

## CHAPITRE 4

# HÉRITAGE ET POLYMORPHISME

### 4.1 HÉRITAGE

Dans le but de faciliter la réutilisation, l'héritage est un des piliers centraux de la programmation orientée objet.

#### 4.1.1 Syntaxe générale

Pour établir une relation d'héritage de type « est-un/e », il faut utiliser « : nomClasseBase ». Gardez en tête que ce que vous souhaitez faire avec l'héritage est de créer des classes générales avec des champs et des fonctionnalités générales, puis avoir des classes dérivées de plus en plus « spécialisées » par les données et services supplémentaires qu'elles comportent.

Imaginons un contexte de jeu de hockey pour lequel on doit définir les joueurs d'une équipe. Une approche pourrait consister à d'abord avoir une classe de base qui définit les aspects (données et services) communs à tout joueur, peu importe sa position. Par exemple :

```
public enum JoueurStatut
{
    PretAJouer,
    Blesse,
    Suspendu,
    Retraite
}

public class JoueurHockey
{
    public string Nom { get; private set; } // le private set fait en sorte
    qu'on peut changer mais seulement à l'intérieur de la classe
    public JoueurStatut Statut { get; set; }
    public int Constitution { get; set; }
    private int numero;
    public int Numero
    {
        get { return this.numero; }
        set { this.numero = value > 99 || this.numero < 0 ? 0 : value; } // on
        s'assure d'avoir un # de chandail valide
    }
}
```

```
    }

    public JoueurHockey(string nom, int numero, JoueurStatut statut =
JoueurStatut.PretAJouer)
    {
        this.Nom = nom;
        this.Numero = numero; //remarquez qu'on utilise le mutateur de la
propriété et non le champ
        this.Statut = statut;
    }

    public void SEntrainer()
    {
        if (this.Statut == JoueurStatut.PretAJouer || this.Statut ==
JoueurStatut.Suspendu)
        {
            Console.WriteLine(Nom + " s'entraîne.");

            if (new Random().Next(1, 100) > this.Constitution)
            {
                Console.WriteLine("Il est plus en forme à la fin de cet
entraînement.");
                this.Constitution++;
            }

            VerifierSiBlessure();
        }
        else
            Console.WriteLine(Nom + " ne peut pas s'entraîner. Il est soit
blessé ou à la retraite.");
    }

    private void VerifierSiBlessure()
    {
        if (new Random().Next(1, 100) < this.Constitution)
        {
            Console.WriteLine(this.Nom + " se blesse à l'entraînement");
            this.Statut = JoueurStatut.Blesse;
        }
    }
}
```

On se retrouve donc avec un joueur de hockey qui a un numéro de chandail, un nom, une constitution et un état qui stipule s'il est apte à jouer, blessé, suspendu ou à la retraite. Il peut aussi s'entraîner.

Maintenant, il existe deux types de joueurs : les gardiens de but et les joueurs d'avant, dont les caractéristiques sont bien différentes. On pourrait les définir à tour de rôle comme suit :

```
public enum GardienStyle
{
    Debout,
    Papillon,
    ToesUp
}

public class Gardien : JoueurHockey
{
    public int HabileteGant { get; private set; }
    public int HabileteBaton { get; private set; }
    public int HabileteFiveHole { get; private set; }
    public int Recuperation { get; private set; }
    public GardienStyle Style { get; private set; }

    public Gardien(string nom, int numero, int habileteGant, int
    habileteBaton, int habileteFiveHole, int recuperation, GardienStyle style
    = GardienStyle.Papillon, JoueurStatut statut = JoueurStatut.PretAJouer)
    {
        this.Nom = nom;
        this.Numero = numero;
        this.HabileteGant = habileteGant;
        this.HabileteBaton = habileteBaton;
        this.HabileteFiveHole = habileteFiveHole;
        this.Recuperation = recuperation;
        this.Style = style;
        this.Statut = statut;
    }

    public void PratiquerGant()
    {
        if (new Random().Next(1, 100) > this.HabileteGant)
        {
            this.HabileteGant++;
            Console.WriteLine(this.Nom + " a amélioré sa technique du côté
            gant suite à une pratique.");
        }
        else
            Console.WriteLine(this.Nom + " s'est pratiqué du côté du gant,
            sans gain technique.");

        VerifierSiBlessure();
    }

    public void PratiquerBaton()
    {
        if (new Random().Next(1, 100) > this.HabileteBaton)
        {
            this.HabileteBaton++;
            Console.WriteLine(this.Nom + " a amélioré sa technique du côté
            bâton suite à une pratique.");
        }
        else
            Console.WriteLine(this.Nom + " s'est pratiqué du côté du bâton,
            sans gain technique.");
    }
}
```

```

        VerifierSiBlessure();
    }

    public void PratiquerFiveHole()
    {
        if (new Random().Next(1, 100) > this.HabileteFiveHole)
        {
            this.HabileteFiveHole++;
            Console.WriteLine(this.Nom + " a amélioré sa technique entre les  
deux jambières suite à une pratique.");
        }
        else
            Console.WriteLine(this.Nom + " s'est pratiqué entre les deux  
jambières, sans gain technique.");

        VerifierSiBlessure();
    }
}

```

Et pour Avant :

```

public class Avant : JoueurHockey
{
    public int Vitesse { get; private set; }
    public int Endurance { get; private set; }
    public int Robustesse { get; private set; }
    public int Maniement { get; private set; }
    public int Lancer { get; private set; }

    public Avant(string nom, int numero, int vitesse, int endurance, int  
robustesse, int maniement, int lancer, JoueurStatut statut =  
JoueurStatut.PretAJouer)
    {
        this.Nom = nom;
        this.Numero = numero;
        this.Vitesse = vitesse;
        this.Endurance = endurance;
        this.Robustesse = robustesse;
        this.Maniement = maniement;
        this.Lancer = lancer;
        this.Statut = statut;
    }

    public void PratiquerPatin()
    {
        if (new Random().Next(1, 100) > this.Vitesse)
        {
            this.Vitesse++;
            Console.WriteLine(this.Nom + " a amélioré son coup de patin suite  
à une pratique.");
        }
        else
            Console.WriteLine(this.Nom + " a pratiqué son coup de patin, sans  
gain technique.");
    }
}

