



La collection HashSet du Framework 3.5

Florian Casabianca

03/09/2007

Remerciements

J'adresse ici tous mes remerciements à l'équipe de rédaction de "developpez.com" pour le temps qu'ils ont bien voulu passer à la correction et à l'amélioration de cet article.

Table des matières

Introduction.....	3
Création d'un HashSet.....	3
L'Insertion.....	4
L'union.....	4
L'intersection.....	5
La différence.....	5
La différence symétrique.....	5
Sous-ensemble.....	5
Sous-ensemble strict.....	6
Sur-ensemble.....	6
Sur-ensemble strict.....	6
L'unicité dans un HashSet.....	7
Conclusion.....	9
Liens.....	9

Introduction

La nouvelle version du Microsoft .NET Framework 3.5 (qui sera livrée avec la prochaine version de Visual Studio® 2008) va introduire une nouvelle collection disponible dans l'espace de nom **System.Collections.Generic**: la classe **HashSet**. Cette dernière fournit une collection haute performance qui ne contient aucun élément en double et dont ces éléments ne sont pas placés dans un ordre particulier.

HashSet implémente les opérations standards des collections telles que Add, Clear, Remove, Contains, mais fournit aussi des opérations d'ensembles comme l'union, l'intersection ou la différence symétrique.

Création d'un HashSet

Il est possible de construire un HashSet de plusieurs manières. La première est d'utiliser le constructeur sans argument:

```
HashSet<int> hset = new HashSet<int>();
```

Le constructeur possède une surcharge qui prend en paramètres un **IEnumerable** générique. Vous pouvez donc par exemple utiliser une liste générique pour initialiser un HashSet:

```
/****** Création d'un HashSet à partir d'un autre IEnumerable *****/  
List<int> maListe = new List<int> { 1, 4, 5, 7, 5 };  
HashSet<int> hset2 = new HashSet<int>(maListe); //le deuxième 5 n'est pas ajouté
```

Dans cet exemple hset2 ne contiendra que quatre éléments. En effet, un objet **HashSet** ne contient pas d'éléments en double. Or la liste générique passée en paramètre contient deux fois le chiffre cinq. Le deuxième 5 ne sera donc pas inséré dans le **HashSet**.

L'insertion

L'insertion d'un nouvel élément se fait avec la méthode **Add**. En fait, il faudrait plutôt ici parler de fonction. En effet, contrairement aux autres listes génériques, la méthode **Add** renvoie ici un booléen indiquant si l'ajout à bien été fait ou non. L'ajout d'un élément peut en effet échouer si cet élément est déjà présent dans le HashSet. Voici un exemple:

```
HashSet<int> hset = new HashSet<int>();
hset.Add(1);
hset.Add(2);
hset.Add(3);
bool ajout = hset.Add(4); //ajout vaut true
ajout = hset.Add(2); //élément non ajouté car déjà présent. ajout vaut false
```

L'union

La méthode **UnionWith** prend en paramètre un **IEnumerable** générique et permet de faire l'union des éléments contenus dans un HashSet avec ceux de n'importe quelle liste générique (les éléments devant être de même type). Bien entendu, les doublons sont éliminés.

```
HashSet<int> hset = new HashSet<int>( new int[] { 1,2,3,4 } );
HashSet<int> hset2 = new HashSet<int>( new List<int> { 1, 4, 5, 7 } );

hset.UnionWith(hset2);
//hset contient maintenant 1,2,3,4,5,7

List<int> liste = new List<int> { 1, 5, 11, 8, 3 };
hset.UnionWith(liste);
//hset contient maintenant 1,2,3,4,5,7,11,8

int[] tab = new int[] { 2, 7, 15, 0 };
hset.UnionWith(tab);
//hset contient maintenant 1,2,3,4,5,7,11,8,15,0
```

La méthode **UnionWith** ne crée pas un nouvel HashSet mais modifie celui sur lequel cette méthode est appelée. Ce point est important car LINQ propose lui aussi des opérations d'union, d'insertion, etc. La différence est que LINQ construit une nouvelle collection.

Voici un exemple d'union avec LINQ:

```
int[] numbersA = { 0, 2, 4, 5, 6, 8, 9 };
int[] numbersB = { 1, 3, 5, 7, 8 };
// l'union des deux tableaux avec LINQ crée un nouvel objet
var uniqueNumbers = numbersA.Union(numbersB);
```

L'intersection

L'intersection se fait avec la méthode **IntersectWith**. Comme avec l'union, cette méthode prend en paramètre un **IEnumerable** générique.

```
HashSet<int> hset = new HashSet<int>(new List<int> { 1, 3, 5, 7, 8 });  
  
hset.IntersectWith(new List<int> { 1, 6, 12, 8, 3 });  
//hsat contient maintenant 1,3,8
```

