

# Au coeur des dictionnaires en .Net 2.0

par Mehdi FEKIH

Date de publication : 10/09/2007

Dernière mise à jour : 10/09/2007

Cet article présente une synthèse des dictionnaires du Framework .Net 2.0

- I - Introduction
- II - Présentation générale
- III - La Complexité des algorithmes
- IV - Les dictionnaires dans le Framework 2.0
  - IV-A - Introduction aux dictionnaires
  - IV-B - Dictionary
  - IV-C - SortedDictionary
  - IV-D - OrderedDictionary
  - IV-E - ListDictionary Vs Hashtable Vs HybridDictionary
  - IV-F - StringDictionary
  - IV-G - NameValueCollection
  - IV-H - KeyedCollection
- V - Conclusion
- Ressource
- Remerciments

## I - Introduction

Le Framework 2.0 compte parmi ses classes une grande panoplie de collections aussi diverses l'une de l'autre et conçues pour un besoin spécifique. Les développeurs .Net se trouvent fréquemment dans une situation où ils ne savent pas laquelle des collections choisir et se tournent généralement vers la plus basique d'entre elles pour arriver à leurs fins. Ce choix rapide et maladroit entraîne souvent le ralentissement de l'algorithme à implémenter et aussi des effets de bords incompréhensibles et difficiles à cerner si on ne s'aperçoit pas que la collection choisie est en fait la cause des malheurs.

Le but de cet article, est d'assister le développeur à choisir le bon dictionnaire selon le cas de figure dans lequel il se trouve. Je vais traiter tout au long de cet article les avantages et les limites de chacun des dictionnaires du Framework tout en me focalisant sur les méthodes les plus intéressantes qu'offrent chacune des classes.

## II - Présentation générale

Du point de vue organisationnel, les collections sont groupées en 3 namespaces :

**System.Collections** : ce namespace regroupe un ensemble de collections non typées permettant de stocker des données hétérogènes dans la même structure dynamique puisque la signature des méthodes accepte le type object.

**System.Collections.Generic** : La grande nouveauté du Framework 2.0, les generics reprennent essentiellement les collections du premier namespace en prenant en charge des données typées.

**System.Collections.Specialized** : comme son nom l'indique, ce namespace regroupe un ensemble de collections spécialisées qui répondent à un besoin spécifique pour un type de donnée spécifique. Contrairement aux collections du namespace System.Collections, celles du namespace Specialized sont fortement typées.

Les collections qui seront traitées dans cet article sont les suivantes :

Namespaces & Classes
<b>System.Collections</b>
Hashtable
<b>System.Collections.Specialized</b>
OrderedDictionary
StringDictionary
NameValueCollection
ListDictionary
HybridDictionary
<b>System.Collections.Generic</b>
Dictionary
SortedDictionary
<b>System.Collection.ObjectModel</b>
KeyedCollection

Du point de vue logique, je regrouperais les collections du Framework en 5 classes. Bien entendu une collection du Framework peut appartenir à plusieurs groupes.

**Les collections basiques** : Ce sont les collections à comportement basique. Elles permettent de stocker des données dynamiquement dans une structure de données en se limitant à des fonctionnalités de base telles que l'ajout, la suppression, l'énumération, etc.

**Les listes séquentielles** : Ces collections suivent une certaine logique d'ordre lors de l'insertion des données. Il existe deux sortes de collections séquentielles dans le Framework 2.0 ; la pile (LIFO : Last In First Out) et la file (FIFO : First In First Out).

**Les dictionnaires** : Ce sont des structures dynamiques qui permettent d'emmagasiner des données sous forme de clé/valeur afin de rendre instantanée la récupération d'une valeur sachant sa clé.

**Les collections spécialisées** : Ce sont des structures spéciales qui représentent des comportements spécifiques applicables généralement à un type donné. Par exemple une structure qui représente une suite binaire ou une structure qui n'accepte que des types strings.

**Les generics** : C'est l'ensemble des collections fortement typées. Elles font parties systématiquement du namespace **System.Collections.Generic**.

### III - La Complexité des algorithmes

Afin de comparer l'efficacité des différentes collections du Framework, nous aurons besoin d'une norme qui permettrait d'évaluer le coût d'exécution des opérations offertes par les collections. La théorie de la complexité algorithmique permet justement d'évaluer la rapidité d'exécution des tâches algorithmiques (méthodes de classes). Tout au long de cet article la complexité dans le pire des cas a été adoptée. Elle permet en fait d'évaluer les algorithmes dans le cas où les paramètres d'entrées engendrent le maximum d'itérations pour converger vers une solution.

Prenons l'exemple d'une liste chaînée représentée par la structure suivante :

5	4	42	2	8	36	9	47
---	---	----	---	---	----	---	----

Dans le meilleur des cas, l'algorithme de la méthode GetAt, qui permet de récupérer une valeur à partir de son index, converge en 1 itération. Ce cas est représenté par l'appel GetAt(0). On dit que le l'algorithme dans le meilleur des cas est en  $O(1)$ .

Dans le pire des cas, l'algorithme de la méthode GetAt converge en 8 itérations. Ce cas est représenté par l'appel GetAt(7). Pour généraliser, on dit que l'algorithme dans le pire des cas est en  $O(n)$ ,  $n$  représente la taille des entrées (taille du tableau dans notre cas).