```

```

        VerifierSiBlessure();
    }

    public void PratiquerLancer()
    {
        if (new Random().Next(1, 100) > this.Lancer)
        {
            this.Lancer++;
            Console.WriteLine(this.Nom + " a amélioré sa technique de lancer suite à une pratique.");
        }
        else
            Console.WriteLine(this.Nom + " a pratiqué son lancer, sans gain technique.");

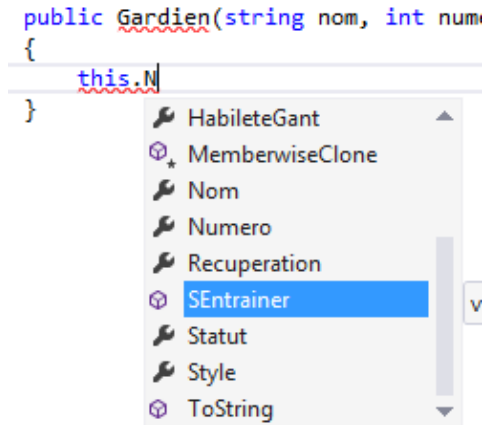
        VerifierSiBlessure();
    }

    public void PratiquerManiementRondelle()
    {
        if (new Random().Next(1, 100) > this.Maniement)
        {
            this.Maniement++;
            Console.WriteLine(this.Nom + " a amélioré sa technique de maniement de rondelle suite à une pratique.");
        }
        else
            Console.WriteLine(this.Nom + " a pratiqué sa technique de maniement de rondelle , sans gain technique.");

        VerifierSiBlessure();
    }
}

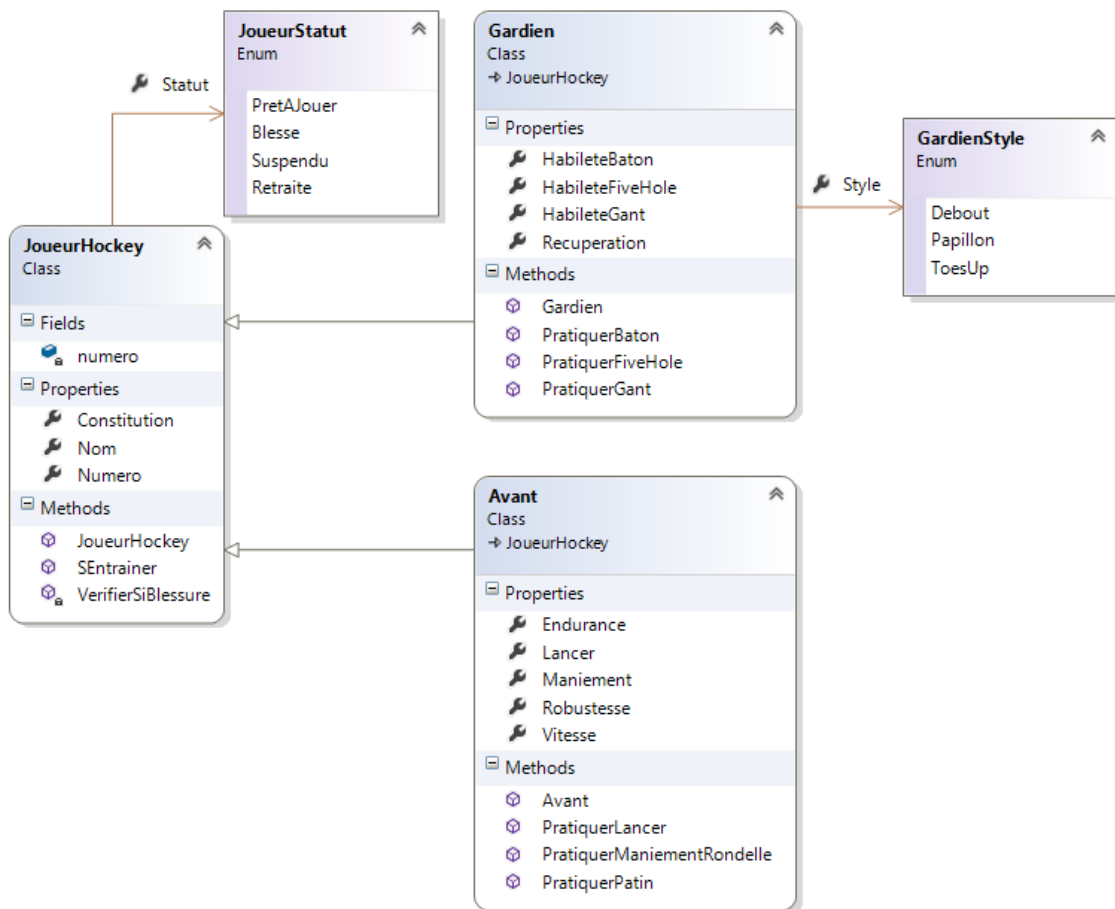
```

Remarquez l'utilisation des deux points et du nom de la classe de base ( : JoueurHockey). De ce fait, on voudra faire en sorte que les classes dérivées Gardien et Avant héritent des champs nom, numero, constitution et statut, ainsi que de la méthode SEntraîner() de la classe JoueurHockey. En effet, vous avez sans doute remarqué en codant les classes que l'Intelli-sense de .NET vous offre l'accès aux champs et aux méthodes publiques de la classe de base :



Puis, chaque classe dérivée a ses propres champs et méthodes supplémentaires.

On se retrouve avec le diagramme de classe suivant :

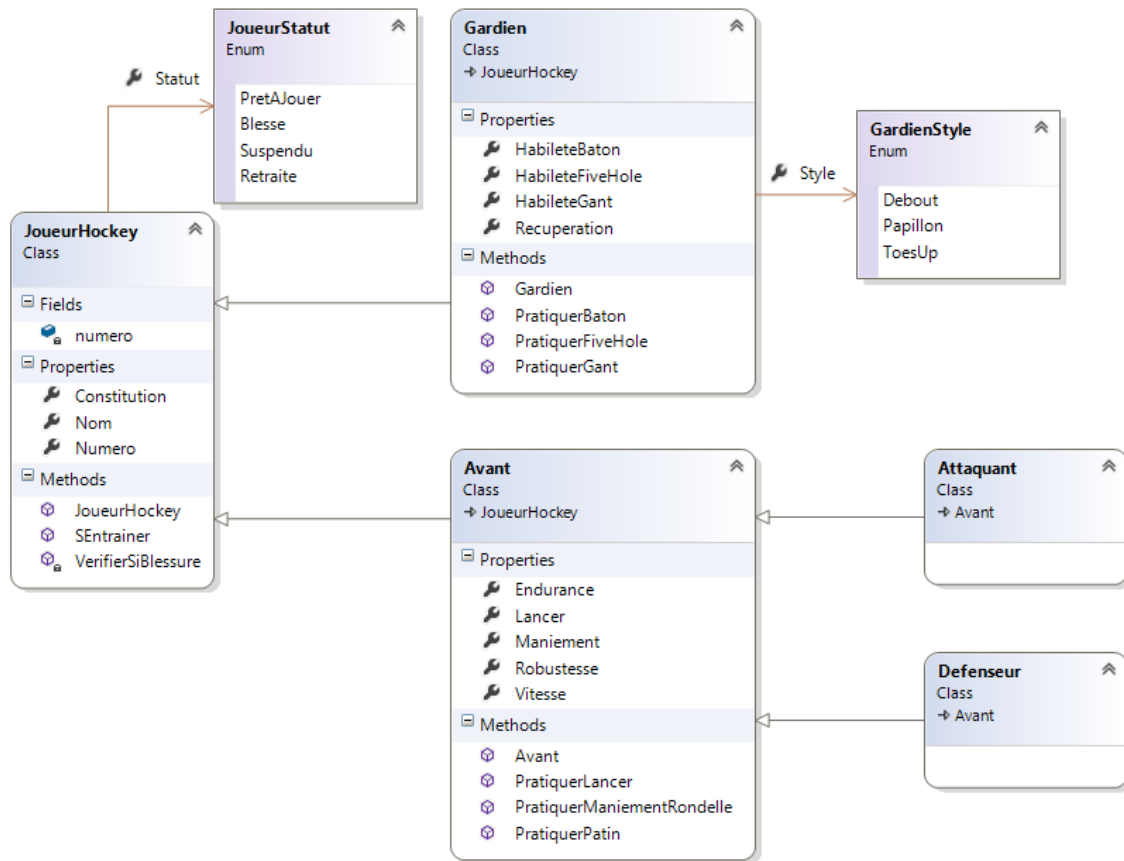


On pourrait aussi considérer qu'un joueur d'avant peut être un attaquant ou un défenseur. Selon notre vision des choses, il est possible qu'on souhaite créer une spécialisation supplémentaire, comme ceci :

```
public class Attaquant : Avant
{
    //...
}

public class Defenseur : Avant
{
    //...
}
```

