

Opérateurs « Count » & « LongCount »

```
public static int Count<TSource>(this IEnumerable<TSource> source);  
public static int Count<TSource>(this IEnumerable<TSource> source, Func<TSource, bool> predicate);  
public static long LongCount<TSource>(this IEnumerable<TSource> source);  
public static long LongCount<TSource>(this IEnumerable<TSource> source, Func<TSource, bool> predicate);
```

Ces deux opérateurs comptent le nombre d'éléments dans une collection. Leur différence se situe dans leur type de retour. Ceux-ci peuvent être accompagnés d'une expression booléenne qui va permettre de filtrer le nombre d'éléments à compter.

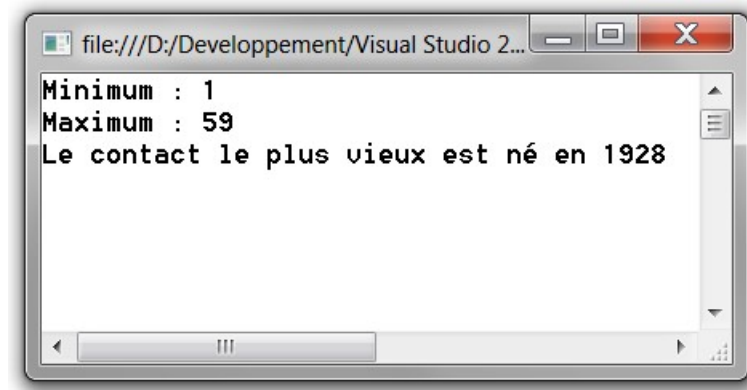
```
int[] ints = new int[] { 5, 4, 7, 52, 36, 59, 24, 1 };  
// Retourne le nombre de l'ensemble des éléments  
Console.WriteLine(string.Format("Nombre total : {0}", ints.Count()));  
// retourne le nombre des éléments pairs  
Console.WriteLine(string.Format("Nombre d'éléments pairs : {0}", ints.Count(i => i % 2 == 0)));
```



Opérateurs « Min » & « Max »

Comme l'indique leurs noms, les opérateurs « Min » et « Max » retournent respectivement la valeur minimale et maximale d'une collection.

```
int[] ints = new int[] { 5, 4, 7, 52, 36, 59, 24, 1 };  
// Retourne la plus petite valeur  
Console.WriteLine(string.Format("Minimum : {0}", ints.Min()));  
// Retourne la plus grande valeur  
Console.WriteLine(string.Format("Maximum : {0}", ints.Max()));  
// Retourne l'année de naissance du contact le plus vieux  
int AnnéeDeNaissance = Contacts.Min(c => c.AnneeDeNaissance);  
Console.WriteLine("Le contact le plus vieux est né en {0}", AnnéeDeNaissance);
```

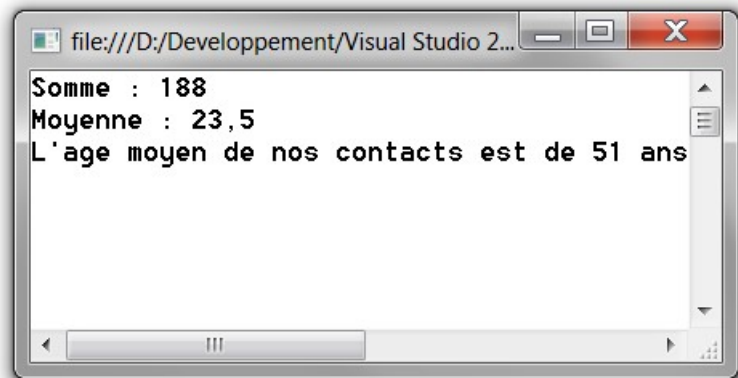


*Voir les différentes surcharges dans la classe « Enumerable »

Opérateurs « Sum » & « Average »

