

# Introduction

---

Nous avons maintenant une bonne idée de ce que sont les objets. Nous savons les définir, les instancier et les utiliser. Nous savons également les rassembler dans des collections que nous allons avoir très vite envie de trier et d'interroger...

« Recherchez-moi toutes les factures de la société Normandy Software du second trimestre 2018 avec un montant HT supérieur à 1500 €. »

L'outil .NET extraordinaire qui va permettre de coder cette recherche en une ligne C# s'appelle LINQ et nous allons en découvrir les utilisations de base.

Autre objectif important : pour l'instant, dès que nos objets sont détruits, leurs informations disparaissent avec eux... Nous allons donc nous intéresser aux moyens de les "persister" en utilisant des fichiers XML .

## LINQ

---

### 1. Qu'est-ce que c'est ?

---

Tout d'abord, LINQ est l'acronyme de *Language Integrated Query* ; donc rien à voir avec LINK, qui est un outil contribuant à la construction de fichiers exécutables. La vocation première de LINQ est d'uniformiser une interface de questionnement des collections de données comme des listes - que nous avons déjà utilisées -, des bases de données et des fichiers XML. Même si parfois le diable se cache dans les détails, on peut dire qu'utiliser LINQ permet de faire abstraction des gestionnaires de bases de données.

L'autre énorme avantage de LINQ est qu'il est fondu dans la grammaire C# permettant ainsi de détecter toute erreur de syntaxe à la compilation et non à l'exécution... Voyons comment.

Sans LINQ : un programme exploite une base de données SQL à travers les classes .NET de gestion de bases de données. Le développeur devra bien connaître le langage SQL pour construire des requêtes correctes à envoyer sur le serveur. Le serveur les exécutera et retournera des résultats sous forme de chaînes à décoder pour en extraire les données utiles. Le développeur utilisera le C# pour construire ses requêtes SQL dans des objets *string*. Il n'y a pas de vérification possible de la commande SQL à la compilation ; toute erreur sera sanctionnée à l'exécution sur le serveur ! Ensuite, le format de retour devra être connu du développeur pour qu'il puisse en extraire correctement les informations...

Avec LINQ : le langage de requête LINQ faisant partie du C#, il n'y a donc plus ces "objets *string*" intermédiaires à construire d'un côté et à décoder de l'autre. La requête est un enchaînement d'appels à des méthodes C# et, si erreurs il y a, alors elles seront détectées à la compilation, évitant ainsi l'exécution de code erroné sur les serveurs ! Cerise sur le gâteau : comme il ne s'agit que de C#, on bénéficie des services de l'assistant de saisie de code IntelliSense que l'on aime tellement.

Côté développeur - donc de notre côté -, LINQ propose une série de méthodes spécialisées qui vont être traduites en requêtes de bases de données. Pour que cette opération soit possible, il faut que la base de données propose un module compatible LINQ provider qui va s'intercaler entre LINQ et son gestionnaire pour effectuer les conversions. Par défaut, les LINQ providers disponibles dans .NET sont :

- *LINQ to Objects*
- *LINQ to XML*
- *LINQ to SQL*
- *LINQ to Entities*
- *LINQ to Datasets*

## 2. Les deux syntaxes LINQ

### a. La syntaxe "développeur SQL"

Le langage SQL (à prononcer *siquoual* dans les milieux branchés) est, depuis longtemps, LE standard pour interroger les bases de données. Alors quoi de plus naturel que de proposer une syntaxe LINQ "à la SQL" ?

Nous allons partir d'un exemple de requête simple pour découvrir cette syntaxe un peu particulière. Dans cet exemple, nous souhaitons filtrer les nombres pairs d'une collection d'entiers ; voici donc la requête LINQ associée.

```
int[] tab = new int[] { 8, 1, 20, 3, 7, 4, 5, 6 };

IEnumerable<int> filtre = from number in tab
                        where number % 2 == 0
                        select number;
```

Alors déjà les développeurs SQL remarquent que la requête est montée "un peu à l'envers". Voyons vite pourquoi...

La partie *from number in tab* qui définit l'itération sur la source pourrait être comparée à un *foreach(int number in tab)*. Elle est placée en début pour qu'IntelliSense connaisse le type de données à traiter et nous suggère des choix appropriés.

La partie *where number % 2 == 0* est le filtre. On ne garde que les *numbers* qui retournent 0 au modulo 2 (donc les chiffres qui divisés par 2 ne laissent aucun reste et donc... les chiffres pairs). Notez la syntaxe "full C#" du contenu du filtre : une erreur de syntaxe entraîne une erreur de compilation assurée.

La partie *select number* définit ce que LINQ doit extraire des données filtrées. Ici, il n'y a pas de discussion, car la collection contient des entiers qui sont par définition des types Valeurs. LINQ retournera chaque élément "au complet". Par contre, si la collection avait contenu des objets de types Références, alors il aurait été possible de ne sélectionner que certains attributs, voire même de créer à la volée des objets anonymes ne regroupant que les attributs nécessaires à une action donnée... Nous verrons cela dans quelques lignes.

Enfin, le retour de la requête est une collection supportant l'interface *IEnumerable*. La requête est prête à être exécutée et son résultat en mémoire est référencé par la variable *filtre*.

Attention au "piège" ! Avec *LINQ*, une requête n'est réellement exécutée QUE lorsque son résultat est parcouru !

En effet, si l'on met un point d'arrêt juste après notre requête, on s'aperçoit que la référence *filter* est allouée, mais que la liste qu'elle pointe est vide alors qu'il y a quelques chiffres pairs dans notre tableau...

Avantage : le résultat est à jour quand on le parcourt !

Inconvénient : à chaque accès à ce résultat - donc à la référence *filter* dans notre exemple -, la requête est réexécutée. Ce sera bien dans certains cas, mais pénalisant dans d'autres. Si vous savez que le résultat de la requête sera stable pendant votre traitement et que vous aurez besoin d'y accéder plusieurs fois, alors vous pourrez le "figer" par un appel à *ToList* ou *ToArray* pour forcer l'exécution de la requête dès sa définition.

```
IEnumerable<int> paire = (from number in tab
                        where number % 2 == 0
                        select number).ToList();
```

Si maintenant on met un point d'arrêt après cette ligne, on s'aperçoit que *paire* pointe sur une collection contenant quatre entrées.