La théorie de la complexité algorithmique utilise la notation Landau ( $O$  : grand  $o$ ) pour représenter l'efficacité. En faisant abstraction des explications mathématiques, voici les différents types de complexités que l'on va rencontrer dans cet article classés du plus rapide au plus lent :

**$O(1)$**  : complexité constante indépendante des paramètres d'entrées.

**$O(\log(n))$**  : complexité logarithmique, le temps d'exécution croît légèrement par rapport à la taille des entrées.

**$O(n)$**  : complexité proportionnelle à la longueur de l'entrée.


## IV - Les dictionnaires dans le Framework 2.0

### IV-A - Introduction aux dictionnaires

Les dictionnaires représentent des structures de données particulières qui permettent d'associer chaque valeur à une clé. La particularité de ce mapping est que la recherche d'une valeur par sa clé est presque instantanée et converge vers une complexité en  $O(1)$  même si dans certains cas elle peut monter jusqu'en  $O(n)$  si la fonction de hachage est mal implémentée. Cette rapidité de recherche est possible grâce aux tables de hachages sur lesquelles reposent les dictionnaires. La question qui se pose donc est : comment cette structure permet de retrouver les valeurs quasi instantanément.

En fait, une table de hachages permet d'associer à chaque clé, quelque soit son type, un entier calculé grâce à une fonction. La valeur entière représente généralement l'index de la clé dans la table et la fonction en question est appelée la fonction de hachage.

Concrètement, dans le Framework 2.0, la méthode `GetHashCode` représente la fonction de hachage. Le Framework dispose de l'implémentation de cette fonction pour quelques types comme le `string`. L'utilisation de `string` comme clé assure donc la performance de recherche dans la table de hachage puisque le Framework assure l'unicité de la valeur de hachage pour chaque combinaison de caractères stockée dans un `string`. Cette unicité empêche le cas où deux clés retournent le même code de hachage de se produire. Ce phénomène est appelé collision et est responsable de la perte de performance de la table de hachage. La méthode `GetHashCode` n'est pas tout le temps implémentée au niveau du Framework, certainement pas dans le cas où on veut utiliser des classes personnalisées comme clé. Dans ce cas, le développeur prend en charge l'implémentation de la fonction de hachage en surchargeant la méthode `GetHashCode` et en implémentant l'interface `IEquatable`. Ce sujet sera traité plus en détail dans la suite de l'article.

 **note msdn** : Par exemple, l'implémentation de la méthode `GetHashCode` fournie par la classe `String` retourne des codes de hachage uniques pour des valeurs de chaîne uniques. Par conséquent, deux objets `String` retournent le même code de hachage s'ils représentent la même valeur de chaîne.

Prenons l'exemple du cas où on veut stocker dans un dictionnaire les noms des pilotes de formule 1 à qui on veut associer leurs totaux de points au cours de la saison.

```
Dictionary<string, int> lesPilotes = new Dictionary<string, int>(2);

lesPilotes.Add("Lewis Hamilton", 70);
lesPilotes.Add("Fernando Alonso", 68);

foreach (string piloteKey in lesPilotes.Keys)
{
    //afficher les clés et leurs codes de hachage
    Console.WriteLine("Clé = {0}, HashCode = {1}",
        piloteKey, piloteKey.GetHashCode());
}

//Clé = Lewis Hamilton, HashCode = 710750954
//Clé = Fernando Alonso, HashCode = 477124616
```

Nous remarquons que les deux clés retournent des codes de hachages différents qui serviront à retrouver directement le score correspondant au nom du pilote.

### IV-B - Dictionary

La classe Dictionary et la plus basique des génériques dans le contexte des structures en clé/valeur. Elle correspond à la classe hashtable (table de hachage) des collections de base. Nous allons voir dans ce qui suit les possibilités d'insertion et de suppression dans la classe Dictionary et son comportement après ces opérations.

### Quand faut-il choisir la classe Dictionary ?

- 1 Collection typée qui stocke des listes de **clé/valeur** ; l'identification des clés similaires se fait en implémentant, la classe qui représente la clé, l'interface **IEquatable**. Il est possible aussi de passer une classe qui implémente **IEqualityComparer** lors de la construction de la collection.
- 2 N'autorise pas les **clés doublons**.
- 3 **Rapide** : accès et suppression en **O(1)**

### Manipulation d'un dictionnaire

```
//Initialiser la taille pour gagner en performance.
Dictionary<string, string> lesPilotes = new Dictionary<string, string>(10);

//Initialiser le dictionnaire
lesPilotes.Add("L. Hamilton", " McLaren Mercedes");
lesPilotes.Add("F. Alonso", " McLaren Mercedes");
lesPilotes.Add("F. Massa", "Ferrari");
lesPilotes.Add("K. Räikkönen", "Ferrari");
lesPilotes.Add("N. Heidfeld", "BMW Sauber");
lesPilotes.Add("R. Kubica", "BMW Sauber");
lesPilotes.Add("G. Fisichella", "Renault");
lesPilotes.Add("H. Kovalainen", "Renault");
lesPilotes.Add("A. Wurz", "Williams");
lesPilotes.Add("M. Webber", "Red Bull");

//lesPilotes.

Afficher(lesPilotes);
// 1 - Pilote : L. Hamilton , Ecurie : McLaren Mercedes
// 2 - Pilote : F. Alonso , Ecurie : McLaren Mercedes
// 3 - Pilote : F. Massa , Ecurie : Ferrari
// 4 - Pilote : K. Räikkönen , Ecurie : Ferrari
// 5 - Pilote : N. Heidfeld , Ecurie : BMW Sauber
// 6 - Pilote : R. Kubica , Ecurie : BMW Sauber
// 7 - Pilote : G. Fisichella , Ecurie : Renault
// 8 - Pilote : H. Kovalainen , Ecurie : Renault
// 9 - Pilote : A. Wurz , Ecurie : Williams
// 10 - Pilote : M. Webber , Ecurie : Red Bull

//Exception "ArgumentNullException" levée lors de l'ajout d'une clé nulle
try
{
    lesPilotes.Add(null, "Ferrari");
}
catch (ArgumentNullException ex)
{
    Console.WriteLine(System.Environment.NewLine + ex.Message);
    //Value cannot be null.
    //Parameter name: key
}

//l'ajout d'une valeur null autorisé
lesPilotes.Add("M. Fekih", null);

//Exception "ArgumentException" levée lors de l'ajout d'une clé existante
try
{
    lesPilotes.Add("L. Hamilton", " McLaren Mercedes");
}
catch (ArgumentException ex)
{
}
```

