

La sérialisation XML avec .NET

Par [Thomas Levesque](#)  


Date de publication : 26 février 2009


Dernière mise à jour : 26 février 2009

Ce tutoriel explique comment sérialiser et désérialiser des objets en XML à l'aide des fonctionnalités du .NET Framework. On verra notamment comment contrôler très finement le schéma des documents XML générés par la sérialisation.
Commentez cet article :

Introduction.....	3
I - La classe XmlSerializer.....	3
I-A - Sérialiser un objet.....	4
I-A-1 - Un exemple simple.....	4
I-A-2 - Membres de type non-primitif.....	5
I-A-3 - Sérialisation d'une collection.....	6
I-B - Désérialiser un objet.....	8
I-B-1 - La même chose, à l'envers.....	8
I-B-2 - Cas particulier des collections.....	9
II - Personnaliser la sérialisation avec les attributs de contrôle.....	9
II-A - L'attribut XmlIgnore.....	9
II-B - L'attribut XmlElement.....	10
II-C - Les attributs XmlArray et XmlArrayItem.....	11
II-D - L'attribut XmlRoot.....	11
II-E - L'attribut XmlAttribute.....	12
II-F - L'attribut XmlEnum.....	12
II-G - Contrôler le format d'une propriété.....	13
III - Gestion de l'héritage.....	14
III-A - Le problème.....	14
III-B - Sérialisation de classes dérivées.....	15
III-B-1 - L'attribut XmlInclude.....	15
III-B-2 - L'attribut XmlElement.....	16
III-C - Collections hétérogènes.....	17
III-C-1 - L'attribut XmlArrayItem.....	17
III-C-2 - L'attribut XmlElement.....	17
III-D - Approches "dynamiques".....	18
IV - Personnalisation avancée avec l'interface IXmlSerializable.....	19
IV-A - La méthode GetSchema.....	20
IV-B - La méthode WriteXml.....	20
IV-C - La méthode ReadXml.....	22
V - Utiliser l'outil XML Schema Definition Tool.....	22
V-A - Générer le schéma à partir des classes.....	22
V-B - Générer les classes à partir du schéma.....	23
Conclusion.....	23
Remerciements.....	24

Introduction

Dans n'importe quel type d'application, on a souvent besoin de  **sérialiser** des objets, c'est-à-dire les "enregistrer" sous une forme qui permettra de les reconstituer ultérieurement, ou encore de les transmettre sur un réseau. Il peut s'agir des préférences de l'utilisateur, de l'état de l'application, d'un document, d'une commande envoyée à un service, etc...


Il existe de nombreuses approches pour sérialiser des objets (binaire, texte brut, formats propriétaires, XML). Depuis quelques années,  **XML** s'impose comme un des formats les plus utilisés. Le .NET Framework fournit différents composants pour manipuler des données en XML : celui auquel on va s'intéresser dans cet article permet de sérialiser en XML quasiment n'importe quel objet, de façon générique.



Pré-requis : cet article s'adresse à des personnes maîtrisant les bases du langage C#.

I - La classe XmlSerializer

La classe *XmlSerializer*, disponible depuis les toutes premières versions de .NET, permet de sérialiser un objet en XML. Pour l'utiliser, il faut :

- Que l' **assembly** *System.Xml* soit référencé dans le projet (c'est généralement le cas par défaut dans un nouveau projet)
- Ajouter dans le fichier source une directive *using* pour le namespace *System.Xml.Serialization*

Pour se faire une idée du format généré par la classe *XmlSerializer*, voici un exemple basique. Soit la classe suivante :

```
public class Person
{
    public int Id { get; set; }
    public string LastName { get; set; }
    public string FirstName { get; set; }
}
```

Si on sérialise un objet de ce type en XML, on obtient le résultat suivant :

```
<?xml version="1.0" encoding="utf-8"?>
<Person xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/
XMLSchema">
  <Id>123</Id>
  <LastName>Dupond</LastName>
  <FirstName>Jean</FirstName>
</Person>
```

On voit tout de suite que ce format a un gros avantage : il est très facilement lisible, et donc compréhensible et modifiable par un humain, ce qui est très utile pour certains types de documents (fichiers de configuration par exemple).



Remarquez que l'élément racine du document XML généré a le nom de la classe sérialisée, et chaque propriété ou champ public de l'objet est représenté par un élément XML. C'est le comportement par défaut de la classe *XmlSerializer*.



Notez que ce format ne correspond à aucun standard particulier, et n'est pas nécessairement interopérable avec d'autres implémentations de la sérialisation XML. En revanche, il est possible, comme on le verra plus loin, de personnaliser ce format pour le faire correspondre au standard voulu.

I-A - Sérialiser un objet

I-A-1 - Un exemple simple

Pour commencer, reprenons l'exemple présenté plus haut. Voici le code qui permet d'obtenir le même résultat :


```
Person p = new Person
{
    Id = 123,
    LastName = "Dupond",
    FirstName = "Jean"
};

XmlSerializer xs = new XmlSerializer(typeof(Person));
using (StreamWriter wr = new StreamWriter("person.xml"))
{
    xs.Serialize(wr, p);
}
```

On a donc effectué les étapes suivantes :

- Création d'une instance de *XmlSerializer* : on passe en paramètre du constructeur le type d'objet à sérialiser
- Ouverture d'un *StreamWriter* sur le fichier de destination
- Sérialisation de l'objet vers ce *StreamWriter*, avec la méthode *Serialize*


Vous voyez qu'il est donc très facile de sauvegarder un objet en XML : la sérialisation proprement dite prend 3 lignes, sans compter les accolades...

 Ici on a utilisé un objet *StreamWriter* (hérité de *TextWriter*), mais la méthode *Serialize* peut aussi prendre en paramètre un *Stream* (*FileStream*, *NetworkStream*, *MemoryStream*, etc) ou un *XmlWriter*. Cela permet de sérialiser des données sur n'importe quel type de support (mémoire, fichier, réseau, etc).

Avant d'aller plus loin, sachez qu'il existe cependant des contraintes à respecter pour que vos objets soient sérialisables :

- Les classes, structures, et énumérations sont sérialisables. Les interfaces **ne sont pas** sérialisables
- Les types à sérialiser doivent être publics
- Seuls les champs et propriétés publics sont sérialisés
- Les propriétés à sérialiser ne doivent pas être en lecture seule (les propriétés qui renvoient une collection sont une exception à cette règle, nous y reviendrons plus tard)
- Le type à sérialiser doit posséder un constructeur par défaut (public et sans paramètres) : cela est nécessaire pour que la désérialisation puisse créer une instance de ce type.
- Les classes qui implémentent *IDictionary* **ne sont pas** sérialisables. Il est cependant possible de contourner cette limitation en implémentant l'interface *IXmlSerializable*, dont on reparlera plus tard.