La différence

La méthode **ExceptWith** supprime tous les éléments d'un HashSet qui sont aussi contenus dans la collection passée en paramètre.

```
HashSet<int>hset = new HashSet<int>( new List<int> { 1, 3, 5, 7, 8 } );  
hset.ExceptWith( new int[] { 3, 7,9 } );  
//hsat contient maintenant 1,5,8
```

La différence symétrique

La méthode **SymmetricExceptWith** modifie le HashSet pour qu'il contienne tous les éléments contenus par soit la classe HashSet, soit la collection passée en paramètre, mais pas les deux.

```
HashSet<int> hset = new HashSet<int>(new List<int> { 1, 3, 5, 7, 8 });  
hset.SymmetricExceptWith(new int[] { 2, 3, 8, 9 });  
//hsat contient maintenant 1,5,7,2,9
```

Sous-ensemble

La méthode **IsSubsetOf** permet de savoir si un HashSet est un sous-ensemble d'un ensemble donné. On rappelle qu'un ensemble A est un sous-ensemble d'un ensemble B si **tous** les éléments de A sont aussi contenus dans B. Ainsi, l'ensemble A{2,4} est un sous-ensemble de l'ensemble B{2,5,4}. L'ensemble B ne doit forcément posséder plus d'éléments que A. Ainsi, A{1,2} est un sous-ensemble de B{2,1}. Deux remarques: un ensemble est toujours un sous-ensemble de lui-même et l'ensemble vide est toujours un sous-ensemble de n'importe quel ensemble (même de lui-même).

```
hset = new HashSet<int>(new List<int> { 1, 3, 5 });  
bool isSubsetOf = hset.IsSubsetOf(new int[] { 1, 2, 3, 8, 9, 5 });  
//isSubsetOf vaut vrai car hset est un sous-ensemble du tableau  
  
hset = new HashSet<int>(new List<int> { 1, 3, 5, 4 });  
isSubsetOf = hset.IsSubsetOf(new int[] { 1, 3, 6, 5 });  
//isSubsetOf vaut faux car hset n'est pas un sous-ensemble du tableau  
  
hset = new HashSet<int>(new List<int> { 1, 3, 5, 6 });  
isSubsetOf = hset.IsSubsetOf(new int[] { 1, 3, 6, 5 });  
//isSubsetOf vaut vrai car hset est un sous-ensemble du tableau
```

Sous-ensemble strict

Un ensemble A est strictement inclus dans un ensemble B si et seulement si A est inclus dans B sans lui être égal. Ainsi, l'ensemble $A\{1,2\}$ est un **sous-ensemble** de $B\{1,2\}$ mais pas un **sous-ensemble strict**. L'ensemble $C\{3,4\}$ est lui un sous-ensemble strict de $D\{3,5,4\}$. L'ensemble vide est toujours un sous-ensemble strict d'un autre ensemble SAUF de lui-même.

```
hset = new HashSet<int>(new List<int> { 1, 3, 5 });
bool isProperSubsetOf = hset.IsProperSubsetOf(new int[] { 1, 2, 3, 8, 9, 5 });
//isProperSubsetOf vaut vrai car hset est un sous-ensemble strict du tableau
```

```
hset = new HashSet<int>(new List<int> { 1, 3, 5 });
isProperSubsetOf = hset.IsProperSubsetOf(new int[] { 1, 3, 5 });
//isProperSubsetOf vaut faux car hset n'est pas un sous-ensemble strict du tableau
```

Sur-ensemble

B est sur-ensemble de A, si tout élément du sous-ensemble A est aussi élément du sur-ensemble B. Il peut par contre y avoir des éléments de B qui ne sont pas éléments de A. Ainsi, $B\{1,5,6\}$ est un sur-ensemble de $A\{6,5\}$. Mais c'est aussi un sur-ensemble de $C\{5,6,1\}$. L'élément vide est un sur-ensemble de lui-même.

```
hset = new HashSet<int>(new List<int> { 1, 3, 5, 6, 7, 8 });
bool isSupersetOf = hset.IsSupersetOf(new int[] { 1, 8, 3 });
//isSupersetOf vaut vrai car hset est un sur-ensemble du tableau
```

```
hset = new HashSet<int>(new List<int> { 3, 8, 1 });
isSupersetOf = hset.IsSupersetOf(new int[] { 1, 8, 3 });
//isSupersetOf vaut vrai car hset est un sur-ensemble du tableau
```

