

CHAPITRE 6

INTERFACES ET MÉTHODES D'EXTENSION

6.1 INTRODUCTION GÉNÉRALE

Dans ce chapitre, nous allons voir ensemble comment créer et implémenter des interfaces, ainsi que les bénéfices de le faire.

6.1.1 *Type Interface*

Une **interface** peut être définie comme un **ensemble de membres abstraits (comportements services) qu'une classe ou une structure peut choisir de supporter (implémenter)**. Elles servent en quelque sorte d'établir une sorte de protocole de comportements attendus pour uniformiser, faciliter/assurer et prédire la communication entre différents composants.

De plus, comme nous le verrons, une classe ou une structure peut choisir de supporter (implémenter) plus d'une interface à la fois.

La librairie .NET vient avec des centaines de types interface prédéfinis qui sont implémentées par d'autres classes et structures.

Par exemple :

- La plupart des structures de données doivent par exemple implémenter certaines interfaces pour permettre leurs énumérations (`IEnumerable/IEnumerator`) ou certaines fonctionnalités propres à la catégorie de structure de données sous-jacente (comme `ICollection`, `IDictionary`, ...);
- Les types d'ADO.NET (pour la communication avec des bases de données) définit une interface nommée `IDbConnection` qui définit un ensemble de membres commun à tous les objets de connection ADO.NET, comme `Open()`, `Close()` et `CreateCommand()`, peu importe que la connection se fasse vient des classes `SqlConnection`, `OleDbConnection` ou `OdbcConnection`.

Par convention, toute interface débute « I... », y compris celles que vous créez.

6.1.2 Interface vs classe abstraite

Une des questions qui revient souvent lorsque'on parle d'interfaces est la suivante : *mais alors quelle est la différence entre une interface et une classe abstraite?*

En effet, les deux constructions peuvent sembler redondantes, mais en réalité elles ont plusieurs différences notoires :

- Lorsque vous définissez une **classe abstraite**, non seulement pouvez-vous définir des méthodes abstraites mais vous pouvez en plus **définir un certain nombre de constructeurs, de champs d'instance, de membres non abstraits** avec leurs implémentations, etc., alors que pour une **interface** vous ne pouvez **définir que des membres abstraits**;
- L'interface polymorphique d'une **classe parent abstraite** souffre d'une limitation en ce sens où **seuls ses types dérivés supportent les (héritent des) membres définis par celle-ci**; les **interfaces** n'ont pas cette limitation, du fait qu'elles **peuvent être implémentées par tout classe ou structure, dans toute hiérarchie, dans n'importe quel espace de nom ou assembly .NET**, ce qui en fait une construction hautement polymorphique.

Par exemple, imaginons qu'on a défini la classe abstraite suivante, pour laquelle on définit une méthode abstraite `Clone()`, qui définit qu'un type peut être cloné :

```
public abstract class TypeClonable
{
    public abstract object Clone();
}
```

De cette façon, seuls les types qui étendent `TypeClonable` peuvent supporter la méthode `Clone()`.

Si on crée une nouvelle ensemble hiérarchique de classes qui n'ont pas `TypeClonable` comme classe de base, alors aucune classe de la nouvelle hiérarchie n'aura accès à la méthode `Clone()` dans son interface polymorphique.

Ainsi, si on souhaite créer une classe Avant qui soit à la fois un JoueurHockey et un TypeClonable, on ne pourra pas le faire, puisque l'héritage multiple n'est pas supportée en C# :

```
1 reference
public abstract class TypeClonable
{
    0 references
    public abstract object Clone();
}

1 reference
public abstract class JoueurHockey { }

0 references
public class Avant : JoueurHockey, TypeClonable { }
Class 'Chapitre6.Avant' cannot have multiple base classes: 'Chapitre6.JoueurHockey' and 'TypeClonable'
```

Par contre, si on définit non pas une classe abstraite, mais une interface, comme ceci :

```
public interface IClonable
{
    object Clone();
}
```

On pourra alors sans problème l'implémenter par Avant en complémentaire à JoueurHockey :

```
public abstract class JoueurHockey { }

public class Avant : JoueurHockey, IClonable { }
```

Une interface similaire, ICloneable, est en fait une interface qui existe déjà dans le framework .NET (System.IClonable) et elle est implémentée par un large nombre de types qui n'ont que très peu ou pas du tout de lien entre eux : System.Array, System.Data.SqlClient.SqlConnection, System.OperatingSystem, System.String, etc.

Même s'ils n'ont pas de parents communs (sauf System.Object, implicitement), ils peuvent tous être traités polymorphiquement via le type interface ICloneable. On voit bien comment cela élargit grandement le potentiel polymorphique de la programmation orientée objet.

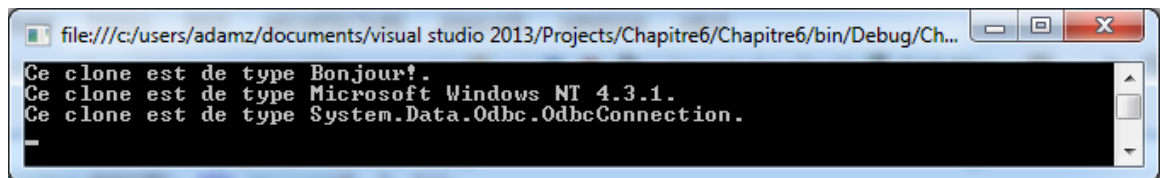
On peut donc imaginer le code suivant où on se déclare un tableau de différents types d'objets pour lequel on les parcourt un à un à faisant une conversion indirecte en type `ICloneable` et en invoquant la méthode `Clone()` :

```
class Program
{
    static void Main(string[] args)
    {
        object[] t = {
            "Bonjour!",
            new OperatingSystem(PlatformID.Win32NT, new Version(4, 3, 1)),
            new System.Data.Odbc.OdbcConnection()
        };

        foreach (ICloneable c in t)
            Console.WriteLine("Ce clone est de type {0}.", c.Clone());

        Console.ReadLine();
    }
}
```

Ce qui donne ceci à la console :



À nouveau, on voit les opérations polymorphiques. On aurait aussi pu simplement créer un tableau de type `ICloneable` directement et obtenir le même résultat, sans plantage quelconque :

```
ICloneable[] t = {
    "Bonjour!",
    new OperatingSystem(PlatformID.Win32NT, new Version(4, 3, 1)),
    new System.Data.Odbc.OdbcConnection()
};
```

- Une dernière différence entre classe abstraite et interface : lorsqu'on définit des **membres abstraits** dans une **classe abstraite**, toute classe dérivée est obligée de fournir une implémentation de chacun d'entre eux.

Imaginons que nous avons la hiérarchie suivante :

```
public abstract class Forme { }

public class Cercle : Forme { }

public class Rectangle : Forme { }

public class Hexagone : Forme { }

public class Cercle3D : Cercle { }
```

Et que nous souhaitons définir une nouvelle méthode qui retourne le nombre de points requis pour générer graphiquement les différentes formes (Points), alors une façon de faire pourrait être d'ajouter Points comme propriété abstraite dans la classe mère Forme, comme ceci :

```
public abstract class Forme
{
    public abstract byte Points { get; }
}
```

Cela oblige Cercle, Rectangle, Hexagone et Cercle3D à toutes les quatre implémenter la propriété Points, alors qu'on pourrait considérer que la notion de « points » ne s'applique pas à des formes comme Cercle et Cercle3D.

Une interface peut alors représenter une solution plus flexible et plus élégante où on peut simplement attacher l'interface (disons IPointu) aux types Rectangle et Hexagone (et leurs types dérivés), tout en faisant en sorte que ces classes héritent de tout ce qui est nécessaire du type Forme :

```
public abstract class Forme { }

public interface IPointu
{
    byte Points { get; }
}

public class Cercle : Forme { }

public class Rectangle : Forme, IPointu
{
    public byte Points { get { return 4; } }
}

public class Hexagone : Forme, IPointu
{
    public byte Points { get { return 6; } }
}

public class Cercle3D : Cercle { }
```

6.2 DÉFINITION ET IMPLÉMENTATION D'INTERFACES

Voyons maintenant ensemble quelques détails supplémentaires sur la définition et l'implémentation d'interfaces.

