Côté cours : Découverte d'Entity Framework

Description du thème

Propriétés	Description
Type de publication	Cours
Intitulé cout	Découverte d' Entity Framework
Intitulé long	Présentation des nouveautés des framework Dotnet 3 et 3.5 ainsi que des classes permettant de mapper des tables relationnelles vers des classes
Formation concernée	BTS Services informatiques aux organisations
Matière	SLAM4 - Réalisation et maintenance de composants logiciels
Présentation	Les éléments nécessaires à la découverte d'Entity Framework sont présentés, le langage de requête Linq, et les nouveaux éléments de programmation dynamique du langage C# ; pour la plupart des notions des exercices sont proposés.
Notions	• Programmation objet • C# • Visual Studio 2008 • Linq • Mapping • Programmation dynamique
Transversalité	
Pré-requis	Les notions de base de la programmation objet, la syntaxe C#.
Outils	Visual Studio 2008, un navigateur, SQL server La base ecoleConduite figure dans l'archive
Mots-clés	Interface, mapping, Linq, expressions lambda, Entity Framework
Durée	8 heures
Auteur(es)	Patrice Grand
Date	Octobre 2009

Introduction

Le support proposé aborde principalement la notion de mapping avec Dotnet, version 3.5. Néanmoins, avant d'aborder le framework, Entity Framework, il convient de maitriser différents éléments du langage ; c'est pourquoi des notions plus anciennes sont présentées (Delegate, Types génériques, les itérateurs, méthodes et classes anonymes, le langage de requête Linq). Le support est sous forme de site avec des exercices d'applications (les corrigés ne sont pas fournis et sont disponibles sur simple demande).

Les évolutions des langages

Nous assistons depuis quelques années à une évolution des langages de programmation. A l'origine ceux-ci (C, C++, Java, C#) pouvaient être qualifiés de fortement *impératifs* et proches de la machine ; les structures algorithmiques s'adossaient aux structures de données telles qu'elles étaient implantées en mémoire, on écrivait t[0] pour obtenir la première valeur d'un tableau (fortement typé), les déclarations de variables pouvaient paraître redondantes (**int** n = 100 % 5 !!, **Personne** p = new Personne() !!), il fallait très précisément indiquer le « comment faire » etc…
Aujourd'hui les langages évoluent, s'affranchissent de plus en plus des contraintes de stockage physique. On demandera t.first(), on itérera sur une structure algorithmique de haut niveau (foreach), on passera moins de temps sur le « comment faire » pour s'attarder sur le « ce que je veux ». Les langages deviennent ainsi plus *déclaratifs.*
La deuxième évolution concerne le rôle du compilateur ; traditionnellement le langage compilait ce qui était codé. Ceci est de moins en moins vrai aujourd'hui ; de nouvelles couches d'abstraction permettent d'écrire du code qui

sera interprété avant la compilation. Le langage C # fournit de nombreux exemples de ce compilateur intelligent capable d'interpréter ce que veut le développeur codant dans un plus haut niveau d'abstraction. On utilise parfois le terme de programmation *dynamique* pour désigner cette tendance.
La première partie du support (rappel des notions nécessaires) illustre ces deux évolutions du langage.

Le mapping avec Linq

La deuxième partie du support présente un nouveau langage de requêtage Linq, pouvant être utilisé sur des objets (listes…), XML, ou des tables relationnelles. Concernant ce dernier cas, le grand intérêt est de profiter de la vérification syntaxique à la compilation (ce qui n'est pas le cas d'un langage hôte) et aussi de l'Intellisence.
La troisième partie présente le framework de mapping dans ces éléments généraux avec des exemples à partir d'un cas de gestion à télécharger.

Sommaire

A-Des éléments significatifs du langage C#

1) Retour sur le type *delegate*

Ceci n'est pas une nouveauté, revenons néanmoins sur ce type un peu particulier car il joue un rôle central dans les nouveaux éléments de syntaxe C# 3 et 3.5.
On dit souvent que le type `delegate` est un type "pointeur sur une fonction". Faisons un retour en arrière pour revenir à son ancêtre en C : pointeur sur fonction.

1.a Les pointeurs sur fonction en C

Rien de tel qu'un petit exemple pour illustrer la notion, imaginons la situation simple :

```
#include <iostream>
using namespace std;

int incremente(int n)
{
    return ++n;
}

void main()
{
    cout<<incremente(9)<<endl;
```

Cet appel affichera 10, par appel de la fonction `incremente`.
Maintenant, déclarons un pointeur sur une fonction de même signature que la fonction `incremente` :

```
#include <iostream>
using namespace std;

int incremente(int n)
{
    return ++n;
}

void main()
{
    int (*ptr)(int);        // déclaration d'un pointeur sur une fonction de signature int f(int)
    cout<<incremente(9)<<endl;
    ptr=incremente;         // référençons ptr avec la fonction incremente
    cout<<ptr(9)<<endl;
    system("pause");
}
```

A l'exécution, on obtient bien sûr, le même résultat :

La ligne *ptr = incremente* laisse penser que le nom d'une fonction est une adresse (un pointeur), confirmons-le en affichant aussi les valeurs des fonctions :

```
void main()
{
    int (*ptr)(int);        // déclaration d'un pointeur sur une fonction de signature int f(int)
    cout<<incremente(9)<<endl;
    ptr=incremente;         // référençons ptr avec la fonction incremente
    cout<<ptr(9)<<endl;
    cout<<incremente<<endl;  // affichage de l'adresse de la fonction incremente
    cout<<ptr<<endl;        // affichage du pointeur de fonction
    system("pause");
}
```

Ce qui donne :

Le pointeur *ptr* pourra, si besoin, référencer une autre fonction ; ajoutons une fonction `decremente` en conservant le code essentiel :

```
#include <iostream>
using namespace std;

int incremente(int n)
{
    return ++n;
}
int decremente(int n)
{
    return --n;
}
void main()
{
    int (*ptr)(int);        // déclaration d'un pointeur sur une fonction de signature int f(int)
    ptr = incremente;       // référençons ptr avec la fonction incremente
    cout<<ptr(9)<<endl;
    ptr = decremente;       // référençons ptr avec la fonction decremente
    cout<<ptr(9)<<endl;
    system("pause");
}
```

Ce qui produira, sans surprise :

Notons que `ptr` ne peut référencer qu'une fonction dont la signature lui a été précisée à la déclaration : *int f(int)*.

Pour terminer, rendons le code plus compact et utilisons le pointeur de fonction comme un argument d'une fonction d'affichage :

```
#include <iostream>
using namespace std;

int incremente(int n)
{
    return ++n;
}
int decremente(int n)
{
    return --n;
}
void affiche(int a, int (*ptr)(int )) //le deuxième argument est un pointeur de fonction
{
    cout<<ptr(a)<<endl;
}
void main()
{
    affiche(9,incremente);
    affiche(9,decremente);
```

Le résultat est identique au programme précédent.

Ainsi, ici, la fonction *affiche* prend un pointeur comme argument. Noter la syntaxe de l'appel *affiche(9, incremente)* ; cet appel ne doit pas préciser l'argument de la fonction *incremente*. Nous obtenons un code très compact, qui certes perd un peu en lisibilité compte tenu des indirections opérées...
L'intérêt principal de ce type de programmation est de proposer une liaison retardée (*late-binding*) en injectant du code à des endroits à priori inattendus.

1.b Les delegate en C#

Écrivons maintenant en C# l'équivalent de la dernière version :

```
using System.Collections.Generic;
using System.Text;

namespace TestDelegate
{
    delegate int monDelegue(int n);    //déclaration du type délégué
    class Program
    {
        static int incremente(int a)
        {
            return ++a;
        }
        static int decremente(int a)
        {
            return --a;
        }
        static void affiche(int b, monDelegue deleg)
        {
            System.Console.WriteLine(deleg(b).ToString());
        }
        static void Main(string[] args)
        {

            affiche(9, incremente);
            affiche(9, decremente);
```

L'effet est bien sûr le même :

Notons néanmoins quelques différences :

1.b.1 Le mot de`legate` déclare un type qui peut être utilisé par la suite

Ainsi nous pourrions écrire :

```
        static void Main(string[] args)
        {
            monDelegue d = incremente;
            affiche(9, d);
            d = decremente;
            affiche(9, d);
```

Dans cette version nous déclarons une variable d de type `monDelegue`. Il y a donc distinction classique entre le type et la variable de ce type ; ceci permet également une signature de la méthode `affiche` un peu moins obscure qu'en C!! (mais cela ne va pas durer :-), cf plus loin)

1.b.2 Une seconde différence tient à la nature des delegate, types plus élaborés qu'en C ; en effet il possible qu'un délégué pointe sur plusieurs fonctions (multicast).

Pour montrer cela, on va un peu modifier les fonctionnalités des fonctions afin de se concentrer sur l'essentiel :

```
namespace TestDelegate
{
    delegate void monDelegue(int n);     //déclaration du type délégué
    class Program
    {
        static void incremente(int a)
        {
            a++;
            System.Console.WriteLine(a.ToString());
        }
        static void decremente(int a)
        {
            a--;
            System.Console.WriteLine(a.ToString());
        }

        static void Main(string[] args)
        {
            monDelegue d = incremente;
           d += decremente;
            d(9);
```

Les fonctions `incremente` et `decremente` affichent maintenant le résultat, le délégué a été modifié en conséquence ; l'opérateur += ajoute une référence au délégué ; ainsi le délégué pointe sur une liste de deux fonctions.

Ce qui donne à l'exécution :

Si l'on trace le code, on constate bien que l'appel de `d(9)` engendre l'exécution des deux méthodes *`incremente`* et *`decremente`*, et cela dans l'ordre de leurs "inscriptions". L'opérateur `-=` retire de la liste pointée la dernière référence des fonctions "inscrites".

Remarques :
- L'initialisation *`monDelegue d = incremente`* est un racourci pour
`monDelegue d = new monDelegue(incremente);`
- Le langage C proposait également ce type de service par l'intermédiaire des *`functors`*.
- La portée de la définition du délégué suit les règles générales de portée de C# (ici la portée est le `namespace`)
- Les *`delegate`* sont à la base des événements.

TP d'application

Créer un nouveau projet console C#, remplacer le code de la classe générée par :

```
using System;
using System.Collections.Generic;
using System.Text;
namespace exoDelegate
{
/*  partie à copier*/
   class Calcul
   {
      public static int somme(int a,int b)
      {
         return a+b;
      }
      public static int produit(int a,int b)
      {
         return a*b;
      }
   }
   class Program
   {
      static void Main(string[] args)
      {
         int result = 0;
         Console.WriteLine("premier nb svp:");
         int n1 = Convert.ToInt32(Console.ReadLine());
         Console.WriteLine("deuxième nb svp:");
         int n2=Convert.ToInt32(Console.ReadLine());
         Console.WriteLine("quelle opération '+' ou '*' : ");
         char choix=(char)Console.Read();
         switch(choix)
         {
            case '+':
            {result = Calcul.somme(n1,n2);break;}
            case '*':
            {result = Calcul.produit(n1,n2);break;}
         }
         Console.WriteLine("resultat:" + result.ToString());
   }
/*fin partie à copier*/
```

1. **Sans modifier la classe Calcul, mettre en œuvre un `delegate` pour effectuer le même traitement.**

2) Les types génériques (ou *templates* ou types paramétrés)

La version 2.0 introduit un nouveau concept, cher à C++, les types génériques (`template` en C).

Ainsi (comme en C) le type générique est annoncé par les balises < et >. Imaginons une classe `Pile` qui ne fait qu'ajouter ou retirer des éléments (dernier entré). Les types génériques permettent de ne présenter qu'une seule interface pour une classe générique permettant de créer des piles d'entiers, de réels ou de chaînes ou autres.

La classe `Pile` pourrait se présenter ainsi :

```
class Pile<T>
{
    public Pile(int n)
    {
        this.mesT = new T[n];
        this.nbElements = 0;
    }
    public void depile()
    {
        if (this.nbElements > 0)
            this.nbElements--;
    }
    private bool estPLeine()
    {
        return this.mesT.Length == this.nbElements;
    }
    public void empile(T unT)
    {
        if (!this.estPLeine())
        {
            this.mesT[this.nbElements] = unT;
            this.nbElements++;
        }
    }
    public int getNbElements()
    {
        return this.nbElements;
    }
    public T leT(int n)
    {
        return this.mesT[n];
    }
    private T[] mesT;
    private int nbElements;
}
```

Le type générique est annoncé par <T> ; nous pourrions, bien sûr, utiliser tout autre identificateur que T. Le type générique T est utilisé à chaque fois que sa référence est exigée.

L'utilisation est simple :

```
class Program
{
    static void Main(string[] args)
    {
        Pile<string> p1 = new Pile<string>(10);
        p1.empile("toto");
        p1.empile("titi");
        p1.empile("tutu");
        p1.dePile();
        for(int i=0; i<p1.getNbElements();i++)
            System.Console.WriteLine((p1.leT(i).ToString()));
```

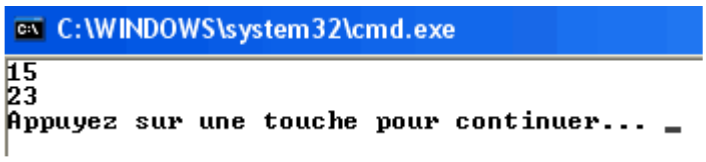
Ce qui provoque :

A l'initialisation d'une pile, il faut indiquer le type réel ; ici le compilateur va générer une classe `Pile` typée par un *string*.

Nous pourrions créer, de la même manière, une pile d'entiers, en utilisant la même interface :

```
Pile<int> p1 = new Pile<int>(3);
p1.depile();
p1.empile(15);
p1.empile(23);
p1.empile(11);
p1.empile(5);
p1.depile();
for (int i = 0; i < p1.getNbElements(); i++)
    Console.WriteLine(p1.leT(i).ToString());
```

Ce qui produit :



Une classe générique n'est pas réduite à un seul type générique, ainsi une classe *`MaClasse`* pourrait être déclarée ainsi :

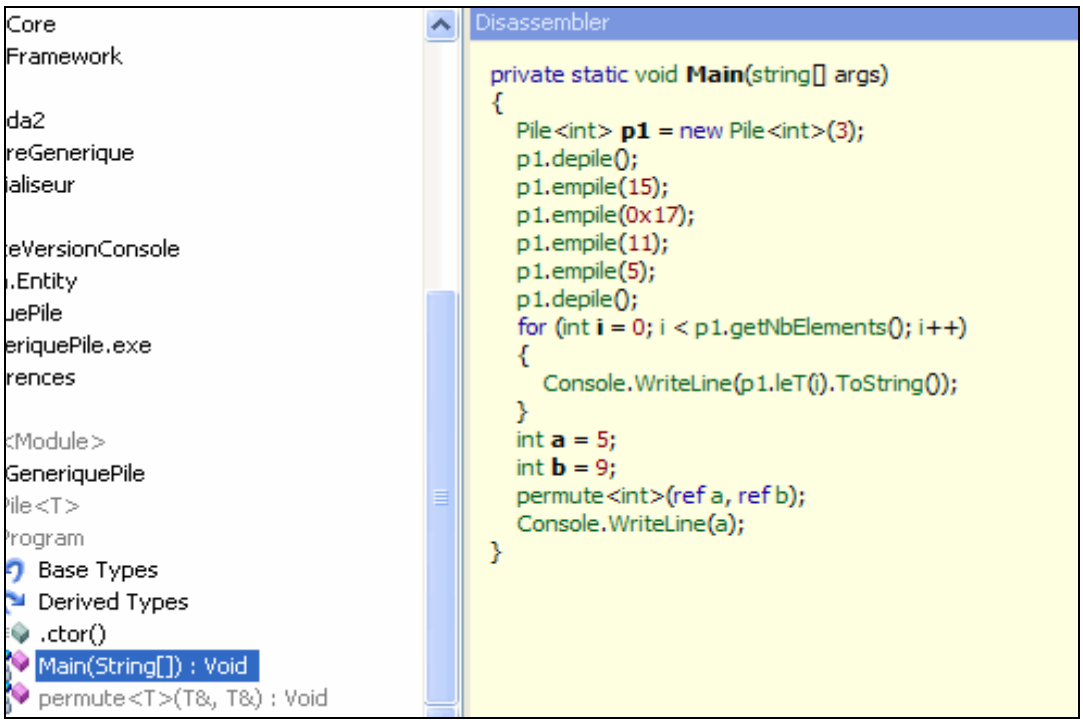
`class MaClasse<T1><T2>`.