I-A-2 - Membres de type non-primitif

Dans notre exemple précédent, la classe *Person* est très simple, et n'a que des propriétés de  **type primitif** (nombres et chaîne de caractères). Compliquons maintenant un peu les choses et voyons ce qu'il se passe lorsque notre classe a des propriétés de type plus complexe, par exemple une propriété *Address* :

```
public Address Address { get; set; }
```

Et voici la définition de la classe *Address* :

```
public class Address
{
    public string Street { get; set; }
    public string ZipCode { get; set; }
    public string City { get; set; }
    public string Country { get; set; }
}
```

Sérialisons maintenant cet objet, comme dans l'exemple précédent :

```
Person p = new Person
{
    Id = 123,
    LastName = "Dupond",
    FirstName = "Jean",
    Address = new Address
    {
        Street = "1, rue du petit pont",
        ZipCode = "75005",
        City = "Paris",
        Country = "France"
    }
};

XmlSerializer xs = new XmlSerializer(typeof(Person));
```

```
using (StreamWriter wr = new StreamWriter("person.xml"))
{
    xs.Serialize(wr, p);
}
```

Voici le résultat obtenu :

```
<?xml version="1.0" encoding="utf-8"?>
<Person xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/
XMLSchema">
  <LastName>Dupond</LastName>
  <FirstName>Jean</FirstName>
  <Id>123</Id>
  <Address>
    <Street>1, rue du petit pont</Street>
    <ZipCode>75005</ZipCode>
    <City>Paris</City>
    <Country>France</Country>
  </Address>
</Person>
```

On peut donc observer que la propriété Address est sérialisée dans un élément XML du même nom, et que les éléments enfants correspondent aux propriétés de Address. Chaque membre public de l'objet à sérialiser est donc parcouru récursivement, jusqu'à tomber sur des objets de type primitif qui ne peuvent plus être "décomposés".

I-A-3 - Sérialisation d'une collection

La sérialisation d'une collection s'effectue suivant le même principe que pour les autres types d'objets, mais il est intéressant d'observer le XML généré. Supposons qu'on veut manipuler un "carnet d'adresses", sous forme d'une liste de personnes. Voyons comment sérialiser cette liste :

```
List<Person> contactBook = new List<Person>();

Person jean = ... // inutile de s'appesantir sur l'initialisation des objets...
Person jacques = ...

contactBook.Add(jean);
contactBook.Add(jacques);

XmlSerializer xs = new XmlSerializer(typeof(List<Person>));
using (StreamWriter wr = new StreamWriter("contactBook.xml"))
{
    xs.Serialize(wr, contactBook);
}
```

Comme on peut le voir, il n'y a rien de très nouveau dans ce code... Voyons maintenant le XML généré :

```
<?xml version="1.0" encoding="utf-8"?>
<ArrayOfPerson xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/
XMLSchema">
  <Person>
    <LastName>Dupond</LastName>
    <FirstName>Jean</FirstName>
    <Id>123</Id>
    <Address>
      <Street>1, rue du petit pont</Street>
      <ZipCode>75005</ZipCode>
      <City>Paris</City>
      <Country>France</Country>
    </Address>
  </Person>
  <Person>
    <LastName>Durand</LastName>
    <FirstName>Jacques</FirstName>
    <Id>521</Id>
```

```

<Address>
  <Street>2, rue du grand tunnel</Street>
  <ZipCode>75018</ZipCode>
  <City>Paris</City>
  <Country>France</Country>
</Address>
</Person>
</ArrayOfPerson>
  
```

Il y a 2 choses à remarquer dans la structure de ce document :

- L'élément racine n'est pas `<List<Person>>` (et pour cause, les caractères `<` et `>` étant des caractères spéciaux en XML), ni même `<ListOfPerson>`, comme on aurait pu s'y attendre, mais `<ArrayOfPerson>`. Mais alors, qu'aurait on eu comme élément racine si on avait utilisé un tableau de `Person` (`Person[]`), ou une `Collection<Person>` ? La même chose ! La sérialisation XML ne fait pas de différence entre les différents types de collection, tout simplement parce que ce n'est pas nécessaire : le type avec lequel on a initialisé le `XmlSerializer` indique le type réel à utiliser.
- Les éléments de la liste n'ont pas de nom, puisqu'ils ne correspondent pas à une propriété d'un objet : ils sont donc sérialisés dans un élément qui a le nom de leur type.

Remarque importante : *il est nécessaire que la collection à sérialiser soit typée, afin que le `XmlSerializer` sache quel type d'objet sérialiser. Si on avait écrit le même code avec une collection non typée, `ArrayList` par exemple, on aurait eu une exception à la sérialisation : en effet, le `XmlSerializer` aurait considéré que les éléments d'une `ArrayList` sont de type `Object`, et en l'absence d'indications complémentaires, n'aurait pas su quoi faire d'un objet de type `Person`. On verra au chapitre III comment gérer ce type de problématique.*

Concernant les collections, il y a encore un point intéressant à observer : la sérialisation d'un objet qui a des propriétés de type collection. Par exemple, supposons qu'on veut maintenant pouvoir enregistrer plusieurs adresses pour une même personne : on va remplacer la propriété `Address` par une propriété `Addresses`, de type `List<Address>` :

```

public List<Address> Addresses { get; set; }

// Et on n'oublie pas de l'initialiser dans le constructeur :
public Person()
{
    this.Addresses = new List<Address>();
}
  
```

Je ne m'attarde pas sur le code de sérialisation, qui est identique à ce qu'on a déjà utilisé plus haut. Le XML obtenu est le suivant :

```

<?xml version="1.0" encoding="utf-8"?>
<Person xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <LastName>Dupond</LastName>
  <FirstName>Jean</FirstName>
  <Id>123</Id>
  <Addresses>
    <Address>
      <Street>1, rue du petit pont</Street>
      <ZipCode>75005</ZipCode>
      <City>Paris</City>
      <Country>France</Country>
    </Address>
    <Address>
      <Street>2, rue du grand tunnel</Street>
      <ZipCode>75018</ZipCode>
      <City>Paris</City>
      <Country>France</Country>
    </Address>
  </Addresses>
</Person>
  
```

On remarque que dans ce cas, *Addresses* est sérialisée sous forme d'un élément *<Addresses>*, et non *<ArrayOfAddress>* : puisque c'est une propriété d'un objet, le nom de la propriété est utilisé de préférence au type.

I-B - Désérialiser un objet


I-B-1 - La même chose, à l'envers...