6.2.1 Définition

Voici quelques règles qu'il faut connaître au moment de créer une interface :

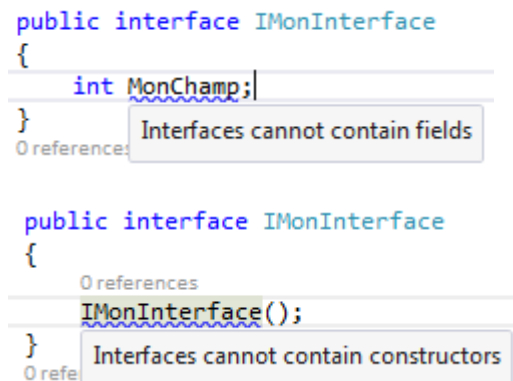
- Une interface est définie avec le mot clé `interface` plutôt que `class`;
- Une interface **ne peut pas hériter d'une classe de base**, mais seulement d'autres interfaces de base;
- Les membres d'une interface ne spécifient jamais le modificateur d'accès (c'est-à-dire qu'ils sont **implicitement publics et abstraits**);
- Une interface **peut contenir des définitions de méthodes ou de propriétés**, mais **pas de constructeurs ni de champs**;

Ainsi, ceci fonctionne (*remarquez le ; suite aux méthodes, mais pas pour les propriétés*):

```
public interface IMonInterface
{
    int MaProprieteLectureSeule { get; }
    string MaPropriete { get; set; }

    void MaMethode(string unParametre);
}
```

Mais ces tentatives se résultent par des erreurs de compilation :



```
public interface IMonInterface
{
    int MonChamp;
}
0 references Interfaces cannot contain fields

public interface IMonInterface
{
    IMonInterface();
}
0 references Interfaces cannot contain constructors
```

```
public interface IMonInterface
{
    O references
    public void MaMethode();
}
```

The modifier 'public' is not valid for this item

- Une interface **peut aussi contenir des événements et des indexeurs**. Nous reviendrons sur les événements dans un chapitre ultérieur;
- **On ne peut pas créer des instances d'interfaces**, comme on le ferait pour une classe non abstraite ou une structure; ceci ne fonctionne pas :

```
static void Main(string[] args)
{
    IPointu p = new IPointu();
}
```

interface Chapitre6.IPointu

Error:
Cannot create an instance of the abstract class or interface 'Chapitre6.IPointu'

En somme, pour qu'une interface prenne son sens, elle doit être implémentée par une classe ou une structure. C'est ce que nous ferons à la prochaine section.

6.2.2 Implémentation

Pour l'implémentation, voici quelques éléments à savoir :

- Si la classe n'a pas de classe de base explicite (c'est-à-dire qu'elle hérite implicitement de `System.Object`), alors vous n'avez qu'à la faire hériter directement de l'interface ou des interfaces :

```
public class MaClasse : IClonable, IPointu { }
```

- Même principe pour les structures (qui dérivent implicitement de `System.ValueType`, pour le rappeler) :

```
public struct MaClasse : IClonable, IPointu { }
```

- Si la classe hérite d'une autre classe, **vous devez obligatoirement indiquer d'abord la classe de base avant de faire hériter de toute interface** :

```
public class Rectangle : Forme, IPointu { }
```

```
public class Rectangle : IPointu, Forme
{
    0 references
```

Base class 'Forme' must come before any interfaces

- L'implémentation d'une interface est une proposition tout ou rien, c'est-à-dire que vous ne pouvez pas choisir d'implémenter seulement certains membres d'une interface : si vous choisissez qu'une classe implémente une interface, elle doit donc implémenter en entier les membres implicitement abstraits demandés.

Mettons un peu à jour notre exemple de formes en tenant compte des considérations nouvelles suivantes :

- La classe abstraite de base `Forme` demande maintenant à toutes ses classes dérivées de devoir définir une méthode `Dessiner()` ;
- Une nouvelle classe dérivée `Triangle` doit être ajoutée, qui implémente `IPointu` en plus de dériver de `Forme` ;
- On ajoute des constructeurs aux classes dérivées qui prennent comme paramètre un nom associé à toute forme.

On peut alors imaginer le code suivant :

```
public abstract class Forme
{
    public string Nom { get; private set; }

    public Forme(string nom)
    {
        this.Nom = nom;
    }

    public abstract void Dessiner();
}

public interface IPointu
{
    byte Points { get; }
}

public class Cercle : Forme
{
    public Cercle(string nom)
        : base(nom) { }
    public override void Dessiner()
    {
        Console.WriteLine("Dessinage du cercle nommé " + this.Nom);
    }
}
```



```
public class Triangle : Forme, IPointu
{
    public Triangle(string nom)
        : base(nom) { }
    public byte Points { get { return 3; } }

    public override void Dessiner()
    {
        Console.WriteLine("Dessinage du triangle nommé " + this.Nom);
    }
}

public class Rectangle : Forme, IPointu
{
    public Rectangle(string nom)
        : base(nom) { }
    public byte Points { get { return 4; } }

    public override void Dessiner()
    {
        Console.WriteLine("Dessinage du rectangle nommé " + this.Nom);
    }
}

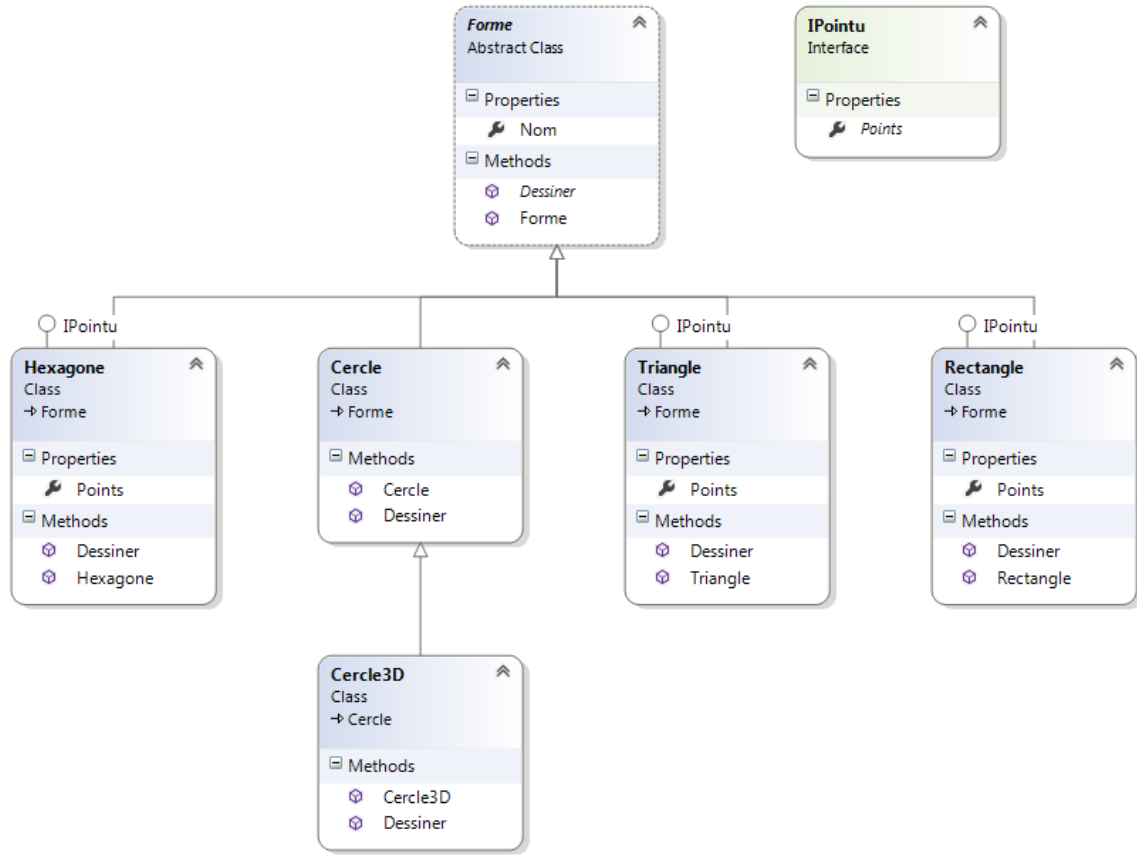
public class Hexagone : Forme, IPointu
{
    public Hexagone(string nom)
        : base(nom) { }
    public byte Points { get { return 6; } }

    public override void Dessiner()
    {
        Console.WriteLine("Dessinage de l'hexagone nommé " + this.Nom);
    }
}

public class Cercle3D : Cercle
{
    public Cercle3D(string nom)
        : base(nom) { }

    public override void Dessiner()
    {
        Console.WriteLine("Dessinage du cercle 3D nommé " + this.Nom);
    }
}
```

Si vous voulez avoir un aperçu de ce à quoi ressemblent les relations entre ces classes de façon plus imagée, vous pouvez toujours utiliser le diagramme de classes de Visual Studio, qui vous montrerait ceci :



Cela permet du même coup de bien voir la différence entre une classe abstraite et une interface.