On aurait alors le diagramme de classes suivant :



Ou encore, si on détermine que dans notre monde simulé il n'y a aucune distinction supplémentaire entre les deux catégories d'avant (ils ont les mêmes caractéristiques et peuvent faire les mêmes actions), on peut plutôt introduire un champ dans **Avant** qui

permet d'identifier s'il s'agit d'un attaquant ou d'un défenseur, par exemple sous forme d'énumération :

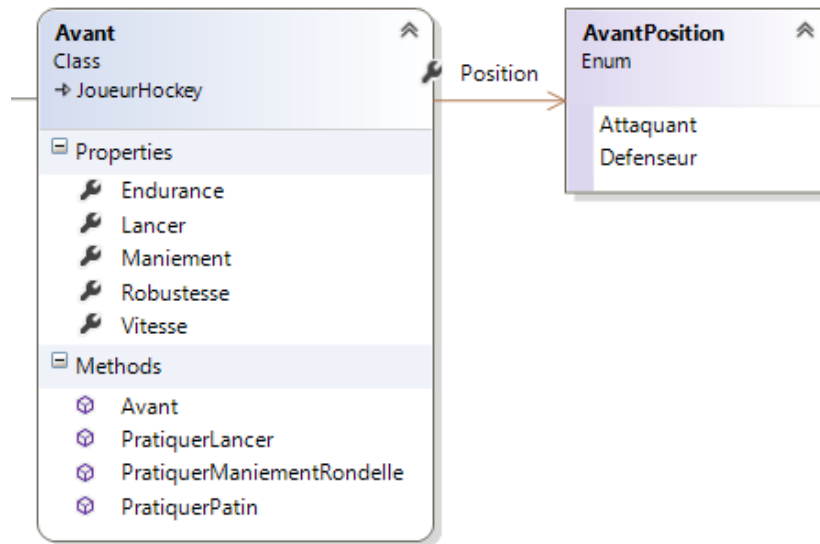
```
public enum AvantPosition
{
    Attaquant,
    Defenseur
}

public class Avant : JoueurHockey
{
    public int Vitesse { get; private set; }
    public int Endurance { get; private set; }
    public int Robustesse { get; private set; }
    public int Maniement { get; private set; }
    public int Lancer { get; private set; }
    public AvantPosition Position { get; private set; }

    public Avant(string nom, int numero, int vitesse, int endurance, int
robustesse, int maniement, int lancer, AvantPosition position,
JoueurStatut statut = JoueurStatut.PretAJouer)
    {
        this.Nom = nom;
        this.Numero = numero;
        this.Vitesse = vitesse;
        this.Endurance = endurance;
        this.Robustesse = robustesse;
        this.Maniement = maniement;
        this.Lancer = lancer;
        this.Position = position;
        this.Statut = statut;
    }
}
```

Le diagramme de classes s'en retrouve changé :





**Il faut donc utiliser l'héritage avec parcimonie**, lorsque vraiment la spécialisation supplémentaire amène à devoir rendre des services particuliers et/ou à stocker des données supplémentaires. Sinon, il se peut qu'une énumération ou une structure de donnée soit suffisante.

Ainsi, grâce à l'héritage, on a un direct accès public aux champs et méthodes de la classe mère sans avoir eu à répéter le code en commun entre Gardien et Avant. C'est évidemment là toute la puissance en ce qui concerne la réutilisation de code.

On peut alors utiliser des objets Gardien et Avant conséquemment :

```
class Program
{
    static void Main(string[] args)
    {
        Avant joueurA = new Avant("Pinocchio", 41, 45, 39, 22, 45, 72,
        AvantPosition.Attaquant);
        Avant joueurB = new Avant("Figaro", 63, 29, 43, 12, 45, 34,
        AvantPosition.Defenseur, JoueurStatut.Suspendu);
        Gardien joueurC = new Gardien("Oswald", 11, 39, 43, 44, 45, statut:
        JoueurStatut.PretAJouer);

        joueurA.PratiquerLancer();
        joueurA.SEntrainer();

        joueurB.PratiquerManiementRondelle();
        joueurB.PratiquerPatin();

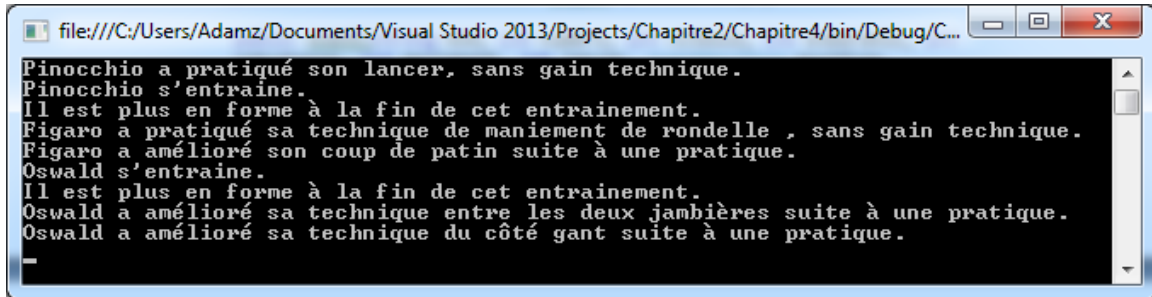
        joueurC.SEntrainer();
        joueurC.PratiquerFiveHole();
        joueurC.PratiquerGant();
    }
}
```

```

        Console.ReadLine();
    }
}

```

Ce qui donnerait, suite à l'exécution :



```

file:///C:/Users/Adamz/Documents/Visual Studio 2013/Projects/Chapitre2/Chapitre4/bin/Debug/C...
Pinocchio a pratiqué son lancer, sans gain technique.
Pinocchio s'entraîne.
Il est plus en forme à la fin de cet entraînement.
Figaro a pratiqué sa technique de maniement de rondelle, sans gain technique.
Figaro a amélioré son coup de patin suite à une pratique.
Oswald s'entraîne.
Il est plus en forme à la fin de cet entraînement.
Oswald a amélioré sa technique entre les deux jambières suite à une pratique.
Oswald a amélioré sa technique du côté gant suite à une pratique.