## b. La syntaxe "développeur C#"

Il se peut que cette première version de syntaxe puisse choquer les développeurs C#. Si pour eux l'intrusion de mots-clés tels que *from*, *where* ou *select* dans une ligne de code est dure à digérer, ils pourront utiliser cette seconde syntaxe, tout aussi équivalente à la première, et qui est :

```
IEnumerable<int> paire = tab.Where(n => n % 2 == 0);
```

On utilise ici la méthode *Where* avec une expression lambda en paramètre. Cette syntaxe lambda a déjà été présentée dans cet ouvrage et peut aussi paraître quelque peu déroutante... mais il faut s'y faire. Comprenez « je prends tous les éléments de la collection *tab* et je ne garde au final que ceux dont le modulo 2 retourne 0 ». Donc tous les éléments de *tab* vont défiler dans *n*. Même constat concernant le moment d'exécution de la requête : elle ne sera exécutée qu'au moment où on lira son résultat. Si vous souhaitez figer le retour une fois pour toutes, alors vous pouvez simplement utiliser une commande de copie dans une collection.

```
IEnumerable<int> paire = tab.Where(n => n % 2 == 0).ToList();
```

## 3. Requêtes et filtres

Les deux exemples précédents ont effectué une requête sur une collection d'objets de type Valeur - des entiers, en l'occurrence. Intéressons-nous maintenant à une collection d'objets de type Références. La requête peut sélectionner des éléments de la collection "au complet" (toutes les propriétés de chaque instance) ou "partiellement" (une sélection adaptée à votre action).

Pour illustrer cette utilisation, nous allons développer une collection de motos. Chaque moto sera représentée par un objet contenant trois propriétés : *Marque*, *Type*, *Puissance*, et une implémentation de la méthode de base *ToString* permettant de synthétiser le contenu.

```

class Moto
{
    public string? Marque { get; set; }
    public string? Type { get; set; }
    public int Puissance { get; set; }
    public override string ToString()
    {
        return $"{Marque} {Type} {Puissance} ch";
    }
}

```

Le contenu de la collection est de type *List* et son contenu est construit "en dur". En réalité, il serait lu depuis une base de données que, pour l'instant, nous ne savons pas utiliser. Alors construisons cette collection avec des valeurs recopiées du site <https://moto-station.com>.

```

var motos = new List<Moto>
{
    new Moto { Marque = "Honda", Type = "CBX 1000", Puissance = 105 },
    new Moto { Marque = "Kawasaki", Type = "Z 1300", Puissance = 120 },
    new Moto { Marque = "Yamaha", Type = "V-Max 1200", Puissance = 145 },
    new Moto { Marque = "Kawasaki", Type = "ZZR 1100", Puissance = 147 },
    new Moto { Marque = "Honda", Type = "CBR1100XX Superbb", Puissance = 164 },
    new Moto { Marque = "Suzuki", Type = "GSX1300R Hayabusa", Puissance = 175 },
    new Moto { Marque = "Kawasaki", Type = "ZX-12R", Puissance = 180 },
    new Moto { Marque = "Suzuki", Type = "GSX-R 1300 Hayabusa", Puissance = 198 },
    new Moto { Marque = "Yamaha", Type = "V-max 1700", Puissance = 200 },
    new Moto { Marque = "Ducati", Type = "Panigale 1299", Puissance = 205 },
    new Moto { Marque = "Ducati", Type = "Panigale V4 S", Puissance = 214 },
    new Moto { Marque = "Kawasaki", Type = "Ninja H2", Puissance = 231 }
};

```

Pour nous remettre en selle par rapport à la section précédente, commençons par récupérer des listes d'objets "complets", comme par exemple tous ceux qui contiennent la marque *Kawasaki*.

```

var kawas = from m in motos where m.Marque == "Kawasaki" select m;
foreach (var kawa in kawas)
{
    Console.WriteLine(kawa);
}

```

Cette requête "syntaxe développeur SQL" se termine par *select m*, qui signifie "récupère l'objet au complet".

L'itération sur les objets filtrés appelle *WriteLine*, et comme *WriteLine* attend en paramètre un objet de type *string*, elle appelle implicitement la méthode *Moto.ToString()*. On arrive donc à l'affichage suivant :

```

Kawasaki Z 1300 120 ch
Kawasaki ZZR 1100 147 ch
Kawasaki ZX-12R 180 ch
Kawasaki Ninja H2 231 ch

```

Maintenant, voyons le cas où seules les propriétés *type* et *puissance* de ces instances nous intéressent. On peut alors demander à LINQ de les stocker dans des objets dits "anonymes".

```
var typesEtPuissances
    = from m in motos where m.Marque == "Kawasaki"
    select new {m.Type, m.Puissance};
foreach (var typeEtPuissance in typesEtPuissances)
{
    Console.WriteLine(
        $"{typeEtPuissance.Type} {typeEtPuissance.Puissance} ch");
}
```

Le `select new {m.Type, m.Puissance}` permet la création d'objets "à la volée" qui ne contiendront que deux propriétés, *Type* et *Puissance*, copiées depuis les instances de type *Moto*. Comme ces objets n'appartiennent pas à une classe clairement définie, leur conteneur ne peut pas être typé. Et c'est ici l'origine du mot-clé *var* !

Ce raccourci très pratique n'a cependant d'intérêt que si l'agrégat *Type Puissance* est utilisé immédiatement. Si l'objet doit être transporté "plus loin", alors il deviendra obligatoire de le typer en lui attribuant une classe connue au moment de la compilation comme celle-ci :

```
class CaracteristiquesMoto
{
    public string? Type { get; set; }
    public int Puissance { get; set; }
    public override string ToString()
    {
        return $"{Type} {Puissance} ch";
    }
}
```

La requête LINQ devient alors :

```
IEnumerable<CaracteristiquesMoto> caracteristiquesMotos
    = from m in motos where m.Marque == "Kawasaki"
    select new CaracteristiquesMoto {Type=m.Type,
    Puissance=m.Puissance};
foreach (var caracteristiquesMoto in caracteristiquesMotos)
{
    Console.WriteLine(caracteristiquesMoto);
}
```

Le retour de la requête étant fortement typé, son résultat devient "transportable" entre les couches d'appels, contrairement au cas précédent. Dans l'exemple, les propriétés de la classe *CaracteristiquesMoto* ont les mêmes noms que ceux de la classe *Moto*. C'est recommandé, mais pas une obligation.