6.3 UTILISATION DES INTERFACES ET DE SES MEMBRES

Maintenant que nous avons quelques classes supportant l'interface **IPointu**, il est l'heure d'en invoquer la fonctionnalité en invoquant la propriété **Points**.

6.3.1. Appel de membres

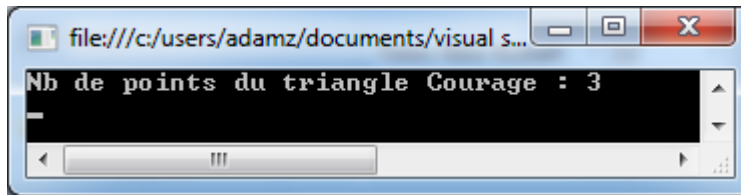
À la base, il n'y a aucun problème à directement appeler la méthode si on a affaire à une instance d'une classe qui implémente **IPointu**. Par exemple, on s'attend à ce que :

```

static void Main(string[] args)
{
    Triangle t = new Triangle("Courage");
    Console.WriteLine("Nb de points du triangle {0} : {1}", t.Nom,
        t.Points);
    Console.ReadLine();
}
  
```

```
}
```

Mène à la sortie suivante :



Dans d'autres cas, on pourrait ne pas être en mesure de savoir si un type implémente l'interface en question.

Imaginons que nous avons un tableau de plusieurs formes :

```
Forme[] formes =  
{  
    new Triangle("Courage"),  
    new Cercle3D("Ballon"),  
    new Cercle("Interdiction"),  
    new Hexagone("Blason"),  
    new Rectangle("Photo")  
};
```

Comme nous l'avons montré pour l'héritage, vous ne serez pas en mesure d'invoquer la propriété `Points` pour n'importe quelle des instances du tableau `formes` sans obtenir des erreurs de compilations :

```
foreach(Forme f in formes)  
    Console.WriteLine("Nb de points de la forme {0} : {1}", f.Nom, f.Points);  
  
Console.ReadLine();
```

'Chapitre6.Forme' does not contain a definition for 'Points'

Afin de déterminer à l'exécution si un type supporte une interface spécifique, on pourra procéder de trois (3) façons, toutes analogues à ce que nous avons vu pour l'héritage entre classes.

La première consiste à **utiliser une conversion explicite**, accompagné d'une gestion d'exception de nature `InvalidCastException`, de sorte que si la conversion est impossible on pourra en rendre compte sans que l'application plante. :

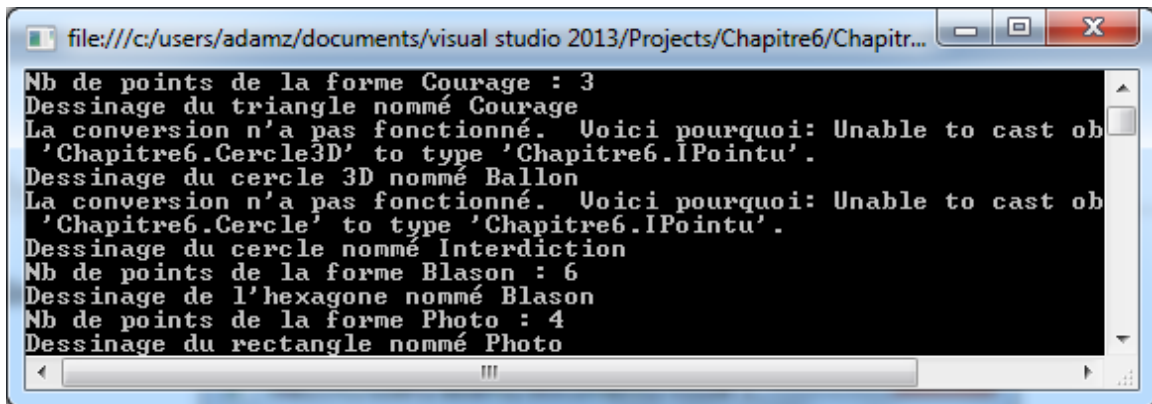
```
foreach (Forme f in formes)  
{  
    try  
    {  
        Console.WriteLine("Nb de points de la forme {0} : {1}", f.Nom,  
            ((IPointu)f).Points);  
    }  
}
```

```
        catch (InvalidCastException e)
        {
            Console.WriteLine("La conversion n'a pas fonctionné. Voici pourquoi: " + e.Message);
        }

        f.Dessiner();
    }

    Console.ReadLine();
```

À la console, vous obtiendrez :



Bien que cela fonctionne, il reste qu'on s'oblige à devoir utiliser de la gestion d'exception plutôt que de faire des vérifications préalables (en ce sens où elles peuvent être faites afin d'éviter à devoir recourir à la gestion d'exception).

6.3.2 As

Vous pouvez également utiliser le mot clé `as` avec les interfaces. Vous pourriez donc modifier le code suivant en éliminant la gestion d'exception, mais en tentant une conversion des instances en des interfaces `IPointu` :

- Si la conversion est impossible, **null** sera retournée comme valeur de retour;
- Si elle est possible, alors **on aura l'instance convertie** dont les membres sont prêts à être invoqués.

Ainsi, on pourrait avoir ce code :

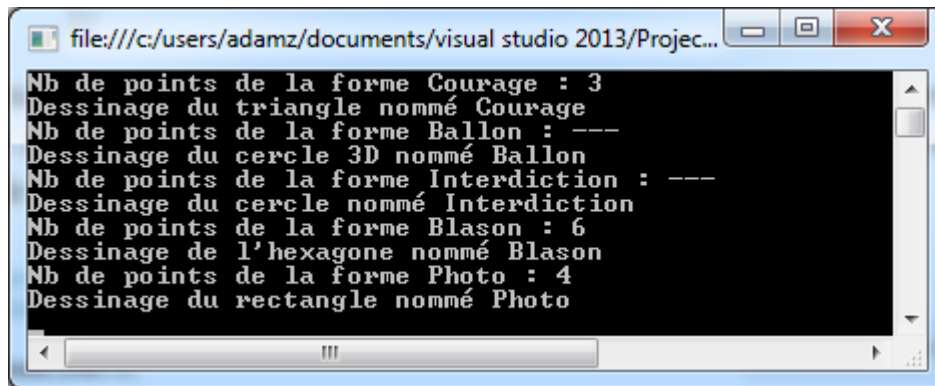
```
foreach (Forme f in formes)
{
    IPointu p = f as IPointu;
```

```
        if (p == null)
            Console.WriteLine("Nb de points de la forme {0} : ---",
                               f.Nom);
        else
            Console.WriteLine("Nb de points de la forme {0} : {1}", f.Nom,
                               p.Points);

        f.Dessiner();
    }

    Console.ReadLine();
```

Qui donnerait ceci en sortie :



6.3.3 Is

Le mot clé `is` est aussi disponible. Rappelez-vous qu'il retourne vrai ou faux selon si la variable est du type demandé ou non. Il faut ensuite faire une conversion explicite avant d'invoquer les membres propres attachés au type en question.

```
foreach (Forme f in formes)
{
    if (f is IPointu)
        Console.WriteLine("Nb de points de la forme {0} : {1}", f.Nom,
                           ((IPointu)f).Points);
    else
        Console.WriteLine("Nb de points de la forme {0} : ---",
                           f.Nom);

    f.Dessiner();
}

Console.ReadLine();
```

La sortie serait en tout point identique qu'avec le `as`.

