

Réflexion

Sommaire

Réflexion	1
1 Introduction.....	2
2 Les bases de la reflection	4
2.1 La classe Assembly.....	4
2.2 La classe Module	5
2.3 Exemple de récupération	6
2.4 Récupérer des informations précises.....	7
2.5 La méthode GetMethodBody.....	14
3 Les attributs d'une assembly.....	15
4 Génération et exécution de code dynamique.....	18
4.1 Génération de code dynamique.....	18
4.2 Exécution dynamique de code	22
4.2.1 Instancier un objet dynamiquement.....	22
4.2.2 Invoquer des méthodes.....	24
4.2.3 Invoquer une propriété	25
4.2.4 Résultat final.....	25
5 Conclusion	26

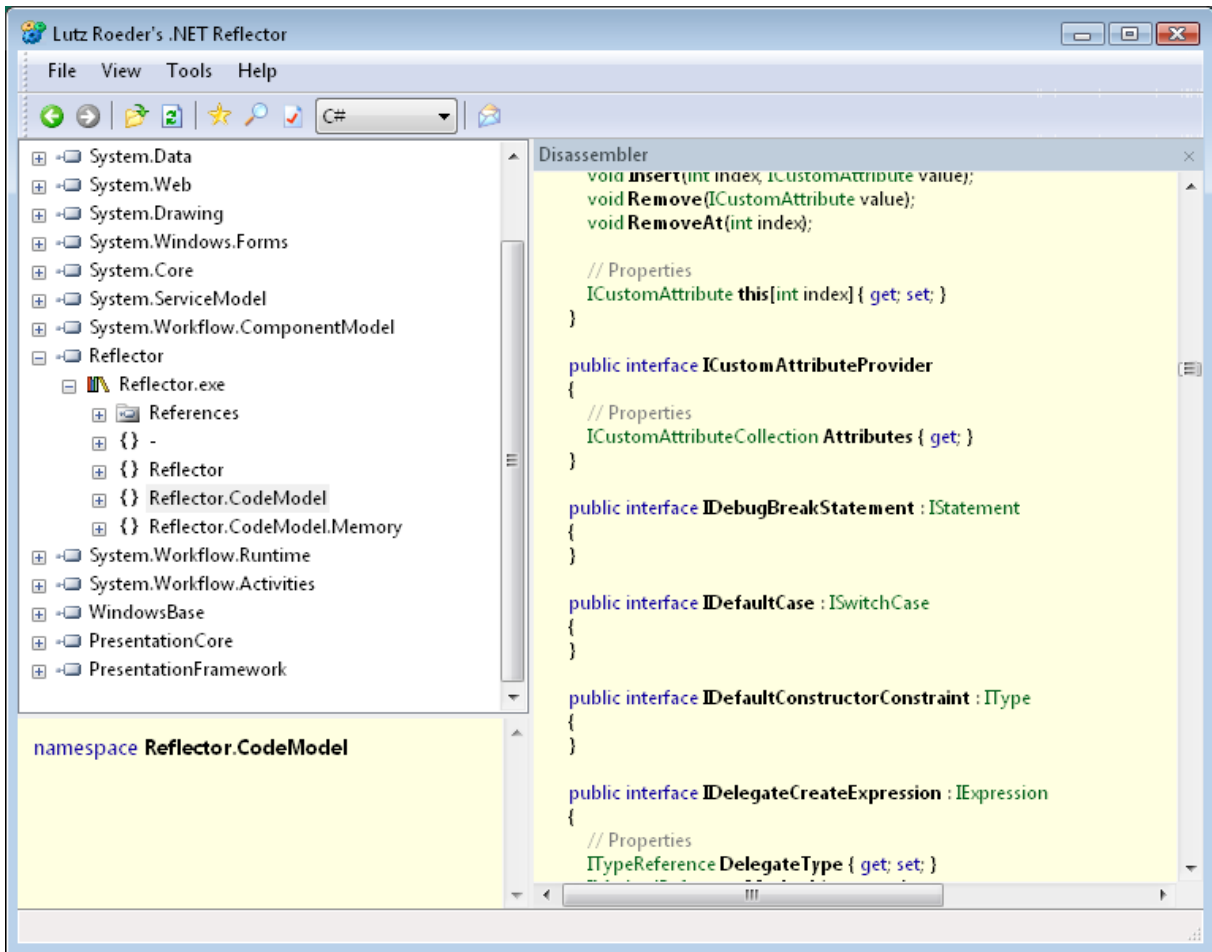
1 Introduction

Alors que jusqu'ici nous devons prévoir à l'avance quelles méthodes et classes nous allons utiliser avant de compiler, la réflexion nous permet entre autre de lister les classes d'une assembly, de les instancier « à la volée » et d'invoquer leurs méthodes au cours de l'exécution de notre application.



Il est difficile de comprendre à première vue le but de la réflexion, mais une fois maîtrisée, elle est d'une grande puissance. Sachez néanmoins que vous ne l'utiliserez pas tous les jours tant les cas qui nécessitent l'utilisation de la réflexion sont rare. Voici deux exemples concrets de ce qu'il est possible de faire avec la réflexion :

- La PropertyGrid en WinForm utilise la réflexion afin d'afficher les membres d'une classe et de les modifier. Vous trouverez un exemple d'utilisation [ici](#).
- L'outil [Reflector](#) dont l'essentiel des fonctionnalités est basé sur la réflexion permet de lister toutes les classes et membres d'une assembly afin de restituer une représentation fidèle d'une assembly :



Dotnet-France

2 Les bases de la réflexion

Les fichiers d'assemblies contiennent plusieurs sections caractérisant votre application :

- Des métadonnées d'assembly (ou manifest) indiquant la version du logiciel, de son copyright, son concepteur etc. et éventuellement les liens vers d'autres fichiers nécessaire à l'assembly pour fonctionner. Ces fichiers, appelés modules, ne contiennent que les informations sur les types ainsi que le code IL associé ou des données binaires constituant des ressources.
- Des informations sur les types décrivant comment sont agencés les éléments, le nom des classes ainsi que leur membres (méthodes, propriétés ...).
- Le code IL à proprement parler. C'est là que se situe tous les codes de vos méthodes.
- Une partie ressources qui contient les éventuels fichiers, images et chaînes de caractères qui sont utilisés dans le programme.

Pour cela, le .NET Framework nous fournit quelques outils contenus dans l'espace de nom `System.Reflection`.