Les types génériques peuvent représenter les types du framework ou tout type créé.

Une méthode peut aussi être générique, dans ses arguments ou pour son type de retour :

```
void permute<T>(ref T a, ref T b)
{
T temp = a;
a = b;
b = temp;
}
int n1 =12;
int n2 = 9;
permute<int>(ref n1, ref n2);
```

Remarque : le dernier appel pourrait être réduit à `permute(ref n1, ref n2)` ; en effet, à la compilation le langage est capable *d'inférer* le type réel à partir du type des arguments. Pour voir ce qui a été généré par le compilateur observons le code (outil *Reflector*) :



Remarque : `Reflector` est petit outil gratuit qui désassemble le fichier *exe* et montre ainsi le code réellement compilé et qui n'est pas toujours le code écrit.
Le framework (à partir de 2.0) propose ainsi des collections génériques :
`List<T>` ou `Dictionnary<Tcle><Tvaleur>`

3) Les itérateurs.

Plutôt que de partir d'une approche théorique sur les itérateurs, posons le problème. Imaginons une situation simple où nous disposons d'une classe `Personne`, réduite à deux attributs (nom et âge) ; ainsi qu'une méthode (redéfinie, `ToString`) qui retourne la chaîne constituée des valeurs des attributs.

```
{
    class Personne
    {
        public Personne(string nom, int age)
        {this.nom = nom; this.age=age;}
        public override string  ToString()
        {
            return this.nom + "  " + this.age.ToString() ;
        }
        private string nom;
        private int age;

    }
```

Nous disposons, par ailleurs, d'une classe conteneur qui contient des personnes : la classe *`Groupe`*. Les objets *`Personne`* de chaque *`Groupe`* sont placés dans un tableau.

```
  class Groupe
 {
      private Personne[] mesPersonnes;
      private int nbPersonnes = 0;
      public Groupe(int n)
      {
          mesPersonnes = new Personne[n];
      }
      public void ajout(Personne p)
      {
          mesPersonnes[this.nbPersonnes++] = p;
      }
      public int getNbPersonnes()
      {
          return this.nbPersonnes;
      }
  }
```

Mais, et c'est là où se situe le problème ; nous ne pouvons pas demander à un `Groupe` d'itérer sur ses `Personne`. Nous ne pouvons pas écrire quelque chose du genre :

```
    static void Main(string[] args)
    {
        Groupe g = new Groupe(3);
        g.ajout(new Personne("toto",12));
        g.ajout(new Personne("titi",15));
        g.ajout(new Personne("tutu",25));
        foreach (Personne p in g)
            Console.WriteLine(p.ToString());
```

La structure itérative `foreach` n'est pas acceptée ; elle ne l'est pas car seules les classes énumérables peuvent bénéficier de l'itérateur `foreach`. Une classe énumérable est une classe qui peut présenter l'itérateur `foreach` pour parcourir sa collection d'objets. En d'autres termes, la classe `Groupe` doit implémenter l'interface

`IEnumerable`, et donc fournir le code de la méthode public *`IEnumerator GetEnumerator()`* (unique méthode de l'interface `IEnumerable`).

Le code de la classe `Groupe` devient :

```
 class Groupe : IEnumerable
{
      private Personne[] mesPersonnes;
      private int nbPersonnes = 0;
      public Groupe(int n)
      {
          mesPersonnes = new Personne[n];
      }
      public void ajout(Personne p)
      {
          mesPersonnes[this.nbPersonnes++] = p;
      }
      public int getNbPersonnes()
      {
          return this.nbPersonnes;
      }
      public IEnumerator GetEnumerator()
      {
          for (int i = 0; i < this.nbPersonnes; i++)
          {
              Personne p = (Personne)this.mesPersonnes[i];
              yield return p;
          }
      }
}
```

Maintenant le code :

```
    static void Main(string[] args)
    {
        Groupe g = new Groupe(3);
        g.ajout(new Personne("toto",12));
        g.ajout(new Personne("titi",15));
        g.ajout(new Personne("tutu",25));
        foreach (Personne p in g)
            Console.WriteLine(p.ToString());
```

affiche bien :

```
file:///C:/Documents and Settings/pat
toto  12
titi  15
tutu  25
```

La méthode `GetEnumerator`, imposée par l'interface, se contente de parcourir le tableau et de retourner les éléments dans une bien étrange instruction `yield return`. Par ailleurs, le type de retour est un `IEnumerator` ; interface qui permet de parcourir une collection, cf la description dans MSDN de l'interface `IEnumerator`:

Membres IEnumerator

Voir aussi Méthodes Propriétés

Réduire tout | Options des membres : Afficher tout

Prend en charge une itération simple sur une collection non générique.

Les tableaux suivants listent les membres exposés par le type IEnumerator .

Propriétés publiques

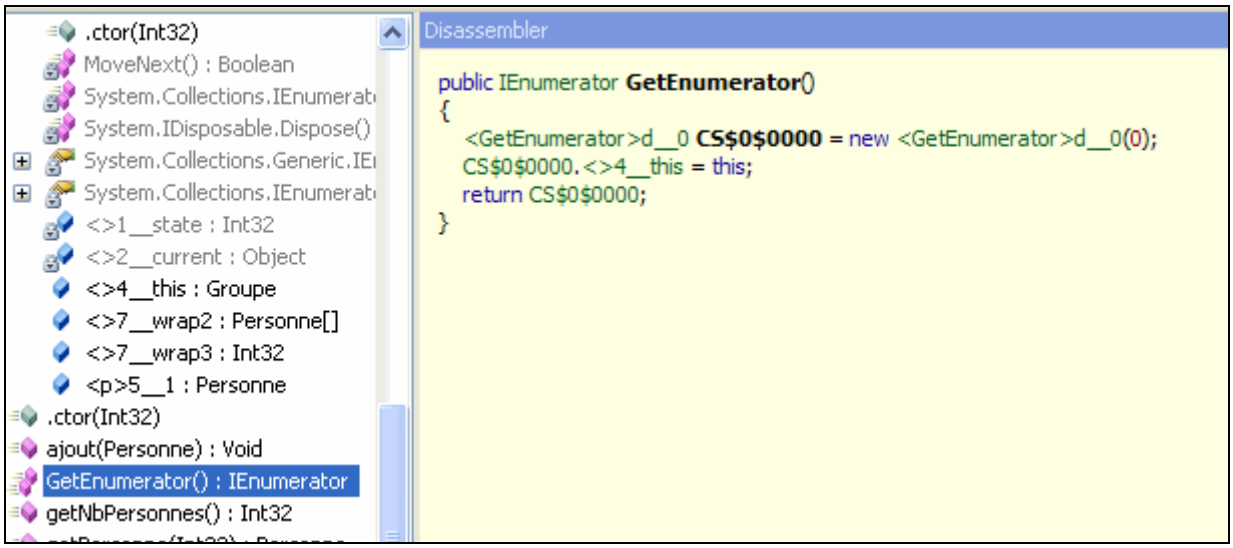
	Nom	Description
	Current	Obtient l'élément en cours dans la collection.

Début

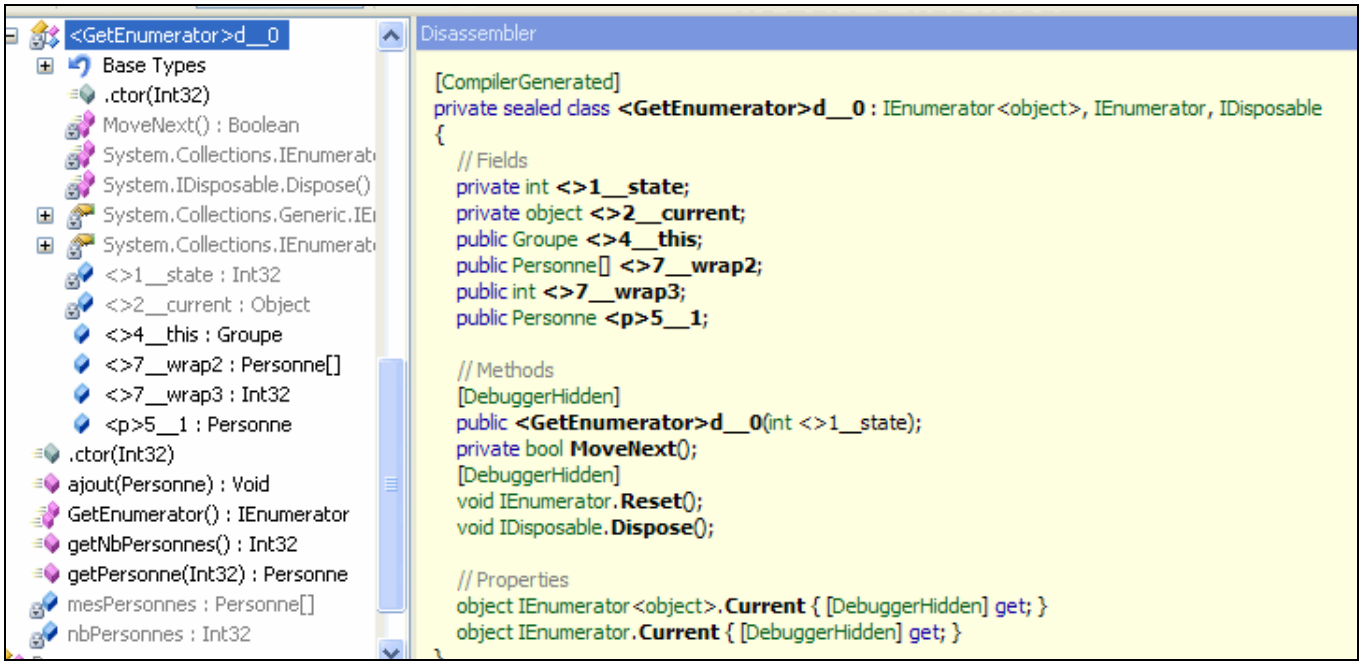
Méthodes publiques

	Nom	Description
	MoveNext	Avance l'énumérateur à l'élément suivant de la collection.
	Reset	Rétablit l'énumérateur à sa position initiale, qui précède le premier élément de la collection.

Pour résumer la situation, une classe conteneur peut bénéficier du `foreach` si elle est énumérable (interface `IEnumerable`) et si elle présente une méthode qui retourne un énumérateur, c'est à dire un mécanisme de parcours des éléments qu'elle veut exposer.
Mais qui construit cet énumérateur ? En fait c'est le framework et ce à partir de l'instruction `yield return`. Pour s'en convaincre, désassemblons le code de `GetEnumerator` (grâce à l'outil `Reflector`).



On peut voir que la méthode `GetEnumerator()` (fenêtre de droite) instancie un objet de la classe `<GetEnumerator>d_0` ; ainsi une classe a été créée automatiquement par le compilateur. Si l'on parcours la fenêtre de gauche on trouve cette classe :



Des méthodes ont été générées, notamment `MoveNext, Reset, Current` ; méthodes demandées dans le contrat de l'interface `IEnumerator` (cf MSDN plus haut).
Il serait par exemple possible aussi de proposer une méthode qui filtre certaines occurrences des personnes :

```
public IEnumerator GetEnumerator()
{
    for (int i = 0; i < this.nbPersonnes; i++)
    {
        Personne p = (Personne)this.mesPersonnes[i];
        if( p.getAge()>21)
            yield return p;
    }
}
```

Remarque : pour la démonstration, on a ajouté un accesseur sur l'âge (`getAge`)
Ceci produira, avec le même `foreach` du `Main` :

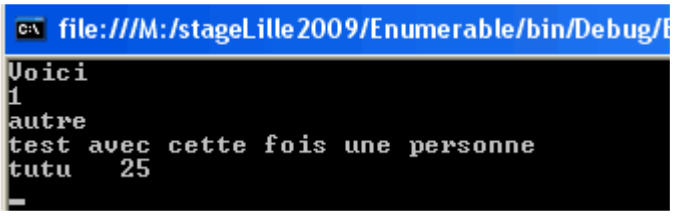
L'instruction `yield return` ne peut figurer que dans des fonctions retournant un `IEnumerator` ou directement un `IEnumerable`. Observons le mécanisme dans ce second cas en dehors de toute classe conteneur. Écrivons une fonction Test (static) :

```
static public IEnumerable test()
{
    yield return "Voici";
    yield return 1;
    yield return "autre";
    yield return "test avec cette fois une personne";
    yield return new Personne("tutu", 25);
}
```

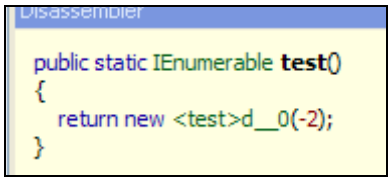
Si nous appelons cette fonction dans le Main, on constate à nouveau ce résultat bien troublant :

```
        foreach (Object o in test())
            Console.WriteLine(o.ToString());
```

Notons au passage que l'on peut itérer sur une fonction puisque celle-ci retourne un IEnumerable !



Désassemblons la fonction :
Comme plus haut, la fonction a un code différent de celui écrit et il y a eu génération d'une classe :



Cette classe dispose de champs qui vont permettre la mémorisation de la valeur et l'état de l'objet (ici chaque valeur "retournée" par `yield return`) courant :

```
private sealed class <test>d__0 : IEnumerable<object>, IEnumerable, IEnumera
{
    // Fields
    private int <>1__state;
    private object <>2__current;

    // Methods
    [DebuggerHidden]
    public <test>d__0(int <>1__state);
    private bool MoveNext();
    [DebuggerHidden]
    IEnumerator<object> IEnumerable<object>.GetEnumerator();
    [DebuggerHidden]
    IEnumerator IEnumerable.GetEnumerator();
    [DebuggerHidden]
    void IEnumerator.Reset();
    void IDisposable.Dispose();

    // Properties
    object IEnumerator<object>.Current { [DebuggerHidden] get; }
    object IEnumerator.Current { [DebuggerHidden] get; }
}
```

La méthode `MoveNext` fait le travail de mise à jour :

```
private bool MoveNext()
{
  switch (this.<>1__state)
  {
    case 0:
      this.<>1__state = -1;
      this.<>2__current = "Voici";
      this.<>1__state = 1;
      return true;

    case 1:
      this.<>1__state = -1;
      this.<>2__current = 1;
      this.<>1__state = 2;
      return true;

    case 2:
      this.<>1__state = -1;
      this.<>2__current = "autre";
      this.<>1__state = 3;
      return true;

    case 3:
      this.<>1__state = -1;
      this.<>2__current = "test avec cette fois une personne";
      this.<>1__state = 4;
      return true;

    case 4:
      this.<>1__state = -1;
      this.<>2__current = new Personne("tutu", 0x19);
      this.<>1__state = 5;
      return true;

    case 5:
      this.<>1__state = -1;
      break;
  }
  return false;
}
```

Noter la dernière affectation dans chaque `case` qui permet d'itérer en avançant pour le prochain appel de `MoveNext`.
Pour terminer ce survol des itérateurs, et pour ceux qui sont allergiques à `yield return`, rien n'interdit d'en rester à une version plus classique de l'itérateur, voici l'itérateur pour la classe `Groupe` :

```
class EnumGroupe : IEnumerator
{
    public EnumGroupe(Groupe g)
    {
        this.g = g;
        this.index = -1;
    }
    public bool MoveNext()
    {
        this.index++;
        return index < g.getNbPersonnes();
    }
    public object Current
    { get { return this.g.getPersonne(index); } }
    public void Reset() { index = -1; }
    private Groupe g;
    private int index;

}
```

Cet itérateur sera bien sûr créé et retourné dans la méthode `GetEnumerator` :

```
public Personne getPersonne(int i){return mesPersonnes[i];}
public IEnumerator GetEnumerator()
{
    EnumGroupe en = new EnumGroupe(this);
    return en;
}
```

TP d'application

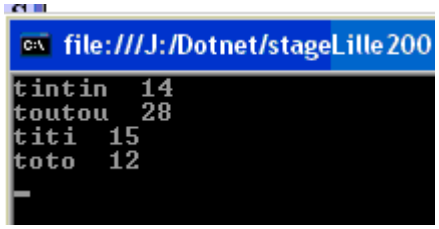
Type générique et itérateur

1. **Écrire une classe générique Pile permettant d'itérer à partir du dernier élément inséré.**

Le code :

```
static void Main(string[] args)
{
  Pile<Personne> p = new Pile<Personne>(4);
  p.depile();
  p.empile(new Personne("toto",12));
  p.empile(new Personne("titi",15));
  p.empile(new Personne("tutu",25));
  p.depile();
  p.empile(new Personne("toutou", 28));
  p.empile(new Personne("tintin", 14));
  p.empile(new Personne("tata", 11));
  foreach (Personne pe in p)
  Console.WriteLine(pe.ToString());
}
```

doit produire :



On vous donne le code de la classe Personne :

```
class Personne
{
   public Personne(string nom, int age)
  {this.nom = nom; this.age=age;}
   public override string ToString()
  {
    return this.nom + " " + this.age.ToString() ;
   }
   private string nom;
   private int age;
}
```

ainsi que celui de la pile générique :

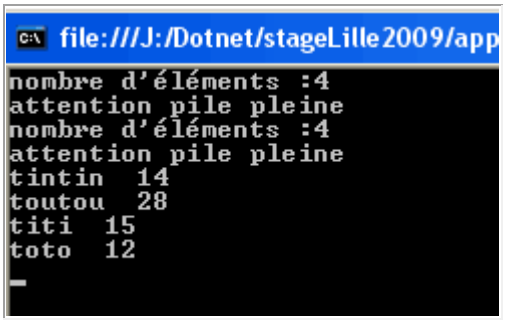
```
class Pile<T>
{
  public Pile(int n)
{
   this.mesT = new T[n];
   this.nbElements = 0;
   }
   public void depile()
  {
     if (this.nbElements > 0)
    this.nbElements--;
   }
   private bool estPLeine()
  {
    return this.mesT.Length == this.nbElements;
    }
    public void empile(T unT)
   {
      if (!this.estPLeine())
      {
    this.mesT[this.nbElements] = unT;
    this.nbElements++;
       }
  }
```

```
        public T leT(int n)
    {
        return this.mesT[n];
      }
     private T[] mesT;
     private int nbElements;
  }
```

Evolution.