```

Notez toutefois que si on a accès aux champs public, on n'a cependant pas accès aux champs ou méthodes privées de la classe de base, s'il y en a. Dans notre exemple, on aurait accès à la propriété `Numero`, mais pas au champ `numero`. Il en va de même pour la propriété `Nom` à partir de `Gardien` ou `Avant`, puisque le mutateur est privé, ainsi que pour méthode privée `VerifierSiBlessure()` :

```

public void PratiquerPatin()
{
    if (new Random().Next(1, 100) > this.Vitesse)
    {
        this.Vitesse++;
        Console.WriteLine(this.Nom + " a amélioré son coup de patin suite à une pratique.");
    }
    else
        Console.WriteLine(this.Nom + " a pratiqué son coup de patin, sans gain technique.");

    VerifierSiBlessure();
}

```

Error:  
'Chapitre4.JoueurHockey.VerifierSiBlessure()' is inaccessible due to its protection level

```

public Avant(string nom, int numero, int vitesse, int endurance, int robustesse, int maniement, int lance
{
    this.Nom = nom;
    string JoueurHockey.Nom

    Error:
    The property or indexer 'Chapitre4.JoueurHockey.Nom' cannot be used in this context because the set accessor is inaccessible

    this.Lancer = lancer;
}

```

```
public Avant(string nom, int numero, int vitesse, int endurance, int r
{
    this.Nom = nom;
    this.Numero = numero; //okay
    this.numero = numero; //ne fonctionne pas
    this.
    this.
    this.
    this.
    this.
    this.
    this.
    Error:
    'Chapitre4.JoueurHockey.numero' is inaccessible due to its protection level
    this.
```

### 4.1.2 Héritage multiple

En C#, une classe donnée ne peut hériter directement que d'une seule et unique classe de base (*c'est possible en C++ non géré/pré .NET*).

Ce n'est donc pas possible, par exemple, d'avoir ce genre de syntaxe :

```
class MaClasseParent { }

class MaClasseParentBis { }

class MaSousClasse : MaClasseParent, MaClasseParentBis
```

Toutefois, comme nous le verrons à un chapitre ultérieur, il est possible pour une classe ou une structure d'implémenter plus d'une interface, qui sont des types qui forcent les classes ou structures qui les implémentent à définir certaines fonctionnalités. Cela permet une solution hybride entre l'héritage simple et la complexité liée à l'héritage multiple. L'interface en soi peut pour sa part dériver de plusieurs interfaces.

### 4.1.3 Base

Présentement, chacune des classes `JoueurHockey`, `Gardien` et `Avant` ont leur propre constructeur, avec des redondances.

Un peu comme ce fut le cas lorsqu'on avait de multiples constructeurs pour une seule classe dans le chapitre précédent, vous avez la possibilité en C# de faire appel à des constructeurs de classes parents à l'aide du mot clé `base`.

Non seulement cela, mais dans notre cas, vous aurez peut-être remarqué que les classes dérivées ne compilent pas pour l'instant à cause d'une erreur de compilation pour les constructeurs :

```
public Avant(string nom, int numero, int vitesse, int endurance, int ro
{
    thi 'Chapitre4.JoueurHockey' does not contain a constructor that takes 0 arguments

public Gardien(string nom, int numero, int habileteGant, int habileteB
{
    thi 'Chapitre4.JoueurHockey' does not contain a constructor that takes 0 arguments
```

### Que se passe-t-il?

En fait, puisqu'on a défini un constructeur personnalisé dans `JoueurHockey`, le compilateur considère que toute classe dérivée doit faire appel à au moins un d'entre eux afin d'instancier adéquatement l'objet (ce qui est le rôle d'un constructeur).

En d'autres termes, on considère que si on a pris la peine de définir un constructeur à la classe de base, c'est qu'il contient des instructions essentielles d'instanciation. À défaut d'utiliser un constructeur personnalisé, un constructeur par défaut sera implicitement appelé.

En définitive, dans notre cas, on n'a donc pas le choix de faire appel au constructeur pré-défini pour la classe-mère. Mais dans tous les cas, on souhaitera aussi ultimement éviter toutes redondances d'affectations de valeurs initiales en utilisant le constructeur de la classe de base pour les champs en communs, et en utilisant le constructeur spécialisé pour les champs de la spécialisation, le tout dans une syntaxe analogue au `this` :

```
public Gardien(string nom, int numero, int habileteGant, int
habileteBaton, int habileteFiveHole, int recuperation, GardienStyle style
= GardienStyle.Papillon, JoueurStatut statut = JoueurStatut.PretAJouer)
: base(nom, numero, statut)
{
    this.HabileteGant = habileteGant;
    this.HabileteBaton = habileteBaton;
    this.HabileteFiveHole = habileteFiveHole;
    this.Recuperation = recuperation;
    this.Style = style;
}
```

Et

```
public Avant(string nom, int numero, int vitesse, int endurance, int
robustesse, int maniemment, int lancer, AvantPosition position,
JoueurStatut statut = JoueurStatut.PretAJouer)
: base(nom, numero, statut)
{
    this.Vitesse = vitesse;
    this.Endurance = endurance;
    this.Robustesse = robustesse;
```

```
        this.Maniement = maniemment;  
        this.Lancer = lancer;  
        this.Position = position;  
    }
```

Sachez que ce mot clé n'est pas limité à cette unique utilisation. **Il peut être employé pour accéder à des membres publics ou protégés des classes parents.**

Par exemple, à partir de `PratiquerBaton()` de la classe `Gardien`, on pourrait invoquer la méthode `SEntrainer()` de la classe mère `JoueurHockey`:

```
public void PratiquerBaton()  
{  
    base.SEntrainer();  
  
    if (new Random().Next(1, 100) > this.HabileteBaton)  
    {  
        this.HabileteBaton++;  
        Console.WriteLine(this.Nom + " a amélioré sa technique du côté  
        bâton suite à une pratique.");  
    }  
    else  
        Console.WriteLine(this.Nom + " s'est pratiqué du côté du bâton,  
        sans gain technique.");  
  
    VerifierSiBlessure();  
}
```

Notez que comme `this`, `base` est souvent implicite et la plupart du temps utilisé pour parer les ambiguïtés possibles ou pour augmenter la compréhensivité du code.

### 4.1.4 Protected

Les membres déclarés comme étant publics sont accessibles de n'importe où, alors qu les membres privés ne sont accessibles que de l'intérieur de la classe auxquels ils appartiennent.

C'est d'ailleurs un des problèmes identifiés : on n'a pas accès à la variable `numero` de la classe `JoueurHockey` à partir d'une de ses classes dérivées (`Gardien` ou `Avant`), ni à la méthode `VerifierSiBlessure()`, comme nous l'avons mentionné plus tôt.

Afin de protéger l'accès des classes extérieures tout en permettant l'accès aux classes dérivées, vous pouvez utiliser le mot clé **protected**, qui peut être utilisé tant devant un champ que devant une méthode. Par exemple :

```
public class JoueurHockey
{
    (...)

    protected int numero;

    (...)

    protected void VerifierSiBlessure()
    {
        (...)
    }
}
```

On a maintenant accès à `numero` et à `VerifierSiBlessure()` du côté de la classe `Gardien` et `Avant`.

On peut également définir la propriété `Nom` en utilisant ceci, si on souhaite pouvoir modifier le nom à partir des classes dérivées :

```
public string Nom { get; protected set; }
```

Ce mot clé est donc fort pratique en ce qui a trait à l'héritage. Il faut seulement faire attention car il présente un léger inconvénient : puisque tout programmeur peut créer une classe dérivée d'une classe d'une librairie existante (à moins que celle-ci soit scellée, bien évidemment), il peut donc accidentellement ou volontairement modifier/violier des règles de validation de données établies par la classe parent pour ce champ à travers une propriété, par exemple. La prudence est donc de mise.