## Manipulation d'un dictionnaire

```
Console.WriteLine(System.Environment.NewLine + ex.Message + System.Environment.NewLine);
//An item with the same key has already been added.
}

//Supprimer des entrées
lesPilotes.Remove("K. Räikkönen");
lesPilotes.Remove("G. Fisichella");

//Afficher des pilotes
Afficher(lesPilotes);

// 1 - Pilote : L. Hamilton , Ecurie : McLaren Mercedes
// 2 - Pilote : F. Alonso , Ecurie : McLaren Mercedes
// 3 - Pilote : F. Massa , Ecurie : Ferrari
// 4 - Pilote : N. Heidfeld , Ecurie : BMW Sauber
// 5 - Pilote : R. Kubica , Ecurie : BMW Sauber
// 6 - Pilote : H. Kovalainen , Ecurie : Renault
// 7 - Pilote : A. Wurz , Ecurie : Williams
// 8 - Pilote : M. Webber , Ecurie : Red Bull
// 9 - Pilote : M. Fekih , Ecurie :
```

Dans l'exemple précédent, des strings ont été utilisé comme clé. Le Framework reconnait deux strings identiques grâce à l'implémentation de GetHashCode. Dans ce qui suit nous allons voir comment utiliser une classe personnalisée ( custom class ) comme clé.

Tout d'abord, il faut comprendre comment le dictionnaire du Framework .Net fonctionne pour reconnaitre deux clés identiques. Le principe reste le même, le dictionnaire va calculer la valeur de hachage de la classe et va essayer de retrouver la même valeur dans les clés déjà utilisées dans la table de hachage. Si le dictionnaire ne trouve aucune clé alors il va pouvoir insérer la valeur sans se soucier de la redondance. Dans l'autre cas où le dictionnaire retrouve la même valeur de hachage. Il va vérifier si les deux instances de la classe qui représente la clé sont égaux. Pour cela, il va donc utiliser la méthode Equals.

On va donc dans le code suivant changer la définition de notre classe Pilote en implémentant la classe générique IEquatable et en surchargeant la méthode GetHashCode.

## L'interface IEquatable

```
public class Pilote
{
    private string nom;

    public string Nom
    {
        get { return nom; }
        set { nom = value; }
    }

    private string prenom;

    public string Prenom
    {
        get { return prenom; }
        set { prenom = value; }
    }

    public Pilote(string prenom, string nom)
    {
        this.Nom = nom;
        this.Prenom = prenom;
    }
}
```

## L'interface IEquatable

```
public override string ToString()
{
    return string.Concat(this.Prenom, " ", this.Nom);
}

public class PiloteEquatable : Pilote, IEquatable<PiloteEquatable>
{
    public PiloteEquatable(string prenom, string nom)
        : base(prenom, nom)
    {
    }

    public override int GetHashCode()
    {
        Trace.WriteLine(" GetHashCode appelé ");
        return string.Concat(this.Prenom, " ", this.Nom).GetHashCode();
    }

    public bool Equals(PiloteEquatable other)
    {
        Trace.WriteLine(" Equals Appelé");
        return this.Nom.Equals(other.Nom) && this.Prenom.Equals(other.Prenom);
    }

    public override bool Equals(object obj)
    {
        return Equals((PiloteEquatable)obj);
    }
}

public static class DictionaryDemo
{
    public static void ExecuteIEquatable()
    {
        Dictionary<Pilote, string> lesPilotes = new Dictionary<Pilote, string>(3);

        //Essayer d'insérer deux pilotes identiques sans l'implémentation de IEquatable
        lesPilotes.Add(new Pilote("Lewis", "Hamilton"), " McLaren Mercedes");
        lesPilotes.Add(new Pilote("Fernando", "Alonso"), " McLaren Mercedes");
        lesPilotes.Add(new Pilote("Lewis", "Hamilton"), " McLaren Mercedes");

        //1 - Pilote : Lewis Hamilton , Ecurie : McLaren Mercedes
        //2 - Pilote : Fernando Alonso , Ecurie : McLaren Mercedes
        //3 - Pilote : Lewis Hamilton , Ecurie : McLaren Mercedes

        //ça fonctionne ; le dictionnaire ne détecte pas le même pilote
        Afficher(lesPilotes);

        Dictionary<PiloteEquatable, string> lesPilotesEquatable = new Dictionary<PiloteEquatable,
string>(5);

        lesPilotesEquatable.Add(new PiloteEquatable("Lewis", "Hamilton"), " McLaren Mercedes");
        lesPilotesEquatable.Add(new PiloteEquatable("Fernando", "Alonso"), " McLaren Mercedes");

        try
        {
            lesPilotesEquatable.Add(new PiloteEquatable("Lewis", "Hamilton"), " McLaren Mercedes");
        }
        catch (ArgumentException ex)
        {
            Console.WriteLine(System.Environment.NewLine + ex.Message);

            //Ajouter une nouvelle fois Lewis Hamilton :
            //GetHashCode appelé
        }
    }
}
```

### L'interface IEquatable

```

//Equals Appelé

//Le dictionnaire detecte deux hashcode identiques => GetHashCode appelé
//Il va donc comparer les instaces grâce à Equals => Equals appelé

//An item with the same key has already been added.
}

Console.WriteLine(lesPilotesEquatable.ContainsKey(new PiloteEquatable("Fernando",
"Alonso"))); // True

lesPilotesEquatable.Remove(new PiloteEquatable("Fernando", "Alonso"));
}
    
```

## IV-C - SortedDictionary

La SortedDictionary est une implémentation particulière de la classe Dictionary précédente. Elle a la particularité d'ordonner les clés de sa table selon une logique définie par l'utilisateur. Cette prise en charge de l'ordonnement influe sur les performances de la collection.