6.3.4 Passage d'interfaces en paramètres

On peut évidemment avoir des méthodes qui prennent des interfaces comme paramètres, comme pour la méthode `Clone()` que nous avons utilisée au tout début du chapitre.

Pour étoffer davantage notre exemple, supposons une seconde interface nommée **IDessiner3D** définie simplement comme suit :

```
public interface IDessiner3D
{
    void Dessiner3D();
}
```

Supposons maintenant que la classe `Cercle3D` implémente aussi cette interface. Sa définition mise à jour devient alors celle-ci :

```
public class Cercle3D : Cercle, IDessiner3D
{
    public Cercle3D(string nom)
        : base(nom) { }

    public override void Dessiner()
    {
        Console.WriteLine("Dessinage du cercle 3D nommé " + this.Nom);
    }

    public void Dessiner3D()
    {
        Console.WriteLine("Dessinage en 3 dimensions du cercle 3D nommé " +
            this.Nom);
    }
}
```

Si on définit une méthode dans `Program` qui prend une interface `IDessiner3D` comme paramètre, on pourra lui passer toute instance dont le type implémente cette interface :

```
class Program
{
    static void Main(string[] args)
    {
        Forme[] formes =
        {
            new Triangle("Courage"),
            new Cercle3D("Ballon"),
            new Cercle("Interdiction"),
            new Hexagone("Blason"),
            new Rectangle("Photo")
        }
    }
}
```

```

    };

    foreach (Forme f in formes)
    {
        if (f is IPointu)
            Console.WriteLine("\nNb de points de la forme {0} : {1}",
                               f.Nom, ((IPointu)f).Points);
        else
            Console.WriteLine("\nNb de points de la forme {0} : ---",
                               f.Nom);

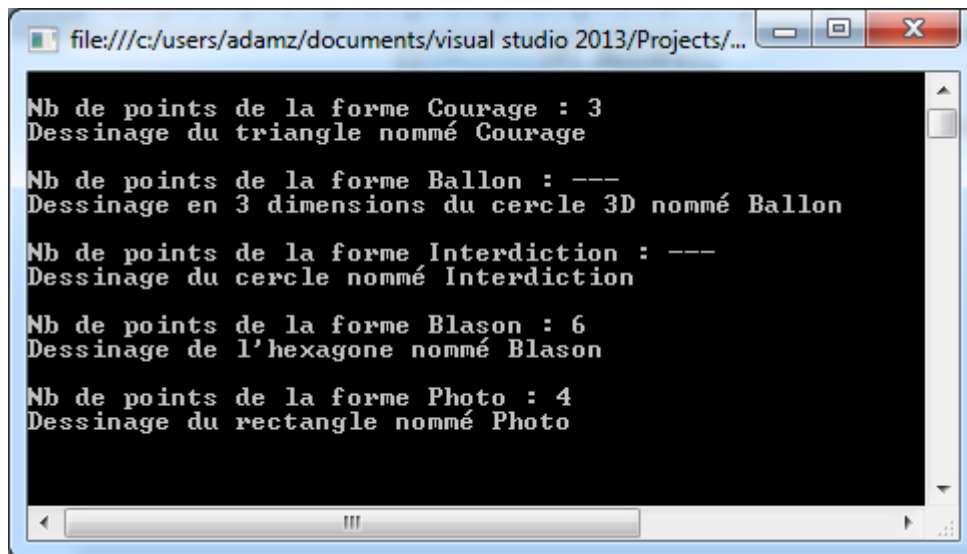
        if (f is IDessiner3D)
            DessinerEn3D((IDessiner3D)f);
        else
            f.Dessiner();
    }

    Console.ReadLine();
}

static void DessinerEn3D(IDessiner3D iDessiner3D)
{
    iDessiner3D.Dessiner3D();
}
}

```

Si vous tentez de passer un type qui ne supporte pas l'interface, une erreur de compilation vous sera donnée :



Le même principe s'applique si vous utilisez une interface comme type de valeur de retour.

6.4 IMPLÉMENTATION D'INTERFACES EXPLICITES

Comme discuté plus tôt, une classe ou une structure peut implémenter plusieurs interfaces si cela est souhaité.

De ce fait, il n'est pas impossible qu'un type implémente plus d'une interface qui définisse des membres identiques. Il importe de comprendre comment on peut et on doit gérer ces situations.

Imaginons que nous avons une nouvelle classe `Octogone` qui implémente plusieurs interfaces qui demandent toutes de définir une méthode `Dessiner()` :

```
public interface IDessinerVersFenetre
{
    void Dessiner();
}

public interface IDessinerVersImprimante
{
    void Dessiner();
}

public interface IDessinerVersMemoire
{
    void Dessiner();
}

public class Octogone : IDessinerVersFenetre, IDessinerVersImprimante,
IDessinerVersMemoire
{
    //???
```

Que faire?

En fait vous avez deux solutions possibles :

- Vous pouvez définir une méthode `Dessiner()` unique qui satisfera l'exigence des trois interfaces à la fois :

```
public class Octogone : IDessinerVersFenetre, IDessinerVersImprimante,
IDessinerVersMemoire
{
    public void Dessiner()
    {
        Console.WriteLine("Dessinage de l'octogone.");
    }
}
```


Cela compile sans problème, mais mène vers un autre : cela ne permet pas à la classe `Octogone` d'avoir des comportements propre à chacun des services `Dessiner()` des interfaces liées.

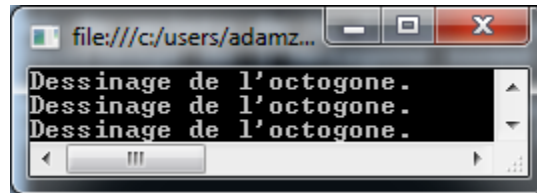
Plus spécifiquement, même si vous faites une conversion explicite d'un octogone vers une des trois interfaces, vous ferez toujours appel à la même méthode `Dessiner()` :

```
Octogone octo = new Octogone();

((IDessinerVersFenetre)octo).Dessiner();
((IDessinerVersImprimante)octo).Dessiner();
((IDessinerVersMemoire)octo).Dessiner();

Console.ReadLine();
```

Effectivement, vous aurez en console :



- Si vous voulez donner des implémentations spécifiques de `Dessiner()` en lien avec chacune des interfaces concernées, vous devez explicitement donner le nom de l'interface devant le nom de la méthode à implémenter.

Dans notre exemple, cela demanderait de faire ceci :

```
public class Octogone : IDessinerVersFenetre, IDessinerVersImprimante,
IDessinerVersMemoire
{
    void IDessinerVersFenetre.Dessiner()
    {
        Console.WriteLine("Dessinage de l'octogone vers la fenêtre
        active.");
    }

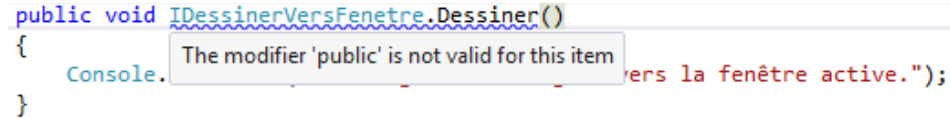
    void IDessinerVersImprimante.Dessiner()
    {
        Console.WriteLine("Dessinage de l'octogone vers l'imprimante.");
    }

    void IDessinerVersMemoire.Dessiner()
    {
        Console.WriteLine("Dessinage de l'octogone en mémoire.");
    }
}
```

C'est ce qu'on appelle une syntaxe d'**implémentation d'interface explicite**.

Remarquez que vous ne pouvez plus utiliser le modificateur d'accès public devant les signatures, sous peine de recevoir le message d'erreur suivant :

```
public void IDessinerVersFenetre.Dessiner()  
{  
    Console.WriteLine("Dessinage de l'octogone vers la fenêtre active.");  
}
```



Ceci est dû au fait que **les membres implémentés explicitement sont réputés être implicitement et automatiquement privés**.