Les opérateurs « Sum » et « Average » retournent respectivement la somme et la moyenne d'une collection.

```
int[] ints = new int[] { 5, 4, 7, 52, 36, 59, 24, 1 };  
// Retourne la somme  
Console.WriteLine(string.Format("Somme : {0}", ints.Sum()));  
// Retourne la moyenne  
Console.WriteLine(string.Format("Moyenne : {0}", ints.Average(i => (float)i)));  
// Retourne l'age moyen des contacts  
Console.WriteLine("L'age moyen de nos contacts est de {0} ans",  
    DateTime.Now.Year - (int)Contacts.Average(c => c.AnneeDeNaissance));
```



Opérateur « GroupBy »

```
public static IEnumerable<IGrouping<TKey, TSource>> GroupBy<TSource, TKey>(this IEnumerable<TSource> source, Func<TSource, TKey> keySelector);
```

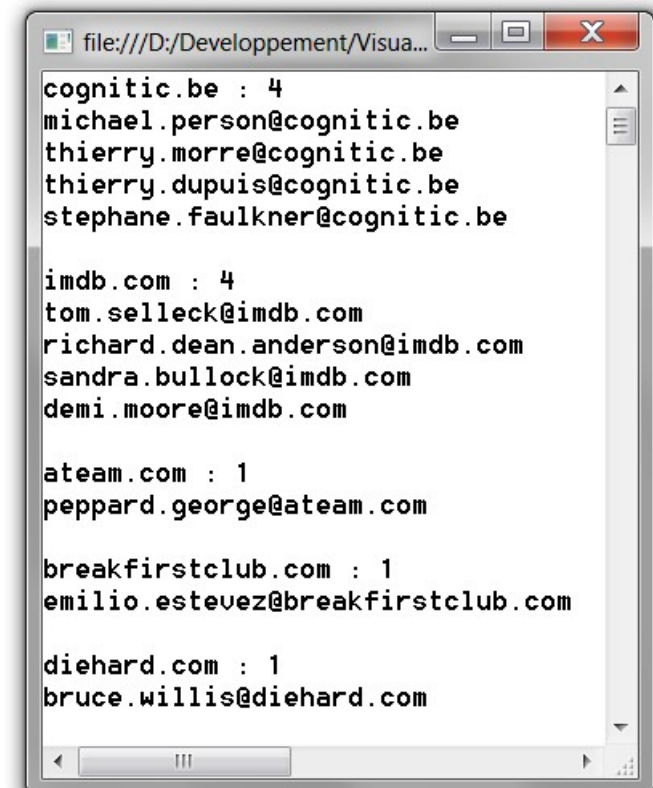
L'opérateur « GroupBy » est un peu différent des autres, par défaut celui-ci travail avec l'interface « IGrouping<Tkey, TElement> » qui hérite de « IEnumerable<T> » et intégrant une propriété « Key ».

```
...public interface IGrouping<out TKey, out TElement> : IEnumerable<TElement>, IEnumerable  
{  
    ...TKey Key { get; }  
}
```

Ce qui donne ceci :

```
IEnumerable<IGrouping<string, Contact>> QueryResult = Contacts  
    .GroupBy(c => c.Email.Substring(c.Email.IndexOf('@') + 1));  
  
foreach (IGrouping<string, Contact> g in QueryResult)  
{  
    Console.WriteLine("{0} : {1}", g.Key, g.Count());  
    foreach (Contact c in g)  
    {  
        Console.WriteLine("{0}", c.Email);  
    }  
    Console.WriteLine();  
}
```