La désérialisation d'un objet se fait grâce à la méthode *Deserialize* de la classe *XmlSerializer*. Voyons comment reconstruire l'objet de l'exemple précédent à partir du fichier :

```
XmlSerializer xs = new XmlSerializer(typeof(Person));  
using (StreamReader rd = new StreamReader("person.xml"))  
{  
    Person p = xs.Deserialize(rd) as Person;  
    Console.WriteLine("Id : {0}", p.Id);  
    Console.WriteLine("Nom : {0} {1}", p.FirstName, p.LastName);  
}
```

On a donc effectué les étapes suivantes :

- Création d'une instance de *XmlSerializer* (comme pour la sérialisation)
- Ouverture d'un *StreamReader* pour lire le fichier source
- Désérialisation de l'objet à partir de ce *StreamReader*, avec la méthode *Deserialize*
- Conversion (cast) de l'objet obtenu vers le type *Person*

 *Notez que la méthode *Deserialize* renvoie un *Object* : il est donc nécessaire d'effectuer un cast vers le type voulu afin de pouvoir le manipuler.*

La désérialisation ne présente donc pas plus de difficultés que la sérialisation, c'est pourquoi on ne s'appesantira pas dessus plus longtemps. A de rares exceptions près, la désérialisation ne posera pas de problème particulier, en utilisant un *XmlSerializer* identique à celui utilisé pour la sérialisation.

I-B-2 - Cas particulier des collections

Je voudrais expliquer ici un aspect spécifique à la désérialisation des collections. J'ai mentionné un peu plus haut qu'une propriété qui renvoie une collection pouvait être en lecture seule, contrairement aux autres propriétés. C'est valable à condition que la collection en question soit explicitement initialisée dans le constructeur de l'objet. En effet, lors de la désérialisation d'une propriété de type collection, le traitement suivant est effectué :

- La valeur actuelle de la propriété est lue à l'aide de l'accesseur *get*
- Si la valeur renvoyée est une référence nulle, la collection est initialisée avec l'accesseur *set*. Si ce dernier n'existe pas (propriété en lecture seule), une exception est levée
- Les éléments de la collection sont désérialisés et ajoutés à la collection

Ce comportement a une conséquence importante : si des éléments sont ajoutés à la collection lors de l'initialisation, il ne seront pas supprimés lors de la désérialisation : les éléments désérialisés seront ajoutés à ceux déjà présents dans la collection.

II - Personnaliser la sérialisation avec les attributs de contrôle

Jusqu'ici, on a laissé le *XmlSerializer* gérer automatiquement le schéma du XML généré. Cependant, dans certains cas, on aura besoin de se conformer à un schéma imposé, ou encore de ne pas sérialiser certains membres. Le namespace *System.Xml.Serialization* fournit plusieurs attributs qui permettent de modifier le comportement du *XmlSerializer* ; ce chapitre décrit les plus couramment utilisés.

II-A - L'attribut *XmlIgnore*

Il est courant que les objets qu'on utilise aient des propriétés liées à leur état durant l'exécution de l'application : dans ce cas, les sérialiser est inutile, et non souhaitable, puisqu'elles perdent tout leur sens hors du contexte de l'application en cours d'exécution. L'attribut *XmlIgnore*, appliqué à un champ ou à une propriété, permet d'indiquer au *XmlSerializer* que le membre en question ne doit pas être sérialisé.

Supposons par exemple que notre classe *Person* ait une propriété *IsSelected*, pour indiquer si la personne est sélectionnée dans une liste au sein de l'application (la pertinence de mettre une telle propriété dans la classe *Person* est très discutable, mais c'est pour les besoins de l'exemple...). Voilà comment exclure cette propriété de la sérialisation :

```
[XmlIgnore]
public bool IsSelected { get; set; }
```

II-B - L'attribut XmlElement

Supposons que, pour une raison ou une autre, on ait besoin de modifier le schéma XML de notre carnet d'adresses pour que les balises soient en français. La première idée qui vient à l'esprit est de renommer les classes et propriétés pour qu'elles soient sérialisées avec le nom voulu... mais cela oblige à modifier tout le code qui utilise la classe *Person* ! Même si la refactorisation du code dans Visual Studio est assez efficace, on n'est jamais certain que ça n'aura pas d'effets indésirables. Et beaucoup de développeurs (dont moi) préfèrent coder avec des identifiants en anglais...

Heureusement, ce cas a été prévu... Il est possible, grâce à l'attribut *XmlElement*, de définir le nom de l'élément XML qui sera généré pour un champ ou une propriété. Exemple avec la classe *Address* :

```
public class Address
{
    [XmlElement("Rue")]
    public string Street { get; set; }
    [XmlElement("CodePostal")]
    public string ZipCode { get; set; }
    [XmlElement("Ville")]
    public string City { get; set; }
    [XmlElement("Pays")]
    public string Country { get; set; }
}
```

Voici un extrait du XML généré :

```
...
<Address>
  <Rue>1, rue du petit pont</Rue>
  <CodePostal>75005</CodePostal>
  <Ville>Paris</Ville>
  <Pays>France</Pays>
</Address>
...
```

Cas particulier des collections : l'attribut *XmlElement* a un effet particulier lorsqu'il est appliqué à une propriété qui renvoie une collection : il spécifie que la collection sera sérialisée comme une séquence d'éléments de même niveau que les autres propriétés de l'objet (et non contenus dans un élément "englobant" du nom de la propriété). Il permet aussi de spécifier le nom des éléments de la séquence. Ce sera plus clair avec un exemple... si on applique l'attribut *XmlElement* à la propriété *Addresses* :

```
[XmlElement("Adresse")]
public List<Address> Addresses { get; set; }
```

On obtient le XML suivant :

```
<?xml version="1.0" encoding="utf-8"?>
<Person xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema" Nom="Dupond">
  <Id>123</Id>
  <Prénom>Jean</Prénom>
  <Adresse>
    <Rue>1, rue du petit pont</Rue>
    <CodePostal>75005</CodePostal>
    <Ville>Paris</Ville>
    <Pays>France</Pays>
  </Adresse>
  <Adresse>
    <Rue>2, rue du grand tunnel</Rue>
    <CodePostal>75018</CodePostal>
    <Ville>Paris</Ville>
    <Pays>France</Pays>
  </Adresse>
</Person>
```

```
</Personne>
```

On voit donc que tous les éléments *<Adresse>* sont directement sous l'élément *<Person>*, sans élément englobant *<Adresses>* comme c'était le cas avant.

L'attribut *XmlElement* permet aussi de spécifier un nom d'élément XML différent selon le type de l'objet. On abordera cet aspect plus en détails dans le chapitre III.