Et comme vous vous en doutez, de l'extérieur du contexte de la classe et ses classes dérivés, les membres `protected` restent inaccessible :

```
static void Main(string[] args)
{
    JoueurHockey j = new JoueurHockey("Pinocchio", 99, JoueurStatut.Blesse);
    j.VerifierSiBlessure();
}

void JoueurHockey.VerifierSiBlessure()
```

Error:  
'Chapitre4.JoueurHockey.VerifierSiBlessure()' is inaccessible due to its protection level

### 4.1.5 Sealed

En C#, vous avez la possibilité d'utiliser le mot clé `sealed` de façon à spécifier qu'une classe ne peut plus être dérivée (elle ne peut donc pas être utilisée comme classe de base d'aucune autre classe). Elle devient en d'autres mots une « *feuille* » dans l'arborescence de vos classes.

Par exemple, vous pourriez vous assurer qu'il n'existe pas de spécialisation plus spécifique à Gardien et Avant comme ceci :

```
public sealed class Gardien : JoueurHockey {...}
```

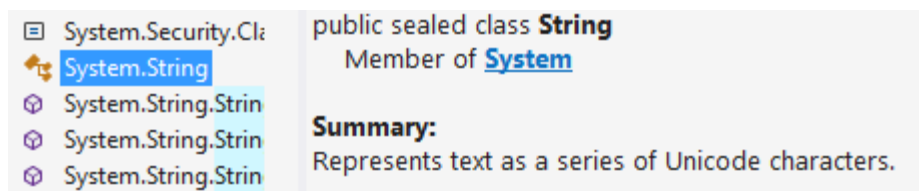
Et :

```
public sealed class Avant : JoueurHockey {...}
```

De cette façon, on ne pourrait plus spécialiser en ajoutant des sous-classes Attaquant et Defenseur comme on aurait pu vouloir le faire plus tôt, sans déclencher une erreur de compilation :

```
public class Attaquant : Avant
{
    'Chapitre4.Attaquant': cannot derive from sealed type 'Chapitre4.Avant'
}
```

Une catégorie de classes qui sont souvent définies comme étant scellées d'emblée sont les classes utilitaires. L'espace de noms `System` définit un bon nombre de classes avec l'attribut `sealed`. Vous pouvez observer le tout en consultant l'explorateur d'objet pour la définition de la classe `String`, par exemple :



Vous ne pourriez donc pas créer une classe dérivée de `String` même si vous le souhaitiez :

```
public class SousString : String
{
    'Chapitre4.SousString': cannot derive from sealed type 'string'
}
```

**Note :**

Les structures sont par définition implicitement scellées : il est donc impossible de dériver une structure d'une autre structure, une classe d'une structure ou une structure d'une classe.

### ***4.1.6 Aggrégation, héritage et délégation***

Nous l'avons mentionné à quelques reprises déjà : l'héritage définit une relation de type « est-un ».

Le deuxième type de relation en programmation orientée objet est l'**aggrégation**, qui établie une relation de type « a-un ».

Cette notion est, comme vous le savez sans doute à ce point-ci, liée aux champs de données d'une classe et permettent à un objet d'en contenir d'autres.

Dans le cadre de notre démonstration de ligue de hockey, on pourrait avoir un nouveau champ dans `JoueurHockey` de type `Contrat` :

```
public class JoueurHockey
{
    public Contrat ContratActuel { get; set; }

    (...)

    public JoueurHockey(Contrat contrat, string nom, int numero, JoueurStatut
statut = JoueurStatut.PretAJouer)
    {
        this.ContratActuel = contrat;
        this.Nom = nom;
        this.Numero = numero;
        this.Statut = statut;
    }

    (...)
}
```

Qui nécessiterait aussi de modifier les signatures des constructeurs des classes `Avant` et `Gardien` comme suit :



```
public Avant(Contrat contrat, string nom, int numero, int vitesse, int
endurance, int robustesse, int maniemment, int lancer, AvantPosition
position, JoueurStatut statut = JoueurStatut.PretAJouer)
    : base(contrat, nom, numero, statut)
```

Et :

```
public Gardien(Contrat contrat, string nom, int numero, int habileteGant,
int habileteBaton, int habileteFiveHole, int recuperation, GardienStyle
style = GardienStyle.Papillon, JoueurStatut statut =
JoueurStatut.PretAJouer)
    : base(contrat, nom, numero, statut)
```

Ce contrat pourrait être défini comme étant une structure qui indique un salaire annuel de base, une durée du contrat et le nom de l'agent négociateur. Contrat possède aussi une propriété Paie qui retourne le montant de la paie (aux deux semaines) liée au contrat :

```
public struct Contrat
{
    public double SalaireBase { get; private set; }
    public int Duree { get; private set; }
    public string Agent { get; private set; }

    public double Paie
    {
        get
        {
            return this.SalaireBase / 27;
        }
    }

    public Contrat(double salaireBase, int duree, string agent) : this()
    {
        this.SalaireBase = salaireBase;
        this.Duree = duree;
        this.Agent = agent;
    }
}
```

Nous avons donc une relation « un joueur de hockey *a-un* contrat », et donc un objet Contrat contenu dans un objet JoueurHockey.

Le principe de la **délégation** consiste alors à exposer les fonctionnalités d'un objet contenu via une méthode ajoutée à la classe conteneur, sans exposer l'objet contenu en tant que tel.

Dans notre exemple, cela revient à ajouter une propriété `Paie` (garder le même nom importe peu, mais cela permet de rester plus consistant) à la classe `JoueurHockey` qui appelle la propriété `Paie` de son objet `Contrat` :

```
public double Paie
{
    get
    {
        return this.ContratActuel.Paie;
    }
}
```

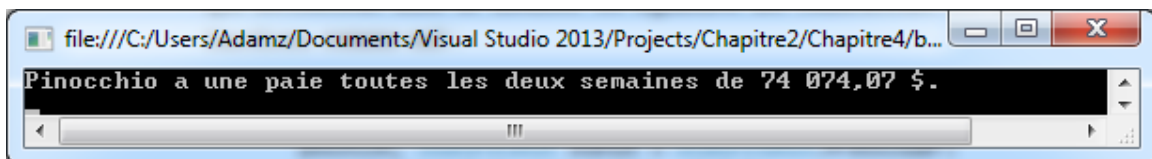
Le code suivant montre une utilisation de cette méthode de délégation :

```
static void Main(string[] args)
{
    Avant joueurA = new Avant(new Contrat(2000000, 4, "Tobby McGuire"),
    "Pinocchio", 41, 45, 39, 22, 45, 72, AvantPosition.Attaquant);

    Console.WriteLine("{0} a une paie toutes les deux semaines de {1:c}.",
    joueurA.Nom, joueurA.Paie);

    Console.ReadLine();
}
```

Et son exécution :



### 4.1.7 Types imbriqués

En C#, il est aussi possible de définir des types (enum, class, struct, interface ou delegate) directement à l'intérieur d'une classe ou d'une structure (au même niveau que les champs et les méthodes, donc).

C'est ce qu'on appelle en anglais des « *nested type* », ou des types imbriqués.