Sur-ensemble strict

Un ensemble A est un stricte sur-ensemble d'un ensemble B si B est inclus dans A sans lui être égal. Ainsi, l'ensemble $A\{1,2,3\}$ est un **sur-ensemble** de $B\{1,2,3\}$ mais pas un **sur-ensemble strict**. L'ensemble $C\{3,4,5\}$ est lui un **sur-ensemble strict** de $D\{3,5\}$. L'ensemble vide n'est pas un sur-ensemble de lui-même.

```
hset = new HashSet<int>(new List<int> { 1, 3, 5, 2 });
bool isProperSupersetOf = hset.IsProperSupersetOf(new int[] { 1, 3, 2 });
//isProperSupersetOf vaut vrai car hset est un sur-ensemble strict du tableau
```

```
hset = new HashSet<int>(new List<int> { 1, 3, 5 });
isProperSupersetOf = hset.IsProperSupersetOf(new int[] { 1, 3, 5 });
//isProperSupersetOf vaut faux car hset n'est pas un sur-ensemble strict du tableau
```

L'unicité dans un HashSet

Jusqu'à présent nous avons exploré la possibilité de la classe **HashSet** à l'aide d'exemples simples utilisant des entiers. Nous avons vu que le code suivant

```
HashSet<int> hset = new HashSet<int>();  
hset.Add(5);  
hset.Add(5);
```

produisait un **HashSet** ne contenant qu'une seule fois le chiffre 5. La classe **HashSet** se sert d'un **EqualityComparer** afin de déterminer si deux éléments sont égaux. Si aucun **EqualityComparer** n'est spécifié, la classe **HashSet** utilise l'**EqualityComparer** par défaut. Dans l'exemple ci-dessus elle utilise l'**EqualityComparer** par défaut pour les `Int32`.

Mais que se passe-t-il si vous décidez d'utiliser un **HashSet** avec vos propres classes ? Prenons le cas d'une classe `Personne` dont voici la spécification:

```
class Personne  
{  
    public int Id { get; set; }  
    public String Nom { get; set; }  
  
    public Personne()  
    { }  
}
```

Sur quels critères le **HashSet** va-t-il se baser pour déterminer que deux objets `Personnes` sont différents?

Faisons un premier essai:

```
Personne p1 = new Personne { Id = 1, Nom = "Toto" }; //une 1ère personne  
Personne p2 = new Personne { Id = 2, Nom = "Toto" }; //une 2ème personne avec le même nom  
Personne p3 = new Personne { Id = 2, Nom = "Toto" }; //la même personne que p2  
  
HashSet<Personne> listePersonnes = new HashSet<Personne>();  
listePersonnes.Add(p1);  
listePersonnes.Add(p2);  
listePersonnes.Add(p3);  
  
foreach (var p in listePersonnes)  
{  
    Console.WriteLine(p.Id);  
}
```

Nous obtenons l'affichage suivant sur la console: 1,2,2. Le **HashSet** contient bien trois éléments alors que pour nous `p2` et `p3` correspondent à la même personne (Identifiants identiques). Pour faire en sorte que le **HashSet** détermine que deux personnes sont identiques si elles possèdent le même identifiant, il va falloir créer un **EqualityComparer** personnalisé.

Cela se fait en quelques lignes:

```

class PersonneEvenComparer : IEqualityComparer<Personne>
{
    public PersonneEvenComparer() { }
    public bool Equals(Personne p1, Personne p2)
    {
        return (p1.Id == p2.Id); //deux personnes sont égales si elles ont le même Id
    }

    public int GetHashCode(Personne p)
    {
        return (p.Id & 1);
    }
}

```

Il ne reste plus qu'à passer une instance de **PersonneEvenComparer** en paramètre du constructeur du HashSet:

```

Personne p1 = new Personne { Id = 1, Nom = "Toto" };
Personne p2 = new Personne { Id = 2, Nom = "Toto" };
Personne p3 = new Personne { Id = 2, Nom = "Toto" };

HashSet<Personne> listePersonnes = new HashSet<Personne>(new PersonneEvenComparer());
listePersonnes.Add(p1);
listePersonnes.Add(p2);
listePersonnes.Add(p3);

foreach (var p in listePersonnes)
{
    Console.WriteLine(p.Id);
}

```

Cette fois-ci nous obtenons l'affichage suivant: 1,2. Le HashSet a déterminé que p2 et p3 étaient égaux et n'a donc pas ajouté p3.

Conclusion

La nouvelle classe HashSet permet de prendre en charge la plupart des opérations mathématiques qui sont généralement réalisées sur des ensembles. De plus, elle s'intègre très bien avec les collections génériques existantes depuis la version 2.0 du Framework.

Liens

MSDN magazine Avril 2007:

<http://msdn.microsoft.com/msdnmag/issues/07/04/CLRInsideOut/Default.aspx?loc=fr#S6>

BCL Team Blog:

<http://blogs.msdn.com/bclteam/archive/2006/11/09/introducing-hashset-t-kim-hamilton.aspx>

HashSet sur MSDN:

[http://msdn2.microsoft.com/en-us/library/bb359438\(VS.90\).aspx](http://msdn2.microsoft.com/en-us/library/bb359438(VS.90).aspx)