Au fait, comment aurions-nous écrit cette requête en "mode classe anonyme" façon développeur C# ?

```
var tps = motos.Where(m => m.Marque == "Kawasaki")
                .Select(m1 => new{ m1.Type, m1.Puissance});
```

Et de façon typée *CaracteristiquesMoto* ?

```
var _tps = motos.Where(m => m.Marque == "Kawasaki")
                .Select(m1 => new CaracteristiquesMoto
                        { Type=m1.Type,
                          Puissance=m1.Puissance
                        });
```

## 4. Quelques calculs

Le requête LINQ ne se limite pas au filtrage de collections. Il est possible de faire tout ce que nous permet une base de données, comme par exemple dénombrer des enregistrements. Si nous souhaitons connaître combien de motos de marque *Honda* figurent dans cette collection, nous pouvons utiliser la syntaxe C# :

```
var nombreDeHonda = motos.Where(m => m.Marque == "Honda").Count();
Console.WriteLine($"Il y a {nombreDeHonda} motos de marque HONDA
dans la liste");
```

Cette syntaxe est logique : on prend la collection, on sélectionne les motos de marque *Honda* puis on les dénombre. Elle peut cependant être simplifiée, car la méthode LINQ *Count* peut prendre en argument une expression lambda qui va nous servir de filtre. On arrive donc à cette commande :

```
var _nombreDeHonda = motos.Count(m => m.Marque == "Honda");
```

Il est possible de demander également des moyennes comme : « Quelle est la puissance moyenne des machines de marque *Kawasaki* ? »

```
var puissanceMoyenneDesKawas
    = motos.Where(m => m.Marque == "Kawasaki")
            .Select(m_ => m_.Puissance)
            .Average();
```

Il est possible de récupérer les valeurs max ou min : « Quelle est la puissance maximale des machines de marque *Suzuki* de cette collection ? »

```
var puissanceMaxiDesSuzuki
    = motos.Where(m => m.Marque == "Suzuki")
            .Select(m => m.Puissance)
            .Max();
Console.WriteLine($"Le moteur Suzuki le plus puissant fait
{puissanceMaxiDesSuzuki} ch");
```

Si on souhaite maintenant afficher le modèle correspondant, on peut demander un classement décroissant des puissances puis retenir le premier de la liste.

```
var modeleLePlusPuissantDesSuzuki
    = motos.Where(m => m.Marque == "Suzuki")
            .OrderByDescending(m => m.Puissance)
            .Select(m => m.Type)
            .First();

Console.WriteLine($"Le modèle Suzuki le plus puissant est
{modeleLePlusPuissantDesSuzuki}");
```

## 5. Regroupement des résultats

Des regroupements peuvent être mis en œuvre avec l'opérateur *GroupBy*. Cet opérateur retourne une collection de type *IEnumerable* d'objets de type *IGrouping*.

*IGrouping* est une interface qui elle-même étend *IEnumerable* en lui ajoutant une propriété *Key*. C'est cette propriété qui définit la clé de regroupement que l'on précise en paramètre de l'opérateur *GroupBy*.

```
namespace System.Linq
{
    ...public interface IGrouping<out TKey, out TElement> : IEnumerable<TElement>, IEnumerable
    {
        ...TKey Key...
    }
}
```

Dans notre exemple, nous allons demander un regroupement par marques de motos.

```
IEnumerable<IGrouping<string?, Moto>> parMarques
    = motos.GroupBy(m => m.Marque);
```

```
IEnumerable<IGrouping<string?, Moto>> parMarques
= motos.GroupBy(m => m.Marque);
```

if (parMarques != null)

```
{
    foreach (var parMarque in parMarques)
    {
        Console.WriteLine(parMarque.Key);
        foreach (var moto in parMarque)
        {
            Console.WriteLine($"{moto.Marque} {moto.Type}");
        }
    }
}
```

Membres non publics

Affichage des résultats

Le développement de l'affichage des résultats

[0] Key = "Honda"

[1] Key = "Kawasaki"

[2] Key = "Yamaha"

[3] Key = "Suzuki"

[4] Key = "Ducati"

Key

[0]

[1]

Affichage brut

Vue "Ducati"

{Ducati Panigale 1299 205 ch}

{Ducati Panigale V4 S 214 ch}

Le type retour de *parMarques* a été déclaré complètement pour bien comprendre le format récupéré. Il pourra évidemment être remplacé par le mot-clé *var* pour plus de simplification.

Le critère de regroupement est défini dans une expression lambda désignant la propriété *Marque*. C'est le contenu de cette propriété qui sera copié dans *Key* de l'objet *IGrouping*. Comme notre exemple contient cinq marques, il y aura cinq objets *IGrouping* retournés.

*GroupBy* propose une surcharge définissant la composition des entrées de la liste contenue dans chaque *IGrouping*.

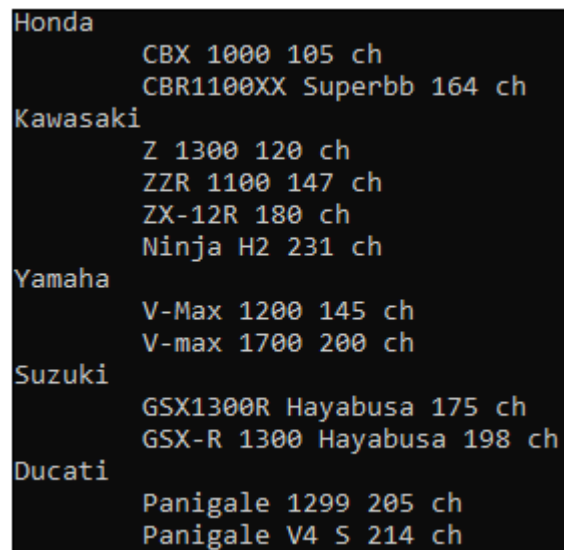
```
var parMarques
    = motos.GroupBy(m => m.Marque, m => new {m.Type, m.Puissance})
```

Ici, c'est une classe anonyme qui est construite à la volée contenant les propriétés *Type* et *Puissance*.

Si aucun format n'est déclaré, ce sont les objets *Moto* au complet qui sont retournés.

Le résultat du regroupement peut être parcouru par deux boucles imbriquées. La première concerne la clé de regroupement - donc la marque - puis la seconde concerne le détail du regroupement.

```
foreach (var parMarque in parMarques)
{
    Console.WriteLine(parMarque.Key);
    foreach (var moto in parMarque)
    {
        Console.WriteLine($"{moto.Type} {moto.Puissance} ch");
    }
}
```



```
Honda
    CBX 1000 105 ch
    CBR1100XX Superbb 164 ch
Kawasaki
    Z 1300 120 ch
    ZZR 1100 147 ch
    ZX-12R 180 ch
    Ninja H2 231 ch
Yamaha
    V-Max 1200 145 ch
    V-max 1700 200 ch
Suzuki
    GSX1300R Hayabusa 175 ch
    GSX-R 1300 Hayabusa 198 ch
Ducati
    Panigale 1299 205 ch
    Panigale V4 S 214 ch
```

## 6. Les jointures

Autres fonctions très utilisées dans le monde des bases de données relationnelles : les jointures ou l'art de relier plusieurs collections ensemble en utilisant des champs communs. LINQ n'est pas en reste sur ce point en offrant la fonctionnalité en syntaxe C#. Nous allons présenter ici la jointure "simple", mais il existe d'autres types de jointures plus complexes dont la présentation déborderait du cadre de cet ouvrage.

