méthode Pop sert à enlever le dernier élément de la pile.

// Elle retourne l'élément enlevé.

Console.WriteLine(

"Le dernier élément ({0}) a été enlevé.",

numbers.Pop());

// La méthode Peek sert à récupérer le dernier élément de la pile, sans le supprimer.

Console.WriteLine(

"Le dernier élément est actuellement {0}.",

numbers.Peek());

**foreach ... in**

**Présentation**

Le mot-clef foreach permet de parcourir des objets implémentant l'interface **IEnumerable**. La syntaxe est :

foreach (typeDeLélément élément in collectionDéléments)

{

// Code utilisant l'élément.

}

Comme dans une boucle for, vous pouvez utiliser break; pour sortir de la boucle ou continue; pour passer à l'élément suivant.

**Exemples**

Avec une liste

Avec la liste créée précédemment, le code suivant...

foreach (string firstName in firstNameList)

{

Console.WriteLine(firstName);

}

...affichera (Tim a été supprimé) : Matt James

**Avec un dictionnaire**

Il y a plusieurs façons de faire, suivant vos besoins. Un dictionnaire possède deux propriétés intéressantes :

* **Keys** qui est de type Dictionary<TKey, TValue>.KeyCollection : c'est une collection qui contient les clefs du dictionnaire.
* **Values** qui est de type Dictionary<TKey, TValue>.ValueCollection : c'est une collection qui contient les valeurs du dictionnaire.

Dans le définition de Dictionary, on voit : Dictionary<TKey, TValue> : IDictionary<TKey, TValue>, ICollection<KeyValuePair<TKey, TValue>>, IEnumerable<KeyValuePair<TKey, TValue>>, IDictionary, ICollection, IEnumerable, ISerializable, IDeserializationCallback. En fait, quand je dis qu'un dictionnaire est une collection de paires clefs/valeur, il faudrait préciser : c'est une collection d'objets de type **KeyValuePair<TKey, TValue>**. Pour que vous compreniez, regardez cet exemple (où openWith est le dictionnaire précédemment créé) :

Console.WriteLine("Liste des clefs :");

// On récupère juste les clefs.

Dictionary<string, string>.KeyCollection myKeyCollection = openWith.Keys;

// On parcourt les clefs (qui sont des objets de type string).

foreach (string key in myKeyCollection)

{

Console.WriteLine(key);

}

// Le \n sert à faire un retour charriot

// (donc à sauter une ligne car on était déjà revenus à la ligne avec WriteLine)

Console.WriteLine("\nListe des valeurs :");

// On récupère juste les valeurs.

Dictionary<string, string>.ValueCollection myValueCollection = openWith.Values;

// On parcourt les valeurs (qui sont des objets de type string)

foreach (string value in myValueCollection)

{

Console.WriteLine(value);

}

Console.WriteLine("\nListe des paires clef/valeur :");

// Quand on utilise foreach pour énumérer les éléments du dictionnaire,

// ces éléments sont récupérés en tant que des objets de type KeyValuePair.

foreach (KeyValuePair<string, string> kvp in openWith)

{

Console.WriteLine(

"openWith[\"{0}\"] vaut {1}",

kvp.Key,

kvp.Value);

}

Le résultat est :

Liste des clefs :

txt

bmp

dib

rtf

Liste des valeurs :

notepad.exe

paint.exe

paint.exe

wordpad.exe

Liste des paires clef/valeur :

openWith["txt"] vaut notepad.exe

openWith["bmp"] vaut paint.exe

openWith["dib"] vaut paint.exe

openWith["rtf"] vaut wordpad.exe

**Avec une pile**

C'est comme avec une liste, sauf que la pile est parcourue de l'élément le plus récent au plus vieux. L'ordre est donc inversé et le code suivant (où numbers est la pile précédemment créée)...

foreach (int i in numbers)

{

Console.WriteLine(i);

}

...affichera :

4

3

2

1

0

Vous disposez maintenant d'une alternative aux tableaux. Faites-en bon usage !

J'espère d'une part vous avoir apporté suffisamment de concret pour éclaircir vos idées, et d'autre part j'espère vous avoir donné envie de continuer la lecture de ce tutoriel.

Dans la prochaine partie, nous nous attaquerons à la GUI ("Interface Utilisateur Graphique") avec WPF. Nous allons donc changer complètement de domaine, alors n'en profitez pas pour oublier ce que je vous ai dit dans cette partie !

Ce tutoriel est loin d'être fini. Il me faudra du temps pour le terminer !

Prochaine partie prévue : "**Créer des applications riches en design avec WPF**". Vous y découvrirez la création moderne et rapide d'interfaces graphiques époustouflantes !

**Exemple**

using System;

using System.Collections.Generic;

using System.Collections.ObjectModel;

public class Demo

{

public static void Main()

{

Collection<string> dinosaurs = new Collection<string>();

dinosaurs.Add("Psitticosaurus");

dinosaurs.Add("Caudipteryx");

dinosaurs.Add("Compsognathus");

dinosaurs.Add("Muttaburrasaurus");

Console.WriteLine("{0} dinosaurs:", dinosaurs.Count);

Display(dinosaurs);

Console.WriteLine("\nIndexOf(\"Muttaburrasaurus\"): {0}",

dinosaurs.IndexOf("Muttaburrasaurus"));

Console.WriteLine("\nContains(\"Caudipteryx\"): {0}",

dinosaurs.Contains("Caudipteryx"));

Console.WriteLine("\nInsert(2, \"Nanotyrannus\")");

dinosaurs.Insert(2, "Nanotyrannus");

Display(dinosaurs);

Console.WriteLine("\ndinosaurs[2]: {0}", dinosaurs[2]);

Console.WriteLine("\ndinosaurs[2] = \"Microraptor\"");

dinosaurs[2] = "Microraptor";

Display(dinosaurs);

Console.WriteLine("\nRemove(\"Microraptor\")");

dinosaurs.Remove("Microraptor");

Display(dinosaurs);

Console.WriteLine("\nRemoveAt(0)");

dinosaurs.RemoveAt(0);

Display(dinosaurs);

Console.WriteLine("\ndinosaurs.Clear()");

dinosaurs.Clear();

Console.WriteLine("Count: {0}", dinosaurs.Count);

}

private static void Display(Collection<string> cs)

{

Console.WriteLine();

foreach( string item in cs )

{

Console.WriteLine(item);

}

}

}

/\* This code example produces the following output:

4 dinosaurs:

Psitticosaurus

Caudipteryx

Compsognathus

Muttaburrasaurus

IndexOf("Muttaburrasaurus"): 3

Contains("Caudipteryx"): True

Insert(2, "Nanotyrannus")

Psitticosaurus

Caudipteryx

Nanotyrannus

Compsognathus

Muttaburrasaurus

dinosaurs[2]: Nanotyrannus

dinosaurs[2] = "Microraptor"

Psitticosaurus

Caudipteryx

Microraptor

Compsognathus

Muttaburrasaurus

Remove("Microraptor")

Psitticosaurus

Caudipteryx

Compsognathus

Muttaburrasaurus

RemoveAt(0)

Caudipteryx

Compsognathus

Muttaburrasaurus

dinosaurs.Clear()

Count: 0

\*/