Quand faut-il choisir la classe SortedDictionary ?	
1	Collection <b>typée</b> qui stocke des listes de <b>clé/valeur ordonnées par les clés</b> ; la classe représentant la clé doit implémenter l'interface <b>IComparable</b> . Dans le cas échéant, l'utilisateur peut passer une classe qui implémente IComparer lors de la construction de la collection.
2	N'autorise pas les <b>clés doublons</b> .
3	<b>Plus lente que la Dictionary</b> : Insertion et suppression en <b>O(log(n))</b>

Pour la détection des clés similaires, la SortedDictionary se base sur la méthode CompareTo, membre de l'interface IComparable. Cette méthode est appelée systématiquement lors de l'ajout, suppression et recherche de clés, d'où la perte de performance constatée par rapport à la classe Dictionary.

Par exemple, si on utilise un type primitif du Framework tel que le string, la collection va automatiquement ordonner les éléments selon l'ordre alphabétique.

### Manipulation simple d'une SortedDictionary

```

SortedDictionary<string,string> lesPilotes=new SortedDictionary<string,string>();

lesPilotes.Add("L. Hamilton", "McLaren Mercedes");
lesPilotes.Add("F. Alonso", "McLaren Mercedes");
lesPilotes.Add("F. Massa", "Ferrari");
lesPilotes.Add("K. Räikkönen", "Ferrari");
lesPilotes.Add("N. Heidfeld", "BMW Sauber");
lesPilotes.Add("R. Kubica", "BMW Sauber");
lesPilotes.Add("G. Fisichella", "Renault");
lesPilotes.Add("H. Kovalainen", "Renault");
lesPilotes.Add("A. Wurz", "Williams");
lesPilotes.Add("M. Webber", "Red Bull");

Afficher(lesPilotes);
//1 - Pilote : A. Wurz , Ecurie : Williams
//2 - Pilote : F. Alonso , Ecurie : McLaren Mercedes
//3 - Pilote : F. Massa , Ecurie : Ferrari
//4 - Pilote : G. Fisichella , Ecurie : Renault
//5 - Pilote : H. Kovalainen , Ecurie : Renault
    
```

## Manipulation simple d'une SortedDictionary

```
//6 - Pilote : K. Räikkönen , Ecurie : Ferrari
//7 - Pilote : L. Hamilton , Ecurie : McLaren Mercedes
//8 - Pilote : M. Webber , Ecurie : Red Bull
//9 - Pilote : N. Heidfeld , Ecurie : BMW Sauber
//10 - Pilote : R. Kubica , Ecurie : BMW Sauber
}

public static void ExecuteIEquatable()
{
    SortedDictionary<PiloteComparable, string> lesPilotes = new SortedDictionary<PiloteComparable,
    string>();

    //Dans le cas d'une classe personnalisée, il faut impérativement implementer IComparable
    lesPilotes.Add(new PiloteComparable("Lewis", "Hamilton"), "McLaren Mercedes");

    //la duplication est detectée par CompareTo et non pas par Equals et GetHashCode
    try
    {
        lesPilotes.Add(new PiloteComparable("Lewis", "Hamilton"), "McLaren Mercedes");
    }
    catch (ArgumentException ex)
    {
        Console.WriteLine(System.Environment.NewLine + ex.Message + System.Environment.NewLine);
        //An entry with the same key already exists.
    }

    lesPilotes.Add(new PiloteComparable("Fernando", "Alonso"), "McLaren Mercedes");
    lesPilotes.Add(new PiloteComparable("Felipe", "Massa"), "Ferrari");
    lesPilotes.Add(new PiloteComparable("Kimi", "Räikkönen"), "Ferrari");
    lesPilotes.Add(new PiloteComparable("Nick", "Heidfeld"), "BMW Sauber");
    lesPilotes.Add(new PiloteComparable("Robert", "Kubica"), "BMW Sauber");
    lesPilotes.Add(new PiloteComparable("Giancarlo", "Fisichella"), "Renault");
    lesPilotes.Add(new PiloteComparable("Heikki", "Kovalainen"), "Renault");
    lesPilotes.Add(new PiloteComparable("Alexander", "Wurz"), "Williams");
    lesPilotes.Add(new PiloteComparable("Mark", "Webber"), "Red Bull");

    Afficher(lesPilotes);
    //1 - Pilote : Alexander Wurz , Ecurie : Williams
    //2 - Pilote : Felipe Massa , Ecurie : Ferrari
    //3 - Pilote : Fernando Alonso , Ecurie : McLaren Mercedes
    //4 - Pilote : Giancarlo Fisichella , Ecurie : Renault
    //5 - Pilote : Heikki Kovalainen , Ecurie : Renault
    //6 - Pilote : Kimi Räikkönen , Ecurie : Ferrari
    //7 - Pilote : Lewis Hamilton , Ecurie : McLaren Mercedes
    //8 - Pilote : Mark Webber , Ecurie : Red Bull
    //9 - Pilote : Nick Heidfeld , Ecurie : BMW Sauber
    //10 - Pilote : Robert Kubica , Ecurie : BMW Sauber

    //ContainsKey utilise que CompareTo
    Console.WriteLine(lesPilotes.ContainsKey(new PiloteComparable("Robert", "Kubica")));
    //True

    //Remove utilise que CompareTo
    lesPilotes.Remove(new PiloteComparable("Giancarlo", "Fisichella"));

    Console.WriteLine(Environment.NewLine + lesPilotes[new PiloteComparable("Mark", "Webber")]);
    //Red Bull
}
```

L'avantage avec l'exemple précédent est que le Framework sait comment comparer deux string et donc situer une chaîne de caractères par rapport à une autre. Ce n'est pas le cas avec une classe personnalisée. Pour remédier à ce problème et "apprendre" au Framework comment comparer deux instances d'une classe personnalisée, il faut dans ce cas implémenter l'interface générique IComparable et par la suite redéfinir la méthode CompareTo.