Créons une collection de motards.

```
class Motard
{
    public string Nom { get; set; }
```



```

        //(...)
    }

    var motards = new List<Motard>();
    var ludo = new Motard { Nom = "Ludo" };
    motards.Add(ludo);
    var celine = new Motard { Nom = "Céline" };
    motards.Add(celine);
    var julien = new Motard { Nom = "Julien" };
    motards.Add(julien);
    var isabelle = new Motard { Nom = "Isabelle" };
    motards.Add(isabelle);

```

Chaque motard pouvant posséder une ou plusieurs motos et chaque moto ne pouvant avoir qu'un propriétaire, on ajoute à notre classe *Moto* une référence de type *Motard*.

```

class Moto
{
    //(...)
    public Motard? Proprietaire { get; set; }
}

```

La liste des motos est regénérée avec des propriétaires pour certaines d'entre elles.

```

motos = new List<Moto>
{
    new Moto { Marque = "Honda", Type = "CBX 1000", Puissance = 105,
    Proprietaire=ludo},
    new Moto { Marque = "Kawasaki", Type = "Z 1300", Puissance = 120,
    Proprietaire=celine},
    new Moto { Marque = "Yamaha", Type = "V-Max 1200", Puissance = 145 },
    new Moto { Marque = "Kawasaki", Type = "ZZR 1100", Puissance = 147 },
    new Moto { Marque = "Honda", Type = "CBR1100XX Superbb",
    Puissance = 164, Proprietaire=julien },
    new Moto { Marque = "Suzuki", Type = "GSX1300R Hayabusa",
    Puissance = 175, Proprietaire=ludo },
    new Moto { Marque = "Kawasaki", Type = "ZX-12R", Puissance = 180 },
    new Moto { Marque = "Suzuki", Type = "GSX-R 1300 Hayabusa",
    Puissance = 198 },
    new Moto { Marque = "Yamaha", Type = "V-max 1700", Puissance = 200 },
    new Moto { Marque = "Ducati", Type = "Panigale 1299", Puissance = 205,
    Proprietaire= isabelle},
    new Moto { Marque = "Ducati", Type = "Panigale V4 S", Puissance = 214,
    Proprietaire= isabelle},
    new Moto { Marque = "Kawasaki", Type = "Ninja H2", Puissance = 231 }
};

```

On va maintenant utiliser une jointure "interne" pour récupérer la liste des motards avec le type de leurs montures à l'aide des mots-clés *join* et *equals*.

```
var query = from motard in motards join moto in motos
             on motard equals moto.Proprietaire
             select new { NomMotard = motard.Nom, TypeMoto = moto.Type };
```

La première partie définit les collections intervenant dans la jointure : *motards* et *motos*. La seconde partie définit la base de cette jointure, à savoir quand l'instance *motard* se retrouvera dans la propriété *Proprietaire* des instances *Moto*. La dernière partie concerne la création d'un objet anonyme reprenant des propriétés des deux côtés.

L'affichage utilise les noms des propriétés des instances anonymes.

```
foreach (var r in query)
{
    Console.WriteLine($"{r.NomMotard} possède un {r.TypeMoto}");
}
```

```
Ludo possède un CBX 1000
Ludo possède un GSX1300R Hayabusa
Céline possède un Z 1300
Julien possède un CBR1100XX Superbb
Isabelle possède un Panigale 1299
Isabelle possède un Panigale V4 S
```

## Persistance des données en XML

Nous savons maintenant définir des classes, les instancier, les inscrire dans des collections que l'on peut interroger. Mais actuellement, dès que le programme s'arrête, toutes les données sont perdues ! On va donc s'intéresser aux moyens de les "persister", c'est-à-dire les écrire sur un support physique comme disque dur, clé mémoire USB, carte SD ou encore sur le cloud dans lequel on loue des espaces de stockage (on ne parle plus des disquettes, qui ne servent qu'à illustrer les boutons de sauvegarde de nos applications...).

La plupart du temps, la persistance d'un objet revient à sauvegarder le contenu de ses propriétés. Quand notre programme relira sa sauvegarde, il commencera par instancier un objet du même type, "vierge" de données, mais avec des méthodes prêtes à fonctionner. Ensuite, il remplira les propriétés lues depuis le fichier. Le code des méthodes de l'objet persisté se trouve donc dans le programme C# compilé et les données dans un fichier à part. Il existe des cas d'utilisation où description des propriétés et leur contenu se trouve dans un même fichier de données, mais nous nous intéressons au cas plus simple de persistance de données uniquement.

Proposer une fonction de sauvegarde des données peut amener à modifier notre modèle quand il contient des collections d'objets faisant référence à d'autres objets. En effet, il est tout à fait courant d'utiliser un objet contenant des références à d'autres objets et d'en faire des collections, comme par exemple une liste d'objets *facture*. Chaque facture contient une référence à une instance client et une collection de références à des objets à facturer. Cette organisation est très pratique à manipuler, car on peut passer de l'un à l'autre aisément, mais elle ne vaut que pour la session courante. Ces "références" ne sont que des adresses mémoire utilisées par notre programme en cours d'exécution. Si le programme est déchargé puis rechargé, ces emplacements mémoire alloués par le .NET seront différents et ça n'a donc aucun sens de les sauvegarder dans le fichier de données. Donc, dans ce cas,

il faut ajouter dans chaque objet des identifiants uniques qui vont servir de clés de jointure entre les listes. La facture ayant l'identifiant 190309 est attribuée au client ayant l'identifiant 640120 et elle contient deux produits portant l'identifiant X2001. Ce sera certainement moins direct que d'utiliser les références, mais ces identifiants seront toujours les mêmes quelles que soient les sessions d'utilisation et ils seront écrits dans le fichier de données. Les jointures pourront se faire à la volée avec des requêtes LINQ.

Les formats de stockage des informations dans les fichiers de sauvegarde peuvent être multiples. On peut opter pour un format binaire si l'on recherche des tailles de stockage réduites et une programmation simple, car les propriétés des objets sont écrites et lues sans conversion. Par contre, l'exploitation des contenus est plus difficile pour d'autres systèmes et les fichiers seront certainement bloqués par les protections des réseaux en cas d'échange. Un format texte ne posera pas ce dernier problème, car il aura le mérite d'être exploitable par d'autres outils (à commencer par un éditeur de texte classique). Structurer le contenu du fichier texte en utilisant une grammaire descriptive normalisée est un véritable atout, et comme le format XML est très largement utilisé, nous allons nous y intéresser...