2. **Ajouter un événement à la classe Pile qui est généré lorsque la pile est pleine, après l'ajout d'un élément.**

Prévoir un gestionnaire par défaut qui s'exécutera automatiquement. Ainsi la sortie suivante :



Il y a eu deux réalisations de l'événement : aux insertions de "Tintin" et "Tata" la première ligne (nombre d'éléments : 4) correspond au gestionnaire par défaut de la classe Pile et le second ("attention pile pleine") correspond à un gestionnaire ajouté.

Pour cela, créer un `delegate` (portée `namespace`) sans argument et ne retournant rien. Déclarer une fonction déléguée dans la classe `Pile` ; créér le gestionnaire d'événement par défaut dans la classe `Pile`, ajouter-le à la fonction déléguée, appeler la fonction déléguée au moment désiré (sur l'ajout quand la pile est pleine). Créer un autre gestionnaire d'événement dans le programme que vous ajoutez à l'événement de l'instance de la pile dans le main. C'est tout :-)

4) Déclaration implicite de type

La version 3.0 propose une simplification de la syntaxe de déclaration des variables ; il pouvait sembler redondant (dans la majorité des cas) de faire le type de déclaration et d'initialisation suivants : `Personne P = new Personne().`
Le langage propose désormais d'utiliser le mot `var` :

```
var p = new Personne();
```

Rien de révolutionnaire là, le compilateur déduira de l'initialisation le type de l'objet. Ceci procure un confort d'utilisation lorsque notamment le type de retour d'une fonction n'est pas évident, par contre :
- Le mot `var` doit être suivi de l'initialisation `(new)` ou de la valeur `( var s = "toto";)`
- Le type est défini à la compilation et ne peut être modifié ensuite
- Il est possible bien sûr (et conseillé dans la majorité des cas) de continuer à typer explicitement les variables dans la très grande majorité des situations.

5) Méthode d'extension

Le framework 3 propose d'enrichir des classes existantes par des méthodes dites "d'extension" ; cette ouverture n'est faite que pour des raisons techniques, liées au désir de MS de ne pas modifier des classes (pour la mise en oeuvre de `linq`, cf plus loin) en étendant des fonctionnalités.

La syntaxe doit utiliser une classe statique (sans constructeur et constituée uniquement de méthodes statiques) :

```
static class ExtensionClass
{
    public static int doubler( this int n)
    {
        return 2*n;
    }
}
```

Noter l'utilisation très particulière ici de `this`.
L'appel à cette extension (sur les int) se fait tout naturellement ainsi :

```
int n = 5;
Console.WriteLine(n.doubler());
```

TP d'application

Méthode d'extension

3. **Écrire une méthode d'extension `retireCar` de la classe `string` qui permet de retirer un caractère d'une chaîne :**

```
string s = "voiture";
  Console.WriteLine( s.retireCar('t'));
```

affiche *voiure*

6) Les méthodes anonymes et les expressions lambda

6.1 Les méthodes anonymes.

La version 2.0 du framework a introduit la notion de méthode anonyme ; comme son nom l'indique, il s'agit d'une méthode qui n'a pas de nom, par contre les méthodes anonymes ne concerne que les délégués.

Reprenons notre exemple : `Pile<T>`. Nous avions ajouté un `delegate` au niveau du `namespace` (cf tp : `delegate void deleg();`) afin de gérer l'événement `pilePleine`. Dans le constructeur de la pile, nous abonnions à l'événement un gestionnaire (par défaut) :

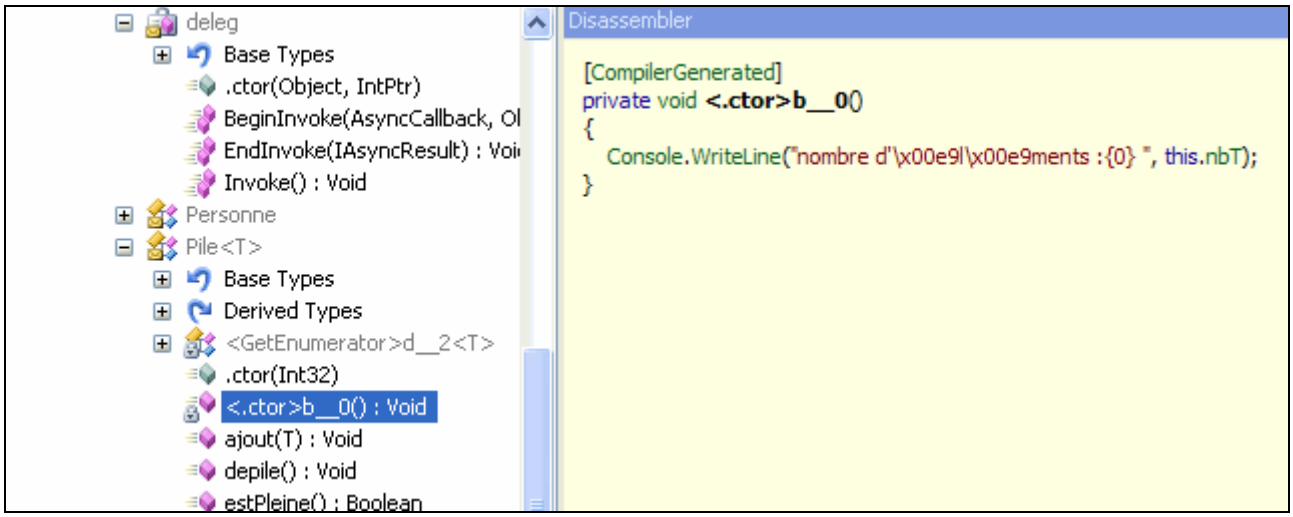
```
 class Pile<T> : IEnumerable
{
      private T[] mesT;
      private int nbT = 0;
      public deleg evtPilePleine ;
      public Pile(int n)
      {
          mesT = new T[n];
          evtPilePleine = evtInitPilePleine;
      }
      public void evtInitPilePleine()
      {
          Console.WriteLine("nombre d'éléments :{0} ", this.nbT);
      }
```

Ainsi, il est possible de ne pas déclarer le gestionnaire d'événement `evtInitPilePleine` en utilisant une méthode anonyme au moment de l'inscription :

```
public Pile(int n)
{
    mesT = new T[n];
    evtPilePleine = delegate()
    {
      Console.WriteLine("nombre d'éléments :{0} ", this.nbT, nbAppels);
    };
```

Le délégué pointe sur une fonction sans nom, dont le code est présenté entre des accolades classiques. L'annonce en est faite grâce au mot `delegate` jouant pour cette occasion un rôle un peu différent. Comment ceci est-il possible ? C'est grâce au compilateur qui va s'occuper de générer explicitement une méthode conforme à la signature du `delegate`. Le compilateur se comporte comme pour l'instruction `yield return` : il allège le travail du développeur en générant automatiquement le code attendu.

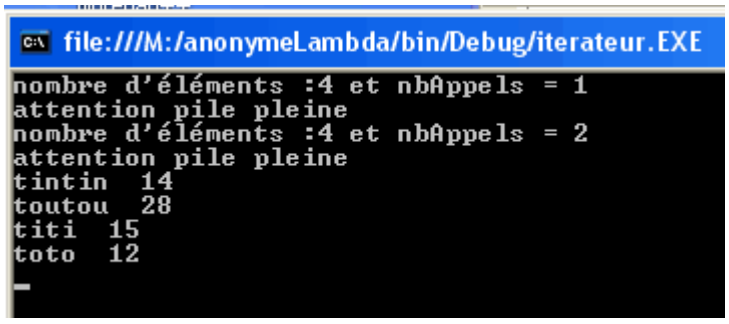
Regardons ce que le compilateur à généré à l'aide de l'outil Reflector :



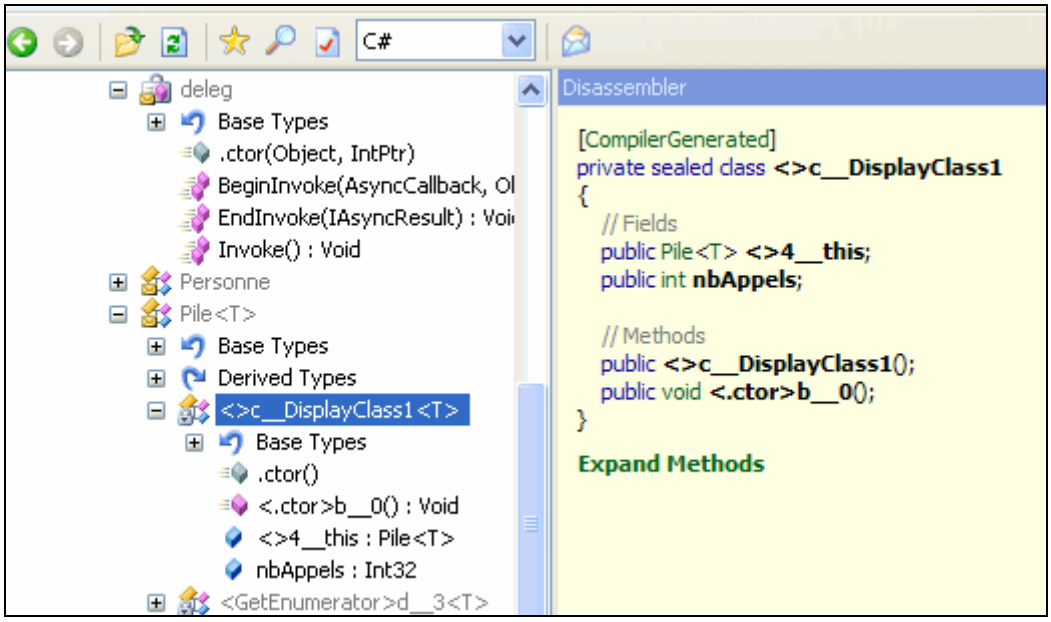
Une nouvelle méthode, générée par le compilateur (attribut `CompilerGenerated`) est visible ; son code est identique à la description `inline` de la méthode.
Le mécanisme semble simple : on peut déclarer `inline` le corps d'une méthode déléguée, le compilateur se charge de créer la bonne méthode. Mais l'intervention du compilateur n'en reste pas là : modifions très légèrement le code `inline`.

```
public Pile(int n)
{
    mesT = new T[n];
    int nbAppels = 0;
    evtPilePleine = delegate() { nbAppels++;
                                 Console.WriteLine("nombre d'éléments :{0} et nbAppels = {1} ", this.nbT, nbAppels);
                                     }
```

On déclare une variable (locale au constructeur) et on demande au délégué d'afficher à chacun de ses appels cette variable incrémentée. Non seulement le compilateur accepte ce code (troublant au niveau de la portée de `nbAppels`) mais il propose un résultat pour le moins déroutant !



L'incrémentation se fait alors que `nbAppels` (variable locale) est détruite à la fin de l'exécution du constructeur !!
`nbAppels` semble détruit...pour le constructeur mais pas pour tout le monde !!
Replongeons-nous dans le code généré par le compilateur:



Le compilateur a créé une classe cette fois (et pas seulement une méthode), `nbAppels` en est devenu un champ ! Chaque nouvel appel de l'événement entraîne une incrémentation du champ `nbAppels`.
Ainsi, selon le contexte de l'appel de la méthode inline, le compilateur génère une méthode ou une classe.

6.2 Les expressions lambda

Une expression lambda peut être utilisée lorsqu'un délégué est normalement nécessaire ; ceci permet de simplifier l'écriture de la fonction. Une expression lambda comprend :
- Des paramètres
- Le signe =>
- Une expression résultat.

Quelques expressions :

```
( n) => n%2
( a, b) => a+b
```

Mise en œuvre :

Définition du type délégué voulu	`delegate bool deleg2(int a,int b);`
Déclaration d'un délégué conforme	`deleg2 d2 = (a, b) => a % b==0;`
Appel du délégué	`if( d2(12, 6))` `  Console.WriteLine("multiple");`

Il peut sembler pénible d'avoir à définir le type `deleg2` ; C# propose un type `delegate` générique (Func) qui allège l'écriture :

Utilisation de Func et déclaration d'un délégué conforme	`Func<int, int, bool> d2 = (a, b) => a % b == 0;`
Appel du délégué	`if( d2(12, 6))` `  Console.WriteLine("multiple");`