2.1 La classe Assembly

Cette classe vous permet de charger/exécuter du code contenu dans une autre assembly. En voici les principaux membres:

- Méthodes Statiques

Méthodes	Description
<code>GetAssembly</code>	Retourne un objet Assembly représentant l'assembly qui contient le type passé en paramètre.
<code>GetCallingAssembly</code>	Retourne un objet Assembly représentant l'assembly qui a fait l'appel à la méthode courante.
<code>GetEntryAssembly</code>	Retourne un objet Assembly représentant l'assembly qui a démarré le processus courant.
<code>ReflectionOnlyLoadFrom</code>	Charge une assembly à partir d'un fichier spécifié. L'assembly est chargée en parcour seul, on ne peut pas en exécuter de codes.
<code>GetExecutingAssembly</code>	Retourne un objet Assembly représentant l'assembly qui exécute le code courant.
<code>Load</code>	Charge une assembly dans le domaine d'application courant.
<code>LoadFile</code>	Charge une assembly à partir d'un fichier.
<code>LoadFrom</code>	Charge une assembly dans le domaine d'application actuel à partir d'un fichier.
<code>ReflectionOnlyLoad</code>	Charge une assembly en parcour seul (sans exécution possible).

- Propriétés

Propriété	Description
<code>EntryPoint</code>	Contient un objet <code>MethodInfo</code> indiquant la méthode de démarrage de l'assembly chargée.
<code>FullName</code>	Retourne le nom complet de l'assembly chargée.
<code>GlobalAssemblyCache</code>	Indique si l'assembly chargée est contenu dans le Global Assembly Cache (GAC).
<code>Location</code>	Retourne le chemin d'accès à l'assembly.
<code>ReflectionOnly</code>	Indique si l'assembly a été chargée en parcour seul ou non.

➤ Méthodes

Méthodes	Description
CreateInstance	Crée une instance d'un type existant dans l'assembly chargée.
GetExportedTypes	Retourne une liste des types visibles en dehors de l'assembly.
GetFile	Crée un flux vers un fichier déclaré dans les ressources de l'assembly.
GetFiles	Identique à <code>GetFile</code> mais appliqué à tous les fichiers présents dans les ressources de l'assembly.
GetModule	Retourne un objet <code>Module</code> lié à l'assembly.
GetModules	Retourne un tableau d'objet <code>Module</code> liés à l'assembly.
GetCustomAttributes	Permet de récupérer un tableau d'attribut ou les informations d'un attribut particulier.
GetLoadedModules	Retourne un tableau contenant tous les modules chargés par l'assembly.
GetName	Retourne un objet <code>AssemblyName</code> contenant les informations sur le nom de l'assembly.
GetSatelliteAssembly	Retourne une assembly annexe (uniquement si elle existe) en utilisant des informations de langue particulier.
GetTypes	Retourne un tableau référençant tous les types déclarés dans les modules de l'assembly.

2.2 La classe Module

Cette classe permet de faire une référence vers des portions de codes ou des ressources externes à l'assembly courante, stockées dans un autre fichier. Ces fichiers annexes sont appelés des modules.

Ces modules sont créés au moment de la compilation de votre projet en code MSIL. Pour cela, vous devez ouvrir une console Visual Studio et compiler votre projet en passant par là. Dans votre commande, pour créer un module, il vous faudra ajouter l'instruction `"/target:module"`.

Note : Pour plus d'informations, allez sur [ce lien](#) qui récapitule les options du compilateur Visual Studio.

Voici les principaux membres de cette classe :

Membre	Description
<code>Assembly</code>	Permet de récupérer un objet <code>Assembly</code> représentant l'assembly lié au module.
<code>FullyQualifiedName</code>	Permet de récupérer le nom du module ainsi que le chemin complet vers le module.
<code>Name</code>	Permet de récupérer le nom du module.
<code>FindTypes</code>	Permet de chercher dans une liste de module ceux qui correspondent aux critères de filtres donnés.
<code>GetCustomAttributes</code>	Permet de récupérer les attributs d'assembly associé au module.
<code>GetField</code>	Retourne un champ spécifique d'un module.
<code>GetFields</code>	Retourne tous les champs d'un module.
<code>GetMethod</code>	Retourne une méthode spécifique d'un module.
<code>GetMethods</code>	Retourne toutes les méthodes d'un module.
<code>GetTypes</code>	Retourne tous les types contenus dans un module.
<code>IsResource</code>	Permet de déterminer si le module est une ressource.