La syntaxe générale est la suivante :

```
public class MaClasseExterne
```

```
{  
    public class MaClasseImbriqueePublique {}  
    private class MaClasseImbriqueePrivee {}  
}
```

Voici quelques caractéristiques à retenir concernant les types imbriqués :

- Les types imbriqués vous permettent d’obtenir un contrôle total sur le niveau d’accès du type interne, du fait qu’**ils peuvent être déclarés comme étant privés** (ce qui est impossible pour une classe non imbriquée, rappelez-vous);
- Puisque le type imbriqué devient en quelque sorte un membre de la classe externe dans laquelle elle est définie, **ce dernier a accès aux membres privés de cette classe externe**;
- Généralement, on utilise les types imbriqués en tant qu’aidant pour la classe externe, c’est-à-dire qu’on aura un type imbriqué privé qui ne sera utilisé que pour supporter un ou plusieurs fonctionnalités (fins de calculs, structure de données temporaires) de la classe externe.

Pour déclarer des variables pour un type imbriqué, il faut évidemment que la portée soit adéquate.

Ainsi, si le type imbriqué est d’accès public, cela ne cause pas de soucis :

```
MaClasseExterne.MaClasseImbriqueePublique A = new  
MaClasseExterne.MaClasseImbriqueePublique();
```

Par contre, si le type imbriqué est privé, comme il est considéré comme un membre, on n’y a pas accès de l’extérieur :

```
MaClasseExterne.MaClasseImbriqueePrivee A = new MaClasseExterne.MaClasseImbriqueePrivee();  
Console.ReadLine();
```

MaClasseExterne.MaClasseImbriqueePrivee.MaClasseImbriqueePrivee()  
Error:  
'Chapitre4.MaClasseExterne.MaClasseImbriqueePrivee' is inaccessible due to its protection level

Dans notre exemple, si on considère que les contrats ne sont utilisés qu’à l’intérieur du cadre des joueurs de hockey (ceci dit, cela ne veut pas dire que ce serait l’idéal dans notre exemple; ce n’est que pour démontrer la chose), alors on pourrait avoir ceci :

```
public class JoueurHockey  
{  
    public struct Contrat  
    {  
        public double SalaireBase { get; private set; }  
        public int Duree { get; private set; }  
    }  
}
```

```

        public string Agent { get; private set; }
        public double Paie
        {
            get
            {
                return this.SalaireBase / 27;
            }
        }

        public Contrat(double salaireBase, int duree, string agent)
            : this()
        {
            this.SalaireBase = salaireBase;
            this.Duree = duree;
            this.Agent = agent;
        }
    }

    public Contrat ContratActuel { get; set; }
    public string Nom { get; protected set; }
    public JoueurStatut Statut { get; set; }
    public int Constitution { get; set; }
    protected int numero;
    public int Numero
    {
        get { return this.numero; }
        set { this.numero = value > 99 || this.numero < 0 ? 0 : value; }
    }

    (...)

    public JoueurHockey(Contrat contrat, string nom, int numero, JoueurStatut
    statut = JoueurStatut.PretAJouer)
    {
        (...)
    }

    (...)
}

```

Enfin, sachez que vous pouvez imbriquer à souhайте pour de multiples niveaux de profondeur, si cela fait du sens dans votre contexte. Par exemple, ceci est valide :

```

public class MaClasseExterne
{
    public class MaClasseImbriqueePublique
    {
        public class MaClasseEncorePlusImbriquee { }
    }
}

```

Et l'appel du côté `Main()` pourrait être le suivant;

```
MaClasseExterne.MaClasseImbriqueePublique.MaClasseEncorePlusImbriquee = new
MaClasseExterne.MaClasseImbriqueePublique.MaClasseEncorePlusImbriquee();
```

## 4.2 POLYMORPHISME

Dans cette section, nous allons explorer le troisième aspect fondamental de la programmation orientée objet : le polymorphisme.

### 4.2.1 Introduction générale

Dans notre classe `JoueurHockey`, nous avons une méthode nommée `SEntrainer()`. Étant donné qu'elle est définie comme étant publique, nous pouvons y accéder à partir de n'importe quel de ses types dérivés.

On l'avait déjà expérimenté dans le jeu d'essai :

```
Avant joueurA = new Avant("Pinocchio", 41, 45, 39, 22, 45, 72,
AvantPosition.Attaquant);
Avant joueurB = new Avant("Figaro", 63, 29, 43, 12, 45, 34,
AvantPosition.Defenseur, JoueurStatut.Suspendu);
Gardien joueurC = new Gardien("Oswald", 11, 39, 43, 44, 45, statut:
JoueurStatut.PretAJouer);

joueurA.PratiquerLancer();
joueurA.SEntrainer();

joueurB.PratiquerManiementRondelle();
joueurB.PratiquerPatin();

joueurC.SEntrainer();
```

Jusqu'ici, la méthode `SEntrainer()` se comporte de la même façon, qu'on l'utilise depuis un objet de classe `JoueurHockey`, `Gardien` ou `Avant`.

Souvent, on voudra se donner la possibilité de modifier partiellement ou totalement ces méthodes fournies par la classe mère.

C'est là une des facettes du polymorphisme : **il fournit une façon pour une sous-classe de définir sa propre version d'une méthode définie par sa classe de base**, en utilisant un procédé nommé « **redéfinition** » ou parfois « substitution » **de méthodes**. C'est le sujet de notre prochaine sous-section.

### 4.2.2 Virtual et redéfinition/substitution de méthodes (override)

La redéfinition de méthodes demande l'utilisation souvent conjointe des mots clés `virtual` et `override` :

- Si une classe de base souhaite définir une méthode qui peut (sans obligation) être redéfinie par une classe dérivée, alors on doit la marquer du mot clé **`virtual`**;
- Si une classe dérivée souhaite se prévaloir de cette possibilité de `changer l'implémentation de la méthode de la classe mère` en la substituant partiellement ou totalement par sa propre implémentation, alors la méthode du même nom chez la classe dérivée doit être accompagnée du mot clé **`override`**.

Ainsi, dans notre exemple, il faudrait ajouter `virtual` à la méthode `SEntainer()` de la classe `JoueurHockey` :

```
public virtual void SEntainer()
{
    (...)
}
```

Puis si on souhaite redéfinir la méthode `SEntainer()` pour les classes `Gardien` et `Avant`, on pourrait avoir quelque chose comme ceci :

```
public sealed class Gardien : JoueurHockey
{
    (...)

    public override void SEntainer()
    {
        base.SEntainer();

        if (new Random().Next(1, 100) > this.Recuperation)
        {
            Console.WriteLine("Vous avez maintenant plus de facilité à  
récupérer votre position devant le filet.");
            this.Recuperation++;
        }
    }
}
```

Et :

```
public sealed class Avant : JoueurHockey
{
    (...)
}
```

```

    public override void SEntrainer()
    {
        base.SEntrainer();

        if (new Random().Next(1, 100) > this.Endurance)
        {
            Console.WriteLine("Vous avez maintenant plus d'endurance grâce à vos efforts à l'entraînement.");
            this.Endurance++;
        }
    }
}

```

Dans les deux cas, on ne fait pas simplement substituer, mais on appelle d'abord la méthode de la classe de base où est définie la méthode `SEntrainer()` originale (dans `JoueurHockey`, évidemment) en utilisant le mot clé `base` et le nom de la méthode :

- Pour le gardien, l'entraînement lui donne en plus la possibilité d'améliorer sa récupération;
- Pour le joueur d'avant, c'est son endurance qui peut devenir meilleure.