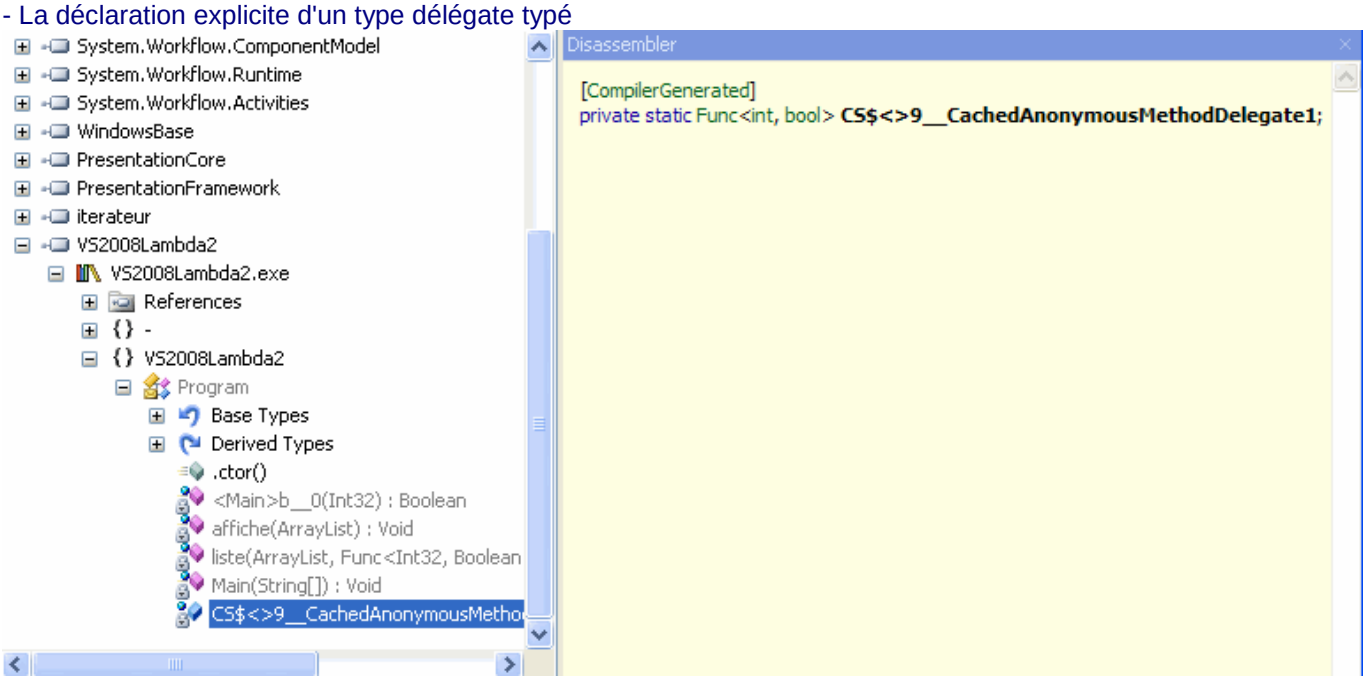
Dans ces deux exemples, l'utilisation d'un délégué n'est bien sûr pas nécessaire pour faire le traitement attendu.
Construisons un exemple où l'utilisation des expressions lambda trouvent plus de sens et de vertu :
L'objectif sera de proposer des filtres sur une liste de nombres (stockée dans une `ArrayList`)

```
class Program
{

    static void Main(string[] args)
    {
        ArrayList  desInts = new  ArrayList();
        for (int i = 1; i <= 100; i++) { desInts.Add(i); }  // construction des valeurs
        affiche( filtre(desInts,(a) => a % 9 == 0));

    }
    static ArrayList filtre(ArrayList ar, Func<int , bool> f)
    {
        ArrayList lesInts = new  ArrayList();
        foreach (int i in ar)
            if (f((int)i))              // appel de la fonction déléguée (f)pour chaque valeur de la liste
                lesInts.Add(i);
        return lesInts;
    }
    static void affiche(ArrayList ar)
    {
        foreach (int i in ar)
            Console.WriteLine(i.ToString());
    }
}
```

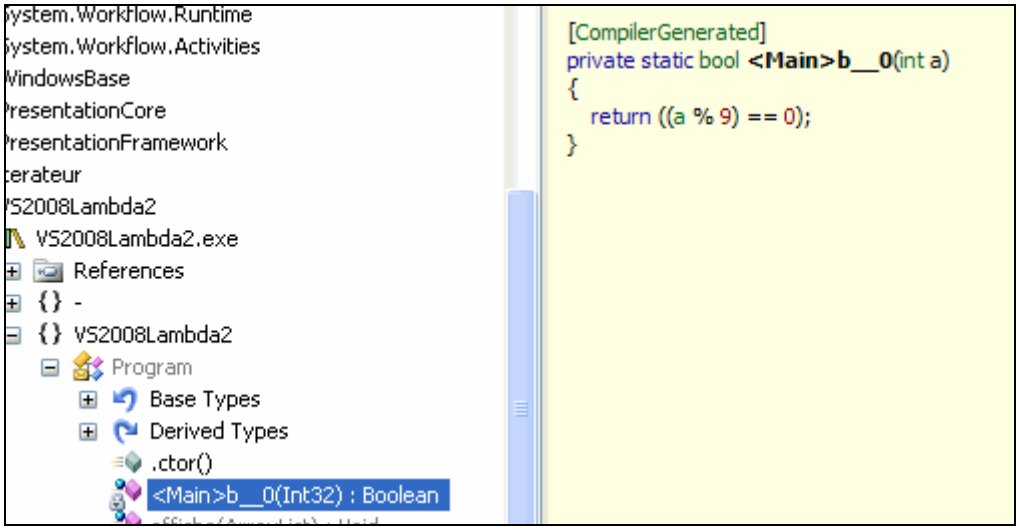
L'appel `filtre(desInts,(a) => a % 9 == 0)` utilise une expression lambda pour filtrer les multiples de 9.

Si nous demandons à Reflector de montrer le code généré par le compilateur, deux choses à noter :

- La déclaration explicite d'un type délégate typé

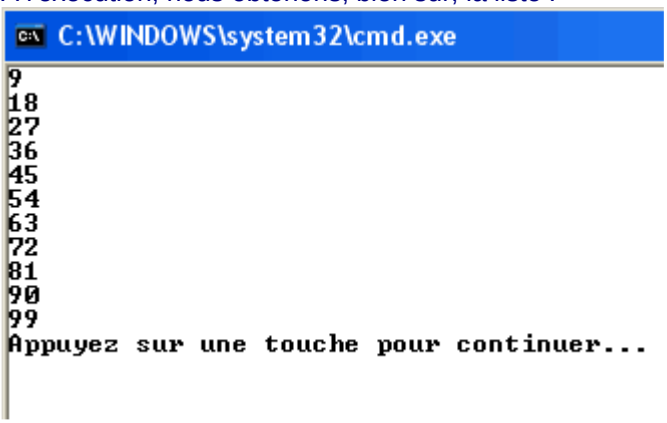


- La définition de la fonction déléguée conforme au `delegate` :



Les expressions lambda sont des facilités offertes au développeur affranchi des déclarations des `delegate` et d'un code impératif.

A l'exécution, nous obtenons, bien sûr, la liste :



Nous pourrions proposer d'autres filtres :

```
static void Main(string[] args)
{
    ArrayList  desInts = new  ArrayList();
    for (int i = 1; i <= 100; i++) { desInts.Add(i); }  // construction des valeurs
    affiche( filtre(desInts,(a) => a >20));
```

Répétons-le encore une fois, l'expression lambda peut être remplacée par du code plus traditionnel ; il serait néanmoins dommage de ne pas profiter de la puissance du compilateur ici.

TP d'application

1.1 Il serait intéressant de proposer le filtre comme méthode d'extension de l' `ArrayList`.

1. **En s'inspirant du code présent dans le support, modifier afin de pouvoir écrire dans le Main :**

```
ArrayList desInts = new ArrayList();
for (int i = 1; i <= 100; i++) { desInts.Add(i); } // construction des valeurs
affiche(desInts.filtre((a) => a % 7 == 0)); // filtre devient une méthode
d'extension de l'ArrayList
```

1.2 Filtrons les capitales

A partir d'une ArrayList initialisée dans le Main :

```
ArrayList capitales = new ArrayList()
{"Paris","Madrid","Londres","Rome","Genève","Dublin","Moscou","Zurich","Prague"
};
```

1. **En s'inspirant du code précédemment vu, écrire un filtre qui permet d'obtenir (par exemple) les capitales qui commence par "P", ou "Ma", ou se termine par "e" ou "ve" ou contiennent "dr" ou pas de "o" ...**

Le filtre ne sera pas une méthode d'extension.

1.3 Filtre générique

Lors des derniers tp, un filtre est utilisé, une fois pour des `int` ou autre fois pour des `string` ; il serait intéressant de proposer une seul filtre générique.

2. **Écrire cette nouvelle fonction filtrer générique ; proposer une version sans méthode d'extension et une avec une méthode d'extension de l'ArrayList.**

7) Les initialiseurs d'objet

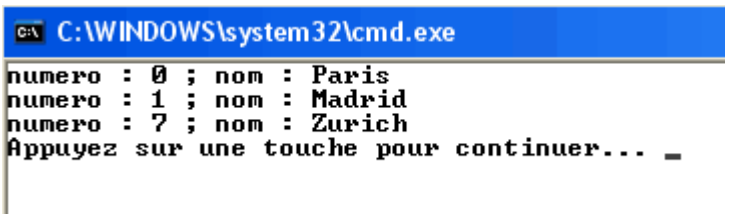
Le dernier point abordé porte sur l'initialisation d'objet, sans déclarer explicitement de classe ; modifions un peu le code du tp "capitales" :

```
static void afficheListeFiltree(ArrayList ar, Func<string, bool> f)
{
    for (int i = 0; i < ar.Count; i++)
         if (f((string)ar[i]))              // appel de la fonction déléguée (f)pour chaque valeur de la liste
         {
             var cap = new { num = i, lib = ar[i] };
             Console.WriteLine("numero : {0} ; nom : {1}", cap.num, cap.lib);
         }
}
```

La ligne surlignée utilise les types anonymes (mot `var`) et initialise un objet cap, sans classe associée. Ceci permet de "récupérer" les champs (num, lib) comme pour une classe "normale". L'appel suivant :

```
ArrayList capitales = new ArrayList() { "Paris", "Madrid", "Londres", "Rome", "Genève", "Dublin", "Moscou", "Zurich", "Prague" };
afficheListeFiltree(capitales,s =>s.Contains("ri"));
```

produira :



B- Le langage LINQ TO Objects

1) Découverte de LINQ (Language INtegrated Query)

Nous avons vu que le framework permettait d'interroger des structures linéaires (tableau, ArrayList...) en utilisant des expressions Lambda (cf la méthode filtrer).
Ainsi, ce que nous avons construit "à la main" est proposé par défaut pour les types tableau et List ; par contre pas pour les ArrayList. Ceci s'utilise avec la méthode d'extension Where.

```
int[] t= new int[100];
for (int i = 0; i < 100; i++) { t[i]=i; }
var desInts = t.Where((n) => n % 2 == 0);
```

- Where est une méthode d'extension pour les tableaux
- Where prend comme seul argument une expression lambda (ou un délégué)
- Le type de retour est une interface générique IEnumerable<T> ; l'utilisation d'un type anonyme (var) allège ici l'écriture !.

Ensuite il faudra parcourir la liste IEnumerable obtenu :

```
foreach (int i in desInts) { Console.WriteLine(i); }
```

On peut remarquer que le type générique a été transformé en <int> ; le compilateur a inféré le type réel.
Mais parallèlement aux expressions lambda, le Framework propose un langage de requétage spécifique. Ecrivons la même "requête" en Linq :

```
var desIntsVersionLinq = from n in t
                         where n %2 ==0
                         select n;
foreach (int i in desIntsVersionLinq) { Console.WriteLine(i); }
```

Nous sommes dans un monde plus connu !! Notons néanmoins l'ordre inattendu des instructions.
Si nous regardons le code généré par le compilateur dans les deux versions :

```
int[] t= new int[100];
for (int i = 0; i < 100; i++) { t[i]=i; }
var desInts = t.Where((n) => n % 2 == 0);
foreach (int i in desInts) { Console.WriteLine(i); }
var desIntsVersionLinq = from n in t
                         where n %2 ==0
                         select n;
foreach (int i in desIntsVersionLinq) { Console.WriteLine(i); }
```

nous constatons que c'est strictement le même code :

```
int[] t = new int[100];
for (int i = 0; i < 100; i++)
{
    t[i] = i;
}
IEnumerable<int> desInts = t.Where<int>(delegate (int n) {
    return (n % 2) == 0;
});
foreach (int i in desInts)
{
    Console.WriteLine(i);
}
IEnumerable<int> desIntsVersionLinq = t.Where<int>(delegate (int n) {
    return (n % 2) == 0;
});
foreach (int i in desIntsVersionLinq)
{
    Console.WriteLine(i);
}
}
```

Ainsi, linq propose une surcouche de requêtage plus naturelle et présentant le double avantage :
- Simplicité (relative :-)) du requêtage et vérification syntaxique à la compilation et non à l'exécution.
- Bénéfice de l'intellisence du code à l'intérieur du requêtage.

2) Des éléments de syntaxe

Utilisons pour cela une liste plus fournie :

Reprenons notre classe Personne :

```
class Personne
{
    public Personne(string nom, int age)
    { this.nom = nom; this.age = age; }
    public string getNom() { return this.nom; }
    public int getAge() { return this.age; }
    private string nom;
    private int age;
}
```

Construisons une liste de Personne :

```
List<Personne> personnes = new List<Personne>{new Personne("Dupond",15), new Personne("Durand",25),new Personne("Rami",18),
                    new Personne("Paton",24),new Personne("Harvis",11),new Personne("Breton",32),new Personne("Perdrix",19),
                    new Personne("Rapport",19),new Personne("Boissette",28),new Personne("Pamelle",45),new Personne("Fétard",15)};
```

Menons des requêtes linq :

```
var desPersonnes = from p in personnes
                   where p.getAge() > 18
                   select p.getNom();
foreach (string  s in desPersonnes) { Console.WriteLine(s); }
```

Ce qui produit :

```
C:\WINDOWS\system32\cmd.exe
Durand
Paton
Breton
Perdrix
Rapport
Boissette
Pamelle
Appuyez sur une touche pour continuer...
```

Quelques requêtes et leur effet :

```
var desPersonnes = from p in personnes
                   where p.getAge() > 30
                   && p.getNom().StartsWith("Pa")
                   select p.getNom();
```

```
C:\WINDOWS\system32\cmd.exe
Pamelle
Appuyez sur une touche pour continuer.
```

```
var desPersonnes = from p in personnes
                   where p.getAge() > 18
                   orderby p.getNom()
                   select p.getNom();
```

```
C:\WINDOWS\system32\cmd.exe
Boissette
Breton
Durand
Pamelle
Paton
Perdrix
Rapport
Appuyez sur une touche pour continuer...
```

```
int nb = (from p in personnes
          where p.getAge() > 18
          select p).Count();
```

```
C:\WINDOWS\system32\c
7
Appuyez sur une touche
```

```
var desPersonnes = from p in personnes
                   where p.getNom().Length > 6
                   select p.getNom();
```

```
C:\WINDOWS\system32\cmd.exe
Perdrix
Rapport
Boissette
Pamelle
Appuyez sur une touche pour continuer...
```

```
var desPersonnes = from p in personnes
                   where p.getNom().Length > 6
                   select new (leNom = p.getNom(),lAge = p.getAge());

foreach (var s in desPersonnes)
{ Console.WriteLine("nom : {0} age : {1}",s.leNom,s.lAge); }
```

```
C:\WINDOWS\system32\cmd.exe
nom : Perdrix age : 19
nom : Rapport age : 19
nom : Boissette age : 28
nom : Pamelle age : 45
Appuyez sur une touche pour continuer
```

```
var desPersonnes = from p in personnes
                   where p.getAge() == personnes.Min((n)=>n.getAge())
                   select new (leNom = p.getNom(),lAge = p.getAge());

foreach (var s in desPersonnes)
{ Console.WriteLine("nom : {0} age : {1}",s.leNom,s.lAge); }
```

```
C:\WINDOWS\system32\cmd.exe
nom : Harvis age : 11
Appuyez sur une touche pour continuer...
```

note : récupère le(s) plus jeune(s)

TP d'application

En copiant le code :

```
List<Personne> personnes = new List<Personne>{new Personne("Dupond",15), new
Personne("Durand",25),new Personne("Rami",18),
new Personne("Paton",24),new Personne("Harvis",11),new Personne("Breton",32),new
Personne("Perdrix",19),
new Personne("Rapport",19),new Personne("Boissette",28),new
Personne("Pamelle",45),new Personne("Fétard",15)};
```

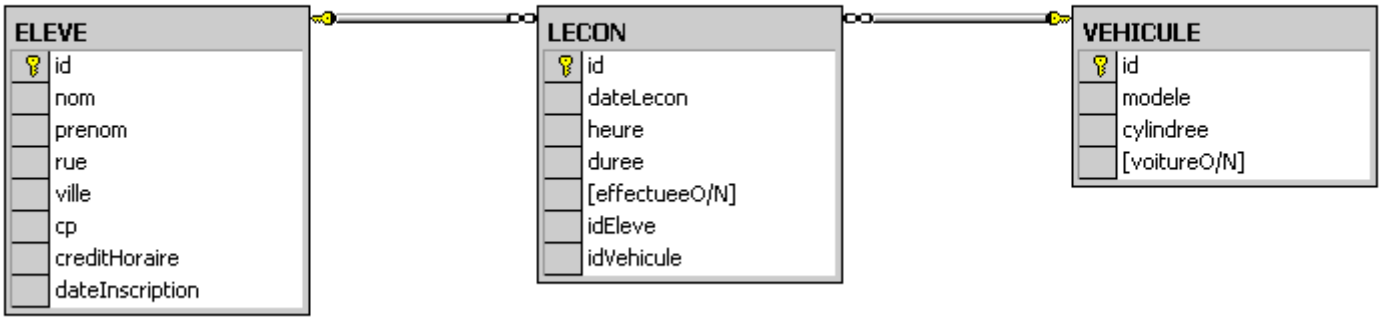
1. **Écrire les requêtes `Linq` qui affichent :**
   a) Le nom des personnes dont le nom contient ""to"
   b) L'age moyen
   c) Le nom et l'age de la (ou des) personne(e) la plus agée
   d) Le nom et l'age de la (ou des) personne(s) dont le nom est le plus court

C- Entity Framework

La bibliothèque de classes Entity Framework propose un mécanisme de mapping entre un modèle relationnel et un modèle objet. Le framework utilise le langage Linq pour interroger les données présentes dans les classes. Ce Framework est disponible avec VS 2008 et son service pack 1 (VS 2008 + SP1 VS 2008) ; ceci correspond au framework dotnet 3.5.