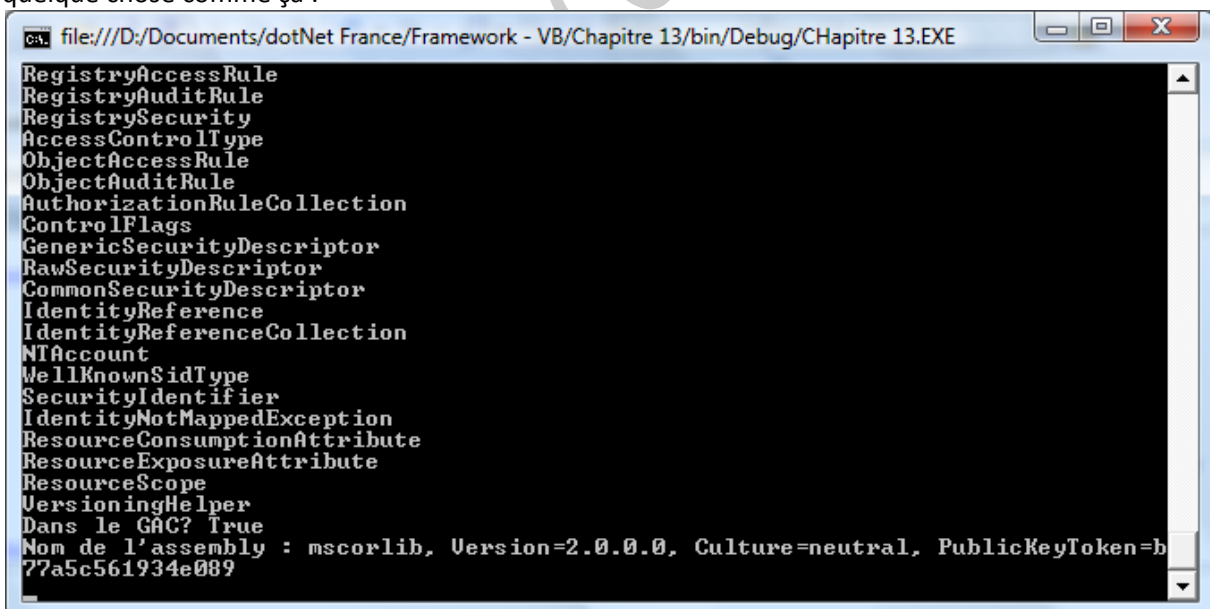
2.3 Exemple de récupération

Vous remarquerez que l'utilisation de ce système n'est pas des plus complexes. Dans l'exemple ci-dessous, nous allons afficher la liste des types disponible dans l'assembly qui a fait appel à notre programme :

```
'VB
Imports System.Reflection
Sub Main()
    Dim ase As Assembly = Assembly.GetCallingAssembly()
    For Each m As Type In ase.GetExportedTypes()
        Console.WriteLine(m.Name)
    Next
    Console.WriteLine("Dans le GAC? {0}" + vbNewLine + "Nom de
l'assembly : {1}", ase.GlobalAssemblyCache, ase.GetName().FullName)
    Console.Read()
End Sub
```

```
//C#
using System.Reflection;
static void Main()
{
    Assembly ase = Assembly.GetCallingAssembly();
    foreach (Type m in ase.GetExportedTypes())
        Console.WriteLine(m.Name);
    Console.WriteLine("Dans le GAC? {0} \n Nom de l'assembly : {1}",
ase.GlobalAssemblyCache, ase.GetName().FullName);
    Console.Read();
}
```

Si vous exécutez ce programme en mode DEBUG dans Visual Studio, vous devriez avoir quelque chose comme ça :



```
RegistryAccessRule
RegistryAuditRule
RegistrySecurity
AccessControlType
ObjectAccessRule
ObjectAuditRule
AuthorizationRuleCollection
ControlFlags
GenericSecurityDescriptor
RawSecurityDescriptor
CommonSecurityDescriptor
IdentityReference
IdentityReferenceCollection
NTAccount
WellKnownSidType
SecurityIdentifier
IdentityNotMappedException
ResourceConsumptionAttribute
ResourceExposureAttribute
ResourceScope
VersioningHelper
Dans le GAC? True
Nom de l'assembly : mscorlib, Version=2.0.0.0, Culture=neutral, PublicKeyToken=b
77a5c561934e089
```

En effet, ce qui fait appel à notre programme en mode debug sous Visual Studio, ce n'est plus le programme mais une autre assembly : mscorlib. Si nous exécutons le même programme sans passer par Visual Studio, nous obtiendrons ceci:

```

D:\Documents\dotNet France\Framework - VB\Chapitre 13\bin\Debug\CHapitre 13.exe
Dans le GAC? False
Nom de l'assembly : CHapitre 13, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null
  
```

2.4 Récupérer des informations précises

La classe `Type` nous donne accès à de nombreuses propriétés et méthodes qui permettent de déterminer si un type est une énumération, si c'est une classe abstraite ou non Ces informations sont accessibles via des propriétés `Is<spécificité>` (Par exemple, `IsSealed` pour déterminer si la classe peut être parente d'une autre ou non etc.) et des méthodes `Get<Informations>` (Par exemple, `GetConstructor` qui retourne des informations sur le constructeur). Concernant les méthodes, elles retournent généralement un objet de type `<Informations>Info` (Par exemple, `ConstructorInfo` pour les informations sur le constructeur) et possèdent toutes une méthode permettant de trouver l'information dont on connaît le nom et l'équivalent pour retourner un tableau d'information (Par exemple, `GetConstructor` pour un constructeur connu et `GetConstructors` pour tous les constructeurs).

Etant donné que les propriétés `Is` sont assez explicites, nous ne détaillerons ici que les méthodes `Get` les plus intéressantes :

Méthodes	Description
<code>GetConstructor</code> <code>GetConstructors</code>	Retourne un ou plusieurs objets <code>ConstructorInfo</code> concernant les constructeurs.
<code>GetEvent</code> <code>GetEvents</code>	Retourne un ou plusieurs objets <code>EventInfo</code> concernant les évènements.
<code>GetField</code> <code>GetFields</code>	Retourne un ou plusieurs objets <code>FieldInfo</code> concernant les champs (variables privées, publiques...).
<code>GetInterface</code> <code>GetInterfaces</code>	Retourne un ou plusieurs objets <code>InterfaceInfo</code> concernant les interfaces utilisées.
<code>GetMember</code> <code>GetMembers</code>	Retourne un ou plusieurs objets <code>MemberInfo</code> concernant les membres.
<code>GetMethod</code> <code>GetMethods</code>	Retourne un ou plusieurs objets <code>MethodInfo</code> concernant les méthodes.
<code>GetNestedType</code> <code>GetNestedTypes</code>	Retourne un ou plusieurs objets <code>Type</code> concernant les types imbriqués (agrégation).
<code>GetProperty</code> <code>GetProperties</code>	Retourne un ou plusieurs objets <code>PropertyInfo</code> concernant les propriétés.

En relation avec ces méthodes, vous pouvez utiliser les propriétés de `BindingFlags` qui permettent de filtrer les informations récupérées. Voici les différentes valeurs possibles et leur utilité :

Valeur	Description
<code>DeclaredOnly</code>	Ajoute les membres déclarés dans le type concerné.
<code>Default</code>	Aucune modification
<code>FlattenHierarchy</code>	Ajoute les membres hérités d'un autre type.
<code>IgnoreCase</code>	Rend la recherche des noms de membres insensible à la casse.
<code>Instance</code>	Ajoute les membres d'instance à la recherche. Ce paramètre est primordial lorsque vous recherchez des membres non-statiques.
<code>NonPublic</code>	Ajoute tous les membres déclarés <code>protected</code> , <code>internal</code> ou <code>private</code> (C#, ou <code>friend</code> en VB.NET)
<code>Public</code>	Ajoute les membres déclarés public.
<code>Static</code>	Ajoute les membres déclarés static

Afin de ne pas limiter les possibilités, il est tout à fait possible d'effectuer un OU logique entre chacune de ces valeurs pour en utiliser plusieurs à la fois. En VB.NET, il est également possible d'utiliser l'opérateur `Not` sur chacune des valeurs afin d'en inverser l'effet.

Vous pouvez obtenir des instances de cette classe soit en passant par `typeof` (ou `GetType` en VB.NET) ou en appelant la méthode `GetType` d'un objet instancié. Vous pouvez également obtenir une ou plusieurs instances de cette classe en utilisant les méthodes appropriées de la classe `Module` ou de la classe `Assembly`.