## 1. Rappels sur le XML

XML, qui est l'acronyme d'*eXtensible Markup Language*, est une recommandation du *Word Wide Web Consortium* (W3C) décrivant une syntaxe d'écriture de données. Un contenu XML est simple à lire pour un humain et facile à décoder pour les machines, car la plupart des langages informatiques proposent des fonctions adaptées. L'organisation repose sur la définition d'éléments (encore appelés balises ou *markup*) pouvant contenir des attributs et éventuellement des valeurs ou d'autres éléments. La définition d'un élément se fait entre les caractères < et > comme ceci :

```
<Moto Marque="BMW" Model="R1200R" />
```

Ici, l'élément *Moto* contient deux attributs. Les valeurs des attributs sont écrites entre "". Dans ce premier exemple, l'élément *Moto* ne possède pas de valeurs, car la ligne se termine par />. Voici maintenant la syntaxe à adopter pour pouvoir définir des valeurs :

```
<Moto Marque="BMW" Model="R1200R">
    (...)
</Moto>
```

C'est entre le duo < et > que figurera la valeur qui sera soit une chaîne texte, soit une liste d'éléments comme celle-ci :

```
<Moto Marque="BMW" Model="R1200R">
    <Couleurs>
        <Couleur>Grise</Couleur>
        <Couleur>Bleu</Couleur>
        <Couleur>Rouge</Couleur>
    </Couleurs>
</Moto>
```

La racine d'un XML ne peut contenir qu'un seul élément. Dans l'exemple suivant, c'est *Catalog* qui contient une collection *Motos* contenant des éléments *Moto*. Chaque *Moto* contient une collection *Couleurs* contenant des éléments *Couleur*. Notons qu'en dehors d'une balise, la valeur n'est pas encadrée par des "".

```
<Catalog Reference="1234">
  <Motos>
    <Moto Marque="BMW" Modèle="R1200R">
      <Couleurs>
        <Couleur>Grise</Couleur>
        <Couleur>Bleu</Couleur>
        <Couleur>Rouge</Couleur>
      </Couleurs>
    </Moto>
    <Moto Marque="Ducati" Modèle="Monster">
      <Couleurs>
        <Couleur>Rouge</Couleur>
      </Couleurs>
    </Moto>
    <Moto Marque="Triumph" Modèle="Bonneville">
      <Couleurs>
        <Couleur>Noir</Couleur>
        <Couleur>Vert</Couleur>
      </Couleurs>
    </Moto>
  </Motos>
</Catalog>
```

Ce format pratique et extensible a fait des émules notamment sur le Web avec le HTML où les balises ont été normalisées pour définir présentations et contenus.

Ainsi, tous les serveurs du monde entier peuvent envoyer des pages web à tous les navigateurs du monde entier pour que tous les utilisateurs du monde entier puissent voir à peu près la même chose...

En C#, nous devons définir nos balises, qui correspondent très souvent aux propriétés de nos objets persistés, et c'est ce que nous allons voir dans les lignes suivantes.

## 2. XML et .NET

### a. Sérialisation/désérialisation d'un modèle de données

Le format XML étant justifié, nous allons nous intéresser à la transformation entre un ensemble d'instances d'objets vers un fichier XML et inversement. On parle ici respectivement de sérialisation et de désérialisation du modèle de données (organisation des classes de l'application). Ce gros travail peut être pris en charge par le .NET. Le développeur doit simplement préciser quelles sont les classes à sérialiser dans son modèle de données avec, éventuellement, quelques informations complémentaires. Ces déclarations se font au moyen de décorations - concept déjà présenté au chapitre Les tests - qui viendront s'insérer dans les définitions de nos classes.

## b. Les décorations de sérialisation XML

Le minimum permettant de déclarer une classe comme étant sérialisable est :

- D'insérer au début du fichier source de la classe *using System.Xml.Serialization*, car c'est l'espace de noms contenant tous les composants nécessaires.
- De préfixer la définition de la classe par la décoration *[Serializable]*.

```
[Serializable]
public class Moto
{
    public string Marque { get; set; }
    public string Modele { get; set; }

    public List<string> Couleurs { get; set; }
}
```

En instanciant un objet *Moto* et en utilisant le mécanisme de sérialisation que nous allons aborder dans quelques lignes, on obtient le contenu XML suivant :

```
<Moto>
  <Marque>BMW</Marque>
  <Modele>R1200R</Modele>
  <Couleurs>
    <string>Grise</string>
    <string>Bleu</string>
    <string>Rouge</string>
  </Couleurs>
</Moto>
```

Sans autre information, les noms des propriétés C# sont repris en éléments XML.

On peut modifier ce comportement pour que certaines propriétés soient traduites en attributs XML comme dans l'exemple suivant où *Marque* et *Modele* deviennent attributs de l'élément *Moto*.

```
<Moto Marque="BMW" Modele="R1200R">
  <Couleurs>
    <string>Grise</string>
    <string>Bleu</string>
    <string>Rouge</string>
  </Couleurs>
</Moto>
```

Pour cela, on va simplement faire précéder les propriétés concernées par *[XmlAttribute]*.

```
[Serializable]
public class Moto
{
    [XmlAttribute]
    public string Marque { get; set; }
    [XmlAttribute]
    public string Modele { get; set; }

    public List<string> Couleurs { get; set; }
}
```

Dans notre exemple, supposons que la traduction de la propriété *Couleurs* en sous-élément de *Moto* convienne, mais que les balises *string* encadrant chacune des entrées soient dérangeantes. On souhaiterait remplacer *string* par *Couleur*. Pour cela, on va simplement préfixer la propriété par un *[XmlArrayItem("Couleur")]*.

```
[Serializable]
public class Moto
{
    [XmlAttribute]
    public string Marque { get; set; }
    [XmlAttribute]
    public string Modele { get; set; }

    [XmlArrayItem("Couleur")]
    public List<string> Couleurs { get; set; }
}
```

On obtient le résultat suivant :

```
<Moto Marque="BMW" Modele="R1200R">
  <Couleurs>
    <Couleur>Grise</Couleur>
    <Couleur>Bleu</Couleur>
    <Couleur>Rouge</Couleur>
  </Couleurs>
</Moto>
```

Dans le même esprit, il est possible de modifier les noms des objets et propriétés C# lors des traductions en éléments et attributs. Par exemple, si l'on souhaite que le fichier XML contienne des éléments et des attributs en anglais tout en conservant les classes et propriétés du modèle de données en français, on va utiliser les décorateurs :

- *XmlAttribute* avec nom de l'attribut XML en paramètre,
- *XmlType* pour redéfinir la balise *Moto*,
- *XmlArray* pour redéfinir la balise contenant la liste des couleurs.

```

[Serializable]
[XmlType("Motorbike")]
public class Moto
{
    [XmlAttribute("Brand")]
    public string Marque { get; set; }
    [XmlAttribute("Model")]
    public string Modele { get; set; }

    [XmlArray("Colors")]
    [XmlArrayItem("Color")]
    public List<string> Couleurs { get; set; }
}