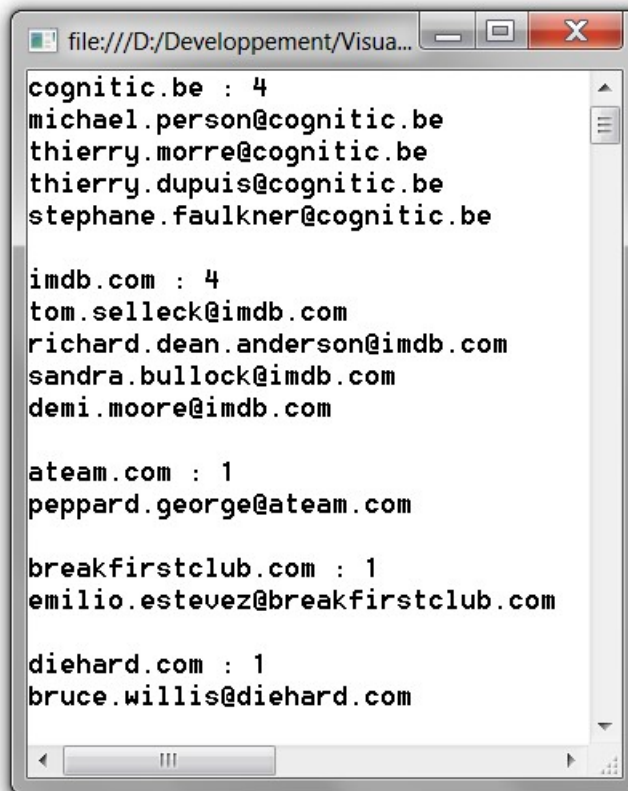
Il existe plusieurs surcharge, ici n'est vue que la plus utilisée.



Opérateur « GroupBy »

De plus, dans le cadre de l'expression régulière, « GroupBy » ne peut être utilisé avec l'opérateur « Select » excepté dans le cadre du « group ... by ... into ... ».

Expression de requête :

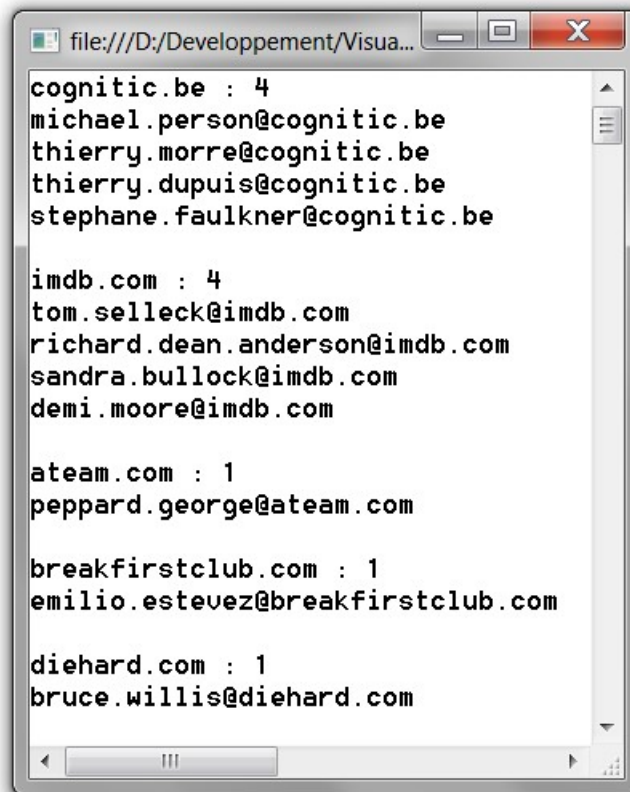


```
IEnumerable<IGrouping<string, Contact>> QueryResult =  
    from c in Contacts  
    group c by c.Email.Substring(c.Email.IndexOf('@') + 1);  
  
foreach (IGrouping<string, Contact> g in QueryResult)  
{  
    Console.WriteLine("{0} : {1}", g.Key, g.Count());  
    foreach (Contact c in g)  
    {  
        Console.WriteLine("{0}", c.Email);  
    }  
    Console.WriteLine();  
}
```

Notons l'absence de l'opérateur « Select » dans la requête.