Comme exemple, nous allons créer un nouveau projet "Bibliothèque de classes" appelé "Chapitre 13 – Lib". Dans ce projet, nous mettrons un fichier de code qui contiendra deux classes appelées "Personnage" et "Voiture", une classe abstraite appelée "IEntites" et une énumération appelée "Enumération" :

```
'VB
Public Enum Enumeration
    Valeur1
    Valeur2
    Valeur3
End Enum

Public MustInherit Class IEntites
    Public Overridable Function rouler() As String
        Return ""
    End Function
    Public Overridable Function sauter() As String
        Return ""
    End Function
End Class
<Serializable()> _
Public Class Personnage
    Inherits IEntites
    Public _nom As String
    Public Property Nom() As String
        Get
            Return _nom
        End Get
        Set(ByVal value As String)
            _nom = value
        End Set
    End Property
```

```
'VB - Suite
Public Sub New()
    _nom = "MonPersonnage"
End Sub

Public Overrides Function sauter() As String
    Return Me._nom + " saute de 1 metre"
End Function

Public Overrides Function ToString() As String
    Return Me._nom
End Function

Public Shared Function parler(ByVal txt As String) As String
    Return "coucou " + txt
End Function
End Class

Public Class Voiture
    Inherits IEntites

    Public _nom As String

    Public Property Nom() As String
        Get
            Return _nom
        End Get
        Set(ByVal value As String)
            _nom = value
        End Set
    End Property

    Public Sub New()
        Me._nom = "MaVoiture"
    End Sub

    Public Overrides Function rouler() As String
        Return Me._nom + " se met à rouler !!"
    End Function

    Public Overrides Function ToString() As String
        Return Me._nom
    End Function

    Public Shared Function parler(ByVal txt As String) As String
        Return "Vroum vroum " + txt
    End Function
End Class
```

```
//C#
public enum Enumeration
{
    Valeur1,
    Valeur2,
    Valeur3
}

public abstract class IEntites
{
    public virtual string rouler() { return ""; }
    public virtual string sauter() { return ""; }
}

[Serializable]
public class Personnage : IEntites
{
    public string nom
    {
        get { return _nom; }
        set { _nom = value; }
    }
    public string _nom;

    public Personnage ()
    {
        nom = "MonPersonnage";
    }
    public override string sauter()
    {
        return this.nom + " saute de 1 metre";
    }

    public override string ToString()
    {
        return this.nom;
    }

    public static string parler(string txt)
    {
        return "coucou " + txt;
    }
}
```

```
//C# - Suite
public class Voiture : IEntites
{
    public string nom
    {
        get { return _nom; }
        set { _nom = value; }
    }
    public string _nom;

    public Voiture()
    {
        this.nom = "MaVoiture";
    }

    public override string rouler()
    {
        return this.nom + " se met à rouler !!";
    }

    public override string ToString()
    {
        return this.nom;
    }

    public static string parler(string txt)
    {
        return "Vroum vroum " + txt;
    }
}
```

Après compilation, vous aurez une DLL qui contiendra ces codes.

Ensuite, nous retournons dans notre classe principale du projet Console et nous chargeons l'assembly créée, puis nous en listons tous les types qu'elle contient. Dans un second temps, nous listerons également les membres déclarés de la classe Voiture :

```
'VB
Sub Main()
    Dim assem As Assembly = Assembly.LoadFrom("../..\\..\\Chapitre 13 -
Lib\\bin\\Debug\\Chapitre 13 - Lib.dll")
    Console.WriteLine("Assembly: {0}", assem.FullName)
    Dim ty() As Type = assem.GetTypes()
    Console.WriteLine("-----Information sur le type-----
-")
    For Each t As Type In ty
        Console.WriteLine("Nom : {0}, Classe : {1}, Enumeration :
{2}, Serializable : {3}", t.Name, t.IsClass, t.IsEnum, t.IsSerializable)
    Next
    Console.WriteLine()
    Console.WriteLine("-----Infos de la classe Voiture-----")
    For Each minfo As MethodInfo In ty(ty.Length -
1).GetMethods(BindingFlags.DeclaredOnly Or BindingFlags.Instance Or
BindingFlags.Public)
        Try
            Dim b() As Byte =
minfo.GetMethodBody().GetILAsByteArray()
            Console.WriteLine("Nom: {2}, Taille du code: {0}octets,
Valeur de retour: {1}", b.LongLength, minfo.ReturnType.Name, minfo.Name)
        Catch ex As Exception
            Console.WriteLine("Nom: {1}, Taille du code: 0octets,
Valeur de retour: {0}", minfo.ReturnType.Name, minfo.Name)
        End Try
    Next
    Console.Read()
End Sub
```

Dotnet-France

```
//C#
static void Main()
{
    Assembly assem = Assembly.LoadFrom(@"..\..\..\Chapitre 13 -
Lib\bin\Debug\Chapitre 13 - Lib.dll");
    Console.WriteLine("Assembly: {0}", assem.FullName);
    Type[] ty = assem.GetTypes();
    Console.WriteLine("-----Information sur le type-----");
    foreach(Type t in ty)
        Console.WriteLine("Nom : {0}, Classe : {1}, Enumeration : {2},
Serializable : {3}", t.Name, t.IsClass, t.IsEnum, t.IsSerializable);
    Console.WriteLine();
    Console.WriteLine("-----Infos de la classe Voiture-----");
    foreach(MethodInfo minfo in ty[ty.Length -
1].GetMethods(BindingFlags.DeclaredOnly | BindingFlags.Instance |
BindingFlags.Public))
    {
        try
        {
            Byte[] b = minfo.GetMethodBody().GetILAsByteArray();
            Console.WriteLine("Nom: {2}, Taille du code: {0}octets,
Valeur de retour: {1}", b.LongLength, minfo.ReturnType.Name, minfo.Name);
        }
        catch (Exception ex)
        {
            Console.WriteLine("Nom: {1}, Taille du code: 0octets, Valeur
de retour: {0}", minfo.ReturnType.Name, minfo.Name);
        }
    }
}

Console.Read();
}
```

Ainsi, à la compilation, nous devrions avoir un résultat similaire à celui-ci :

```
Assembly: Chapitre 13 - Lib, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null
-----Information sur le type-----
Nom : Enumeration, Classe : False, Enumeration : True, Serializable : True
Nom : IEntites, Classe : True, Enumeration : False, Serializable : False
Nom : Personnage, Classe : True, Enumeration : False, Serializable : True
Nom : Voiture, Classe : True, Enumeration : False, Serializable : False

-----Infos de la classe Voiture-----
Nom: get_nom, Taille du code: 12octets, Valeur de retour: String
Nom: set_nom, Taille du code: 9octets, Valeur de retour: Void
Nom: rouler, Taille du code: 22octets, Valeur de retour: String
Nom: ToString, Taille du code: 12octets, Valeur de retour: String
```

Dans la première liste, nous pouvons constater que l'énumération et la classe sont tous les deux présents. Dans la classe Voiture nous avons listé toutes les méthodes, apparaissent les accesseurs, la méthode rouler, et la méthode surchargé `ToString`. Les méthodes héritées et la méthode statique ne sont pas affichées car les BidingFlags ne concordent pas.