1) Etude d'un exemple : école de conduite

On dispose d'un modèle de données simple de gestion de cours de conduite, auto ou moto:

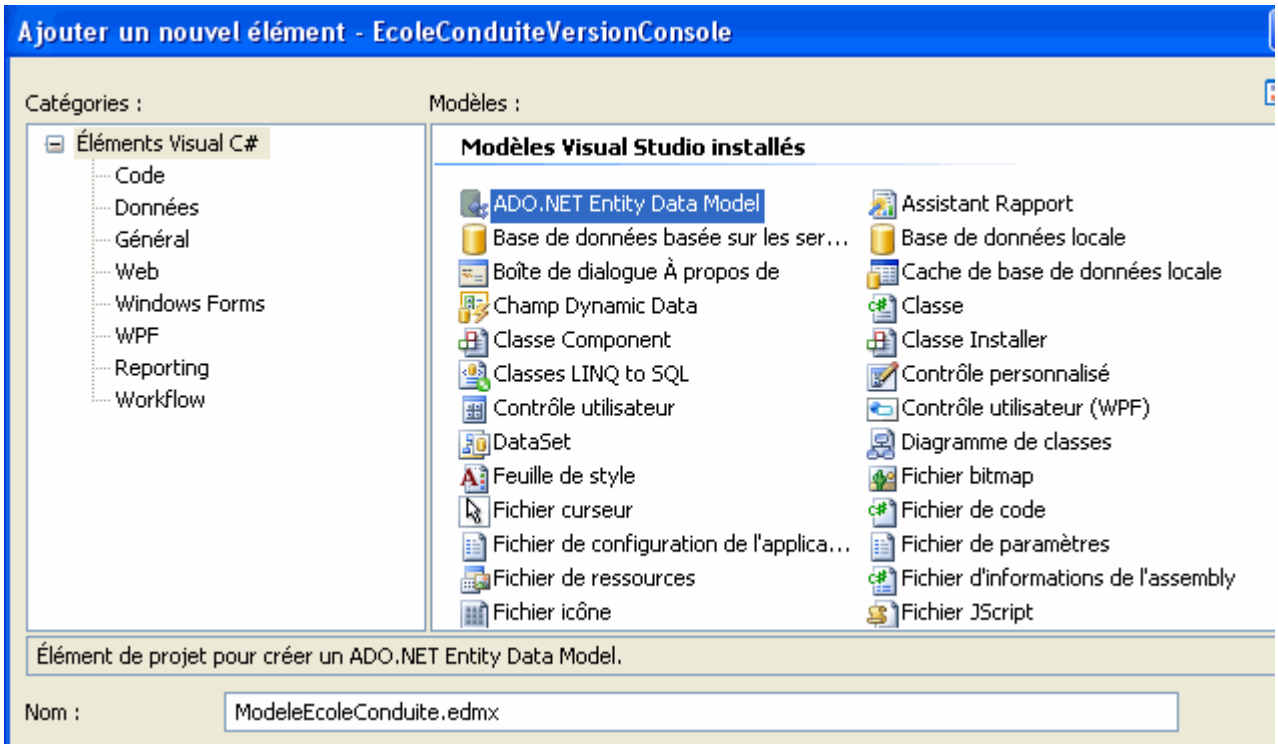


- Une leçon concerne un véhicule, auto ou moto (si c'est une voiture, on connait son modèle, sinon la cylindrée de la moto)
- Lorsqu'une leçon est éffectuée, le crédit horaire de l'élève concerné est décrémenté.

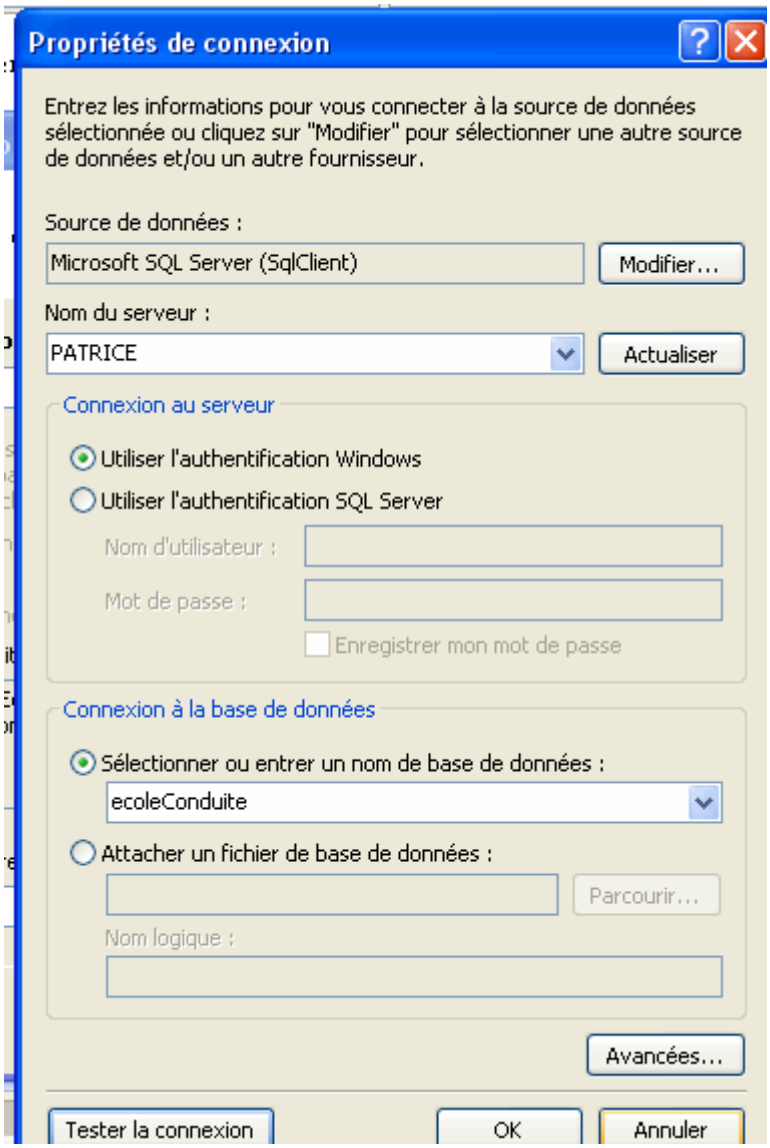
La base de donnée est sous SQL Server et dispose d'une procédure stockée pour l'insertion d'un nouvel élève. Récupérer la base à restaurer (fichier joint).

2)Prise en main en mode console

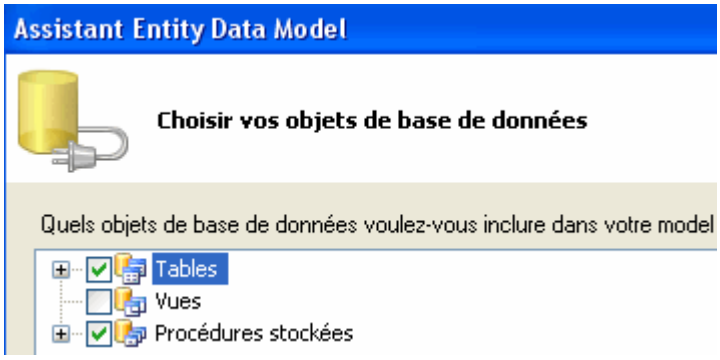
Créons un nouveau projet, de type console. Ajoutons à ce projet un nouvel élément, de type Entity Data Model -EDM- :



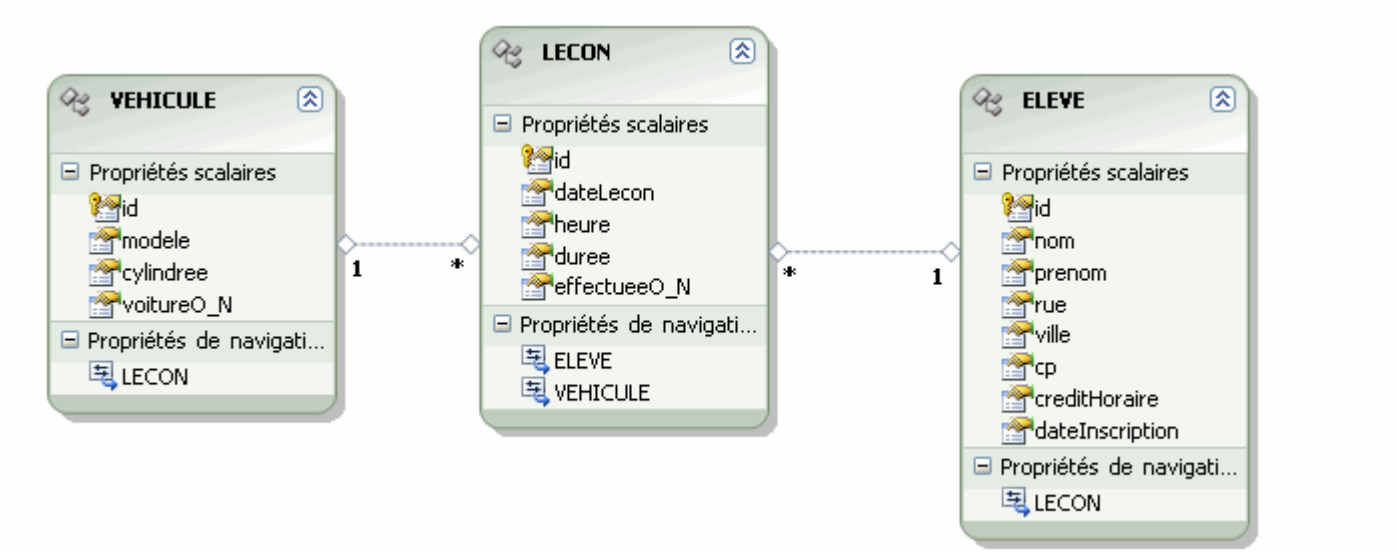
C'est ce modèle, généré par Visual Studio, qui va piloter le mapping. Créons une nouvelle connexion :



Sélectionnons la base de données (ecoleConduite)
Indiquons que nous importons les tables et procédures stockées :



Après avoir terminé la configuration de la connexion on peut observer le modèle de classe généré :



Remarque : si le modeleur ajoute une classe dtProperties, supprimez-la du modèle visuel, il s'agit d'on bogue référencé.

2.1Code généré

Le diagramme est de "type" uml (champ, cardinalités) ; observons les classes générées :
- Une classe principale, dérivant d'ObjectContext:

```
/// </summary>
public partial class ecoleConduiteEntities : global::System.Data.Objects.ObjectContext
{
    /// <summary>
    /// Initialise un nouvel objet ecoleConduiteEntities à l'aide de la chaîne de conne
    /// </summary>
    public ecoleConduiteEntities() :
            base("name=ecoleConduiteEntities", "ecoleConduiteEntities")
```

C'est à partir de cette classe que les différents traitements de mapping seront effectués.
- Une classe par table, dérivant d'EntityObject :

```
public partial class ELEVE : global::System.Data.Objects.DataClasses.EntityObject
{
    /// <summary>
    /// Créez un nouvel objet ELEVE.
    /// </summary>
    /// <param name="id">Valeur initiale de id.</param>
    /// <param name="nom">Valeur initiale de nom.</param>
    /// <param name="creditHoraire">Valeur initiale de creditHoraire.</param>
    /// <param name="dateInscription">Valeur initiale de dateInscription.</param>
    public static ELEVE CreateELEVE(int id, string nom, int creditHoraire, global::Syste
    {
        ELEVE eLEVE = new ELEVE();
        eLEVE.id = id;
        eLEVE.nom = nom;
        eLEVE.creditHoraire = creditHoraire;
        eLEVE.dateInscription = dateInscription;
        return eLEVE;
    }
```

Le modèle de mapping est de type 1-1 (une table => une classe), la construction des instances propose une méthode static CreateEleve, mais attention seuls les champs Not Null de la base sont présents dans les arguments ici !
La génération automatique distingue la notion de champ -privé-, par exemple _id, de celle de propriété -publique-, par exemple id.
La déclaration partial permettra de surcharger la classe sans intervenir dans le fichier généré.
Remarque : le fait que la classe Eleve hérite d'une classe technique témoigne d'une forte adhérence entre les "classes métiers" et les classes techniques ; on pourrait néanmoins s'affranchir de cette dépendance en implémentant les (nombreuses...) interfaces dont dépend EntityObject.
- Des propriétés de navigation dans les deux sens.
Par exemple la classe LECON contient une référence sur un Elève (de nom ELEVE) et sur un véhicule (de nom VEHICULE). La classe ELEVE contient une collection de leçons :
public global::System.Data.Objects.DataClasses.EntityCollection<LECON> LECON
Il est possible de modifier le nom des propriétés de navigation -cf plus loin-

2.2 Exemples d'utilisation des classes

2.2.a Opérations ajout/modification/suppression

2.2.a.1 Ajout

Nous allons commencer par ajouter un nouvel élève :

```
static void Main(string[] args)
{
    ecoleConduiteEntities monModele = new ecoleConduiteEntities();  //ligne 1
    ELEVE e = ELEVE.CreateELEVE(135, "Durand", 23, DateTime.Today); // ligne 2
    monModele.AddToELEVE(e);                                        // ligne 3
    monModele.SaveChanges();                                        // ligne 4
```

La ligne 1 crée une instance de la classe centrale (ObjectContext) du FrameWork
La ligne 2 utilise le "constructeur statique"
La ligne 3 ajoute l'instance au contexte
La ligne 4 sauve en base le modèle.
Si nous ouvrons l'explorateur de serveur et la table elève, on vérifie l'insertion :

130	Ramon	Majoub	21 rue Haute ...	Aulnay ...	93600	23	02/03/2009 00:...
134	Farci	Johann	45 rue Albert ...	Montreuil ...	93100	18	23/01/2009 00:...
135	Durand	*NULL*	*NULL*	*NULL*	*NULL*	23	15/06/2009 00:...

2.2.a.2 Modification

Modifions le crédit horaire du premier élève :

```
ELEVE e = monModele.ELEVE.First();
Console.WriteLine(e.creditHoraire);
e.creditHoraire--;
monModele.SaveChanges();
```

L'affichage indique le crédit courant :

L'ouverture de la table confirme la mise à jour :

id	nom	prenom	rue	ville	cp	creditHoraire
124	Martin	Jean	23 rue Blanche ...	Montreuil ...	93100	24
128	Moisin	Annie	4 rue Péri ...	Romainville ...	93230	20

2.2.a.3 Suppression

Pour supprimer la première leçon :

```
monModele.DeleteObject(monModele.LECON.First());
monModele.SaveChanges();
```

Attention : la suppression entraîne la suppression en cascade des objets enfants.

2.2.b Chargement des données en mémoire.