II-C - Les attributs XmlArray et XmlArrayItem

On utilise l'attribut *XmlArray* pour personnaliser le nom de l'élément XML correspondant à une propriété qui renvoie une collection. Voyons donc comment faire pour que la propriété *Adresses* soit sérialisée dans un élément *Adresses* (en français, avec un seul "d") :

```
[XmlArray("Adresses")]
public List<Address> Adresses { get; set; }
```

Malheureusement, ce n'est pas encore suffisant : les éléments de la liste sont sérialisés dans un élément XML *Address*, et non *Adresse*. On peut corriger ça avec l'attribut *XmlArrayItem*, qui permet d'indiquer l'élément XML à utiliser pour les éléments de la collection :

```
[XmlArray("Adresses")]
[XmlArrayItem("Adresse")]
public List<Address> Adresses { get; set; }
```

On obtient bien maintenant la forme souhaitée :

```
<?xml version="1.0" encoding="utf-8"?>
<Person xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/
XMLSchema" Nom="Dupond">
  <Id>123</Id>
  <Prénom>Jean</Prénom>
  <Adresses>
    <Adresse>
      <Rue>1, rue du petit pont</Rue>
      <CodePostal>75005</CodePostal>
      <Ville>Paris</Ville>
      <Pays>France</Pays>
    </Adresse>
    <Adresse>
      <Rue>2, rue du grand tunnel</Rue>
      <CodePostal>75018</CodePostal>
      <Ville>Paris</Ville>
      <Pays>France</Pays>
    </Adresse>
  </Adresses>
</Person>
```

II-D - L'attribut XmlRoot

L'attribut *XmlRoot* est similaire à *XmlElement*, mais s'applique à une classe et non à un champ ou propriété. Il permet de définir quel élément XML utiliser pour sérialiser l'objet en tant que racine du document XML. Exemple avec la classe *Person* :

```
[XmlRoot("Personne")]
public class Person
{
    ...
}
```

On obtient le XML suivant :

```
<Personne xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/
XMLSchema">
  <Id>123</Id>
  <Nom>Dupond</Nom>
  <Prénom>Jean</Prénom>
  ...
```

II-E - L'attribut XmlAttribute

Il peut arriver qu'on veuille sérialiser un champ ou une propriété, non pas sous forme d'un élément XML, mais sous forme d'un attribut de son élément parent ; c'est le rôle de l'attribut *XmlAttribute*. Voyons comment mettre la propriété *Id* sous forme d'un attribut :

```
[XmlAttribute("Id")]
public int Id { get; set; }
```


On obtient le XML suivant :

```
<?xml version="1.0" encoding="utf-8"?>
<Personne xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/
XMLSchema" Id="123">
  <Nom>Dupond</Nom>
  <Prénom>Jean</Prénom>
  ...
```

II-F - L'attribut XmlEnum

Supposons maintenant qu'on veuille pouvoir indiquer dans l'adresse s'il s'agit de l'adresse de livraison, de facturation, ou les deux. On va donc créer une énumération *AddressType* :

```
[Flags]
public enum AddressType
{
    None = 0,
    Ship = 1,
    Bill = 2
}
```

 **Remarque** l'attribut *Flags* sur l'énumération : il indique que les valeurs peuvent être combinées par un OU binaire (opérateur `|`). Cet attribut n'est habituellement pas obligatoire, mais il l'est dès lors qu'on veut sérialiser cette énumération en XML. Si l'attribut *Flags* n'est pas présent, une exception sera levée si on cherche à sérialiser une valeur qui ne fait pas explicitement partie de l'énumération.

On ajoute à la classe *Adresse* une propriété *AddressType* (en la traduisant au passage...)


```
[XmlElement("TypeAdresse")]
public AddressType AddressType { get; set; }
```

Par défaut, chaque membre de l'énumération est sérialisé avec le nom sous lequel il a été déclaré :

```
...
<Adresse>
  <Rue>1, rue du petit pont</Rue>
  <CodePostal>75005</CodePostal>
  <Ville>Paris</Ville>
  <Pays>France</Pays>
  <TypeAdresse>Ship Bill</TypeAdresse>
```

```
</Adresse>
```

```
...
```

 On voit ici qu'une combinaison de valeurs de l'énumération est sérialisée en indiquant les noms des différentes valeurs, séparés par des espaces.

Maintenant, on voudrait que le XML ne contienne pas "None", "Ship" ou "Bill", mais "Aucun", "Livraison", "Facturation". C'est là que l'attribut *XmlEnum* entre en jeu :

```
[Flags]
public enum AddressType
{
    [XmlEnum("Aucun")]
    None = 0,
    [XmlEnum("Livraison")]
    Ship = 1,
    [XmlEnum("Facturation")]
    Bill = 2
}
```

Et on obtient le XML suivant :

```
...
<Adresse>
  <Rue>1, rue du petit pont</Rue>
  <CodePostal>75005</CodePostal>
  <Ville>Paris</Ville>
  <Pays>France</Pays>
  <TypeAdresse>Livraison Facturation</TypeAdresse>
</Adresse>
...
```

Notez que, puisque les différentes valeurs de la combinaison sont séparées par des espaces, il ne faut pas mettre d'espace dans le nom de la valeur. Si vous le faites, la sérialisation fonctionnera, mais la désérialisation lèvera une exception, car seul le premier mot aura été pris en compte.

II-G - Contrôler le format d'une propriété

Dans cette section, je ne vais pas présenter un nouvel attribut de contrôle, mais une "astuce" pour pouvoir contrôler le format dans lequel les données sont sérialisées. Dans certains cas, le format par défaut peut ne pas convenir, par exemple dans le cas d'une date. Ajoutons à la classe *Person* une date de naissance *DateOfBirth*, qu'on veut sérialiser dans un élément XML *<DateDeNaissance>* :

```
[XmlElement("DateDeNaissance")]
public DateTime DateOfBirth { get; set; }
```

Lors de la sérialisation de cette propriété, on obtient le résultat suivant :

```
...
<DateDeNaissance>1980-10-12T00:00:00</DateDeNaissance>
...
```

La date est donc au format américain (année-mois-jour), avec l'heure à la fin. Or, on voudrait avoir la date au format français (jour/mois/année), sans l'heure. Malheureusement il n'existe aucun attribut de sérialisation pour réaliser cela, mais on peut s'en sortir en utilisant ceux qu'on connaît déjà...

Pour commencer, définissons une nouvelle propriété *DateOfBirthFormatted* qui va renvoyer la date de naissance dans le format voulu, et accepter une date dans ce même format pour modifier la date de naissance :

```
public string DateOfBirthFormatted
```

```
{
    get { return DateOfBirth.ToString("dd/MM/yyyy", CultureInfo.InvariantCulture); }
    set { DateOfBirth = DateTime.ParseExact(value, "dd/MM/yyyy",
        CultureInfo.InvariantCulture); }
}
```

Maintenant, on voudrait sérialiser cette propriété à la place de *DateOfBirth*. Il suffit pour cela d'appliquer à *DateOfBirth* l'attribut *XmlIgnore*, pour la "masquer", et de renommer l'élément XML correspondant à *DateOfBirthFormatted* :

```
[XmlIgnore]
public DateTime DateOfBirth { get; set; }

[XmlElement("DateDeNaissance")]
public string DateOfBirthFormatted
{
    get { return DateOfBirth.ToString("dd/MM/yyyy", CultureInfo.InvariantCulture); }
    set { DateOfBirth = DateTime.ParseExact(value, "dd/MM/yyyy",
        CultureInfo.InvariantCulture); }
}
```

Et on obtient bien le résultat voulu :

```
...
<DateDeNaissance>12/10/1980</DateDeNaissance>
...
```

Cette astuce est particulièrement pratique pour les dates, mais peut aussi être utilisée pour n'importe quelle propriété dont on souhaite contrôler soi-même le format.