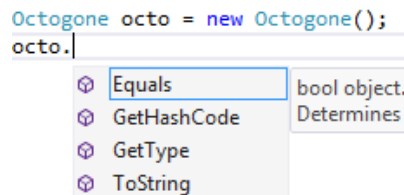
La raison à cela est qu'on souhaite éviter des ambiguïtés au moment d'invoquer la méthode à plusieurs implémentations à partir d'une instance d'un type les implémentant.

Dans notre exemple, on cherche donc à éviter une ambiguïté au moment de faire :

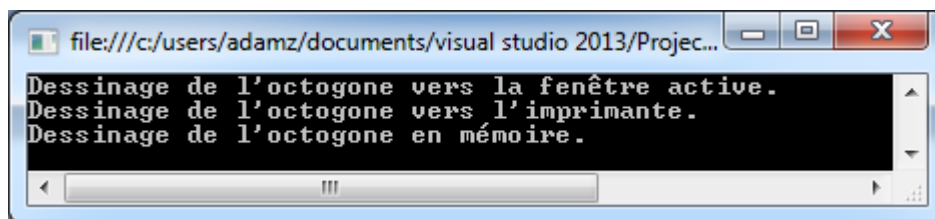
```
octo.Dessiner();
```

En effet, même si vous tapez `octo.`. Et que vous regardez dans la liste de choix contextuelle, vous ne retrouverez pas la méthode `Dessiner()` (puisque'elle est privée) :

```
Octogone octo = new Octogone();  
octo.  
Equals  
GetHashCode  
GetType  
ToString
```



Par contre, le même code utilisé plus tôt permettrait d'invoquer chacune des implémentations en lien avec la bonne interface (puisque'implicitement du côté de l'interface la méthode à implémenter est publique) :



6.5 INTERFACES, HIÉRARCHIES ET HÉRITAGE MULTIPLE

Les interfaces **peuvent être organisées dans une hiérarchie d'interfaces**, c'est-à-dire que des interfaces peuvent implémenter d'autres interfaces. Ce faisant, comme pour une classe qui hérite d'une autre, une interface qui en étend une seconde « hérite » des membres abstraits de cette dernière.

La raison pour laquelle on dit « hérite » entre guillemets est qu'en fait il n'y a pas un réel héritage qui se produit : **une interface dérivée étend tout simplement sa propre définition à celles des interfaces parents.**

Elle est **particulièrement utile dans les cas où on souhaiterait étendre les fonctionnalités d'une interface pré-existante sans corrompre toutes les classes et les structures qui implémentent déjà l'interface en question.**

Imaginons qu'on souhaite étendre notre interface `IDessiner` afin de lui ajouter des méthodes à implémenter de dessin avancé :

```
public interface IDessiner
{
    void Dessiner();
}

public interface IDessinerAvance : IDessiner
{
    void DessinerAvecEchelle(double scalaire);
    void DessinerMiroir();
}
```

Une classe qui implémente `IDessinerAvance` devra alors implémenter les trois (3) méthodes :

```
public class Image : IDessinerAvance
{
    public void Dessiner()
    {
        Console.WriteLine("L'image est dessinée...");
    }

    public void DessinerAvecEchelle(double scalaire)
    {
        Console.WriteLine("L'image est dessinée avec un rapport de x{0}...",
            scalaire);
    }
}
```

```
public void DessinerMiroir()
{
    Console.WriteLine("L'image est dessinée à l'envers
    horizontalement...");
}
}
```

Cela permet alors ce genre de manipulations :

```
Image image = new Image();

image.Dessiner();
image.DessinerAvecEchelle(2.4);
image.DessinerMiroir();

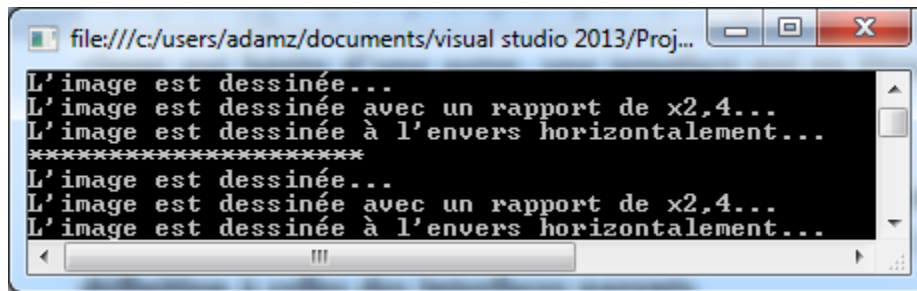
Console.WriteLine("*****");

//-- ou en convertissant explicitement vers IDessinerAvance ---
IDessinerAvance iDessin = image as IDessinerAvance;

if (iDessin != null)
{
    iDessin.Dessiner();
    iDessin.DessinerAvecEchelle(2.4);
    iDessin.DessinerMiroir();
}

Console.ReadLine();
```

Qui donnerait ceci à l'affichage :



Par ailleurs, les interfaces elles-mêmes peuvent étendre plusieurs interfaces. Les conséquences sont les mêmes que celle présentées précédemment.

Pour faire un résumé, on pourrait dire que **les interfaces sont particulièrement utiles** :

- Lorsqu'on a une hiérarchie à l'intérieur de laquelle seulement un sous-ensemble des types dérivés doit supporter des comportements communs;
- Lorsqu'on doit modéliser un comportement commun à travers plusieurs hiérarchies non explicitement reliées entre elles.

6.6 IENUMERABLE ET IENUMERATOR

6.6.1 Introduction

Depuis le début de la matière, nous avons constamment utilisé la structure de contrôle `foreach` afin d'itérer sur des tableaux d'éléments. Par exemple :

```
int[] mesEntiers = { 5, 2, 4, 1, 8 };

foreach (int i in mesEntiers)
    Console.WriteLine(i);
```

Il est possible d'utiliser `foreach` pour itérer à travers tous les éléments non seulement de tableaux, mais d'autres types de collections autres que `Array`, comme `List`, par exemple.

Encore plus précisément, tout type qui supporte une méthode nommée `GetEnumerator()` peut être évaluée par une structure de contrôle `foreach`.

Imaginons ces classes-ci, par exemple :

```
public class JoueurHockey
{
    public String Nom { get; private set; }
    public int Numero { get; private set; }

    public JoueurHockey(String nom, int numero)
    {
        this.Nom = nom;
        this.Numero = numero;
    }
}

public class Equipe
{
    public JoueurHockey[] Joueurs { get; private set; }

    public Equipe()
    {
        Joueurs = new JoueurHockey[4]
        {
            new JoueurHockey("Mathieu Latotale", 24),
            new JoueurHockey("Cédric Laforest", 11),
            new JoueurHockey("Éric Tousignant", 33),
            new JoueurHockey("Stéphane Atik", 87)
        };
    }
}
```

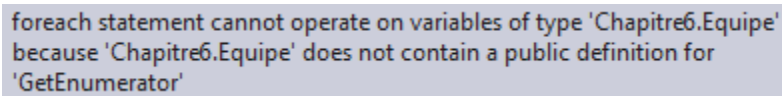
Et tentons d'utiliser un `foreach` pour énumérer tous les joueurs de l'équipe dans le `Main()` :

```
static void Main(string[] args)
{
    Equipe equipe = new Equipe();

    foreach (JoueurHockey joueur in equipe)
        Console.WriteLine(joueur.Nom + " (#" + joueur.Numero + ")");

    Console.ReadLine();
}
```

Cela génère une erreur de compilation :



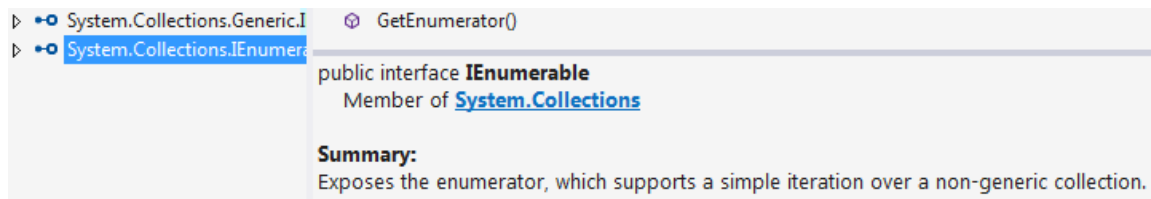
```
foreach statement cannot operate on variables of type 'Chapitre6.Equipe'
because 'Chapitre6.Equipe' does not contain a public definition for
'GetEnumerator'
```

6.6.2 Syntaxe générale

La définition de la méthode `GetEnumerator()` est exigée par tout type implémentant l'interface `IEnumerable`.