```

Nous obtenons alors ce nouveau format après sérialisation :

```

<Motorbike Brand="BMW" Model="R1200R">
  <Colors>
    <Color>Grise</Color>
    <Color>Bleu</Color>
    <Color>Rouge</Color>
  </Colors>
</Motorbike>

```

On complète le modèle de données en ajoutant une classe *Catalogue* contenant une propriété *Reference* et une collection d'objets *Moto*, le tout avec des balises traduites en anglais.

```

[Serializable]
[XmlType("Catalog")]
public class Catalogue
{
    [XmlAttribute]
    public string Reference { get; set; }

    [XmlArray("Motorbikes")]
    public List<Moto> Motos { get; set; }
}

```

## c. XmlSerializer : écrire et lire

Notre modèle de données ayant les décorations adéquates, nous l'alimentons en données en utilisant la syntaxe "initialiseurs d'objets" déjà abordée afin de présenter sa sérialisation.

```

var monCatalogue = new Catalogue
{
    Reference = "C4567",
    Motos = new List<Moto> {
        new Moto
        {

```

```

        Marque = "BMW",
        Modele = "R1200R",
        Couleurs = new List<string> { "Grise", "Bleu", "Rouge" }
    },
    new Moto
    {
        Marque = "Ducati",
        Modele = "Monster",
        Couleurs = new List<string> { "Rouge" }
    },
    new Moto
    {
        Marque = "Triumph",
        Modele = "Bonneville",
        Couleurs = new List<string> { "Noir", "Vert" }
    },
}
};

```

L'objet *monCatalogue* est maintenant prêt à être sérialisé. Pour cela, nous allons faire appel à la classe *XmlSerializer*.

Dans notre exemple, les informations XML vont être persistées dans un fichier. Elles pourraient cependant être envoyées sur un socket réseau. *XmlSerializer* respecte la philosophie objet en faisant abstraction du support de sérialisation. Il le considère comme étant une instance de type de la classe abstraite *System.IO.Stream*. Nous devons donc préparer un flux "spécialisé fichier" qui est un objet *System.IO.FileStream* héritant de *System.IO.Stream* pour recevoir la sérialisation de *monCatalogue*.

```

FileStream streamEcriture
    = new FileStream("Test.xml", FileMode.OpenOrCreate,
                    FileAccess.Write,
                    FileShare.None);

```

Tout est maintenant prêt pour utiliser la classe *XmlSerializer*, qu'il faut commencer par instancier.

```

XmlSerializer monXmlSerializer
    = new XmlSerializer(typeof(Catalogue));

```

Notons que le type du modèle de données est passé au constructeur de *XmlSerializer*. Ici, c'est la syntaxe *typeof(Catalogue)* qui est utilisée pour transmettre l'information, mais cela aurait pu être *monCatalogue.GetType()*.

C'est la méthode *Serialize* qui doit être utilisée pour effectuer l'opération d'écriture. Elle prend en paramètres le flux destination et la référence sur l'objet à sérialiser. Une fois l'opération terminée, il faut penser à refermer le flux fichier. Voici le code d'écriture complet :



```

FileStream streamEcriture
    = new FileStream("Test.xml", FileMode.OpenOrCreate,
                    FileAccess.Write,
                    FileShare.None);

XmlSerializer monXmlSerializer
    = new XmlSerializer(typeof(Catalogue));

monXmlSerializer.Serialize(streamEcriture, monCatalogue);

streamEcriture.Close();

```

Son exécution génère le fichier *Test.xml* ayant le contenu suivant :

```

<?xml version="1.0"?>
<Catalog xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema" Reference="C4567">
  <Motorbikes>
    <Motorbike Brand="BMW" Model="R1200R">
      <Colors>
        <Color>Grise</Color>
        <Color>Bleu</Color>
        <Color>Rouge</Color>
      </Colors>
    </Motorbike>
    <Motorbike Brand="Ducati" Model="Monster">
      <Colors>
        <Color>Rouge</Color>
      </Colors>
    </Motorbike>
    <Motorbike Brand="Triumph" Model="Bonneville">
      <Colors>
        <Color>Noir</Color>
        <Color>Vert</Color>
      </Colors>
    </Motorbike>
  </Motorbikes>
</Catalog>

```

Cet affichage du contenu de notre fichier correctement « indenté » est obtenu dans un navigateur internet. Ce même fichier chargé dans Visual Studio aura tout son contenu sur la même ligne. Il est heureusement possible d'agir sur l'indentation avec l'aide d'un objet *XmlWriter* et de son collègue *XmlWriterSettings* comme le montre le code suivant :

```
// Ecriture avec contrôle d'indentation
var xmlWriterSettings
    = new XmlWriterSettings() { Indent = true };
XmlSerializer serializer
    = new XmlSerializer(typeof(Catalogue));
using (XmlWriter xmlWriter
    = XmlWriter.Create("test.xml", xmlWriterSettings))
{
    serializer.Serialize(xmlWriter, monCatalogue);
}
```

Contrairement à ce que son nom pourrait laisser penser, la classe *XmlSerializer* effectue également l'opération de lecture et de décodage du fichier XML. En effet, elle propose une méthode *Deserialize* prenant en paramètre un flux que l'on spécialisera en type fichier dans notre exemple. Le retour de cette méthode est de type *object*, car elle doit être générique à tous les modèles possibles. Il faut donc penser à le "caster" sur le type du modèle de données pour satisfaire le compilateur.

```
FileStream streamLecture
    = new FileStream("Test.xml", FileMode.Open,
                    FileAccess.Read,
                    FileShare.None);

XmlSerializer monXmlSerializerDeLecture
    = new XmlSerializer(typeof(Catalogue));

Catalogue? catalogueLu
    = (Catalogue?)monXmlSerializerDeLecture.Deserialize(streamLecture);

streamLecture.Close();
```

Le code source de ce petit exemple se trouve dans le répertoire *Chap13\XML\demoXML* du fichier ZIP relatif à cet ouvrage.

### 3. XSD.EXE, un outil de conversion

Jusqu'à présent, les utilisations se sont toujours faites en commençant par écrire le code des classes (*code first*) qui ensuite génèrent les fichiers XML. En tant que développeur, il est fréquent d'être confronté à l'inverse, c'est-à-dire écrire le code C# à partir de la structure d'un fichier XML existant (*database first*). Dans le domaine industriel par exemple, des normes existent pour décrire les fonctionnalités de certains équipements. Ces normes complexes s'appuient parfois sur des modèles de données écrits en XML. Le travail qui consisterait à transcrire manuellement en classes ces schémas contenant des milliers de lignes serait colossal. Heureusement, Visual Studio fournit un outil extraordinaire nommé XSD.EXE qui économise bien du temps et de l'énergie dans ce cas...