Nous aurions pu procéder sensiblement de la même façon pour lister les champs de la classe ou les valeurs de l'énumération grâce à la méthode `GetFields` (ou `GetField` si nous connaissons le nom du champ à récupérer).

2.5 La méthode `GetMethodBody`

Vous avez pu remarquer dans l'exemple précédent qu'on utilisait la méthode `GetMethodBody`. Cette méthode retourne un objet `MethodBody` qui permet de se rapprocher du code IL. En effet, cette classe vous fournit quelques propriétés et méthodes permettant d'accéder au code IL :

Valeur	Description
<code>ExceptionHandlingClauses</code>	Contient une liste contenant toutes les clauses de gestion d'exceptions (bloc Try-Catch)
<code>InitLocals</code>	Indique si les variables locales sont initialisées à leur valeur par défaut.
<code>LocalVariables</code>	Contient la liste de toutes les variables locales utilisées.
<code>MaxStackSize</code>	Indique le nombre maximal de variables stockées dans le tas de la mémoire lorsque la méthode s'exécute.
<code>GetILAsByteArray</code>	Retourne un tableau d'octet représentant le code MSIL.

Dotnet-France Association

3 Les attributs d'une assembly

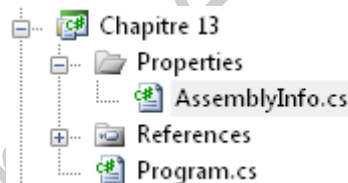
Nous avons vu dans la partie précédente que lorsque nous affichons les informations d'une assembly, nous voyons notamment sa version ou sa culture. Toutes les informations sur notre assembly se trouvent dans le fichier AssemblyInfo.cs/vb qui est dans le dossier Properties de chacun de vos projets.

```

AssemblyInfo.cs
using System.Reflection;
using System.Runtime.CompilerServices;
using System.Runtime.InteropServices;

// Les informations générales relatives à un assembly
// l'ensemble d'attributs suivant. Changez les valeurs
// associées à un assembly.
[assembly: AssemblyTitle("Chapitre 13")]
[assembly: AssemblyDescription("")]
[assembly: AssemblyConfiguration("")]
[assembly: AssemblyCompany("")]
[assembly: AssemblyProduct("Chapitre 13")]
[assembly: AssemblyCopyright("Copyright © 2008")]
[assembly: AssemblyTrademark("")]
[assembly: AssemblyCulture(")]
  
```

Un exemple du contenu du fichier AssemblyInfo.cs.



Le fichier AssemblyInfo.cs se trouve dans le dossier Properties.

Comme vous pouvez le constater avec l'image d'exemple, chaque attribut est précédé du préfixe

```
'VB
Assembly:
//C#
assembly:
```

Veillez à ne pas l'oublier !

Vous allez pouvoir indiquer dans AssemblyInfo n'importe quelle information grâce aux attributs d'assembly. Nous allons lister la plupart des attributs intéressants contenus dans le Framework .NET et ensuite utiliser chacun d'eux dans un exemple complet :

Valeur	Description
<code>AssemblyAlgorithmId</code>	Vous permet de définir quel type de hash est utilisé pour créer l’empreinte des fichiers de l’assembly dans le manifest.
<code>AssemblyCompany</code>	Permet de spécifier le nom de l’entreprise qui a produit l’assembly.
<code>AssemblyConfiguration</code>	Permet de définir si votre assembly est en mode DEBUG ou RELEASE.
<code>AssemblyCopyright</code>	Permet de spécifier le Copyright appliqué à l’assembly.
<code>AssemblyCulture</code>	Permet de spécifier la culture de l’assembly (voir chapitre 15).
<code>AssemblyDefaultAlias</code>	Permet de spécifier un alias afin de raccourcir le nom de l’assembly.
<code>AssemblyDescription</code>	Permet de spécifier une description de l’assembly.
<code>AssemblyFileVersion</code>	Permet de définir la version du fichier de l’assembly (exe ou dll) dans le système de fichier seulement.
<code>AssemblyInformationalVersion</code>	Permet de définir la version de l’assembly à titre informatif, elle ne sera pas utilisée par le Runtime et n’engendre pas de conflit de version.
<code>AssemblyTitle</code>	Permet de spécifier le nom de l’assembly tel qu’il apparaîtra dans le manifest.
<code>AssemblyTrademark</code>	Permet de spécifier une Trademark.
<code>AssemblyVersion</code>	Permet de spécifier la version de l’assembly qui sera utilisée par le runtime. La version est composée de quatre nombres séparés par des points. Le premier nombre représentant les mises à jour majeures et le derniers les révisions de version. Vous pouvez utiliser un astérisque (*) à la place des troisièmes et quatrièmes nombres pour utiliser les valeurs mise à jour dynamiquement.

```
'VB
<Assembly: AssemblyAlgorithmId(AssemblyHashAlgorithm.SHA1)>
<Assembly: AssemblyCompany("Dotnet-France")>
<Assembly: AssemblyConfiguration("RELEASE")>
<Assembly: AssemblyCopyright("Copyright © 2008")>
<Assembly: AssemblyCulture("fr")>
<Assembly: AssemblyDefaultAlias("MonAssembly")>
<Assembly: AssemblyDescription("Une superbe assembly !")>
<Assembly: AssemblyFileVersion("1.6.2.3")>
<Assembly: AssemblyInformationalVersion("1.6.2.3")>
<Assembly: AssemblyTitle("Assembly Dotnet-France")>
<Assembly: AssemblyTrademark("Patatoïde")>
<Assembly: AssemblyVersion("1.6.*.*")>
```

```
//C#  
[assembly: AssemblyAlgorithmId(AssemblyHashAlgorithm.SHA1)]  
[assembly: AssemblyCompany("Dotnet-France")]  
[assembly: AssemblyConfiguration("RELEASE")]  
[assembly: AssemblyCopyright("Copyright © 2008")]  
[assembly: AssemblyCulture("fr")]  
[assembly: AssemblyDefaultAlias("MonAssembly")]  
[assembly: AssemblyDescription("Une superbe assembly !")]  
[assembly: AssemblyFileVersion("1.6.2.3")]  
[assembly: AssemblyInformationalVersion("1.6.2.3")]  
[assembly: AssemblyTitle("Assembly Dotnet-France")]  
[assembly: AssemblyTrademark("Patatoïde")]  
[assembly: AssemblyVersion("1.6.*.*")]
```