À l'exécution, étant donné le jeu d'essai suivant, qui est le même que plus tôt modifié en tenant compte des contrats :

```

static void Main(string[] args)
{
    Avant joueurA = new Avant(new JoueurHockey.Contrat(2000000, 4, "Tobby McGuire"), "Pinocchio", 41, 45, 39, 22, 45, 72, AvantPosition.Attaquant);
    Avant joueurB = new Avant(new JoueurHockey.Contrat(3500000, 2, "Tobby McGuire"), "Figaro", 63, 29, 43, 12, 45, 34, AvantPosition.Defenseur, JoueurStatut.Suspendu);
    Gardien joueurC = new Gardien(new JoueurHockey.Contrat(5750000, 3, "Ken Thomas"), "Oswald", 11, 39, 43, 44, 45, statut: JoueurStatut.PretAJouer);

    joueurA.PratiquerLancer();
    joueurA.SEntrainer();

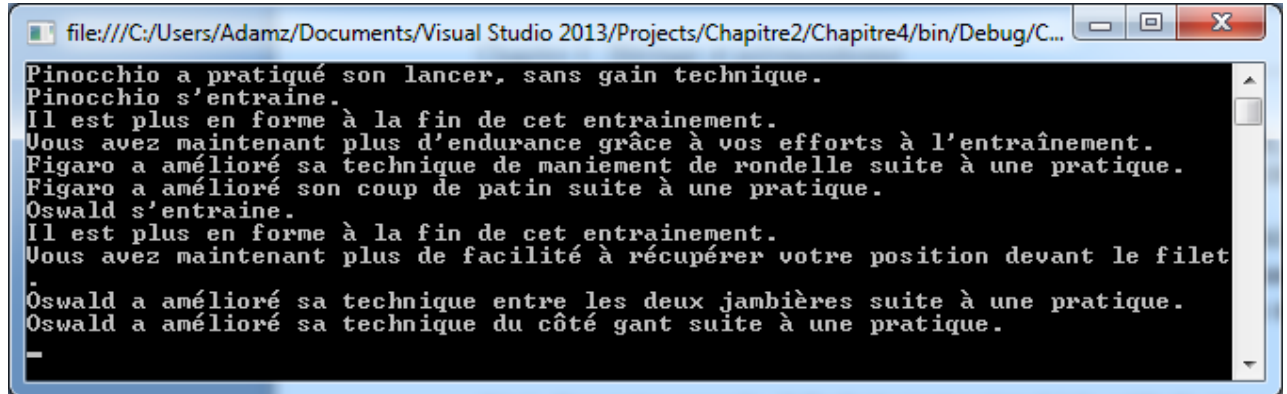
    joueurB.PratiquerManiementRondelle();
    joueurB.PratiquerPatin();

    joueurC.SEntrainer();
    joueurC.PratiquerFiveHole();
    joueurC.PratiquerGant();

    Console.ReadLine();
}

```

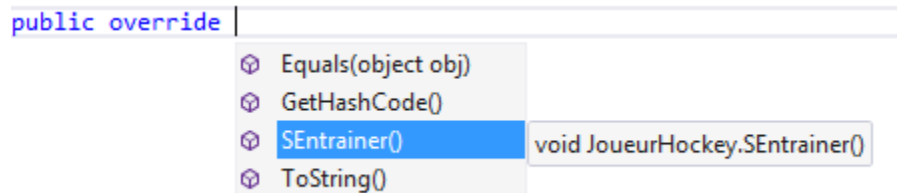
On verra une sortie toutefois un peu différente :



```
file:///C:/Users/Adamz/Documents/Visual Studio 2013/Projects/Chapitre2/Chapitre4/bin/Debug/C...
Pinocchio a pratiqué son lancer, sans gain technique.
Pinocchio s'entraîne.
Il est plus en forme à la fin de cet entraînement.
Vous avez maintenant plus d'endurance grâce à vos efforts à l'entraînement.
Figaro a amélioré sa technique de maniement de rondelle suite à une pratique.
Figaro a amélioré son coup de patin suite à une pratique.
Oswald s'entraîne.
Il est plus en forme à la fin de cet entraînement.
Vous avez maintenant plus de facilité à récupérer votre position devant le filet.
Oswald a amélioré sa technique entre les deux jambières suite à une pratique.
Oswald a amélioré sa technique du côté gant suite à une pratique.
```

L'appel à la méthode de la classe de base du même nom est tout à fait optionnelle, ceci dit.

Remarquez aussi que dans *Visual Studio*, aussitôt après avoir tapé `override`, on vous indique toutes les méthodes de classes de base dans l'hérarchie qui sont substituables :



Et que par défaut on fait directement appel à la définition de la méthode originale :

```
public override void SEntrainer()
{
    base.SEntrainer();
}
```

### 4.2.3 Scellage de membres virtuels

Plus tôt, nous avons utilisé le mot clé `sealed` avec pour objectif de prévenir toute dérivation d'une classe.

**Dans certains cas, il est possible que vous ne souhaitiez pas sceller une classe en entier, mais seulement éviter que certaines fonctionnalités soient redéfinies à nouveau par d'autres sous-types.**

Vous pouvez pour cela utiliser le mot clé `sealed` à une méthode de redéfinition.



Si, par exemple, nous retirons `sealed` comme modificateur de la classe `Avant`, mais que nous ajoutons `sealed` à `SEntainer()`, comme ceci :

```
public class Avant : JoueurHockey
{
    (...)

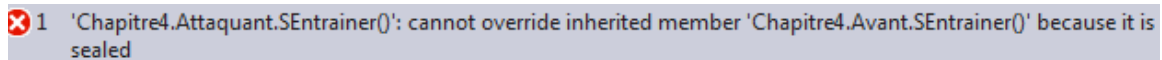
    public override sealed void SEntainer()
    {
        base.SEntainer();

        if (new Random().Next(1, 100) > this.Endurance)
        {
            Console.WriteLine("Vous avez maintenant plus d'endurance grâce à vos efforts à l'entraînement.");
            this.Endurance++;
        }
    }
}
```

Cela pourrait faire du sens, car on pourrait considérer que la méthode `SEntainer()` pour un joueur de l'avant peut altérer la constitution, le statut et l'endurance, mais rien d'autre, peu importe quelle classe spécialisée on pourrait faire dériver d'`Avant`.