Pour charger des données du contexte, on peut, soit utiliser le langage Linq (ce que nous ferons couramment) soit utiliser les expressions lambda :

```
/* en utilisant Linq*/
 var req = from el in monModele.ELEVE
           where el.id == 135
           select el;
 ELEVE e1 = req.First();
 Console.WriteLine(e1.nom);

 /* en utilisant les expressions lambda*/
 ELEVE e2 = monModele.ELEVE.First((ELEVE el) => el.id == 135);
 Console.WriteLine(e2.nom);
```

Remarque : la définition de la requête req n'entraîne pas son exécution, seule l'accès par une méthode spécifique (ici First) charge en mémoire le résultat de la requête.
Si une requête retourne plusieurs occurrences, on utilise la méthode ToList de la requête et on itère sur le résultat:

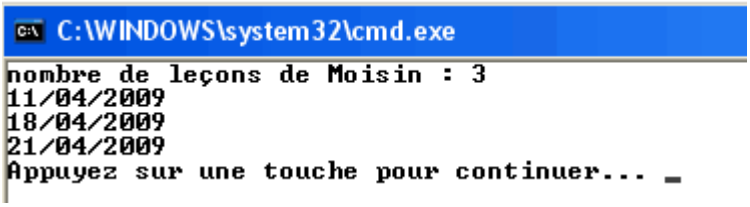
```
var req = from v in monModele.VEHICULE
          where v.cylindree > 500
          select v;
var liste = req.ToList();
foreach (VEHICULE v in liste)
    Console.WriteLine(v.id + " " + v.cylindree.ToString());
```

Remarque : c'est ici la méthode ToList qui appelle l'exécution de la requête mais on pourrait aussi demander l'itération directement sur l'objet req car req est de type *IQueryable* itérable avec foreach.

```
C:\WINDOWS\system32\cmd.exe
123POI93   750
1245GH93   750
Appuyez sur une touche pour continuer... _
```

Le langage Linq permet ainsi de nous affranchir de jointures chères à SQL :

```
var req2 = from l in monModele.LECON
           where l.ELEVE.nom == "Moisin"
           select l;
Console.WriteLine("nombre de leçons de Moisin : {0}", req2.Count());
var liste = req2.ToList();
foreach (LECON l in liste)
    Console.WriteLine(l.dateLecon.ToShortDateString());
```



Mais, pas de miracle néanmoins, le code suivant ne peut fonctionner (même si le compilateur ne signale pas d'erreur):

```
foreach (LECON l in liste)
    Console.WriteLine(l.VEHICULE.id);
```

En effet, la requête Linq est traduite en SQL ; foreach ne peut itérer sur des données inexistantes (ici les informations du véhicule) !!
Pour s'en convaincre, voici la requête exécutée :

```
SELECT
1 AS [C1],
[Extent1].[id] AS [id],
[Extent1].[dateLecon] AS [dateLecon],
[Extent1].[heure] AS [heure],
[Extent1].[duree] AS [duree],
[Extent1].[effectueeO/N] AS [effectueeO/N],
[Extent1].[idEleve] AS [idEleve],
[Extent1].[idVehicule] AS [idVehicule]
FROM  [dbo].[LECON] AS [Extent1]
INNER JOIN [dbo].[ELEVE] AS [Extent2] ON [Extent1].[idEleve] = [Extent2].[id]
WHERE N'Moisin' = [Extent2].[nom]
```

2.2.c Chargement des objets connexes

Pour charger les données connexes (celles qui sont atteignables grâce aux propriétés de navigation) il fait explicitement l'indiquer dans la reqête l'insertion souhaitée, ainsi :

```
var req2 = from e in monModele.ELEVE.Include("LECON")
           where e.nom == "Moisin"
           select e;
var liste = req2.ToList();
ELEVE el =liste.First();
foreach (var l in el.LECON)
    Console.WriteLine("date : {0} heure : {1}", l.dateLecon.ToShortDateString(), l.heure);
```

C'est la méthode Include qui demande l'insertion des leçons de l'élève (rappel : LECON est le nom par défaut de la propriété de navigation élève => leçon). Nous pourrions "plonger" plus loin dans l'insertion en réutilisant la méthode Include : from e in monModele.ELEVE.Include("LECON").Include(...
Après avoir récupéré l'élève, on peut parcourir ses leçons grâce à foreach sur les leçons de l'élève ; le résultat est sans surprise :

```
C:\WINDOWS\system32\cmd.exe
date : 11/04/2009 heure : 19
date : 18/04/2009 heure : 11
date : 21/04/2009 heure : 11
Appuyez sur une touche pour continuer...
```

Voici une seconde version, un peu différente dans le chargement des données, qui utilise la méthode *Find* :

```
var req2 = from e in monModele.ELEVE.Include("LECON")
           select e;
Console.WriteLine(req2.getRequeteSQL());
var liste = req2.ToList();
ELEVE el = liste.Find((ELEVE e) => e.id == 128);
foreach (var l in el.LECON)
    Console.WriteLine("date : {0} heure : {1}", l.dateLecon.ToShortDateString(), l.heure);
```

Ici, toutes les lignes de la table ELEVE sont chargées en mémoire (ainsi que les leçons associées) ; la méthode Find sur la liste obtenue permet d'extraire, par une expression lambda, l'élève voulu. La méthode Find s'utilise à partie de la clé de la table -id ==128-
On pourrait également utiliser une jointure pour charger explicitement des données connexes
:

```
var req2 = from l in monModele.LECON
           from v in monModele.VEHICULE
           where l.ELEVE.nom == "Moisin"
           && l.VEHICULE.id == v.id
           select new { date = l.dateLecon, imma = v.id };

Console.WriteLine(req2.getRequeteSQL());
var liste = req2.ToList();
foreach (var v in liste)
    Console.WriteLine("date : {0} imma : {1}", v.date.ToShortDateString(), v.imma);
```

Remarque : noter l'emploi des classe anonyme pour récupérer les champs (select new {...)

Voici la jointure générée :

```
SELECT
1 AS [C1],
[Extent1].[dateLecon] AS [dateLecon],
[Extent1].[idVehicule] AS [idVehicule]
FROM  [dbo].[LECON] AS [Extent1]
INNER JOIN [dbo].[ELEVE] AS [Extent2] ON [Extent1].[idEleve] = [Extent2].[id]
WHERE N'Moisin' = [Extent2].[nom]
```

et le résultat obtenu :

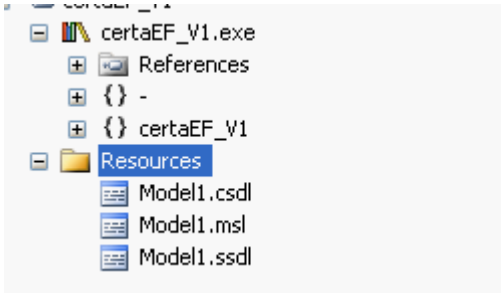
```
date : 11/04/2009 imma : 1245GH93
date : 18/04/2009 imma : 1245GH93
date : 21/04/2009 imma : 123POI93
Appuyez sur une touche pour continuer...
```

Remarque : pour visualiser les requêtes SQL exécutées, on écrit et utilise ici une méthode d'extension de l'interface IQueryable:

```
public static class ObjectQueryExtension
{
    public static string getRequeteSQL<T>(this IQueryable<T> query)
    {
        return ((ObjectQuery<T>)query).ToTraceString();
    }
}
```

3) Quelques commentaires.

Les différents framework de mapping utilisent des stratégies différentes de chargement des données (base=>mémoire). Le problème concerne les objets "connexes" aux données appelées. Jusqu'à quelle profondeur ce chargement se fait-il ? Par exemple, si l'on demande toutes les leçons, le framework charge-t-il la voiture associée à chaque leçon ? Le choix délibéré d'Entity Framework est de ne charger (par défaut) que les données explicitement demandées : on parle de chargement explicite. Il n'en pas de même pour tous les frameworks de mapping. Ce fonctionnement par défaut garantit de ne solliciter la base de données que selon des besoins clairement explimés par le développeur. Nous avons vu plus haut que le chargements d'objets connexes se faisait grâce à la méthode include sur la requête.
Le modeleur EDM (Entity Data Model) permettant, à l'aide de l'assistant -cf plus haut-, de construire la modélisation graphique utilise en fait trois fichiers XML qui décrivent la structure des classes, des relations et du mapping ; ces 3 fichiers sont compilés comme ressource dans le projet :

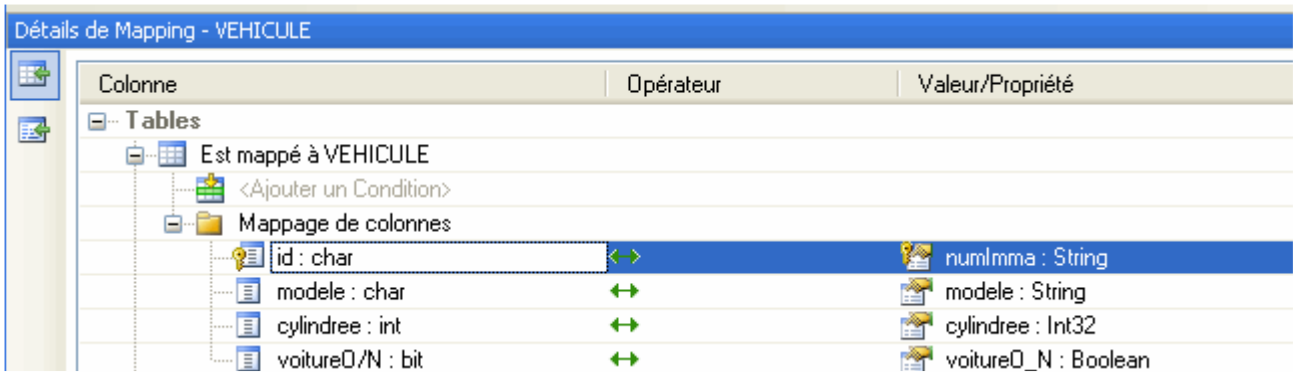


4) Quelques manipulations

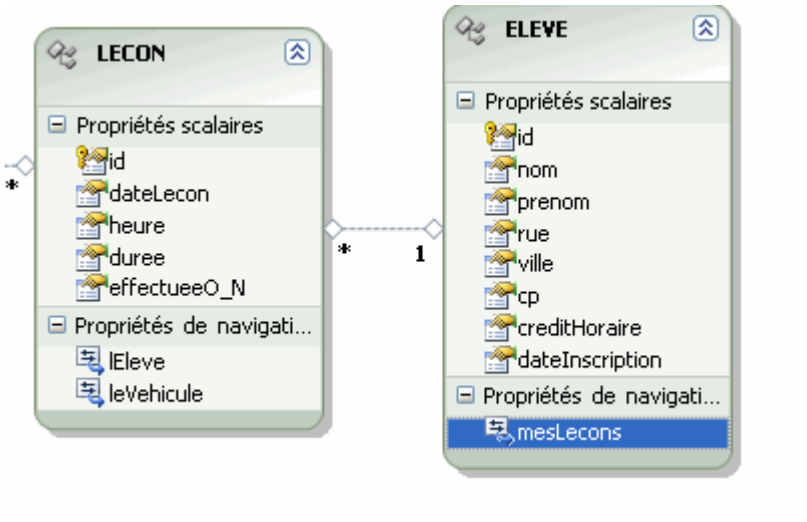
4.a Modifications sur le modèle

On peut modifier le nom des champs des classes ; ainsi, transformons id de Vehicule par un numImma moins connotée base de données.

A partir du modeleur de mapping, on peut voir la modification de mapping :



On peut aussi modifier les proprietés de navigation :



4.b Modification par le code

On peut ajouter de nouveaux attributs ou méthodes dans les classes "métiers". L'architecture basée ici sur des classes partial encourage fortement à ne pas intervenir sur le code généré. Le plus cohérent (et simple) est donc d'ajouter une classe partial, de même nom que la classe à enrichir et ceci dans un fichier distinct :

```
public partial class    LECON
{
      public void setCommentaire(string commentaire)
      {
          this.commentaire = commentaire;
      }
      public LECON() { }
      public LECON(string id, DateTime date, int heure, int duree, ELEVE el, VEHICULE ve)
      {
          this._id = id;
          this._dateLecon = date;
          this._heure = heure;
          this._duree = duree;
          this.effectueeO_N = false;
          this.ELEVE = el;
          this.VEHICULE = ve;
      }
      private string commentaire;

}
```

Remarques : nous avons ajouté un constructeur "classique" ; comme le code généré dans le contexte utilise un constructeur par défaut, il est nécessaire d'ajouter un constructeur par défaut explicite. Nous pouvions aussi surcharger le "constructeur statique". L'ajout d'un attribut "commentaire" n'offre que très peu d'intérêt puisqu'il n'y aura pas de mapping...
L'appel du constructeur de LECON peut prendre la forme suivante :

```
ELEVE elv = monModele.ELEVE.First((ELEVE el) => el.id == 135);
VEHICULE veh = monModele.VEHICULE.First((VEHICULE v) => v.numImma == "4411FG93");
LECON le = new LECON("56", DateTime.Now, 15, 2, elv, veh);
monModele.AddToLECON(le);
monModele.SaveChanges();
```

Remarque : nous avons utilisé des expressions lambda, on pouvait, bien sûr, faire appel à Linq

TP d'application

Application Console

1. **Faire les modifications sur la classe Vehicule (modification de l'id) et sur la classe Lecon (ajout de constructeurs)**
2. **Tester en ajoutant une leçon en utilisant Linq (en lieu et place des expressions Lambda)**

Lorsqu'une leçon est effectuée, son champ `effectueO/N` passe à `TRUE` et le crédit horaire de l'élève est décrémenté en conséquence.

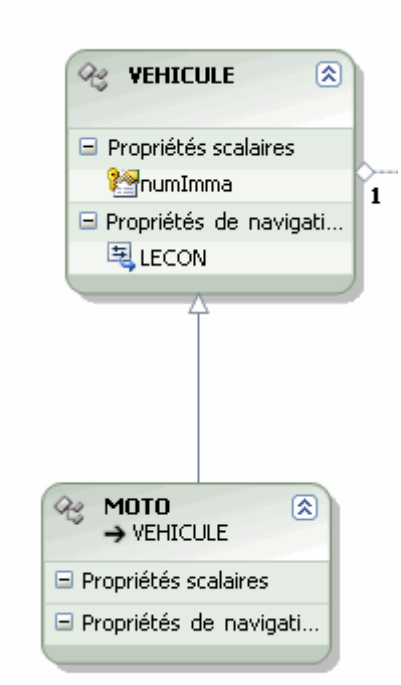
3. **Écrire une méthode setEffectuee() qui effectue ce traitement.**
4. **Tester**

4.c Mise en oeuvre de l'héritage

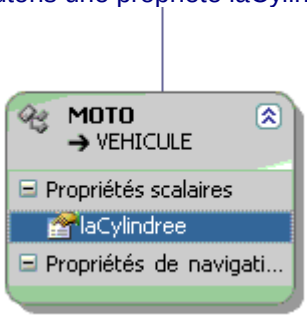
La classe vehicule regroupe voitures (modèle) et motos (cylindrée) ; il serait utile de faire dériver deux classes (auto et moto) d'une même classe vehicule.