Les explications seront très succinctes : Nous utilisons l'algorithme de hash SHA1 pour créer l'empreinte de l'assembly. C'est une version Release de notre assembly. La culture de notre assembly est le français (fr). Nous pouvons importer notre assembly en utilisant l'alias MonAssembly. La version de notre assembly est la 1.6, le numéro de build et de révisions sont mis à jour dynamiquement à chaque génération.

4 Génération et exécution de code dynamique

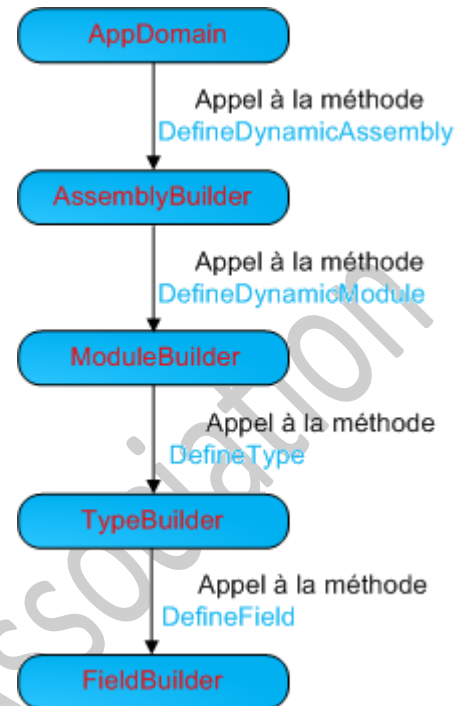
4.1 Génération de code dynamique

Nous savons à présent comment récupérer des informations au sujet d'une assembly chargée dans le programme. Dans cette partie, nous allons nous intéresser à la manière utilisée pour créer du code exécutable de façon dynamique.

En coordination avec les éléments de `System.Reflection`, vous devrez rajouter ceux de l'espace de nom `System.Reflection.Emit`. Dans cet espace de nom, vous pourrez trouver toute une série de classes `<Membre>Builder` qui permettent de concevoir les types d'une assembly ainsi que leurs membres (méthodes, propriétés, champs, constructeurs...).

Pour créer une assembly avec des membres, vous devez suivre un ordre logique de création. Dans tous les cas, il est important de retenir qu'il n'y a aucun constructeur de ces classes, vous devez obligatoirement passer par les méthodes adéquates de la classe qui les englobe comme illustré sur le schéma ci-contre.

L'illustration montre le cheminement à effectuer afin de créer une assembly dynamiquement et comment y ajouter un champ (variable).



Le tableau ci-dessous récapitule les différentes classes disponibles pour concevoir les différents éléments d'une assembly :

Classes	Description
<code>AssemblyBuilder</code>	Contient une assembly.
<code>ConstructorBuilder</code>	Contient un constructeur.
<code>EnumBuilder</code>	Contient une énumération.
<code>EventBuilder</code>	Contient une propriété événement
<code>FieldBuilder</code>	Contient un champ (variable de classe, propriété d'énumération).
<code>LocalBuilder</code>	Contient une variable locale à une méthode.
<code>MethodBuilder</code>	Contient une méthode d'un type (structure ou classe).
<code>ModuleBuilder</code>	Contient un module lié à une assembly.
<code>ParameterBuilder</code>	Contient une liste de paramètres.
<code>PropertyBuilder</code>	Contient une propriété.
<code>TypeBuilder</code>	Contient un type contenu dans un module (Classe, Structure...)

Dans chacune des méthodes permettant de créer l'un de ces objets, vous pouvez utiliser les énumérations associées (par exemple `MethodAttributes` pour la méthode `DefineMethod`) afin de définir la visibilité de chacun de vos membres.

Afin de mieux comprendre comment on crée une assembly complète, voici un code qui va créer une assembly de façon dynamique. Dans cette assembly, le code va ajouter une classe contenant "FirstClass", elle-même contenant un champ nommé "FirstVar" et une méthode "GetValeur" :

```
'VB
Imports System.Reflection.Emit
Imports System.Reflection

Module partie22
    Sub Main()
        'Créé le nom de l'assembly
        Dim nom_assembly As AssemblyName = New
AssemblyName("DynamicAssembly")
        'Créé l'assembleur d'assembly en mode parcourt/écriture
        Dim assem As AssemblyBuilder =
AppDomain.CurrentDomain.DefineDynamicAssembly(nom_assembly,
AssemblyBuilderAccess.RunAndSave)

        'On créé le module associé à l'assembly
        Dim modu As ModuleBuilder =
assem.DefineDynamicModule("DynamicAssembly", "Test.dll", True)

        'Récupère un module et créé une classe FirstClass
        Dim classe As TypeBuilder = modu.DefineType("FirstClass",
TypeAttributes.Class Or TypeAttributes.Public)

        'Créé un champ FirstVar de type String en public
        Dim variable As FieldBuilder = classe.DefineField("FirstVar",
GetType(String), FieldAttributes.Public)

        'Créé une méthode GetValeur
        Dim methode As MethodBuilder = classe.DefineMethod("GetValeur",
MethodAttributes.Public, GetType(String), Nothing)
        'Récupère le générateur de code IL
        Dim code_methode As ILGenerator = methode.GetILGenerator()
        'Entre l'opérateur de retour
        code_methode.Emit(OpCodes.Ret)

        classe.CreateType()

        assem.Save("Test.dll")
    End Sub
End Module
```