Opérateur « GroupBy »

Expression de requête :



```
file:///D:/Developpement/Visua...  
cognitic.be : 4  
michael.person@cognitic.be  
thierry.morre@cognitic.be  
thierry.dupuis@cognitic.be  
stephane.faulkner@cognitic.be  
  
imdb.com : 4  
tom.selleck@imdb.com  
richard.dean.anderson@imdb.com  
sandra.bullock@imdb.com  
demi.moore@imdb.com  
  
ateam.com : 1  
peppard.george@ateam.com  
  
breakfirstclub.com : 1  
emilio.estevez@breakfirstclub.com  
  
diehard.com : 1  
bruce.willis@diehard.com
```

```
IEnumerable<IGrouping<string, Contact>> QueryResult =  
    from c in Contacts  
    group c by c.Email.Substring(c.Email.IndexOf('@') + 1);  
  
foreach (IGrouping<string, Contact> g in QueryResult)  
{  
    Console.WriteLine("{0} : {1}", g.Key, g.Count());  
    foreach (Contact c in g)  
    {  
        Console.WriteLine("{0}", c.Email);  
    }  
    Console.WriteLine();  
}
```

Notons l'absence de l'opérateur « Select » dans la requête.

Opérateur « GroupBy »

Dans ce cas, comment utiliser l'opérateur « GroupBy » avec les types anonymes ?

```
var QueryResult = Contacts
    .Select(c => new { Email = c.Email,
                     FullName = string.Format("{0} {1}", c.Nom, c.Prenom) })
    .GroupBy(c => c.Email.Substring(c.Email.IndexOf('@') + 1));

foreach (var group in QueryResult)
{
    Console.WriteLine("{0} : {1}", group.Key, group.Count());
    foreach (var element in group)
    {
        Console.WriteLine("{0}", element.FullName);
    }
    Console.WriteLine();
}
```

En utilisant l'opérateur « Select » avant le « GroupBy » ...



Opérateur « GroupBy »

Expression de requête :

```
var QueryResult = from c2 in (from c in Contacts
                               select new { Email = c.Email,
                                             FullName = string.Format("{0} {1}", c.Nom, c.Prenom) })
                  group c2 by c2.Email.Substring(c2.Email.IndexOf('@') + 1);

foreach (var group in QueryResult)
{
    Console.WriteLine("{0} : {1}", group.Key, group.Count());
    foreach (var element in group)
    {
        Console.WriteLine("{0}", element.FullName);
    }
    Console.WriteLine();
}
```

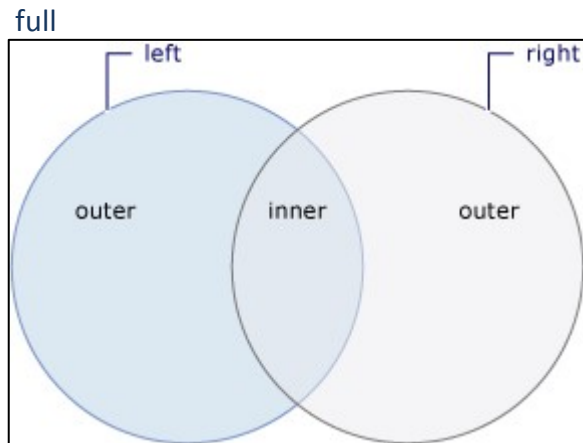
... qui se traduira par une sous-requête dans l'expression de requête.



Opérateur « Join »

L'opérateur « Join » est utile pour associer des séquences différentes sur base de valeurs pouvant être comparée pour définir une égalité.

La jointure est une opération importante dans les requêtes qui ciblent les sources de données dont les relations ne peuvent pas être suivies directement. Dans la programmation orientée objet, cela pourrait signifier **une corrélation entre objets qui n'est pas modélisée***.



Quand on parle de jointures, il y en a 3 qui reviennent régulièrement :

Croisée (Cross Join), Interne (Inner Join), Externe (Outer Join - Left, Right & full). Celles-ci peuvent être basées sur une égalité « Equi Join » et ou non « Non Equi Join ».