Dans la classe vehicule, conservons uniquement le numéro d'immatriculation :

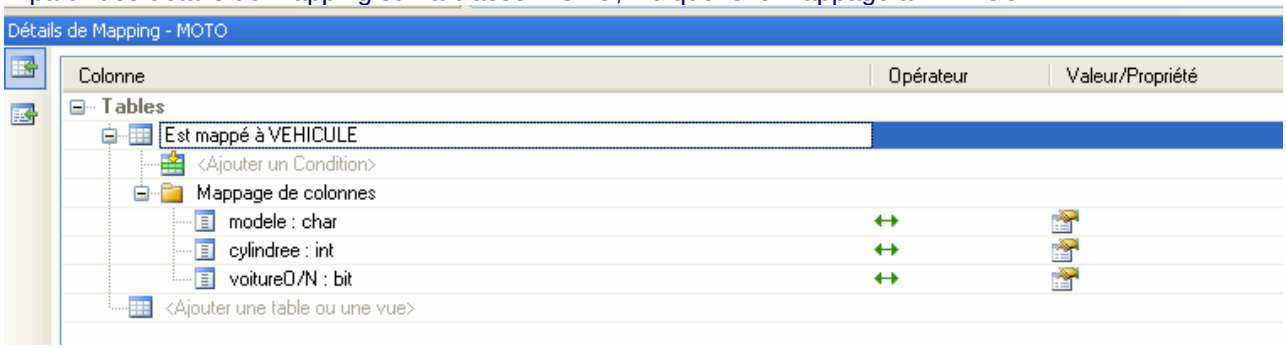
Dans le modeleur, à partir de la boite à outils, ajoutons une nouvelle entity, classe MOTO, ainsi que la relation d'héritage ; supprimons la propriété id générée:



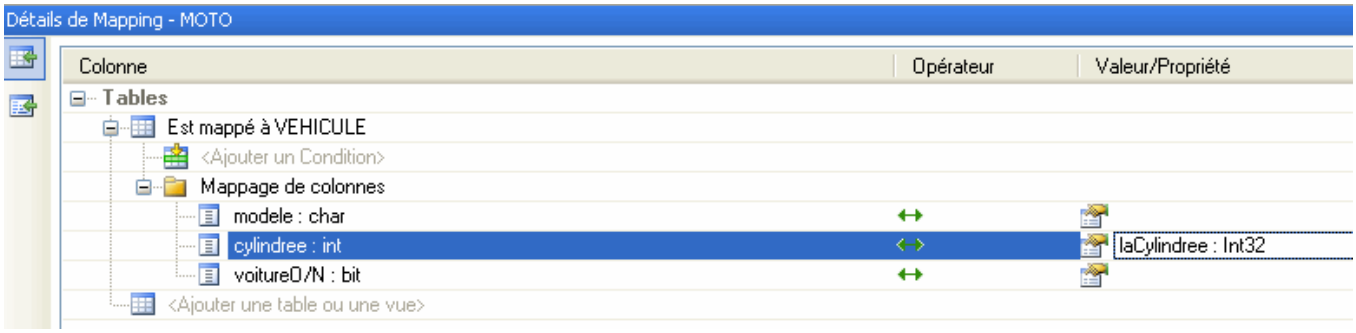
Ajoutons une propriété laCylindree (de type int32).



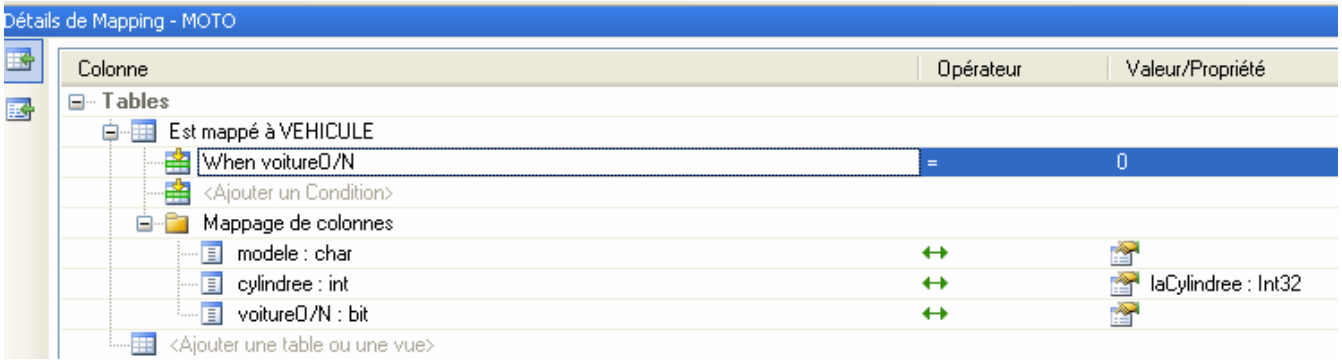
A partir des détails de mapping sur la classe MOTO, indiquons le mappage à VEHICULE :



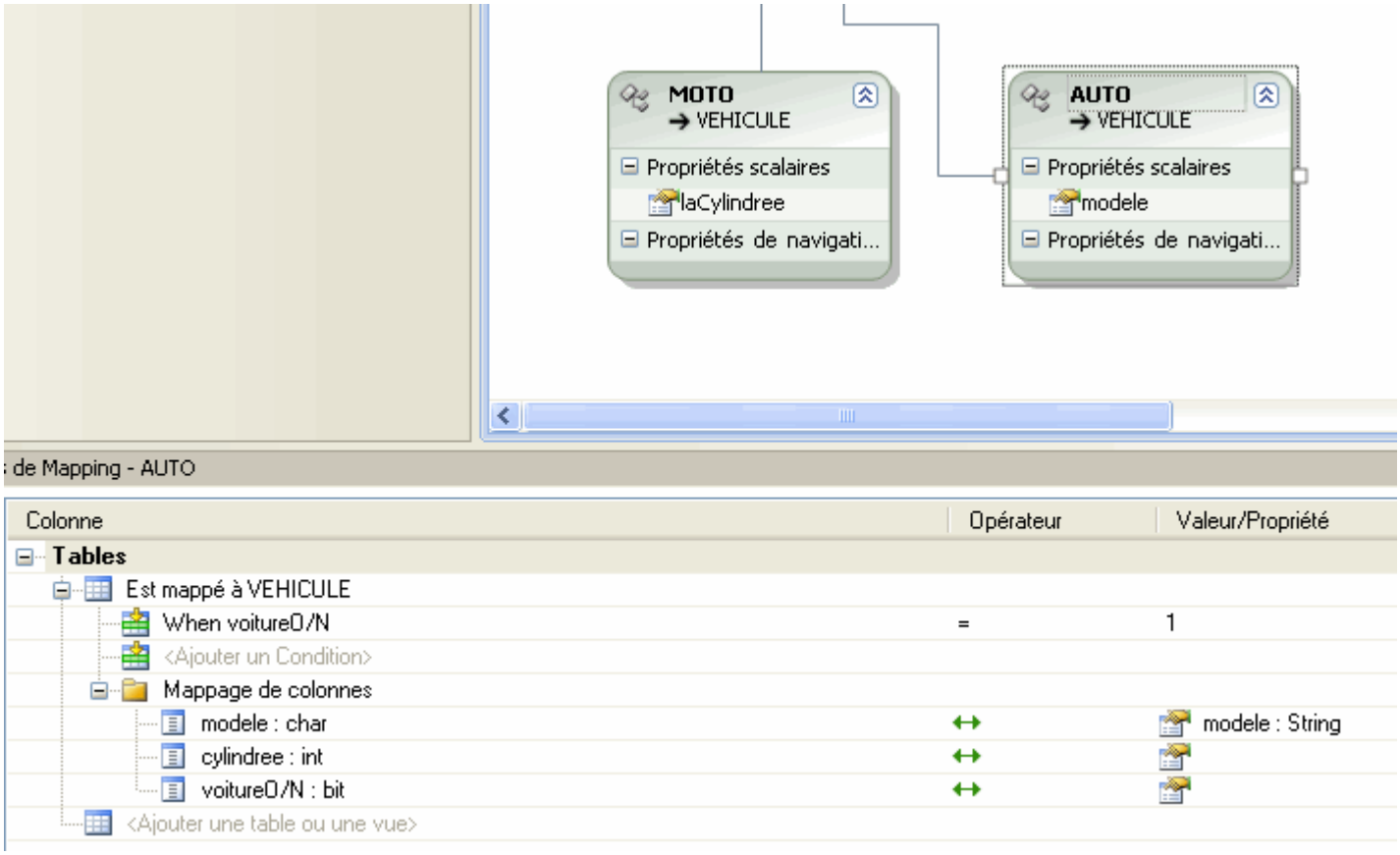
Indiquons le mapping de la propriété :



Définissons la condition de filtre :



Procédons de même pour la classe AUTO :



Testons maintenant :

```
ecoleConduiteEntities monModele = new ecoleConduiteEntities();
MOTO m = MOTO.CreateMOTO("4578JK94");
m.laCylindree = 900;
monModele.AddToVEHICULE(m);
monModele.SaveChanges();
```

La base de donnée a été mise à jour :

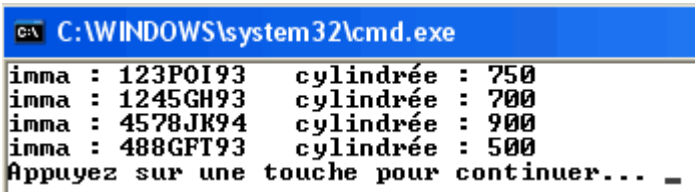
id	modele	cylindree	voitureO/N
123POI93	NULL	750	False
1245GH93	NULL	700	False
4411FG93	Renault Clio	NULL	True
456AZE93	Renault Clio	NULL	True
4578JK94	NULL	900	False
488GFT93	NULL	500	False

Le champ voitureO/N est bien passé aussi à false.
On peut afficher les motos ainsi :

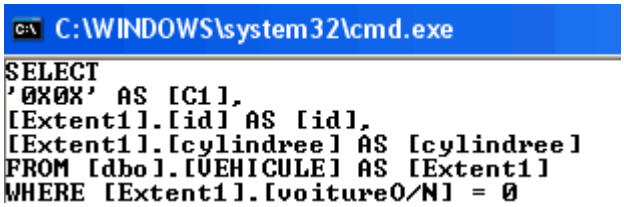
```
var req = (from m in monModele.VEHICULE.OfType<MOTO>()
           select m);
var liste = req.ToList();

foreach (var m in liste)
    Console.WriteLine("imma : {0} cylindrée : {1}",m.numImma,m.laCylindree);
```

Remarque : on peut regréter que le contexte ne connaisse pas directement le type MOTO ...

Ce qui produit :



Si l'on observe la requête générée, on constate sans surprise le filtre sur le champ voitureO/N :



TP d'application

Modifier le modèle comme sur le support (classes moto et auto). Créer une auto, afficher toutes les autos. Afficher ensuite les dates des leçons d'un véhicule de numéro d'immatriculation fourni.

4.d Utilisation d'une procédure stockée

La base contient une procédure stockée permettant l'insertion d'un nouvel élève ; pour l'appeler, il suffit de l'ajouter dans le modeleur :

Remarque : étrangement, il faut déclarer un type de retour, alors que la procédure stockée n'en a pas !!
L'appel se fait à partir du contexte:

```
monModele.nouvelEleve("Robert", DateTime.Now, "jean", "rue petit", "93100", "montreuil", 2
```

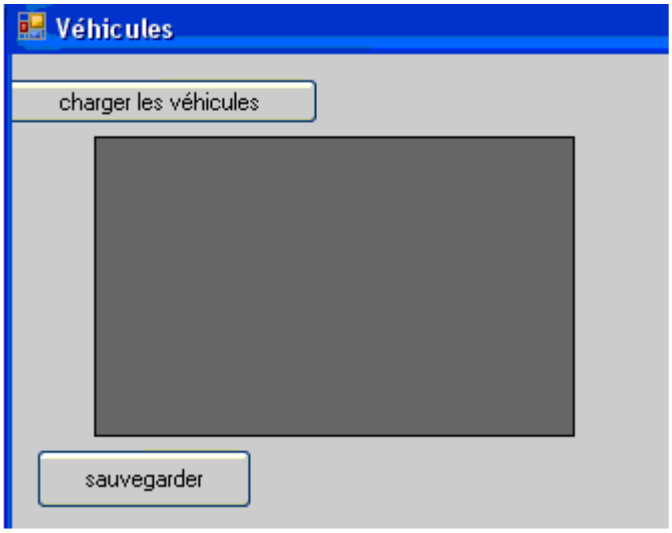
5) Le binding

Le binding ou liaison de données permet de créer un lien bidirectionnel entre un composant graphique et une source de données au sens large (Table, ArrayList, objet....).

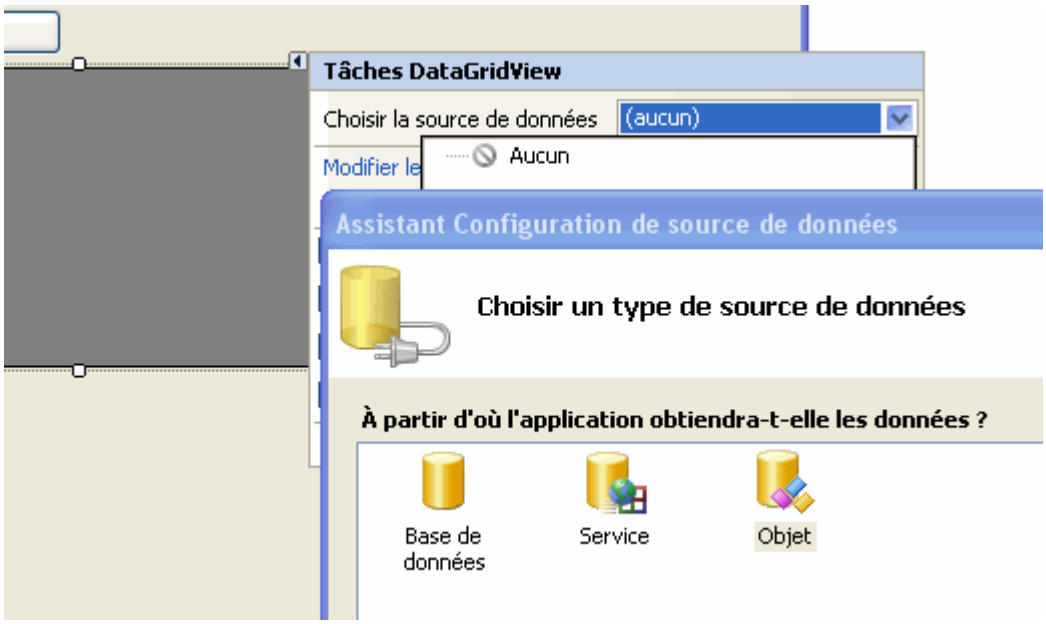
Le mécanisme est très proche de celui mis en oeuvre avec une source de données ADO. Nous allons juste montrer ici quelques exemples.

5.1 Binding simple.

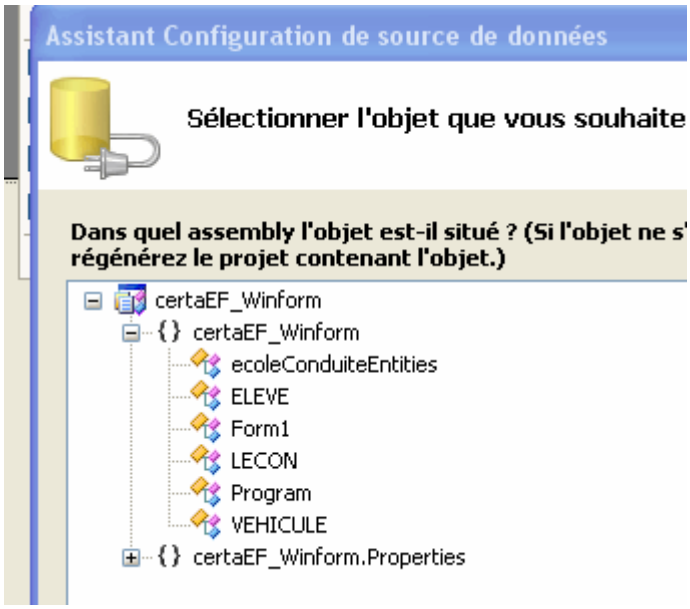
Nous allons gérer les véhicule à l'aide d'un DataGridView.
Construisons un formulaire avec deux boutons, l'un pour charger, l'autre pour sauvegarder. Ajoutons un dataGridView :



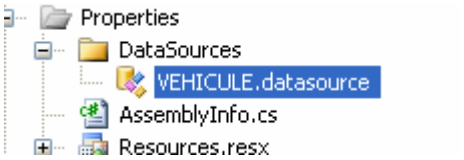
Indiquons la source de données pour le DataGridView, la classe VEHICULE ; paramétrons le dataGridView et ajoutons une nouvelle source de données, de type objet :



Sélectionnons la classe VEHICULE :



La source de données a été ajoutée au projet et un composant de Binding ajouté au formulaire :



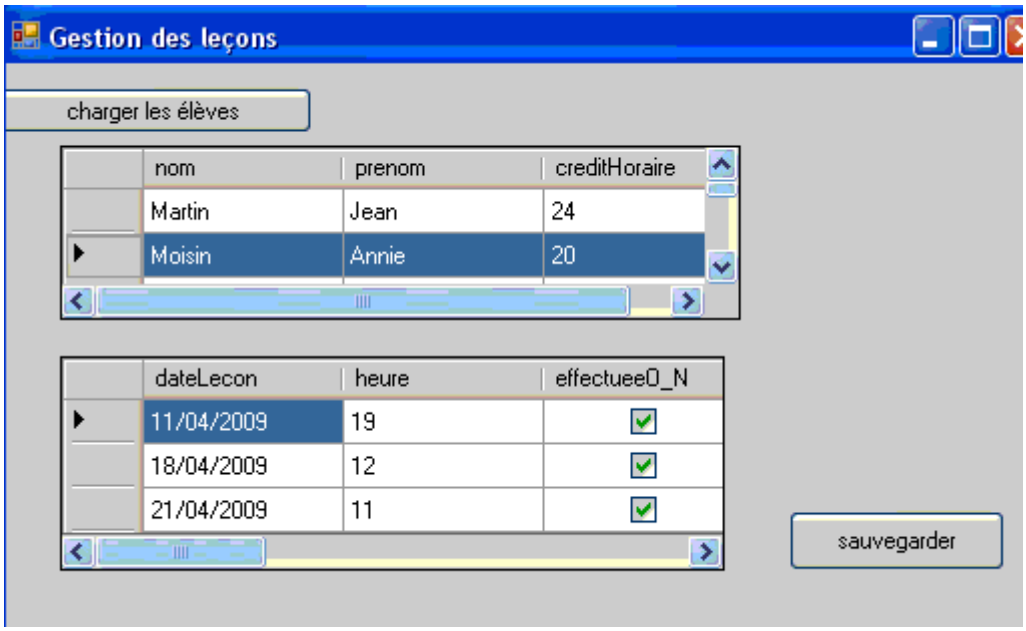
Après avoir chargé le modèle de ses véhicules avec une requête Linq, il ne reste plus qu'à lier le composant de binding à la source et le DataGridView au composant de binding :

```
public partial class Form1 : Form
{
    public Form1()
    {
      InitializeComponent();
      monModele = new ecoleConduiteEntities();
     }
    private void btnCharger_Click(object sender, EventArgs e)
    {
        var req = from v in monModele.VEHICULE
                  select v;
       vEHICULEBindingSource.DataSource = req;
       dataGridView.DataSource = vEHICULEBindingSource;
    }
    private void btnsauver_Click(object sender, EventArgs e)
    {
        monModele.SaveChanges();
    }
    private ecoleConduiteEntities monModele;
}
```

On pouvait ajouter un BindingNavigator pour un parcours par occurrence.