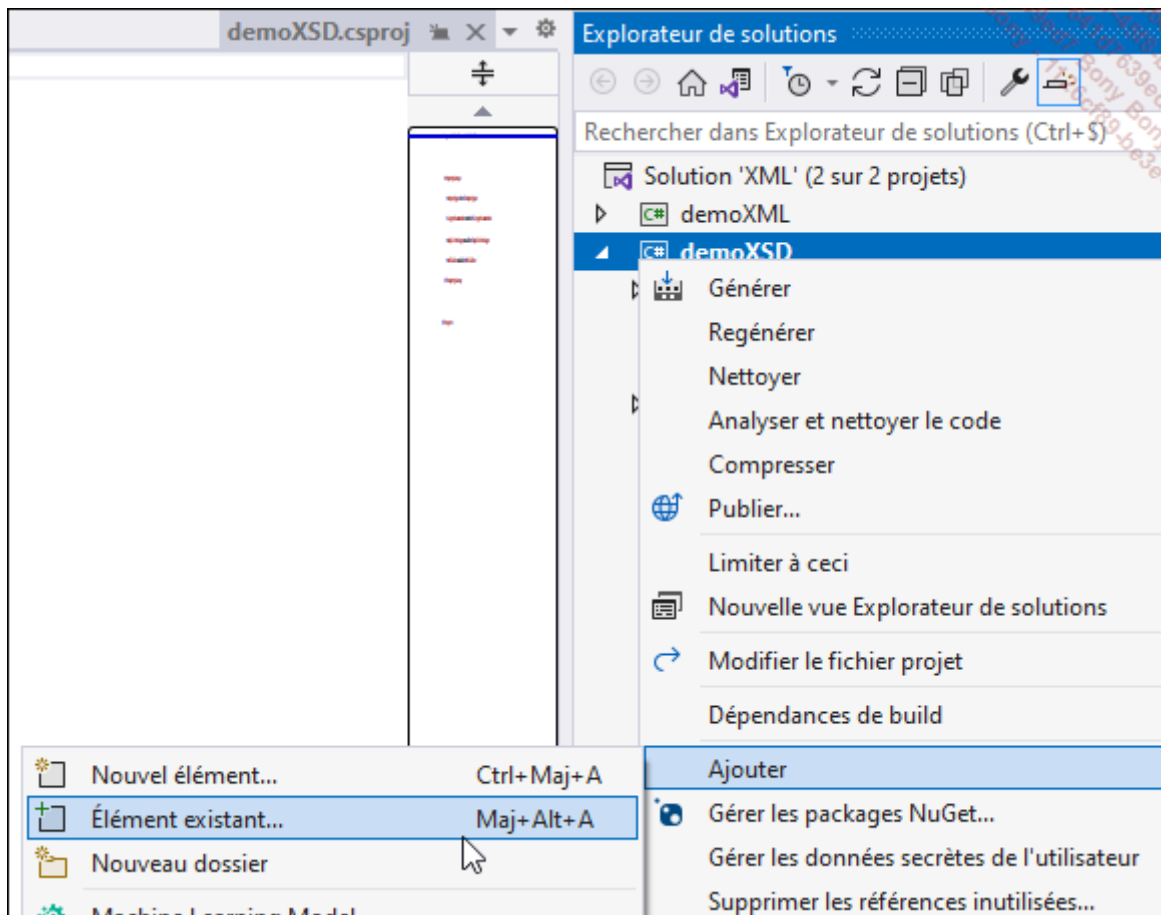
Cet outil s'utilise depuis une console Visual Studio dont le raccourci se trouve dans le dossier **Visual Studio 2022** du menu **Démarrer** de Windows. L'outil va tout d'abord être utilisé pour extraire du fichier XML existant un "schéma" d'organisation des informations. Ce schéma va être écrit dans un fichier temporaire d'extension XSD en utilisant une grammaire XML.

```
C:\demo_XSD>xsd test3.xml
Microsoft (R) Xml Schemas/DataTypes support utility
[Microsoft (R) .NET Framework, Version 4.6.1055.0]
Copyright (C) Microsoft Corporation. All rights reserved.
Écriture du fichier 'C:\demo_XSD\test3.xsd'.
```

C'est ce fichier temporaire qui va ensuite être utilisé par l'outil pour écrire les classes C#.

```
C:\demo_XSD>xsd test3.xsd /c
Microsoft (R) Xml Schemas/DataTypes support utility
[Microsoft (R) .NET Framework, Version 4.6.1055.0]
Copyright (C) Microsoft Corporation. All rights reserved.
Écriture du fichier 'C:\demo_XSD\test3.cs'.
```

Une fois le fichier source généré, il suffit de l'importer dans un projet Visual Studio en faisant un clic droit sur le projet puis en choisissant l'option **Ajouter Élément existant...**



Ensuite, on dispose des mêmes outils que précédemment pour désérialiser le fichier XML. L'objet instancié propose un modèle de données qui s'apparente à l'organisation du XML.

```
var fs = new FileStream("test3.xml", FileMode.Open);
var p = (project)new XmlSerializer(typeof(project)).Deserialize(fs);

foreach (var c in p.component)
{
    //...
}
```

## 5. LINQ to XML

LINQ to XML permet de charger des blocs XML en mémoire, d'y effectuer des requêtes de toutes sortes et même de modifier et de sauvegarder un nouveau contenu. LINQ to XML n'est pas l'unique moyen d'exploiter du XML en .NET, mais c'est certainement le plus simple et le plus moderne ! Une réserve tout de même : le contenu XML doit être entièrement chargé en mémoire avant de pouvoir être travaillé. Et donc, en cas de traitements de flux XML "continus", il faudra préférer les services des classes *System.Xml.XmlReader* et *System.Xml.XmlWriter*. Il existe également la classe *System.Xml.XmlDocument* qui propose une interface DOM (*Document Object Model*) d'accès et de modification de contenus XML entièrement chargés en mémoire. Mais intéressons-nous à LINQ to XML en gardant à l'esprit que ces paragraphes ne restent qu'une présentation de ce module.