III - Gestion de l'héritage

III-A - Le problème

Supposons qu'on spécialise notre classe *Person* pour gérer les employés d'une entreprise. On va donc créer une classe *Employee*, héritée de *Person* :

```
public class Employee : Person
{
    public string Company { get; set; }
    public string Position { get; set; }
    public double Salary { get; set; }
}
```

Voyons maintenant ce qui se passe quand on affecte un *Employee* là où un objet *Person* est attendu. Créons d'abord une classe *ContactBook* qui va servir de conteneur à notre liste de personnes, en incluant les informations sur le propriétaire du carnet d'adresse :

```
public class ContactBook
{
    public ContactBook()
    {
        this.Contacts = new List<Person>();
    }

    public Person Owner;
    public List<Person> Contacts { get; set; }
}
```

Et ajoutons un employé à ce carnet d'adresses :

```

ContactBook contactBook = new ContactBook();
contactBook.Owner = new Employee
{
    Id = 3,
    LastName = "Dugenou",
    FirstName = "Gérard",
    Company = "SuperSoft",
    Position = "CEO",
    Salary = 2147483647
};
contactBook.Contacts.Add(
    new Employee
    {
        Id = 123,
        LastName = "Dupond",
        FirstName = "Jean",
        Company = "SuperSoft",
        Position = "Developer",
        Salary = 40000
    }
);

```

Si on essaie maintenant de sérialiser *contactBook*, on obtient une superbe exception :

InvalidOperationException : Erreur lors de la génération du document XML

Comme c'est un peu vague comme description, on regarde la propriété InnerException :

InvalidOperationException : Le type ArticleXmlSerialization.Employee n'était pas attendu. Utilisez l'attribut XmlInclude ou SoapInclude pour spécifier les types qui ne sont pas connus statiquement.

Voilà qui est plus explicite... Alors, que s'est-il passé exactement ? En fait, le *XmlSerializer* qu'on a créé connaît statiquement le type *Person* (explicitement référencé dans *ContactBook*), mais ne s'attend pas à rencontrer une instance d'un type autre que *Person*, et ne sait pas sérialiser ces objets.

Normalement, c'est là que vous vous dites : "C'est nul la sérialisation XML, ça casse le principe de l'héritage !". Heureusement, il y a bien sûr une solution (et même plusieurs) à ce problème... c'est ce que nous allons voir dans les sections suivantes.

III-B - Sérialisation de classes dérivées

III-B-1 - L'attribut XmlInclude

L'attribut *XmlInclude*, appliqué à une classe, permet de spécifier les types dérivés de cette classe que le *XmlSerializer* peut s'attendre à rencontrer. Appliquons-le à la classe *Person* pour indiquer au *XmlSerializer* l'existence de la classe *Employee* :

```

[XmlInclude(typeof(Employee))]
public class Person : PersonBase
{
    ...
}

```


Avec cet attribut, la sérialisation du *ContactBook* fonctionne, et donne le résultat suivant :

```

<?xml version="1.0" encoding="utf-8"?>
<ContactBook xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <Owner xsi:type="Employee" Id="3">
    <Nom>Dugenou</Nom>
    <Prénom>Gérard</Prénom>
    <Adresses />
    <Company>SuperSoft</Company>
  </Owner>

```

```
<Position>CEO</Position>
<Salary>2147483647</Salary>
</Owner>
<Contacts>
  <Person xsi:type="Employee" Id="123">
    <Nom>Dupond</Nom>
    <Prénom>Jean</Prénom>
    <Adresses />
    <Company>SuperSoft</Company>
    <Position>Developer</Position>
    <Salary>40000</Salary>
  </Person>
</Contacts>
</ContactBook>
```

 **Remarquez l'attribut `xsi:type="Employee"` : il permet d'indiquer le type réel de l'objet.**
Notez aussi que dans la collection `Contacts`, l'`Employee` n'est pas sérialisé dans un élément `<Employee>`, mais dans un élément `<Person>` avec l'attribut `xsi:type`.

III-B-2 - L'attribut `XmlElement`


Bien que dans beaucoup de cas, l'attribut `XmlInclude` permette d'obtenir le résultat voulu, il peut arriver qu'on ne puisse pas intervenir au niveau de la déclaration de la classe de base : par exemple, si la classe `Person` est définie dans un autre assembly sur lequel on n'a pas la main. L'attribut `XmlElement` permet généralement de pallier ce problème. Cet attribut, dont on a déjà parlé plus haut, permet de changer le nom de l'élément XML correspondant à une propriété. Mais il a aussi une autre utilité : il permet de spécifier les différents types que peut avoir la valeur d'une propriété, ainsi que le nom de l'élément XML pour chacun de ces types. Pour l'exemple, supprimons l'attribut `XmlInclude` de la classe `Person`, et ajoutons à la propriété `Owner` 2 attributs `XmlElement` :

```
[XmlElement("Propriétaire_Personne", typeof(Person))]
[XmlElement("Propriétaire_Employé", typeof(Employee))]
public Person Owner { get; set; }
```

Cela signifie que si `Owner` est de type `Person` (respectivement `Employee`), il sera sérialisé sous forme d'un élément `<Propriétaire_Personne>` (respectivement `<Propriétaire_Employé>`). Le XML généré est le suivant :

```
<?xml version="1.0" encoding="utf-8"?>
<ContactBook xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <Propriétaire_Employé Id="3">
    <Nom>Dugenou</Nom>
    <Prénom>Gérard</Prénom>
    <Adresses />
    <Company>SuperSoft</Company>
    <Position>CEO</Position>
    <Salary>2147483647</Salary>
  </Propriétaire_Employé>
  <Contacts>
    <Person Id="123">
      <Nom>Abitbol</Nom>
      <Prénom>Georges</Prénom>
      <Adresses />
    </Person>
  </Contacts>
</ContactBook>
```

Grâce à l'attribut `XmlElement`, on a maintenant un élément XML spécifique selon que `Owner` soit un objet `Person` ou `Employee`, ce qui est quand même plus pratique que l'attribut `xsi:type`.

 **Notez que si l'on avait omis le nom de l'élément dans l'attribut `XmlElement` (en spécifiant seulement le type), le nom du type aurait été utilisé comme nom d'élément.**

III-C - Collections hétérogènes

Dans la section précédente, on a vu qu'on pouvait sérialiser une collection de *Person* contenant des instances de *Employee*, grâce à l'attribut *XmlInclude*. Mais comme on l'a déjà indiqué plus haut, cet attribut ne peut pas toujours être utilisé. Voyons donc deux autres méthodes pour arriver à ce résultat.