5.2 Binding lié à deux composants

Nous désirons visualiser (et éventuellement) modifier les leçons d'un élève sélectionné :

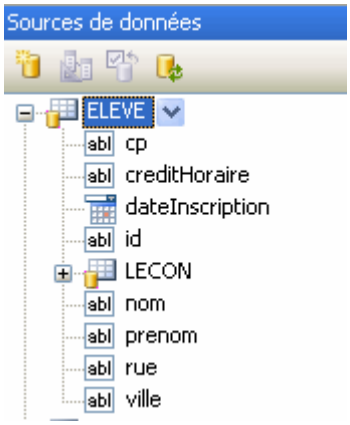


Le premier DataGridView sera bindé aux élèves, le second à la relation entre l'élève et ses leçons (comme pour binding associé à des tables).

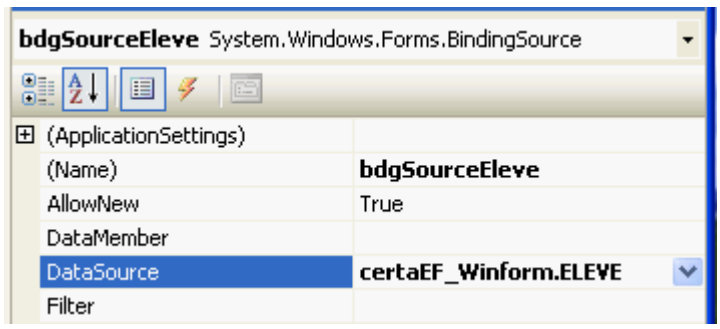
5.2.a Le DatagridView associé aux élèves

La manipulation est identique à celle décrite pour le binding source ; mais on peut déclarer au préalable la source de données objet :

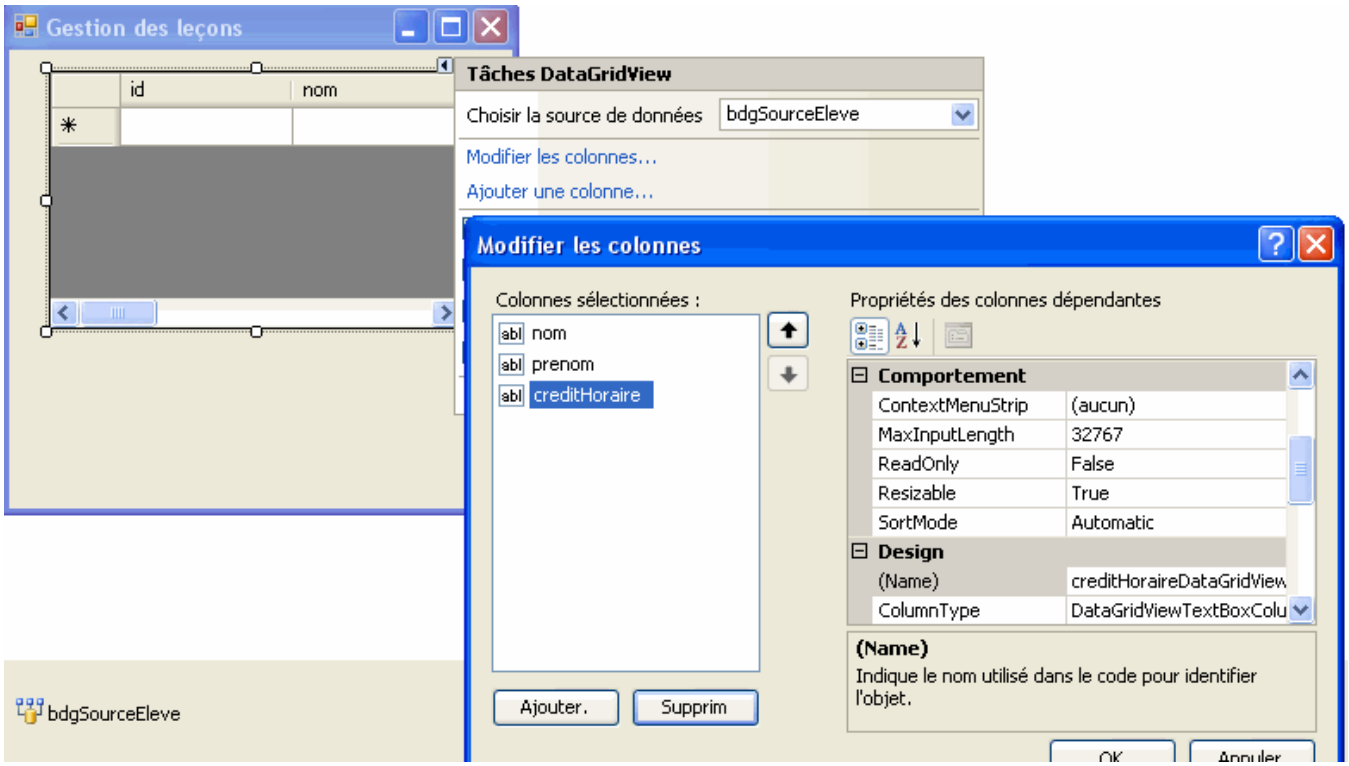
A partir du menu Données/Ajouter une nouvelle source de données, ajoutons la source pointant sur les élèves, cette source apparaît dans la fenêtre :



Ajoutons un composant de binding au formulaire, dont la source est la source créée ci-dessus :

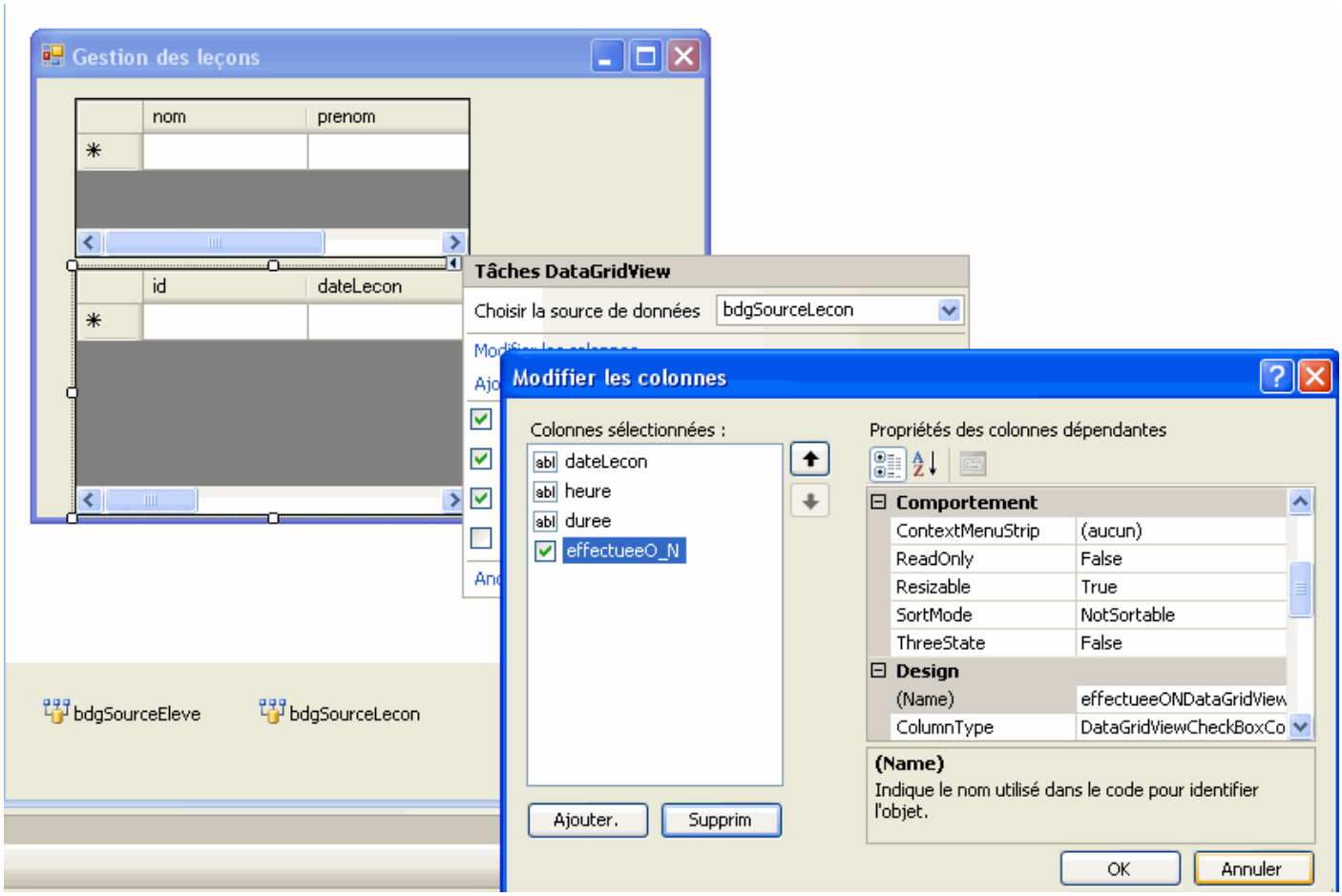


Ajoutons un DataGridView dont la source de données est ce composant de binding, ne conservons que certaines données :



5.2.b Le DataGridView associé aux leçons

Procédons de la même manière (création de la source de données, paramétrage du composant de binding, paramétrage du DataGridView) :



En toutes rigueur, la liaison à la source "LECON" n'est pas nécessaire ; elle est seulement utile pour paramétrer simplement (modification des colonnes) le DataGridView.

5.2.c Chargement des données

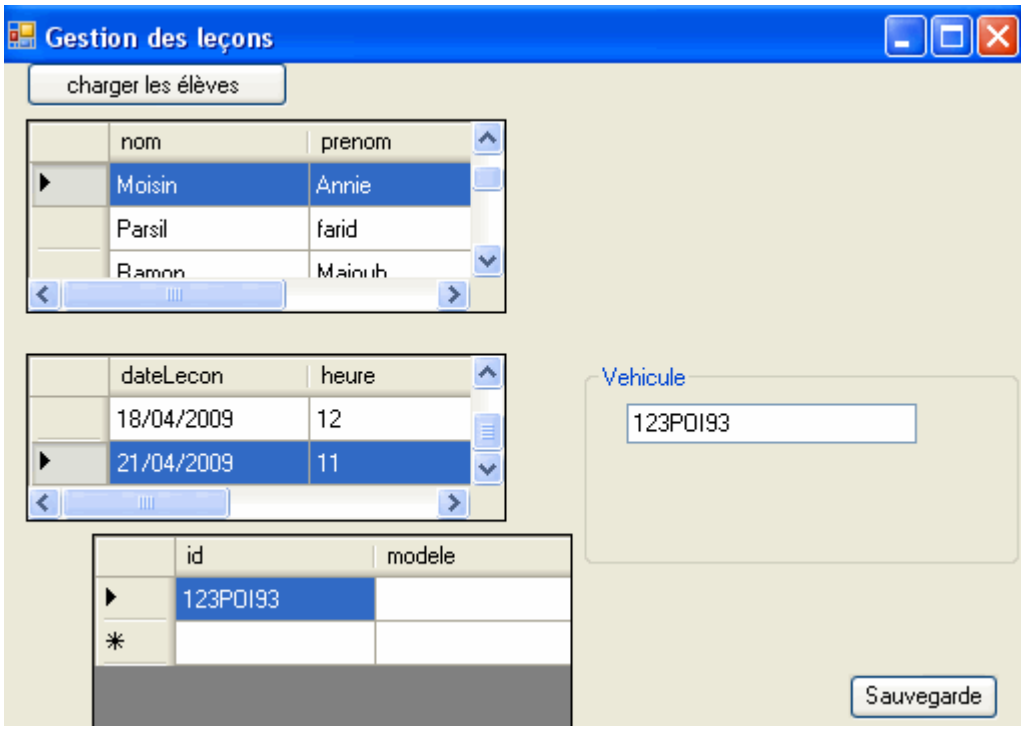
Ceci se fera (par exemple) à partir d'un bouton sprécifique dont le code de l'événement click est le suivant :

```
public partial class Form2 : Form
{
    public Form2()
    {
        InitializeComponent();
        monModele = new ecoleConduiteEntities();
    }
    private void btnCharger_Click(object sender, EventArgs e)
    {
        var req = from el in monModele.ELEVE.Include("LECON")
                  select el;
        bdgSourceEleve.DataSource = req;
        dataGridView1.DataSource = bdgSourceEleve;
        bdgSourceLecon.DataSource = bdgSourceEleve;
        bdgSourceLecon.DataMember = "LECON";
        dataGridView2.DataSource = bdgSourceLecon;
    }
    private ecoleConduiteEntities monModele;

    private void btnSauve_Click(object sender, EventArgs e)
    {
        monModele.SaveChanges();
    }
}
```

Notez l'appel à Include des leçons connexes à chaque élève.
La sauvegarde des données appelle la méthode *SaveChanges*.

5.3 Binding lié à trois composants

Terminons en ajoutant chaque véhicule associé à une leçon :



Deux versions sont proposées, l'une avec un DataGridView et l'autre une zone de texte ; les paramétrages sont identiques à ceux vus plus haut, le code de chargement diffère légèrement :

```
var req = from el in monModele.ELEVE.Include("LECON.VEHICULE")
          select el;
bdgSourceEleve.DataSource = req;
dataGridView1.DataSource = bdgSourceEleve;
bdgSourceLecon.DataSource = bdgSourceEleve;
bdgSourceLecon.DataMember = "LECON";
dataGridView2.DataSource = bdgSourceLecon;
bdgSourceVehicule.DataSource = bdgSourceLecon;
bdgSourceVehicule.DataMember = "VEHICULE";
dataGridView3.DataSource = bdgSourceVehicule;
```

Notez l'appel à Include qui charge les leçons et les véhicules associés
Pour lier la zone de texte, ceci peut se faire en mode conception.

Tp d'application.

Créer un formulaire permettant de créer un nouvel élève en utilisant la procédure stockée fournie avec SqlServer (*pInsertEleve*)
Créer un formulaire permettant de créer une nouvelle leçon pour un élève inscrit.