L'opérateur « Join » en LINQ ne reprend qu'une seule forme de jointure (« Inner Join » basée sur une égalité). Cependant nous allons voir comment réaliser les autres jointures en utilisant LINQ.

Pour les « Cross Join » ou les « Non Equi Join », nous ne pourrions pas utiliser l'opérateur « Join ». Cependant nous pourrions contourner le problème par l'utilisation de plusieurs clauses « from » et l'utilisation de clauses « where ».

*Cette corrélation est fortement utilisée en « LINQ To SQL » & « LINQ To Entities »

Opérateur « Join »

Afin de comprendre les jointures en LINQ, nous allons ajouter une nouvelle classe « RDV » et de nouvelles données (Liste « RendezVous ») à notre environnement.

```
public class RDV
{
    public string Email { get; set; }
    public DateTime Date { get; set; }
}
```

```
List<RDV> RendezVous = new List<RDV>();
RendezVous.AddRange(new RDV[] {
    new RDV(){ Email = "stephane.faulkner@cognitic.be", Date = new DateTime(2012,5,12)},
    new RDV(){ Email = "peppard.george@ateam.com", Date = new DateTime(2011,8,14)},
    new RDV(){ Email = "bruce.willis@diehard.com", Date = new DateTime(2012,6,19)},
    new RDV(){ Email = "bruce.willis@diehard.com", Date = new DateTime(2012,6,20)},
    new RDV(){ Email = "michael.person@cognitic.be", Date = new DateTime(2012,04,19)},
});
```

Opérateur « Join »

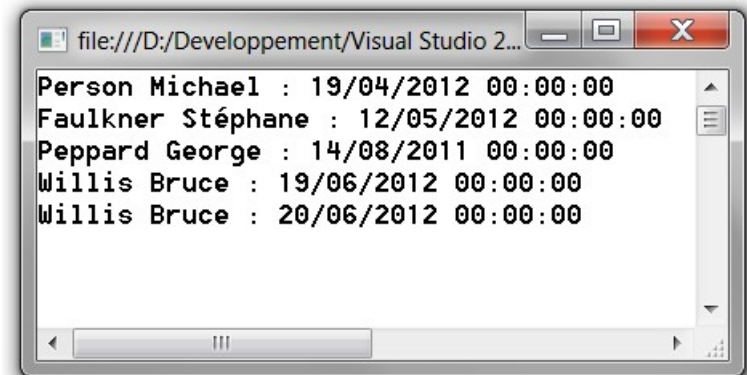
```
public static IEnumerable<TResult> Join<TOuter, TInner, TKey, TResult>(this IEnumerable<TOuter> outer,
    IEnumerable<TInner> inner,
    Func<TOuter, TKey> outerKeySelector,
    Func<TInner, TKey> innerKeySelector,
    Func<TOuter, TInner, TResult> resultSelector);
```

Inner Join :

Obtenir l'email, le nom, le prénom du contact et la date de tous les rendez-vous.

```
var QueryResult = Contacts.Join(RendezVous,
    c => c.Email,
    rdv => rdv.Email,
    (c, rdv) => new {
        Email = c.Email,
        Nom = c.Nom,
        Prenom = c.Prenom,
        DateRDV = rdv.Date});

foreach (var jointure in QueryResult)
{
    Console.WriteLine("{0} {1} : {2}", jointure.Nom, jointure.Prenom, jointure.DateRDV);
}
```

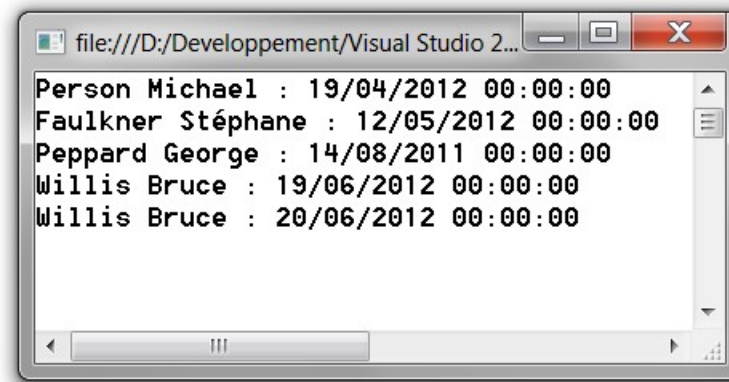


Opérateur « Join »