III-C-1 - L'attribut XmlArrayItem

On a déjà vu plus haut que l'attribut *XmlArrayItem* permettait de définir le nom de l'élément XML à utiliser pour les éléments d'une collection. Voyons maintenant comment il permet aussi de spécifier le type des éléments de la collection :

```
[XmlArrayItem("Personne", typeof(Person))]
[XmlArrayItem("Employé", typeof(Employee))]
public List<Person> Contacts { get; set; }
```

Ce code indique que la collection *Contacts* peut avoir des éléments de type *Person* ou *Employee*, et qu'ils seront sérialisés sous forme d'éléments XML *<Personne>* ou *<Employé>*. On obtient le XML suivant :

```
<?xml version="1.0" encoding="utf-8"?>
<ContactBook xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/
XMLSchema">
  <Propriétaire_Employé Id="3">
    <Nom>Dugenou</Nom>
    <Prénom>Georges</Prénom>
    <Adresses />
    <Company>SuperSoft</Company>
    <Position>CEO</Position>
    <Salary>2147483647</Salary>
  </Propriétaire_Employé>
  <Contacts>
    <Personne Id="3">
      <Nom>Abitbol</Nom>
      <Prénom>Georges</Prénom>
      <Adresses />
    </Personne>
    <Employé Id="123">
      <Nom>Dupond</Nom>
      <Prénom>Jean</Prénom>
      <Adresses />
      <Company>SuperSoft</Company>
      <Position>Developer</Position>
      <Salary>40000</Salary>
    </Employé>
  </Contacts>
</ContactBook>
```

On a donc maintenant dans *Contacts* des éléments XML différents selon le type du contact.

III-C-2 - L'attribut XmlElement


Eh oui, encore lui ! On a déjà vu qu'il permettait de sérialiser une propriété renvoyant une collection sous forme d'une séquence d'éléments XML, ou encore de spécifier le nom de l'élément XML selon le type de l'objet. Ces deux fonctionnalités peuvent être combinées. Voici un exemple avec la propriété *Contacts* :

```
[XmlElement("Personne", typeof(Person))]
[XmlElement("Employé", typeof(Employee))]
public List<Person> Contacts { get; set; }
```

Remarquez qu'on a quasiment le même code que dans la section précédente, avec *XmlElement* à la place de *XmlArrayItem*. On obtient le XML suivant :

```
<?xml version="1.0" encoding="utf-8"?>
<ContactBook xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/
XMLSchema">
  <Propriétaire Employé Id="3">
    <Nom>Dugenou</Nom>
    <Prénom>Georges</Prénom>
    <Adresses />
    <Company>SuperSoft</Company>
    <Position>CEO</Position>
    <Salary>2147483647</Salary>
  </Propriétaire Employé>
  <Personne Id="3">
    <Nom>Abitbol</Nom>
    <Prénom>Georges</Prénom>
    <Adresses />
  </Personne>
  <Employé Id="123">
    <Nom>Dupond</Nom>
    <Prénom>Jean</Prénom>
    <Adresses />
    <Company>SuperSoft</Company>
    <Position>Developer</Position>
    <Salary>40000</Salary>
  </Employé>
</ContactBook>
```

Les contacts sont donc maintenant au même niveau que le propriétaire, avec un élément XML différent selon leur type.


Attention à rester cohérent lors de l'utilisation de tous ces attributs... par exemple, si on avait spécifié que la propriété Owner devait aussi être sérialisée avec un élément <Personne> ou <Employé>, on n'aurait plus pu distinguer le propriétaire des contacts... Cela aurait d'ailleurs levé une exception lors de la création du XmlSerializer.

III-D - Approches "dynamiques"

Tous les attributs qu'on a vu plus haut permettent, de façon assez simple, de spécifier les types dérivés que le *XmlSerializer* peut rencontrer, ainsi que la manière de les sérialiser. L'inconvénient de cette approche est qu'elle est complètement statique : il faut connaître à l'avance tous les types dérivés des classes qu'on veut sérialiser... Dans certains cas, c'est tout simplement impossible.

Heureusement, il est possible de spécifier, lors de l'appel au constructeur de *XmlSerializer*, les types qu'il peut rencontrer. Il suffit d'utiliser une surcharge de ce constructeur qui prend en paramètre un tableau de types. Ce tableau contient tous les types qui ne sont pas connus statiquement dans la classe à sérialiser. Voici comment faire dans le cas de notre *ContactBook*, qui ne connaît pas statiquement le type *Employee*

```
XmlSerializer xs = new XmlSerializer(typeof(ContactBook), new Type[] { typeof(Employee) });
```

Le résultat obtenu est le même que lorsqu'on a utilisé *XmlInclude* sur la classe *Person* :

```
<?xml version="1.0" encoding="utf-8"?>
<ContactBook xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/
XMLSchema">
  <Personne xsi:type="Employee" Id="3">
    <Nom>Dugenou</Nom>
    <Prénom>Georges</Prénom>
    <Adresses />
    <Company>SuperSoft</Company>
    <Position>CEO</Position>
    <Salary>2147483647</Salary>
  </Personne>
  <Contacts>
    <Personne Id="3">
      <Nom>Abitbol</Nom>
      <Prénom>Georges</Prénom>
```

```
<Adresses />
</Personne>
<Personne xsi:type="Employee" Id="123">
  <Nom>Dupond</Nom>
  <Prénom>Jean</Prénom>
  <Adresses />
  <Company>SuperSoft</Company>
  <Position>Developer</Position>
  <Salary>40000</Salary>
</Personne>
</Contacts>
</ContactBook>
```

Cette technique n'offre pas autant de souplesse que l'utilisation des attributs de contrôle, mais elle permet de s'affranchir des dépendances entre classes de base et classes héritées. Pour plus de contrôle, on peut aussi utiliser la classe *XmlAttributeOverrides*, qui permet de redéfinir les attributs de sérialisation pour chaque membre de chaque classe :

```
XmlAttributeOverrides overrides = new XmlAttributeOverrides();

XmlAttributeAttributes attributesOwner = new XmlAttributeAttributes();
attributesOwner.XmlElements.Add(new XmlElementAttribute("Propriétaire_Personne", typeof(Person)));
attributesOwner.XmlElements.Add(new XmlElementAttribute("Propriétaire_Employé", typeof(Employee)));
overrides.Add(typeof(ContactBook), "Owner", attributesOwner);

XmlAttributeAttributes attributesContacts = new XmlAttributeAttributes();
attributesContacts.XmlArrayItems.Add(new XmlArrayItemAttribute("Personne", typeof(Person)));
attributesContacts.XmlArrayItems.Add(new XmlArrayItemAttribute("Employé", typeof(Employee)));
overrides.Add(typeof(ContactBook), "Contacts", attributesContacts);

XmlSerializer xs = new XmlSerializer(typeof(ContactBook), overrides);
```

Cette technique est un peu laborieuse à mettre en œuvre, mais peut aisément être automatisée par l'utilisation de fichiers de configuration. Cela permet d'avoir un code indépendant des différentes classes dérivées que l'on peut rencontrer dans l'arborescence des objets à sérialiser.