Voici une implémentation de IComparable de notre classe de test Pilote :

#### L'interface générique IComparable

```
public class PiloteComparable : Pilote, IComparable<PiloteComparable>
{
    public PiloteComparable(string prenom, string nom)
        : base(prenom, nom)
    {
    }

    public int CompareTo(PiloteComparable other)
    {
        Trace.WriteLine("CompareTo appelé");

        if (this.Prenom.CompareTo(other.Prenom) == 0)
            return this.Nom.CompareTo(other.Nom);
        else return this.Prenom.CompareTo(other.Prenom);
    }
}
```

Maintenant, on va utiliser la classe PiloteComparable comme clé de notre collection SortedDictionary :

#### Manipulation d'une SortedDictionary

```
SortedDictionary<PiloteComparable, string> lesPilotes = new SortedDictionary<PiloteComparable,
string>();

//Dans le cas d'une classe personnalisée, il faut impérativement implémenter IComparable
lesPilotes.Add(new PiloteComparable("Lewis", "Hamilton"), "McLaren Mercedes");

//la duplication est détectée par CompareTo et non pas par Equals et GetHashCode
try
{
    lesPilotes.Add(new PiloteComparable("Lewis", "Hamilton"), "McLaren Mercedes");
}
catch (ArgumentException ex)
{
    Console.WriteLine(System.Environment.NewLine + ex.Message + System.Environment.NewLine);
    //An entry with the same key already exists.
}

lesPilotes.Add(new PiloteComparable("Fernando", "Alonso"), "McLaren Mercedes");
lesPilotes.Add(new PiloteComparable("Felipe", "Massa"), "Ferrari");
lesPilotes.Add(new PiloteComparable("Kimi", "Räikkönen"), "Ferrari");
lesPilotes.Add(new PiloteComparable("Nick", "Heidfeld"), "BMW Sauber");
lesPilotes.Add(new PiloteComparable("Robert", "Kubica"), "BMW Sauber");
lesPilotes.Add(new PiloteComparable("Giancarlo", "Fisichella"), "Renault");
lesPilotes.Add(new PiloteComparable("Heikki", "Kovalainen"), "Renault");
lesPilotes.Add(new PiloteComparable("Alexander", "Wurz"), "Williams");
lesPilotes.Add(new PiloteComparable("Mark", "Webber"), "Red Bull");

Afficher(lesPilotes);
//1 - Pilote : Alexander Wurz , Ecurie : Williams
//2 - Pilote : Felipe Massa , Ecurie : Ferrari
//3 - Pilote : Fernando Alonso , Ecurie : McLaren Mercedes
//4 - Pilote : Giancarlo Fisichella , Ecurie : Renault
//5 - Pilote : Heikki Kovalainen , Ecurie : Renault
//6 - Pilote : Kimi Räikkönen , Ecurie : Ferrari
//7 - Pilote : Lewis Hamilton , Ecurie : McLaren Mercedes
//8 - Pilote : Mark Webber , Ecurie : Red Bull
//9 - Pilote : Nick Heidfeld , Ecurie : BMW Sauber
```

### Manipulation d'une SortedDictionary

```
//10 - Pilote : Robert Kubica , Ecurie : BMW Sauber

//ContainsKey utilise que CompareTo
Console.WriteLine(lesPilotes.ContainsKey(new PiloteComparable("Robert", "Kubica")));
//True

//Remove utilise que CompareTo
lesPilotes.Remove(new PiloteComparable("Giancarlo", "Fisichella"));

Console.WriteLine(Environment.NewLine + lesPilotes[new PiloteComparable("Mark", "Webber")]);
//Red Bull
```

### IV-D - OrderedDictionary

La OrderedDictionary est une implémentation particulière de la table de hachage. Elle permet de disposer d'un accès indexé en consultation et suppression. Elle est généralement utile pour contrôler l'ordre des éléments dans le dictionnaire mais reste néanmoins une collection non typée.

#### Quand faut-il choisir la classe OrderedDictionary ?

- |   |  |
|---|--|
| 1 | Collection <b>non typée</b> qui stocke des listes de <b>clé/valeur</b> . L'accès aux couples est possible soit <b>par clé ou par index</b> |
| 2 | N'autorise pas les <b>clés doublons</b> .  |
| 3 | <b>Rapide</b> : accès et suppression en <b>O(1)</b>  |

Le code qui suit montre comment utiliser les fonctions d'insertion et de suppression par index et par clé dans la classe OrderedDictionary :