De cette façon, tous ces types annoncent qu'ils sont capables d'exposer leur contenu à l'appelant.

Dans l'explorateur d'objets, vous pouvez constater que cette interface ne définit que cette méthode :



Ou si vous préférez :

```
public interface IEnumerable
{
    IEnumerator GetEnumerator();
}
```

On peut observer que la méthode retourne une référence vers une interface de type `System.Collections.IEnumerator`.

Cette interface fournit l'infrastructure nécessaire pour que l'appelant puisse itérer à travers les objets internes contenus par tout conteneur compatible avec l'interface `IEnumerable`.

Elle est définie comme suit :

```
public interface IEnumerator
{
    bool MoveNext();
    object Current { get; }
    void Reset();
}
```

Sachant que :

- La méthode `MoveNext()` **avance la position interne du curseur vers l'élément suivant** et retourne `true` tant et aussi longtemps que l'élément suivant existe; sinon `false`;
- La propriété en lecture seule `Current` **récupère l'élément** courant;
- La méthode `Reset()` **réinitialise le curseur** de sorte qu'il pointe vers le premier élément.

Si nous souhaitons faire en sorte que notre type `Equipe` supporte ces interfaces, deux choix s'offrent à vous :

- **Implémenter vos propres versions** personnalisées de `GetEnumerator()`, `MoveNext()`, `Current` et `Reset()`;
- **Déléguer ces services aux implémentations déjà existantes** pour le type de conteneur concerné.

La seconde est beaucoup plus simple et c'est celle que nous utiliserons.

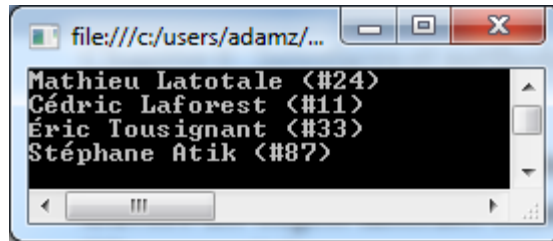
Ainsi, pour ajouter le support d'énumération à votre type, vous devez d'une part ajouter une instruction `using` vers `System.Collections` (puisque votre conteneur est un `Array`), puis spécifier que vous désirez implémenter l'interface `IEnumerable`. Vous devrez ensuite implémenter `GetEnumerator()` :

```
public class Equipe : IEnumerable
{
    public JoueurHockey[] Joueurs { get; private set; }
```

```
public Equipe()
{
    Joueurs = new JoueurHockey[4]
    {
        new JoueurHockey("Mathieu Latotale", 24),
        new JoueurHockey("Cédric Laforest", 11),
        new JoueurHockey("Éric Tousignant", 33),
        new JoueurHockey("Stéphane Atik", 87)
    };
}

public IEnumerator GetEnumerator()
{
    return Joueurs.GetEnumerator();
}
}
```

Immédiatement, votre code du côté du `Main()` devrait être maintenant fonctionnel et vous permettre d'utiliser un `foreach` sur une équipe :



Par ailleurs, vous pouvez aussi de cette façon manuellement manipuler une variable de type `IEnumerator` :

```
static void Main(string[] args)
{
    Equipe equipe = new Equipe();
    IEnumerator i = equipe.GetEnumerator();

    while (i.MoveNext())
        Console.WriteLine(((JoueurHockey)i.Current).Nom + " (#" +
            ((JoueurHockey)i.Current).Numero + ")");

    Console.ReadLine();
}
```

Si vous voulez éviter que de l'extérieur on puisse récupérer un énumérateur, vous pouvez toujours implémenter explicitement l'interface de façon à la rendre explicitement privé :

```
IEnumerator IEnumerable.GetEnumerator()
{
    return Joueurs.GetEnumerator();
}
```


Par ailleurs, sachez que vous pouvez faire un clic droit sur l'interface et demander d'implémenter pour vous toutes les bases de l'interface à implémenter :



6.6.3 Itérateurs et yield

Dans les premières versions de .NET, cette façon de faire était la seule méthode possible pour permettre l'énumération avec un `foreach`.

Il y existe maintenant une alternative pour concevoir des types qui fonctionnent avec les `foreach` via des itérateurs.

Un **itérateur** est un **membre qui spécifie comment les éléments internes d'un conteneur devraient être retournés lorsque traités par un `foreach`**.

Même si la méthode de l'itérateur doit tout de même être nommée `GetEnumerator()` et retourner une variable de type `IEnumerator`, la classe en soi n'a pas à implémenter l'interface conventionnelle.

Vous pourriez par exemple modifier votre classe `Equipe` comme suit :

```
public class Equipe
{
    public JoueurHockey[] Joueurs { get; private set; }

    public Equipe()
    {
        Joueurs = new JoueurHockey[4]
        {
            new JoueurHockey("Mathieu Latotale", 24),
            new JoueurHockey("Cédric Laforest", 11),
            new JoueurHockey("Éric Tousignant", 33),
            new JoueurHockey("Stéphane Atik", 87)
        };
    }

    public IEnumerator GetEnumerator()
    {
        foreach (JoueurHockey joueur in Joueurs)
            yield return joueur;
    }
}
```

Remarquez qu'on utilise une boucle `foreach` interne pour énumérer un à un les éléments du `Array`.

Remarquez également l'utilisation du mot clé `yield` devant `return` :

- Le mot clé **yield** est utilisé pour **spécifier le ou les valeurs à être retournées au `foreach` appelant**;
- Lorsque l'instruction `yield return` est atteinte, la **position courante dans le conteneur est mémorisée** et l'exécution repartira de cette position la prochaine fois que l'itérateur sera appelé.

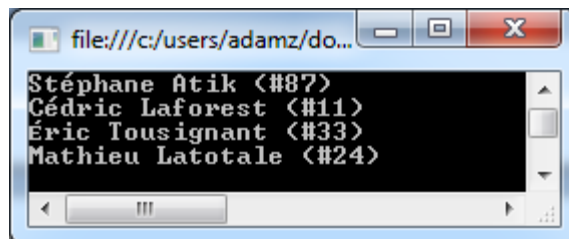
Sachez par ailleurs que vous n'êtes pas tenus d'utiliser un `foreach` dans votre méthode d'itérateur pour retourner son contenu. Vous pourriez fort bien écrire quelque chose comme ceci :

```
public IEnumerator GetEnumerator()  
{  
    yield return Joueurs[0];  
    yield return Joueurs[1];  
    yield return Joueurs[2];  
    yield return Joueurs[3];  
}
```

Et cela retournerait les joueurs dans l'ordre indiqué. Dans la même optique, vous pouvez retourner les éléments dans l'ordre que vous désirez :

```
public IEnumerator GetEnumerator()  
{  
    yield return Joueurs[3];  
    yield return Joueurs[1];  
    yield return Joueurs[2];  
    yield return Joueurs[0];  
}
```

Vous verriez à l'exécution qu'effectivement l'ordre diffère alors :



Évidemment, ce n'est pas l'idéal de procéder de cette façon, car si on modifie le tableau référencé on pourrait générer des erreurs d'éléments inexistants ou ne pas couvrir tous les

éléments. Néanmoins, cela permet de démontrer la flexibilité dans la définition d'un itérateur. Vous pouvez même décider de retourner d'autres champs locaux, résultats de calculs, ou champs d'instances à votre guise pour autant, idéalement, qu'ils soient du même type que l'énumération.