IV - Personnalisation avancée avec l'interface *IXmlSerializable*

Comme on l'a vu dans les chapitres précédents, les attributs de contrôle offrent une certaine souplesse pour maîtriser le schéma du document XML généré. Mais il peut arriver, pour une raison ou une autre, que ça ne suffise pas... Dans ce cas, il reste une option plus radicale : gérer soi-même la façon dont est sérialisée la classe, élément par élément. Pour cela, il faut implémenter l'interface *IXmlSerializable*. Quand le *XmlSerializer* rencontre une classe qui implémente cette interface, il utilise l'implémentation fournie par la classe à la place de l'implémentation par défaut.

Dans ce chapitre, on va voir comment implémenter *IXmlSerializable*, au travers de l'exemple suivant (inspiré d'une discussion sur le forum Developpez.com, l'auteur se reconnaîtra...). Soit une classe *Voiture* :

```
public class Voiture
{
    public string Modele { get; set; }
    public string Constructeur { get; set; }
    public int Cyindree { get; set; }
}
```

Pour une raison quelconque, on ne souhaite pas sérialiser cette classe sous sa forme par défaut, mais sous la forme suivante :

```
<?xml version="1.0" encoding="utf-8"?>
<Voiture xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/
XMLSchema">
  <!--Hello world from IXmlSerializable !-->
  <Modele Constructeur="Volkswagen">Coccinelle</Modele>
  <Cyindree>1584</Cyindree>
```

</Voiture>

On a donc une propriété de la classe (*Constructeur*) qui est sérialisée en tant qu'attribut d'une autre propriété (*Modele*). Les attributs de contrôle qu'on a vu jusqu'ici ne permettent pas d'obtenir ce résultat (à moins que je n'aie mal cherché, dans ce cas j'attends vos suggestions !). On va donc implémenter l'interface *IXmlSerializable* pour obtenir le résultat souhaité.

IV-A - La méthode GetSchema

Commençons par expédier tout de suite cette méthode, qui... ne sert à rien ! La documentation MSDN nous dit à ce propos :

*Cette méthode est réservée et ne doit pas être utilisée. Lorsque vous implémentez l'interface *IXmlSerializable*, vous devez retourner la valeur référence null.*

Inutile donc d'insister plus longuement sur ce point : l'implémentation de cette méthode se résume à l'instruction *return null*.

IV-B - La méthode WriteXml

C'est cette méthode qui est appelée lorsqu'on sérialise la classe en XML. Elle prend en paramètre un *XmlWriter*, qui permet de composer facilement un document XML. Lorsque cette méthode est appelée, l'élément racine de l'objet (<Voiture> dans le cas présent) est déjà créé, il reste donc seulement à sérialiser les propriétés de l'objet.

On va donc créer :

- Un commentaire "Hello world from *IXmlSerializable*"
- Un élément <Modele>, qui contiendra un attribut *Constructeur* et le nom du modèle sous forme de texte
- Un élément <Cylindree> qui contiendra la cylindrée du véhicule

```
public void WriteXml(System.Xml.XmlWriter writer)
{
    // Commentaire XML
    writer.WriteComment("Hello world from IXmlSerializable !");
    // On ouvre l'élément <Modele>, sans le refermer
    writer.WriteStartElement("Modele");
    // Ajoute de l'attribut Constructeur="..."
    writer.WriteAttributeString("Constructeur", this.Constructeur);
    // On écrit le nom du modèle dans l'élément <Modele>
    writer.WriteString(this.Modele);
    // Fermeture de l'élément <Modele>
    writer.WriteEndElement();
    // Ajout de l'élément Cylindrée avec son contenu
    writer.WriteElementString("Cylindree", this.Cylindree.ToString()); // <Cylindree>...</Cylindree>
}
```

Sérialisons maintenant notre objet Voiture de la même façon que d'habitude :

```
Voiture v = new Voiture
{
    Modele = "Coccinelle",
    Constructeur = "Volkswagen",
    Cylindree = 1584
};
XmlSerializer xs = new XmlSerializer(typeof(Voiture));
using (StreamWriter wr = new StreamWriter("voiture.xml"))
{
    xs.Serialize(wr, v);
}
```

Vous pouvez ouvrir le fichier *voiture.xml* et vérifier qu'on obtient bien le résultat indiqué plus haut.

Supposons maintenant qu'on veuille ajouter à *Voiture* des informations sur le propriétaire. Notre bonne vieille classe *Person* va donc reprendre du service :

```
public Person Proprietaire { get; set; }
```

Et là, grand moment de solitude : va-t-on devoir sérialiser manuellement toute la classe *Person* aussi ?! Je vous rassure, la réponse est non... On va en fait utiliser un *XmlSerializer* ad hoc pour sérialiser la propriété *Proprietaire*. Complétons donc notre méthode *WriteXml* :

```
// Proprietaire
XmlAttributeOverrides overrides = new XmlAttributeOverrides();
XmlAttributeOverrides attr = new XmlAttributes();
attr.XmlRoot = new XmlRootAttribute("Proprietaire");
overrides.Add(typeof(Person), attr);
XmlSerializer xs = new XmlSerializer(typeof(Person), overrides);
xs.Serialize(writer, this.Proprietaire);
```

On obtient le XML suivant :

```
<?xml version="1.0" encoding="utf-8"?>
<Voiture xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/
XMLSchema">
  <!--Hello world from IXmlSerializable !-->
  <Modele Constructeur="Volkswagen">Coccinelle</Modele>
  <Cylindree>1584</Cylindree>
  <Proprietaire xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://
www.w3.org/2001/XMLSchema" Id="3">
    <Nom>Abitbol</Nom>
    <Prénom>Georges</Prénom>
    <DateDeNaissance>21/02/1945</DateDeNaissance>
    <Adresses />
  </Proprietaire>
</Voiture>
```

On voit donc qu'il est possible d'utiliser un *XmlSerializer* pour sérialiser les propriétés de type complexe dans l'implémentation de *WriteXml*. Notez cependant l'emploi de *XmlAttributeOverrides* pour forcer le nom de l'élément *<Proprietaire>* : sans cela, la propriété *Proprietaire* aurait été sérialisée dans un élément *<Personne>*. Selon le schéma XML souhaité, on aurait pu utiliser une autre approche, en englobant l'objet *Person* dans un élément *<Proprietaire>* :

```
writer.WriteStartElement("Proprietaire");
XmlSerializer xs = new XmlSerializer(typeof(Person));
xs.Serialize(writer, this.Proprietaire);
writer.WriteEndElement();
```

Et on aurait eu le résultat suivant :

```
<?xml version="1.0" encoding="utf-8"?>
<Voiture>
  <!--Hello world from IXmlSerializable !-->
  <Modele Constructeur="Volkswagen">Coccinelle</Modele>
  <Cylindree>1584</Cylindree>
  <Proprietaire>
    <Personne xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/
XMLSchema" Id="3">
      <Nom>Abitbol</Nom>
      <Prénom>Georges</Prénom>
      <DateDeNaissance>21/02/1945</DateDeNaissance>
      <Adresses />
    </Personne>
  </Proprietaire>
</Voiture>
```

Pour la suite, on retiendra ce dernier schema.