### Manipulation d'un OrderedDictionary

```
OrderedDictionary lesPilotes = new OrderedDictionary();

lesPilotes.Add("L. Hamilton", "McLaren Mercedes");
lesPilotes.Add("F. Alonso", "McLaren Mercedes");
lesPilotes.Add("F. Massa", "Ferrari");

//insertion indexée
lesPilotes.Insert(3, "K. Räikkönen", "Ferrari");
lesPilotes.Insert(4, "N. Heidfeld", "BMW Sauber");
lesPilotes.Insert(5, "R. Kubica", "BMW Sauber");

lesPilotes.Add("G. Fisichella", "Renault");
lesPilotes.Add("H. Kovalainen", "Renault");
lesPilotes.Add("A. Wurz", "Williams");
lesPilotes.Add("M. Webber", "Red Bull");

Afficher(lesPilotes);
//1 - Pilote : L. Hamilton , Ecurie : McLaren Mercedes
//2 - Pilote : F. Alonso , Ecurie : McLaren Mercedes
//3 - Pilote : F. Massa , Ecurie : Ferrari
//4 - Pilote : K. Räikkönen , Ecurie : Ferrari
//5 - Pilote : N. Heidfeld , Ecurie : BMW Sauber
//6 - Pilote : R. Kubica , Ecurie : BMW Sauber
//7 - Pilote : G. Fisichella , Ecurie : Renault
//8 - Pilote : H. Kovalainen , Ecurie : Renault
//9 - Pilote : A. Wurz , Ecurie : Williams
//10 - Pilote : M. Webber , Ecurie : Red Bull

//Ajouter une nouvelle entrée à la tête du dictionnaire
lesPilotes.Insert(0, "Lewis Hamilton", "McLaren Mercedes");
```

### Manipulation d'un OrderedDictionary

```
//Modification indexée
lesPilotes[0] = "McLaren";
//Modification par clé
lesPilotes["Lewis Hamilton"] = "McLaren Mercedes";

Afficher(lesPilotes);
//1 - Pilote : Lewis Hamilton , Ecurie : McLaren Mercedes
//2 - Pilote : L. Hamilton , Ecurie : McLaren Mercedes
//3 - Pilote : F. Alonso , Ecurie : McLaren Mercedes
//4 - Pilote : F. Massa , Ecurie : Ferrari
//5 - Pilote : K. Räikkönen , Ecurie : Ferrari
//6 - Pilote : N. Heidfeld , Ecurie : BMW Sauber
//7 - Pilote : R. Kubica , Ecurie : BMW Sauber
//8 - Pilote : G. Fisichella , Ecurie : Renault
//9 - Pilote : H. Kovalainen , Ecurie : Renault
//10 - Pilote : A. Wurz , Ecurie : Williams
//11 - Pilote : M. Webber , Ecurie : Red Bull
```

### IV-E - ListDictionary Vs Hashtable Vs HybridDictionary

Le choix d'un dictionnaire simple et non typé en 2.0 dépend fortement de la taille des éléments de la collection.

Comment choisir parmi ListDictionary, Hashtable et HybridDictionary ?	
1	ListDictionary, Hashtable, HybridDictionary sont des collections <b>non typées</b> qui stockent des listes de <b>clé/valeur</b> .
2	<b>ListDictionary</b> : Utile dans le cas où la taille de la collection est petite. MSDN recommande que la taille soit inférieure à <b>10 éléments</b> .
3	<b>Hashtable</b> : Efficace pour les collections de grande taille ; <b>supérieures à 10 éléments</b>
4	<b>HybridDictionary</b> : Utile dans le cas où on <b>ne connaît pas</b> à l'avance la taille du dictionnaire. Lorsque la taille d'une HybridDictionary est inférieure à 10, elle utilise dans ce cas une ListDictionary. Lorsque la taille dépasse les 10 éléments, la HybridDictionary transfère les données vers une Hashtable.
5	La classe Dictionary reste néanmoins la plus efficace si les types des clés et valeurs sont uniques.

Le code qui suit représente un ensemble de benchmark qui permet de confirmer les performances des dictionnaires cités par rapport au nombre d'éléments insérés :

### ListDictionary Vs Hashtable Vs HybridDictionary

```
long tempsD = DateTime.Now.Ticks;
long tempsF = DateTime.Now.Ticks;

Hashtable hash = new Hashtable(10000);
ListDictionary list = new ListDictionary();

tempsD = DateTime.Now.Ticks;

for (int i = 0; i < 10000; i++)
{
    hash.Add(i, i);
}

tempsF = DateTime.Now.Ticks;

Console.WriteLine("Le temps mis pour insérer 10000 éléments dans une Hashtable est {0}", tempsF -
    tempsD);
```