Par exemple, ceci compilerait :

```
public class Equipe
{
    public JoueurHockey[] Joueurs { get; private set; }
    public string Nom { get; private set; }
    public Equipe()
    {
        this.Nom = "Les Sabres Laser de Saint-Ours-de-Timon";
        Joueurs = new JoueurHockey[4]
        {
            new JoueurHockey("Mathieu Latotale", 24),
            new JoueurHockey("Cédric Laforest", 11),
            new JoueurHockey("Éric Tousignant", 33),
            new JoueurHockey("Stéphane Atik", 87)
        };
    }

    public IEnumerator GetEnumerator()
    {
        string s = "test";

        yield return Joueurs[3];
        yield return s;
        yield return s == "test" ? "vrai" : "faux";
        yield return Joueurs[1];
        yield return Nom;
        yield return Joueurs[0];
    }
}
```

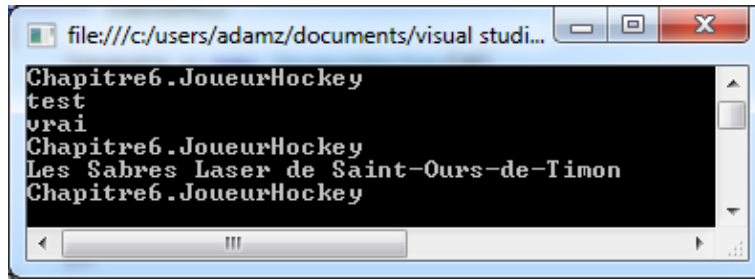
Mais du côté du `Main()`, il faudrait faire un `foreach` en utilisant le type `Object` sous peine d'avoir un plantage à l'exécution :

```
static void Main(string[] args)
{
    Equipe equipe = new Equipe();

    foreach (Object o in equipe)
        Console.WriteLine(o);

    Console.ReadLine();
}
```

On obtiendrait alors ceci :



6.6.4 Itérateur nommé

Vous pouvez par ailleurs créer des méthodes qu'on appelle des itérateurs nommés.

Ces méthodes peuvent prendre n'importe quantité de paramètres, mais ils doivent retourner une valeur de type `IEnumerable` (en contraste avec `IEnumerator` pour l'itérateur normal). On peut s'en servir justement pour permettre des énumérations plus particulières.

Par exemple, imaginons que nous souhaitons permettre d'utiliser un `foreach` sur les joueurs de l'équipe dans l'ordre normal, mais aussi dans l'ordre inverse des joueurs dans le conteneur.

Nous pourrions définir la méthode suivante :

```
public IEnumerable GetJoueurs(bool ordreInverse)
{
    if (ordreInverse)
        for (int i = Joueurs.Length - 1; i >= 0; i--)
            yield return Joueurs[i];
    else
        foreach (JoueurHockey j in Joueurs)
            yield return j;
}
```

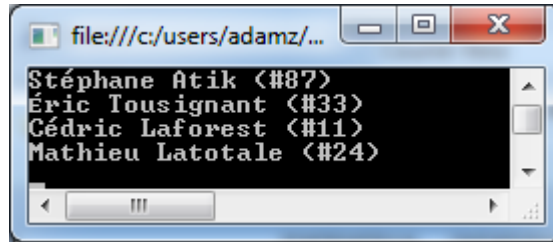
Puis y faire référence dans un `foreach` du côté appelant, comme ceci :

```
static void Main(string[] args)
{
    Equipe equipe = new Equipe();

    foreach (JoueurHockey joueur in equipe.GetJoueurs(true))
        Console.WriteLine(joueur.Nom + " (" + joueur.Numero + ")");

    Console.ReadLine();
}
```

Effectivement, on peut voir que cela fonctionne à l'exécution :



6.7 MÉTHODES D'EXTENSION

L'utilisation de méthodes d'extension permet d'ajouter une ou plusieurs fonctionnalités à des classes (des types) pour lesquelles nous n'avons pas accès au code de base (p. ex. des types contenus dans des bibliothèques de classes).

6.7.1 Définition

Lorsque vous créez des *assemblies*, la définition des types qu'ils contiennent sont en quelque sorte normalement « finaux ». Dans certains contextes, il pourrait s'avérer être intéressant de pouvoir injecter certaines fonctionnalités ou forcer un type à supporter un ensemble de membres. Les méthodes d'extension introduites par C# .NET 2008 permettent de donner l'illusion en ajoutant des fonctionnalités tout en donnant l'illusion qu'elles appartiennent au type précompilé (c'est-à-dire du type de la bibliothèque préexistante).

Cependant, il y a une première **restriction** assez forte à cela :

- Des méthodes d'extension ne peuvent être ajoutées qu'à des classes statiques (et donc les méthodes d'extension doivent aussi être statiques).

La deuxième contrainte est la suivante :

- Chaque méthode d'extension peut soit être appelée à partir de la bonne instance en mémoire ou de façon statique en passant par la classe statique en question.

Enfin, dernière règle :

- Dans la définition des méthodes d'extension, il faut utiliser le mot clé **this** devant le type et le nom de la variable du premier paramètre (et uniquement celui-là; il peut y avoir plus d'un paramètre). Ce premier paramètre indique le type devant faire appel à la méthode en question.

Voyons en détails un exemple pour éclaircir le tout. Considérons tout d'abord la classe statique suivante ajoutée à un projet :

```
public static class MesExtensions
{
    public static void AfficherAssembly(this object o)
    {
        Console.WriteLine(o.GetType().FullName +
            " appartient à l'assembly " + Assembly.GetAssembly(o.GetType()));
    }

    public static int InverserChiffres(this int i)
    {
        char[] chiffres = i.ToString().ToCharArray();
        Array.Reverse(chiffres);

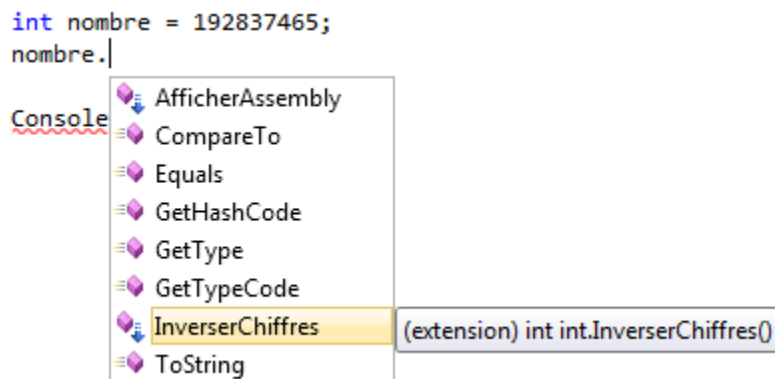
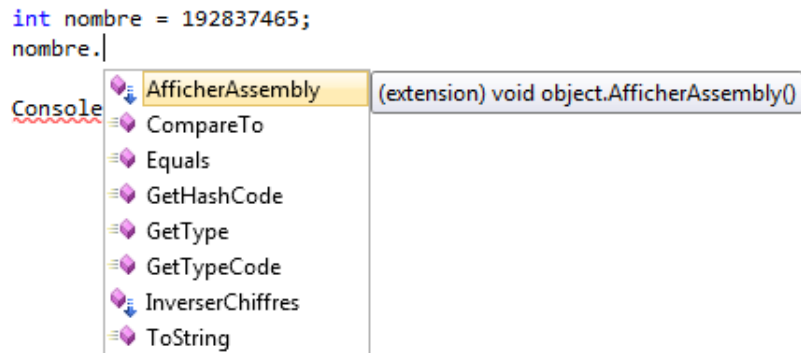
        return int.Parse(new string(chiffres));
    }
}
```


Remarquez l'utilisation de `this int i` et `this object o` dans la signature des méthodes. Le mot clé `this`, comme nous l'avons mentionné, indique qu'il s'agit d'une méthode d'extension. Le type du premier paramètre `int` (ou `object`) indique pour sa part qu'un entier (ou n'importe quel type dans le cas de `object`) est traité par la méthode et peut invoquer cette méthode.

En d'autres termes, le type pour la méthode d'extension `AfficherAssembly()` étant `object`, tout type (du fait que toutes les classes héritent implicitement de la classe mère `Object`) aura maintenant accès à cette méthode (de façon non statique : nous verrons comment dans un instant); par contre, dans le de la seconde méthode d'extension, `InverserChiffres()`, le type du premier (et seul) paramètre est un entier, ce qui fait en sorte que `InverserChiffres()` constituera une extension pour le type `int` seulement (et ses descendants, s'il en avait).