Expression de requête :

```
var QueryResult = from c in Contacts
                  join rdv in RendezVous on c.Email equals rdv.Email
                  select new {
                      Email = c.Email,
                      Nom = c.Nom,
                      Prenom = c.Prenom,
                      DateRDV = rdv.Date};

foreach (var jointure in QueryResult)
{
    Console.WriteLine("{0} {1} : {2}", jointure.Nom, jointure.Prenom, jointure.DateRDV);
}
```



Opérateur « Join »

L'utilisation des types anonymes n'est pas obligatoire dans le cadre des jointures, son emploi résulte en effet de ce que nous allons sélectionner.

Exemple : Obtenir les contacts ayant pris rendez-vous.

```
IEnumerable<Contact> QueryResult = Contacts.Join(RendezVous,  
                                                  c => c.Email,  
                                                  rdv => rdv.Email,  
                                                  (c, rdv) => c);
```

Expression de requête :

```
IEnumerable<Contact> QueryResult = from c in Contacts  
join rdv in RendezVous on c.Email equals rdv.Email  
select c;
```

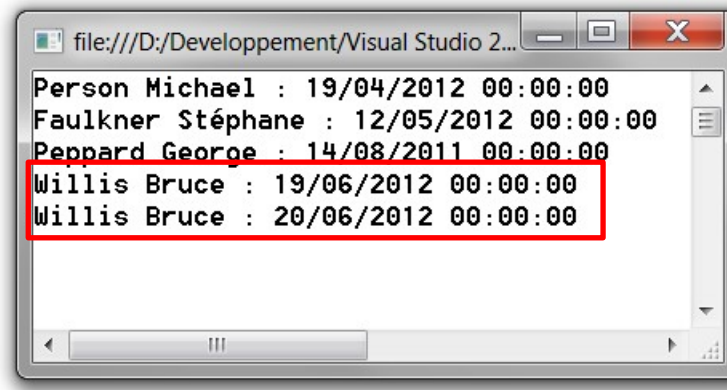
Nous obtenons bien une séquence de Contacts

Opérateur « GroupJoin »

```
public static IEnumerable<TResult> GroupJoin<TOuter, TInner, TKey, TResult>(this IEnumerable<TOuter> outer,
    IEnumerable<TInner> inner,
    Func<TOuter, TKey> outerKeySelector,
    Func<TInner, TKey> innerKeySelector,
    Func<TOuter, IEnumerable<TInner>, TResult> resultSelector);
```

L'opérateur « GroupJoin » travaille de manière comparable à l'opérateur « Join », à ceci près que l'opérateur « Join » ne passe qu'un seul élément de la séquence externe et un élément de la séquence interne à la fonction « resultSelector ».

Cela signifie que si plusieurs éléments de la séquence intérieure (inner) correspondent à un élément de la séquence extérieure (outer), nous aurons plusieurs lignes dans notre « result set ».



L'opérateur « GroupJoin » va, quant à lui, produire une structure de donnée hiérarchique. Il va associer pour chaque élément de la séquence extérieure les éléments de la séquence intérieure qui le concerne.

Si aucun élément de la séquence intérieure n'existe, il retournera une séquence vide.

Il s'apparente donc à une Jointure externe gauche (« Left Join »).