IV-C - La m thode ReadXml

La m thode ReadXml est appel e lors de la d s rialisation d'un objet   partir d'un document XML : c'est la r ciproque de WriteXml. Le param tre fourni en entr e, de type XmlReader, va nous permettre de lire le contenu du XML pour reconstituer l'objet. Voil  comment faire pour notre classe Voiture :

```
public void ReadXml(System.Xml.XmlReader reader)
{
    while (reader.Read())
    {
        if (reader.NodeType == XmlNodeType.Element)
        {
            if (reader.Name == "Modele")
            {
                this.Constructeur = reader.GetAttribute("Constructeur");
                if (reader.Read())
                {
                    if (reader.NodeType == XmlNodeType.Text)
                    {
                        this.Modele = reader.Value;
                    }
                }
            }
            else if (reader.Name == "Cylindree")
            {
                if (reader.Read())
                {
                    if (reader.NodeType == XmlNodeType.Text)
                    {
                        this.Cylindree = int.Parse(reader.Value);
                    }
                }
            }
            else if (reader.Name == "Proprietaire")
            {
                if (reader.Read())
                {
                    if (reader.NodeType == XmlNodeType.Element)
                    {
                        XmlSerializer xs = new XmlSerializer(typeof(Person));
                        this.Proprietaire = xs.Deserialize(reader) as Person;
                    }
                }
            }
        }
    }
}
```

C'est nettement plus long que pour la s rialisation, mais relativement simple quand on y regarde de plus pr s... On boucle sur les  l ments de premier niveau, et on descend l'arborescence si n cessaire pour r cup rer leur contenu. Notez,   nouveau, l'usage de *XmlSerializer* pour d s rialiser la propri t  *Proprietaire*.

V - Utiliser l'outil XML Schema Definition Tool

Le XML Schema Definition Tool (xsd.exe) est un outil fourni avec le .NET framework qui permet de g n rer un sch ma XSD   partir d'un ensemble de classes, ou au contraire de g n rer les classes correspondant   un sch ma XSD.

V-A - G n rer le sch ma   partir des classes

Si on doit  changer des fichiers XML g n r s par la s rialisation XML, il est souvent utile de fournir   ses partenaires le sch ma XSD qui d crit la structure de ces fichiers. L'outil xsd.exe permet de faire cela tr s facilement,   partir de l'assembly contenant la ou les classes   s rialiser. L'outil tient compte de tous les attributs de contr le appliqu s aux classes et membres, et g n re un sch ma auquel les objets s rialis s se conforment.

Pour utiliser cette fonctionnalité, on appelle xsd.exe de la façon suivante en lui passant le nom de l'assembly :

```
> xsd.exe ArticleXmlSerialization.exe
```

Cette commande génère un ou plusieurs fichiers .xsd avec le schéma de tous les types publics de l'assembly. On peut aussi spécifier de générer le schéma seulement pour certains types :

```
> xsd.exe ArticleXmlSerialization.exe /type:ArticleXmlSerialization.Person /  
type:ArticleXmlSerialization.Address
```

Notez que par défaut, xsd.exe ne sait pas générer le schéma XSD d'une classe qui implémente *IXmlSerializable*. Pour que cela fonctionne, il faut appliquer à la classe un attribut *XmlSchemaProvider*, qui indique quelle méthode statique de la classe fournit les informations sur le schéma. Pour plus d'informations à ce sujet, consultez la documentation MSDN de la classe *XmlSchemaProviderAttribute*.

V-B - Générer les classes à partir du schéma

Il arrive souvent qu'on doive manipuler par le code des documents XML conformes à un schéma spécifique. Les traiter "manuellement" avec un *XmlReader* ou un *XmlDocument* est possible, mais peut vite devenir laborieux... Heureusement, l'outil xsd.exe permet aussi de générer automatiquement des classes à partir d'un schéma XSD. Sérialiser des instances de ces classes produira des documents XML conformes au schéma.

Pour générer les classes à partir d'un fichier .xsd, on utilise la commande suivante :

```
> xsd.exe schema.xsd /classes
```



Cette commande produit un fichier source C# nommé schema.cs, contenant la définition des classes correspondant au schéma XML. D'autres options permettent de spécifier le langage du code généré, ou encore le namespace dans lequel les classes sont générées. Notez que les classes générées sont déclarées avec le mot-clé *partial*, ce qui permet de les enrichir en ajoutant des membres dans un autre fichier.

L'outil xsd.exe est également capable d'inférer un schéma XSD à partir d'un document XML. Cela est pratique si on doit traiter des documents XML dont le schéma n'a pas été fourni.

```
> xsd.exe document.xml
```

Cette commande génère un fichier document.xsd, à partir duquel on peut générer les classes correspondantes. Notez cependant que seuls les éléments et attributs présents dans le fichier XML d'origine sont pris en compte dans la génération du schéma, et donc des classes. Si vous utilisez cette méthode, il faut donc choisir un document XML exhaustif, i.e. qui contient tous les éléments et attributs autorisés.

Conclusion

Nous voilà au terme de ce tutoriel qui, je l'espère, vous aura permis de comprendre comment exploiter les nombreuses possibilités de la sérialisation XML. J'ai avant tout cherché à présenter les fonctionnalités les plus susceptibles de servir dans des applications courantes. Si par hasard, vous ne trouvez pas ici les fonctionnalités correspondant à vos besoins, sachez que cet article est loin d'être exhaustif: j'ai fait l'impasse sur certaines fonctionnalités, comme certains attributs de contrôle, les attributs spécifiques à  **SOAP**, ou encore la gestion des namespaces XML. Pour en savoir plus sur les fonctionnalités avancées de la sérialisation XML, je vous invite à explorer la documentation MSDN du namespace  **System.Xml.Serialization**.

Remerciements

Je tiens ici à remercier l'équipe .NET de Developpez.com pour ses relectures attentives et ses suggestions, et en particulier **Skalp** qui s'est impliqué dès le début de la rédaction de cet article, et **StormimOn** pour la relecture orthographique.