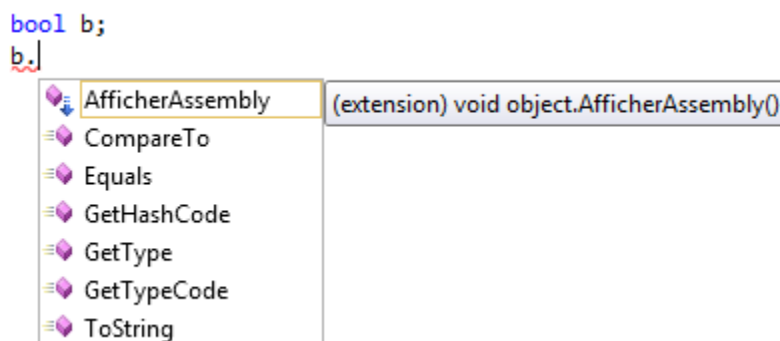
La variable `o` ou `i` sert alors à spécifier quelle doit être le traitement à effectuer pour l'instance qui fait appel à cette méthode d'extension.

Par exemple, après avoir défini ces méthodes, l'intelliSense vous indique les informations suivantes pour, disons, une variable quelconque (dans ce cas-ci un entier).



Remarquez l'icône qui diffère () pour indiquer qu'il s'agit d'une méthode d'extension. Le message d'aide indique également cette caractéristique. Vous pouvez également observer dans la signature de ces méthodes que le premier paramètre (`this int i` ou `this object o`) ne sont pas considérés comme de véritables paramètres, puisqu'ils ne servent qu'à indiquer quel type étendre et éventuellement faire le traitement approprié en lien avec la fonctionnalité étendue.

Comme `InverserChiffres()` étend `int` et rien d'autre, si on a plutôt une variable de type booléenne, par exemple, on n'aura pas accès à cette méthode d'extension particulière :



6.7.2 Invocation à un niveau d'instance

Pour invoquer ces méthodes dans un contexte donné à l'intérieur d'une application qui a une vue sur ces méthodes d'extension (parce qu'elles appartiennent au même projet ou parce que vous avez référencé l'*assembly* qui les contient), il suffit de procéder comme présenté à la sous-section précédente, c'est-à-dire, par exemple :

```
static void Main(string[] args)
{
    int nombre = 192837465;

    Console.WriteLine("Nombre initial: " + nombre);
    Console.WriteLine("Nombre inversé: " + nombre.InverserChiffres());
    nombre.AfficherAssembly();

    Console.WriteLine();

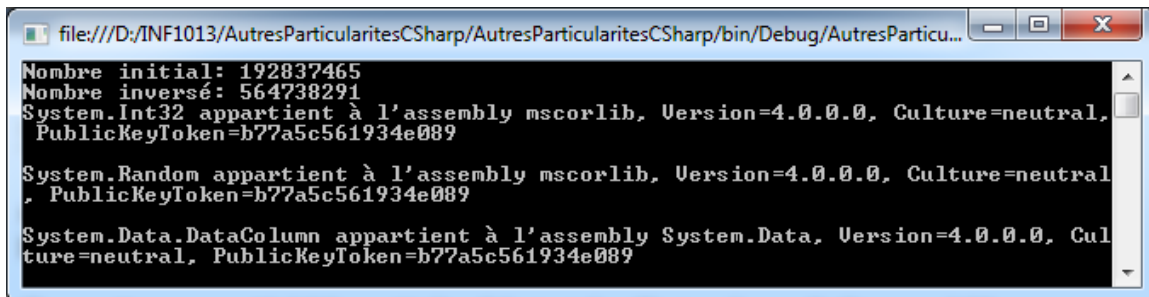
    Random rnd = new Random();
    rnd.AfficherAssembly();

    Console.WriteLine();

    System.Data.DataColumn col = new System.Data.DataColumn();
    col.AfficherAssembly();

    Console.ReadLine();
}
```

Ce qui donnera, en sortie :



6.7.3 Invocation de façon statique

Comme vous l'avez vu à la fin de la section 7.4.1, le « fameux » premier paramètre n'apparaît plus. En réalité, bien que cela soit plus intuitif d'invoquer à un niveau d'instance tel que vu en 7.4.2, ce qui se passe réellement sous le capot est une invocation standard de la méthode statique telle que définie.

Le code précédent est tout à fait équivalent (et permis) que le code suivant :


```
static void Main(string[] args)
{
    int nombre = 192837465;

    Console.WriteLine("Nombre initial: " + nombre);
    Console.WriteLine("Nombre inversé: " +
        MesExtensions.InverserChiffres(nombre));
    MesExtensions.AfficherAssembly(nombre);

    Console.WriteLine();

    Random rnd = new Random();
    MesExtensions.AfficherAssembly(rnd);

    Console.WriteLine();

    System.Data.DataColumn col = new System.Data.DataColumn();
    MesExtensions.AfficherAssembly(col);

    Console.ReadLine();
}
```

6.7.4 Portée d'une méthode d'extension

Les méthodes d'extension n'étant que des méthodes statiques pouvant être invoquées à partir d'une instance du type étendue, il est important de comprendre que vous n'avez pas un accès direct aux membres privées ou protégés de cette classe, mais seulement à ce qui est publique.

Ainsi, si on considère la classe `Robot` suivante :

```
public class Robot
{
    private int vitesse;

    public int Vitesse { get { return this.vitesse; }
        set{ this.vitesse = value; }
    }
}
```

On ne pourrait avoir accès au champ privé `vitesse`, mais seulement à la propriété `Vitesse`. Par exemple :

```
public static void Accelerer(this Robot robot)
{
    robot.vitesse++; //erreur de compilation
    robot.Vitesse++; //correct!
}
```

6.7.5 Utilisation de bibliothèques d'extension

Une des possibilités intéressante avec les méthodes d'extension est de créer ses propres bibliothèques DLL en utilisant des espaces de nom particuliers composées uniquement de méthodes d'extension qu'on importe au besoin dans les projets qui le nécessitent.

Microsoft recommande à cet effet de placer les méthodes d'extension en lien avec des types spécifiques dans des *assemblies* (et des espace de noms) dédiés, de façon à :

- Importer seulement les méthodes d'extension nécessaires;
- Ne pas encombrer inutilement les boîtes de choix proposées dynamiquement grâce à l'intelliSense lorsque vous programmez.

Dans l'exemple précédent, on créerait alors deux *assemblies* : un pour les extensions du type `Object` (incluant la méthode `AfficherAssembly()`) et un autre pour les extensions à `int` (incluant la méthode `InverserChiffres()`).

6.7.6 Extension d'interfaces

Il est également possible d'étendre les fonctionnalités d'une interface.

Supposons que nous avons une interface de base pour des fonctions mathématiques telle que ceci :

```
public interface IMath
{
    int Additionner(int x, int y);
}
```

Comme vous devez le savoir, toute classe implémentant cette interface se devrait de définir la méthode `Additionner()` :

```
public class Mathematiques : IMath
{
    public int Additionner(int x, int y)
    {
        return x + y;
    }
}
```

Maintenant, si vous désirez étendre l'interface en lui ajoutant une méthode comme `Soustraire()`, vous ne pouvez pas seulement donner la signature de la méthode comme ceci :

```
public static int Soustraire(this IMath math, int x, int y);
```

Plutôt, vous devez immédiatement donner le corps de la méthode, comme toute autre méthode d'extension. Cela est compréhensible du fait que :

- Comme nous l'avons dit, en réalité, c'est la méthode statique qui est véritablement appelée, avec l'instance comme paramètre, même si dans l'écriture du code, pour la compréhensibilité on permet de rendre cela transparent;
- Il n'y aurait aucun moyen de s'assurer que tous les types (classes) du projet qui implémentent l'interface en question puissent donner une définition adéquate de cette nouvelle fonctionnalité (cela serait plutôt étrange!!!).

Ainsi, vous devriez écrire quelque chose comme :

```
public static int Soustraire(this IMath math, int x, int y)
{
    return x - y;
}
```

Suite à cela, vous avez alors accès à cette nouvelle méthode :