### a. Lecture

Lors de l'utilisation de LINQ to XML, la lecture des propriétés ne se fait plus de façon automatique comme avec *XmlSerializer*. Il faut connaître parfaitement l'organisation des données XML pour lire ce qui importe dans les traitements à venir. Les données lues pourront être stockées dans des classes anonymes pour des utilisations immédiates ou des classes déclarées pour des utilisations transversales. Voyons comment LINQ to XML va nous permettre d'exploiter les données de notre fichier d'exemple. Tout d'abord, la classe *XDocument* doit être instanciée, car c'est l'élément central qui va contenir le bloc XML et permettre son exploitation. Elle propose un jeu très complet de méthodes, et même s'il est possible de découvrir dynamiquement le schéma d'un fichier XML, il est plus facile de les utiliser en le connaissant. Cela tombe bien : nous connaissons l'organisation de notre catalogue de motos.

Commençons par le charger en mémoire en utilisant la méthode de type *static* de la classe *XDocument* :

```
var x = XDocument.Load("Test.xml");
```

La racine du fichier XML est un élément nommé *Catalogue* contenant tout d'abord un attribut *Reference*. Pour lire la valeur de cette *Reference*, on va demander à l'instance *XDocument* un accès au nœud *Catalogue*. Cet accès se fait par la méthode *Element* qui prend en paramètre le nom à sélectionner. Attention, il faut correctement orthographier ce nom, sinon la méthode lèvera une exception en cas d'erreur d'identifiant.

```
var referenceLue = x.Element("Catalogue").Attribute("Reference").value;
```

Ce code minimaliste pourrait facilement être amélioré pour résister à d'éventuelles erreurs de contenu du fichier XML.

```
var referenceLue =  
x.Element("Catalogue")?.Attribute("Reference")?.value;
```

En effet, le fait de rajouter un `?` en sortie d'une méthode permet d'interrompre l'exécution de la séquence - donc jusqu'à son `;` - si le retour de cette méthode vaut *null*. Donc si une erreur se glisse lors de l'écriture du XML, il n'y aura pas d'exception levée. Par contre, la variable *referenceLue* vaudra *null*...

Ensuite, le nœud *Catalogue* contient un élément *Motos* qui lui-même contient une collection d'éléments *Moto*. C'est cette collection qu'il faut charger en passant par le nœud *Motos*. *Motos* est un élément de *Catalogue* qui est un élément de l'instance *XDocument*. On va retranscrire cette affiliation en commande LINQ to XML.

```
var motos = x.Element("Catalogue")?.Element("Motos");
```

La variable *motos* est de type *XElement* et la suite du fichier XML est une collection d'éléments que l'on va pouvoir parcourir à l'aide de la méthode *Elements()* (*Elements* avec un *s*) de l'objet *motos*.

```
foreach (var moto in motos?.Elements("Moto")) {...}
```

À nouveau, c'est un objet *XElement* qui est instancié à chaque tour, et en jetant un œil sur le fichier XML, on comprend qu'il contient deux attributs, *Marque* et *Modele*, et une collection nommée *Couleurs* contenant des valeurs de *Couleur*. Il est temps de stocker ces informations dans des objets C# de type *Moto* conçus dans les exercices précédents.

```
var m = new Moto  
{  
    Marque = moto.Attribute("Marque")?.value,  
    Modele = moto.Attribute("Modele")?.value,  
    Couleurs = new List<string>()  
};  
  
var couleurs = moto.Element("Couleurs");  
foreach (var color in couleurs?.Elements("Couleur"))  
{  
    if( color.value != null)  
        m.Couleurs.Add(color.value);  
}
```

## b. Écriture

LINQ to XML permet également de modifier le contenu XML et de le persister. Il est possible d'ajouter des attributs à des éléments en utilisant la méthode *SetAttributeValue* de *XElement*. Voici par exemple comment ajouter un attribut *puissance* à chaque élément *Moto*.

```
foreach (var moto in motos?.Elements("Moto"))
{
    if (moto.Attribute("Marque")?.Value == "BMW")
        moto.SetAttributeValue("Puissance", "109");
    else if (moto.Attribute("Marque")?.Value == "Ducati")
        moto.SetAttributeValue("Puissance", "135");
    else if (moto.Attribute("Marque")?.Value == "Triumph")
        moto.SetAttributeValue("Puissance", "68");
}
```

Un peu plus compliqué maintenant : ajoutons une nouvelle moto à notre collection. Il s'agira d'une moto de marque *Honda*, de modèle *Hornet* ayant une puissance de 93 chevaux et pouvant avoir les couleurs *rouge* et *noir*.

La première question à se poser est : "Où effectuer l'insertion ?". Dans notre cas, nous avons pris soin de créer un élément *motos* contenant une liste d'éléments de type *moto* et c'est donc ici que va s'effectuer l'insertion. Un objet *XElement* propose une méthode *Add* permettant de lui ajouter un sous-élément (également de type *XElement*). Ce sous-élément peut être créé à la volée en définissant ses attributs et éventuellement des "sous-sous-éléments" comme c'est le cas pour les couleurs. Voici le code d'ajout de notre *Honda*.

```
motos.Add(new XElement("Moto", new object[]
{
    new XAttribute("Marque", "Honda"),
    new XAttribute("Modele", "Hornet"),
    new XAttribute("Puissance", "93"),
    new XElement("Couleurs", new object[]
    {
        new XElement("Couleur", "Rouge"),
        new XElement("Couleur", "Noir"),
    })
}));
```

*XElement* propose plusieurs surcharges de son constructeur. Celui qui est utilisé ici prend en second paramètre un tableau d'objets les moins typés possible : *object[]*. Cela permet de définir dynamiquement l'organisation du nouvel élément, à savoir trois attributs et une collection de sous-éléments.

Pour sauvegarder les modifications, on utilisera la méthode *Save* de l'instance *XDocument*.

```
x.Save("testModifs.xml");
```

## c. Interrogations

On peut naturellement faire des requêtes avec LINQ to XML à la mode SQL ou à la mode C# directement dans les objets de type *XElement*. Demandons à notre collection de *motos* les modèles dont la puissance est supérieure à 100 chevaux et affichons les résultats.

```

var motosPlusDe100cv =
    from el in motos.Elements()
    where Int16.Parse(el.Attribute("Puissance").Value) > 100
    select el;

foreach (var motoPlusDe100cv in motosPlusDe100cv)
{
    Console.WriteLine($"{motoPlusDe100cv.Attribute("Marque")?.Value}" +
        $"{motoPlusDe100cv.Attribute("Modele")?.Value}");
}

```

Voici maintenant la même requête en mode C#.

```

var _motosPlusDe100cv =
    motos.Elements()
    .Where(el => Int16.Parse(el.Attribute("Puissance").Value) > 100);
foreach (var _motoPlusDe100cv in _motosPlusDe100cv)
{
    Console.WriteLine($"{_motoPlusDe100cv.Attribute("Marque")?.Value}" +
        $"{_motoPlusDe100cv.Attribute("Modele")?.Value}");
}

```