## ListDictionary Vs Hashtable Vs HybridDictionary

```
//Le temps mis pour insérer 10000 éléments dans une Hashtable est 40000

tempsD = DateTime.Now.Ticks;

for (int i = 0; i < 10000; i++)
{
    list.Add(i, i);
}

tempsF = DateTime.Now.Ticks;

Console.WriteLine("Le temps mis pour insérer 10000 éléments dans une ListDictionary est {0}",
    tempsF - tempsD);
//Le temps mis pour insérer 10000 éléments dans une ListDictionary est 6720000

Console.ReadKey();

/***** Test Dictionary et ListDictionary : Cas de classes personnalisées : Dictionary est plus
rapide *****/

list.Clear();

Dictionary<PiloteEquatable, string> lesPilotes = new Dictionary<PiloteEquatable, string>(10);

tempsD = DateTime.Now.Ticks;
lesPilotes.Add(new PiloteEquatable("Fernando", "Alonso"), "McLaren Mercedes");
lesPilotes.Add(new PiloteEquatable("Felipe", "Massa"), "Ferrari");
lesPilotes.Add(new PiloteEquatable("Kimi", "Räikkönen"), "Ferrari");
lesPilotes.Add(new PiloteEquatable("Nick", "Heidfeld"), "BMW Sauber");
lesPilotes.Add(new PiloteEquatable("Robert", "Kubica"), "BMW Sauber");
lesPilotes.Add(new PiloteEquatable("Giancarlo", "Fisichella"), "Renault");
lesPilotes.Add(new PiloteEquatable("Heikki", "Kovalainen"), "Renault");
lesPilotes.Add(new PiloteEquatable("Alexander", "Wurz"), "Williams");
lesPilotes.Add(new PiloteEquatable("Mark", "Webber"), "Red Bull");
tempsF = DateTime.Now.Ticks;

Console.WriteLine("Le temps mis pour insérer 10 éléments dans une Dictionary est {0}", tempsF -
    tempsD);
//Le temps mis pour insérer 10 éléments dans une Dictionary est 30000

tempsD = DateTime.Now.Ticks;
list.Add(new PiloteEquatable("Fernando", "Alonso"), "McLaren Mercedes");
list.Add(new PiloteEquatable("Felipe", "Massa"), "Ferrari");
list.Add(new PiloteEquatable("Kimi", "Räikkönen"), "Ferrari");
list.Add(new PiloteEquatable("Nick", "Heidfeld"), "BMW Sauber");
list.Add(new PiloteEquatable("Robert", "Kubica"), "BMW Sauber");
list.Add(new PiloteEquatable("Giancarlo", "Fisichella"), "Renault");
list.Add(new PiloteEquatable("Heikki", "Kovalainen"), "Renault");
list.Add(new PiloteEquatable("Alexander", "Wurz"), "Williams");
list.Add(new PiloteEquatable("Mark", "Webber"), "Red Bull");
tempsF = DateTime.Now.Ticks;

Console.WriteLine("Le temps mis pour insérer 10 éléments dans une ListDictionary est {0}", tempsF -
    tempsD);
//Le temps mis pour insérer 10 éléments dans une List est 890000

/***** Test Dictionary et ListDictionary : Cas de types simples *****/

list.Clear();

Dictionary<string, string> lesPilotesstring = new Dictionary<string, string>(10);

tempsD = DateTime.Now.Ticks;
lesPilotesstring.Add("L. Hamilton", "McLaren Mercedes");
lesPilotesstring.Add("F. Alonso", "McLaren Mercedes");
lesPilotesstring.Add("F. Massa", "Ferrari");
lesPilotesstring.Add("K. Räikkönen", "Ferrari");
```

### ListDictionary Vs Hashtable Vs HybridDictionary

```

lesPilotesstring.Add("N. Heidfeld", "BMW Sauber");
lesPilotesstring.Add("R. Kubica", "BMW Sauber");
lesPilotesstring.Add("G. Fisichella", "Renault");
lesPilotesstring.Add("H. Kovalainen", "Renault");
lesPilotesstring.Add("A. Wurz", "Williams");
lesPilotesstring.Add("M. Webber", "Red Bull");
tempsF = DateTime.Now.Ticks;

Console.WriteLine("Le temps mis pour insérer 10 éléments dans une Dictionary est {0}", tempsF -
tempsD);
//Le temps mis pour insérer 10 éléments dans une Dictionary est 0

tempsD = DateTime.Now.Ticks;
list.Add("L. Hamilton", "McLaren Mercedes");
list.Add("F. Alonso", "McLaren Mercedes");
list.Add("F. Massa", "Ferrari");
list.Add("K. Räikkönen", "Ferrari");
list.Add("N. Heidfeld", "BMW Sauber");
list.Add("R. Kubica", "BMW Sauber");
list.Add("G. Fisichella", "Renault");
list.Add("H. Kovalainen", "Renault");
list.Add("A. Wurz", "Williams");
list.Add("M. Webber", "Red Bull");
tempsF = DateTime.Now.Ticks;

Console.WriteLine("Le temps mis pour insérer 10 éléments dans une List est {0}", tempsF - tempsD);
//Le temps mis pour insérer 10 éléments dans une List est 0

Console.ReadKey();

```

### IV-F - StringDictionary

La stringDictionary est l'équivalent de la Hashtable sauf qu'elle est fortement typée et n'accepte que des strings pour les clés et les valeurs. Les clés sont insensibles à la casse et seront converties en minuscule lors de l'insertion.

#### Quand faut-il choisir la classe stringDictionary ?

- 1 Collection **typée** qui stocke des listes de **clé/valeur de strings**. La collection est insensible à la casse et sauvegarde ses clés en minuscule.
- 2 N'autorise pas les **clés doublons**.
- 3 Utile pour les Framework précédents le 2.0 où les generics n'existaient pas encore.
- 4 **Rapide** : Insertion et suppression en **O(1)**

### IV-G - NameValueCollection

La NameValueCollection est similaire à la stringDictionary du fait que les deux collections sont fortement typées et n'acceptent que des strings. La NameValueCollection offre une fonctionnalité exclusive qui permet d'insérer des clés en double. On peut en effet avoir plusieurs valeurs pour une seule et unique clé. D'autre part la NameValueCollection permet au développeur de disposer d'un accès indexé.

#### Quand faut-il choisir la classe NameValueCollection ?

- 1 Collection **typée** qui stocke des listes de **clé/valeur de strings**. La collection offre aussi un accès indexé.
- 2 **Autorise les clés doublons**.
- 3 **Rapide** : Insertion et suppression en **O(1)**

Le code qui suit montre comment la NameValueCollection permet d'avoir plusieurs valeurs pour la même clé et comment itérer par index sur la collection

### Manipulation d'un NameValueCollection

```

NameValueCollection ecuries = new NameValueCollection();

ecuries.Add("McLaren Mercedes", "L. Hamilton");
ecuries.Add("McLaren Mercedes", "F. Alonso");

foreach (string pilote in ecuries.GetValues("McLaren Mercedes"))
{
    Console.WriteLine(pilote);
}
//L. Hamilton
//F. Alonso

ecuries.Add("Ferrari", "F. Massa");
ecuries.Add("Ferrari", "F. Massa");

//accès indexé
for (int i = 0; i < ecuries.Count; i++)
{
    Console.WriteLine(" les pilotes de l'écurie {0} sont : {1} ", ecuries.GetKey(i),
        ecuries.Get(i));
}

//les pilotes de l'écurie McLaren Mercedes sont : L. Hamilton,F. Alonso
//les pilotes de l'écurie Ferrari sont : F. Massa,F. Massa
    
```

## IV-H - KeyedCollection

La dernière classe étudiée dans cet article est la KeyedCollection. C'est une classe abstraite générique qui stocke non pas des listes de clé/valeur mais plutôt des éléments (qui constituent les valeurs) dont la clé est définie par ses propriétés. La KeyedCollection est la plus intéressante des dictionnaires en 2.0 mais reste néanmoins une des moins connues. En effet, elle permet une certaine transparence par rapport à la manipulation des paires clé/valeur et par suite les méthodes d'insertion et de suppression sont plus évidentes à utiliser.