Dotnet

```
//C#
using System.Reflection.Emit;
using System.Reflection;
static void Main()
{
    //Crée le nom de l'assembly
    AssemblyName nom_assembly = new AssemblyName("DynamicAssembly");

    //Crée l'assembleur d'assembly en mode parcourt/écriture
    AssemblyBuilder assem =
AppDomain.CurrentDomain.DefineDynamicAssembly(nom_assembly,
AssemblyBuilderAccess.RunAndSave);

    //On crée le module associé à l'assembly
    ModuleBuilder modu = assem.DefineDynamicModule("DynamicAssembly",
"Test.dll", true);

    //Récupère un module et crée une classe FirstClass
    TypeBuilder classe = modu.DefineType("FirstClass",
TypeAttributes.Class | TypeAttributes.Public);

    //Crée un champ FirstVar de type String en public
    FieldBuilder variable = classe.DefineField("FirstVar",
typeof(String), FieldAttributes.Public);

    //Crée une méthode GetValeur
    MethodBuilder methode = classe.DefineMethod("GetValeur",
MethodAttributes.Public, typeof(String), null);

    //Récupère le générateur de code IL
    ILGenerator code_methode = methode.GetILGenerator();

    //Entre l'opérateur de retour
    code_methode.Emit(OpCodes.Ret);

    classe.CreateType();

    assem.Save("Test.dll");
}
```

Ce code est très simple. Il fait appel successivement aux méthodes permettant de créer les membres de l'assembly ainsi que ceux de la classe.

Afin de générer les instructions des méthodes, vous devez utiliser la méthode `Emit` présente sur tous les concepteurs d'éléments permettant d'exécuter des instructions (`MethodBuilder`, `ConstructorBuilder`). Dans cette méthode `Emit`, vous passerez en argument l'un des champs statique de la classe `OpCode`, suivi éventuellement de ce que l'instruction doit utiliser comme valeur.

Générer du code IL dans les méthodes est une opération longue et nécessite l'apprentissage du code MSIL. Aussi, ce point ne sera pas abordé dans ce cours. Cependant, si vous souhaitez en apprendre d'avantage, vous pouvez aller sur [cette section](#) du MSDN.

Une fois que votre classe est dynamiquement créée, vous devez appeler la méthode `CreateType` afin de créer la classe dans l'assembly avec tous les membres qu'elle contient (C'est un peu l'équivalent à la méthode `Flush` utilisée sur les objets de flux).

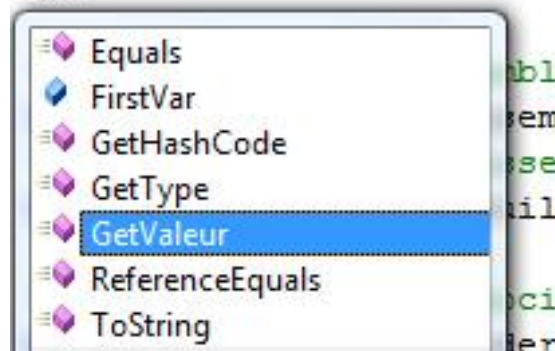
Enfin, si vous souhaitez conserver une trace de l'assembly créée, vous pouvez appeler la méthode `Save` de l'objet `AssemblyBuilder`. Si le nom passé en paramètre est identique à celui que vous avez indiqué pour créer le module, un seul fichier sera généré ; dans le cas contraire, deux fichiers seront générés : Un fichier portant le nom passé dans la méthode `Save` qui contiendra notre assembly et un autre portant le nom passé dans la méthode `DefineDynamicModule` qui sera le module lié à l'assembly.

Afin de tester si la génération a bien fonctionné, vous pouvez importer l'assembly ainsi générée dans l'un de vos projets :

Références :

Nom de la référe...	Type	Vers...	Copie locale	Chemin d'accès
System	.NET	2.0.0.0	False	C:\Windows\Microsc
System.Data	.NET	2.0.0.0	False	C:\Windows\Microsc
System.Deployment	.NET	2.0.0.0	False	C:\Windows\Microsc
System.Xml	.NET	2.0.0.0	False	C:\Windows\Microsc
Test	.NET	0.0.0.0	True	D:\Documents\dotN

```
Dim fc As FirstClass = Ne
fc.
```



Vous pouvez constater que notre méthode `GetValeur` ainsi que la variable `FirstVar` sont bien présentes dans la classe `FirstClass`.

4.2 Exécution dynamique de code

Maintenant que nous savons comment récupérer des informations et comment générer du code, nous allons voir comment utiliser des objets et leurs membres sans les connaître avant la compilation. Nous allons ainsi pouvoir récupérer la liste des classes d'une assembly, les instancier, et invoquer les méthodes de chacun des objets sans connaître le contenu de l'assembly.

Dans les exemples à venir, nous reprendrons la librairie de classe vue dans la partie précédente, et nous allons instancier deux objets et invoquer leurs méthodes, propriétés et méthodes statique.

Remarque : Dans un souci de rapidité du code, nous admettons que nous connaissons les noms des éléments de l'assembly.

4.2.1 Instancier un objet dynamiquement

Première étape, nous allons instancier nos classes sans les connaître.

Tout d'abord nous devons utiliser la classe `Assembly` pour ouvrir notre assembly et récupérer tous les types à l'intérieur :

```
'VB
Dim assembly As Assembly = assembly.LoadFrom("Chapitre 13 - Lib.dll")

For Each type As Type In assembly.GetTypes()
Next

//C#
Assembly assembly = Assembly.LoadFrom("Chapitre 13 - Lib.dll");

foreach (Type type in assembly.GetTypes())
{
}
```

Nous chargeons donc notre librairie de classes et nous récupérons l'ensemble des types qu'elle contient.

Ensuite deux méthodes s'offrent à nous pour instancier nos classes.

La première consiste simplement à utiliser une méthode statique de la classe `Activator.CreateInstance` à laquelle on passe en paramètre le type que l'on souhaite instancier :

```
'VB
Dim entites As Object
For Each type As Type In assembly.GetTypes()
    If (Not type.IsAbstract And Not type.IsEnum And type.IsVisible) Then
        entites = Activator.CreateInstance(type)
    End If
Next

//C#
Object entites;
foreach (Type type in assembly.GetTypes())
{
    entites = Activator.CreateInstance(type);
}
```

La méthode `CreateInstance` crée une instance de chacune des classes dans l'assembly, et ses nombreuses surcharges vous permettront de passer des paramètres à vos constructeurs.