Opérateur « GroupJoin »

Exemple :

Pour tous contacts, obtenir les noms, prénoms et date de rendez-vous éventuels.

```
var QueryResult = Contacts.GroupJoin(RendezVous,
                                     c => c.Email,
                                     rdv => rdv.Email,
                                     (c, rdvs) => new {
                                         Email = c.Email,
                                         Nom = c.Nom,
                                         Prenom = c.Prenom,
                                         RendezVous = rdvs});

foreach (var jointure in QueryResult)
{
    Console.WriteLine("{0} {1} :", jointure.Nom, jointure.Prenom);
    if (jointure.RendezVous.Count() > 0)
    {
        foreach (RDV rdv in jointure.RendezVous)
        {
            Console.WriteLine("{0}", rdv.Date);
        }
    }
    else
    {
        Console.WriteLine("Aucun");
    }
    Console.WriteLine();
}
```



Opérateur « GroupJoin »

Expression de requête :

Dans le cadre de l'expression de requêtes, nous devons ajouter l'expression « into » à notre jointure.

```
var QueryResult = from c in Contacts
                  join rdv in RendezVous on c.Email equals rdv.Email into grdvs
                  select new { Nom = c.Nom, Prenom = c.Prenom, RendezVous = grdvs };

foreach (var jointure in QueryResult)
{
    Console.WriteLine("{0} {1} :", jointure.Nom, jointure.Prenom);
    if (jointure.RendezVous.Count() > 0)
    {
        foreach (RDV rdv in jointure.RendezVous)
        {
            Console.WriteLine("{0}", rdv.Date);
        }
    }
    else
    {
        Console.WriteLine("Aucun");
    }
    Console.WriteLine();
}
```



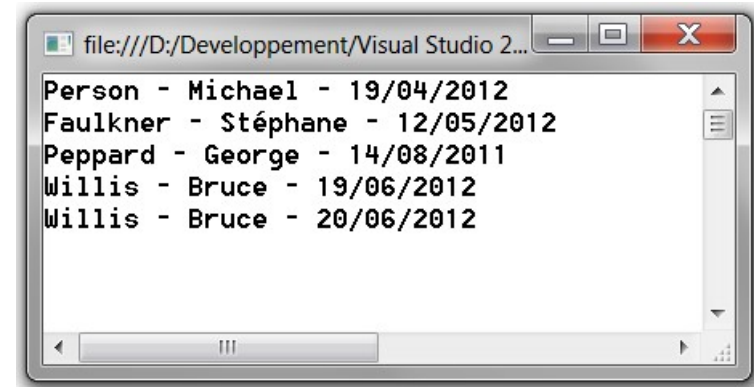
Utilisation de clés composites

Dans le cadre de jointures, nous sommes parfois amené à gérer les clés étrangères utilisant plusieurs champs (clé composite) et « LINQ » n'échappe à la règle. Afin de résoudre ce « problème » nous devons utiliser les classes anonymes.

```
var QueryResult = from c in Contacts
                  join rdv in RendezVous on new { c.ID, c.Email } equals new { rdv.ID, rdv.Email }
                  select new { c.Nom, c.Prenom, rdv.Date };

var QueryResult2 = Contacts.Join(RendezVous,
                                c => new { c.ID, c.Email },
                                rdv => new { rdv.ID, rdv.Email },
                                (c, rdv) => new { c.Nom, c.Prenom, rdv.Date });

foreach (var r in QueryResult)
{
    Console.WriteLine("{0} - {1} - {2}", r.Nom, r.Prenom, r.Date.ToShortDateString());
}
```



Multiple clause « from »

Nous venons de voir que les jointures en « LINQ » sont des jointures internes basée sur une égalité. Comment dans ce cas faire une jointure croisée (« Cross Join ») ou une « Non Equi Join »?

Elles ne sont possible que dans le cadre des expressions de requêtes en utilisant plusieurs clauses « from ».

Exemple de « Cross Join » :

```
var QueryResult = from c in Contacts
                  from rdv in RendezVous
                  select new { c.Nom, c.Prenom, rdv.Date };

foreach (var r in QueryResult)
{
    Console.WriteLine("{0} - {1} - {2}", r.Nom, r.Prenom, r.Date.ToShortDateString());
}
```