### Quand faut-il choisir la classe KeyedCollection ?

- |   |   |
|---|---|
| 1 | <b>Classe abstraite typée</b> qui stocke des valeurs d'objets dont la clé est définie par les propriétés de ses éléments. |
| 2 | N'autorise pas les <b>clés doublons</b> .   |
| 3 | Accès et suppression par clé ou index.  |
| 4 | <b>Rapide</b> : Insertion et suppression en <b>O(1)</b>   |

Pour pouvoir utiliser la classe, il faut tout d'abord créer sa propre collection en héritant de la classe KeyedCollection et en spécifiant le type de la clé et de la valeur. La méthode protégée **GetKeyFromItem** permet de définir la logique de l'extraction de la clé à partir de l'élément défini.

Le code suivant permet d'avoir une idée sur l'intérêt de l'utilisation de la KeyedCollection et montre comment créer sa propre collection :

### Manipulation d'une KeyedCollection

```

public class PiloteKeyedCollection : KeyedCollection<string,Pilote>
{
    //permet définir la clé par rapport à la classe Pilote
    protected override string GetKeyForItem(Pilote item)
    {
        return string.Concat(item.Prenom, " ", item.Nom);
    }
}
    
```

## Manipulation d'une KeyedCollection

```
public static class KeyedCollectionDemo
{
    public static void Execute()
    {
        PiloteKeyedCollection lesPilotes = new PiloteKeyedCollection();

        lesPilotes.Add(new Pilote("Lewis", "Hamilton"));
        lesPilotes.Add(new Pilote("Fernando", "Alonso"));

        try
        {
            lesPilotes.Add(new Pilote("Fernando", "Alonso"));
            //An item with the same key has already been added.
        }
        catch (ArgumentException ex)
        {
            Console.WriteLine(ex.Message);
        }

        lesPilotes.Add(new Pilote("Felipe", "Massa"));
        lesPilotes.Add(new Pilote("Kimi", "Räikkönen"));
        lesPilotes.Add(new Pilote("Nick", "Heidfeld"));
        lesPilotes.Add(new Pilote("Robert", "Kubica"));
        lesPilotes.Add(new Pilote("Giancarlo", "Fisichella"));
        lesPilotes.Add(new Pilote("Heikki", "Kovalainen"));
        lesPilotes.Add(new Pilote("Alexander", "Wurz"));
        lesPilotes.Add(new Pilote("Mark", "Webber"));

        //Lewis Hamilton
        //Fernando Alonso
        //Felipe Massa
        //Kimi Räikkönen
        //Nick Heidfeld
        //Robert Kubica
        //Giancarlo Fisichella
        //Heikki Kovalainen
        //Alexander Wurz
        //Mark Webber

        Afficher(lesPilotes);

        Console.ReadKey();

        Console.WriteLine("Le pilote Lewis Hamilton appartient-il à la collection ? : {0}",
lesPilotes.Contains("Lewis Hamilton"));
        //Le pilote Lewis Hamilton appartient-il à la collection ? : True

        Console.ReadKey();

        //insertion indexée
        lesPilotes.Insert(0, new Pilote("David", "Coulthard"));

        lesPilotes.RemoveAt(10);

        //affichage par index
        AfficherParIndex(lesPilotes);
        //David Coulthard
        //Lewis Hamilton
        //Fernando Alonso
        //Felipe Massa
        //Kimi Räikkönen
        //Nick Heidfeld
        //Robert Kubica
        //Giancarlo Fisichella
        //Heikki Kovalainen
        //Alexander Wurz
    }
}
```

### Manipulation d'une KeyedCollection

```
        Console.ReadKey();
    }

    private static void Afficher(PiloteKeyedCollection lesPilotes)
    {
        Console.WriteLine();
        foreach (Pilote pilote in lesPilotes)
        {
            Console.WriteLine(pilote);
        }
        Console.WriteLine();
    }

    private static void AfficherParIndex(PiloteKeyedCollection lesPilotes)
    {
        Console.WriteLine();

        for (int i = 0; i < lesPilotes.Count; i++)
        {
            Console.WriteLine(lesPilotes[i]);
        }
        Console.WriteLine();
    }
}
```

## V - Conclusion

Nous avons vu dans cet article que le Framework 2.0 offre aux développeurs .Net plusieurs structures de hachages différentes. Le choix judicieux d'un dictionnaire permet un gain de performance lors de l'exécution du code et surtout un gain en productivité non négligeable.

L'intérêt principal de cet article était de persuader les développeurs de ne plus avoir recours systématiquement aux classes Hashtable et Dictionary, mais plutôt d'étudier leur besoins par rapport aux différentes collections disponibles dans le Framework.

## Ressource

Télécharger le code source : [Cliquez ici](#)

## Remerciments

Je remercie toute l'équipe .Net et spécialement Jérôme Lambert (**Cardi**) pour leur soutien et support tout au long de la rédaction de cet article.

Merci à Eric Reboisson (**elitost**) pour la relecture finale de cet article.