Nous testons également si les type sont bien instanciable et visible à l'extérieur de l'assembly pour éviter les conflits avec les autres classes qui seraient déclarées internal (comme c'est le cas si le projet est écrit/compilé en VB.NET)

La seconde méthode consiste à récupérer le constructeur de chaque type trouvé dans notre librairie, et de l'invoquer :

```
'VB
Dim entites As Object
For Each type As Type In assembly.GetType()
    If (Not type.IsAbstract And Not type.IsEnum And type.IsVisible) Then
        Dim arguments() As Type = type.EmptyTypes
        Dim constructeur As ConstructorInfo =
type.GetConstructor(arguments)
        entites = constructeur.Invoke(New Object() {})
    End If
Next

//C#
Object entites;
foreach (Type type in assembly.GetType())
{
    Type[] arguments = Type.EmptyTypes;
    ConstructorInfo constructeur = type.GetConstructor(arguments);
    entites = constructeur.Invoke(new object[] { });
}
```

Nous créons d'abord un tableau `arguments` de type `Type` et nous indiquons qu'il est vide (notre constructeur ne prenant pas de paramètres) grâce à `Type.EmptyTypes`.

Ensuite nous récupérons le constructeur de chaque type de l'assembly qui contient les mêmes arguments que passé en paramètre.

Nous pouvons maintenant invoquer notre constructeur. Pour cela, nous allons utiliser la méthode `Invoke` de l'objet constructeur. Celle-ci prend en paramètre les arguments à passer à notre constructeur. Ici, nous instancions un tableau d'objet vide car notre constructeur ne prend aucun paramètre.

Enfin, dans chaque cas nous assignons le résultat de l'instanciation à un objet `Entites` de type `Object`.

4.2.2 Invoquer des méthodes

Nous avons pu instancier notre objet, nous allons maintenant invoquer chacune des méthodes qu'il contient, et ce, sans se soucier de leur nom.

Pour cela, nous récupérons la liste des méthodes de chaque type, et nous les invoquons une à une (Ces codes sont à ajouter dans la boucle foreach principale).

```
'VB
For Each methode As MethodInfo In type.GetMethods(BindingFlags.Instance
Or BindingFlags.Public Or BindingFlags.DeclaredOnly)
    If (Not methode.IsSpecialName) Then
        Console.WriteLine("Méthode " + methode.Name + " renvoie : "
+ methode.Invoke(entites, New Object() {}))
    End If
Next

//C#
foreach (MethodInfo methode in type.GetMethods(BindingFlags.Instance |
BindingFlags.Public | BindingFlags.DeclaredOnly))
{
    Console.WriteLine("Méthode " + methode.Name + " renvoie : " +
methode.Invoke(Entites, new object[] { }));
}
```

Nous avons donc récupéré toutes les méthodes de chaque classe, avec les BindingFlags Instance, Public et DeclaredOnly qui nous permettent de n'avoir que celles que nous avons créées tout en omettant les méthodes héritées. Le test nous permet d'exclure les propriétés déclarées dans les classes.

Ensuite nous affichons le nom de la méthode, puis sa valeur de retour. Pour récupérer la valeur de retour, nous invoquons la méthode. Pour cela nous mettons en tant que premier paramètre l'objet contenant la méthode, c'est-à-dire l'instance de notre objet contenu dans Entites, et en second paramètre, les arguments de la méthode, ici un tableau vide car aucune méthode ne possède d'argument dans notre exemple.

Nous pouvons également invoquer des méthodes statiques, pour cela, nous pouvons omettre le premier paramètre de la méthode `Invoke`.

```
'VB
Console.WriteLine("Méthode statique : " +
type.GetMethod("parler").Invoke(Nothing, New Object() {"Bertrand"}))

//C#
Console.WriteLine("Méthode statique : " +
type.GetMethod("parler").Invoke(null, new object[] { "Bertrand" }));
```

Ici, nous fournissons le nom de la méthode statique car il n'y en a qu'une seule, mais nous aurions pu utiliser une boucle foreach combinée à la méthode `GetMethods` et au `BindingFlags Static`.

Nous ne fournissons pas de premier paramètre, en effet nous avons à faire à une méthode statique, aucune instance d'objet n'est nécessaire. Enfin, nous passons un paramètre à notre méthode statique : une chaîne de caractère.

4.2.3 Invoquer une propriété

Invoquer une propriété est quasi semblable à ce que nous avons vu jusqu'à présent. Une fois que nous aurons vu le cas de la propriété, vous devriez être plus à l'aise avec l'ensemble des méthodes disponibles.

```
'VB
Console.WriteLine("Propriété nom : " +
type.GetProperty("Nom").GetValue(entites, Nothing))

//C#
Console.WriteLine("Propriété nom : " +
type.GetProperty("Nom").GetValue(Entites, null));
```

Nous fournissons donc le nom de la propriété car nous n'en avons qu'une dans notre classe, mais là encore, nous aurions pu utiliser la méthode `GetProperties`.

La petite différence par rapport à l'invocation de méthode est qu'ici, nous allons utiliser la méthode `GetValue`. Celle-ci prend en premier paramètre l'instance de l'objet concerné, et en second paramètre, nous pouvons fournir un index pour les accesseurs à index.

4.2.4 Résultat final

Après avoir vu en détail le code de notre exemple, je vous propose de voir ce que cela donne si nous l'exécutons après l'avoir complété un petit peu. Nous vous invitons à jeter un œil dans les sources fournis avec les cours afin de voir l'exemple au complet dans les projets Chapitre 13 et Chapitre 13 – Lib.

```
Méthode sauter renvoie : MonPersonnage saute de 1 metre
Méthode ToString renvoie : MonPersonnage
Methode statique : coucou Bertrand
Propriété nom : MonPersonnage
Méthode rouler renvoie : MaVoiture se met à rouler !!
Méthode ToString renvoie : MaVoiture
Methode statique : Vroum vroum Bertrand
Propriété nom : MaVoiture
```

5 Conclusion

Nous vous avons présenté ici les bases de la réflexion, l'espace de nom System.Reflection est cela dit très vaste et vous trouverez de nombreuses méthodes et propriétés capables de répondre à vos besoins. N'hésitez pas à utiliser Reflector pour vous rendre compte des possibilités de la Réflexion.

A la fin de ce chapitre, vous devriez être capable de :

- Utiliser les classes Assembly et Module afin de récupérer des métadonnées.
- Connaitre les attributs de bases à utiliser dans AssemblyInfo.
- Récupérer les membres d'une assembly, d'un module ou d'un type, et les invoquer.
- Savoir utiliser les classes « Builder » afin de construire des assemblies, des modules et des types.

Dans tous les cas, n'hésitez pas à aller visiter le [MSDN](http://msdn.microsoft.com) qui peut vous apporter beaucoup d'aide dans vos développements :