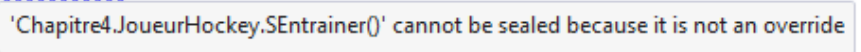
Si par exemple on créait une classe `Attaquant` et qu'on tentait de redéfinir `SEntainer()`, vous obtiendriez une erreur de compilation :

```
public class Attaquant : Avant
{
    public override void SEntainer()
    {
        base.SEntainer();
    }
}
```



Notez que `sealed` ne peut être utilisé **que** conjointement avec le mot clé `override`. Vous ne pourriez pas ajouter `sealed` dans la définition originale dans la classe `JoueurHockey` sans obtenir un message d'erreur :

```
public sealed virtual void SEntainer()
{
    if (this.Statut == Joue
```



Cela va de soi : *pourquoi permettre la redéfinition avec le mot clé **virtual** pour ensuite indiquer que la méthode ne peut être redéfinie?*

#### 4.2.4 Classes abstraites

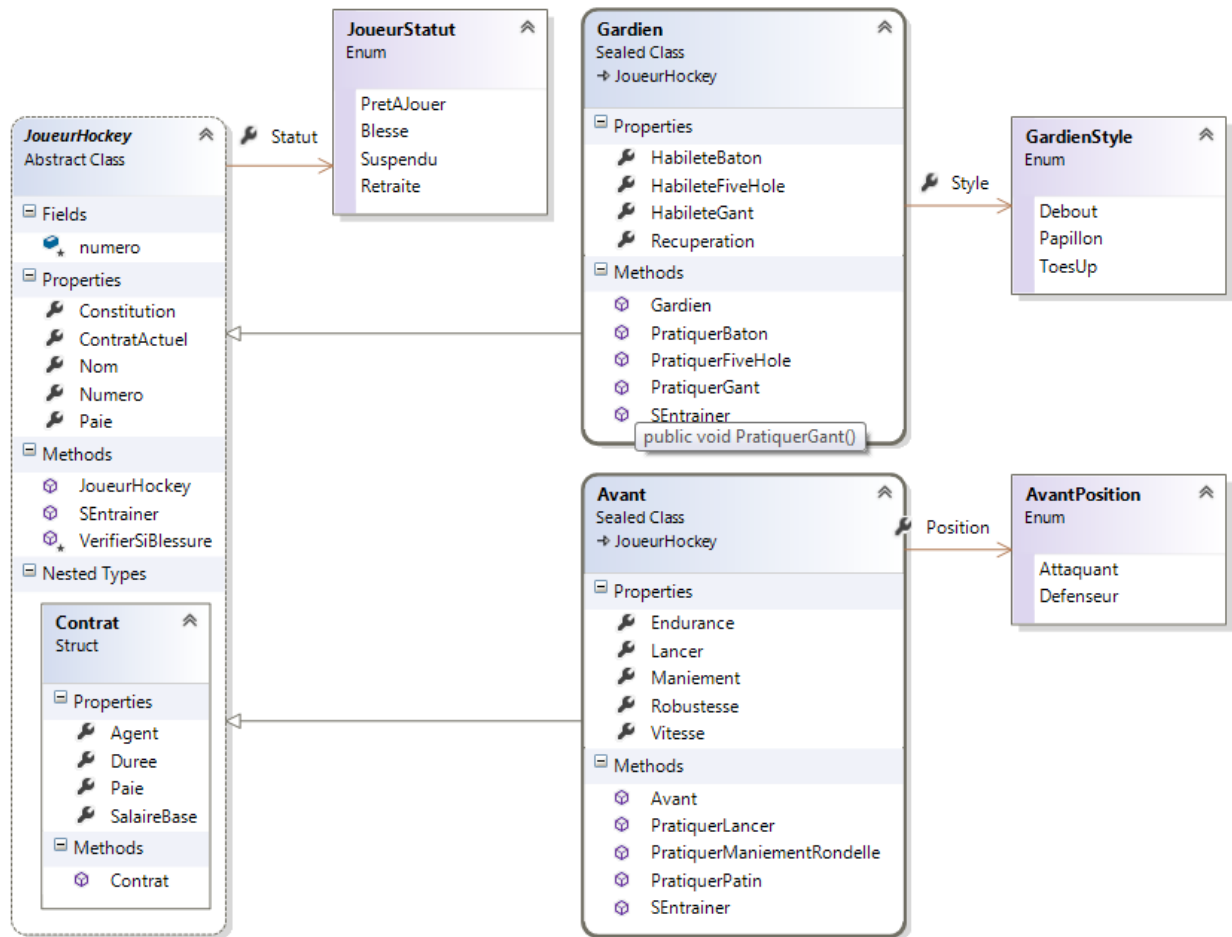
Jusqu'ici, la classe `JoueurHockey` a été utilisée pour fournir certains membres (des champs, des propriétés et des méthodes) pour ses classes descendantes `Gardien` et `Avant`, dont un est redéfinissable (`SEntreainer()`).

On peut s'interroger sur la pertinence de créer des instances de la classe `JoueurHockey` : en effet, il n'y a pas de raison de créer un joueur de hockey qui n'a pas soit ses attributs comme joueur d'avant ou comme gardien de but. On dira à cet effet que la classe `JoueurHockey` est un concept « trop général », ou un concept « **abstrait** ».

Pour définir une classe comme étant abstraite, il suffit d'utiliser le mot clé `abstract` devant le nom de la classe :

```
public abstract class JoueurHockey
```

On a à ce moment-ci le diagramme de classes suivant :



Dès lors, il deviendra impossible d'instancier un objet de classe JoueurHockey :

```

JoueurHockey joueur = new JoueurHockey(new JoueurHockey.Contrat(2000000, 4, "Tobby McGuire")

```

Error:  
Cannot create an instance of the abstract class or interface 'Chapitre4.JoueurHockey'

#### 4.2.4.1 MEMBRES ABSTRAITS

Une fois qu'une classe est définie comme étant abstraite, elle peut également définir des membres abstraits.

**Les membres abstraits peuvent être utilisés à chaque fois que vous souhaitez définir un membre qui ne fournit pas une implémentation par défaut à même la classe abstraite, mais qui doit obligatoirement être défini par toute classe dérivée.**

Ce faisant, les méthodes virtuelles et abstraites forment ce qu'on appelle une **interface polymorphique** qui doit être traitée en tout ou en partie (au minimum les membres abstraits) par les descendants.

### **Note :**

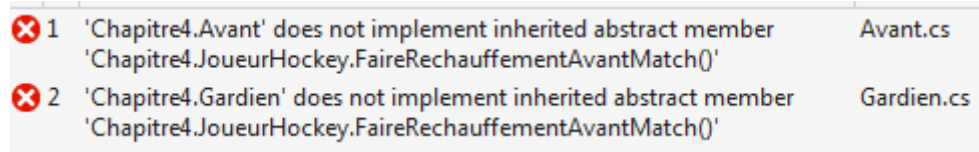
Alors que les méthodes déclarées comme étant virtuelles peuvent facultativement être redéfinies par des descendants, les méthodes abstraites doivent obligatoirement être redéfinies.

Par exemple, on pourrait vouloir forcer Gardien et Avant à implémenter une méthode `FaireRechauffementAvantMatch()`. Plutôt que de définir au niveau de `JoueurHockey` ce qui constitue un réchauffement d'avant-match, on veut que ce soit chacune des classes « concrètes » qui le définissent, la raison étant que de toute façon le réchauffement est uniquement dépendant du type de joueur.

Dans `JoueurHockey`, on ajouterait alors ceci :

```
public abstract void FaireRechauffementAvantMatch();
```

En compilant, vous auriez les messages d'erreurs de compilation suivant :



On vous demande donc d'implémenter ces méthodes. Vous pourriez par exemple les définir comme suit, d'abord pour Gardien:

```
public override void FaireRechauffementAvantMatch()
{
    Console.WriteLine(this.Nom + " pratique ses déplacements gauche/droit
    et reçoit des dizaines de lancers. Il est fin prêt!");
}
```

Et pour Avant :

```
public override void FaireRechauffementAvantMatch()
{
    Console.WriteLine(this.Nom + " pratique ses lancers et son maniement
    de la rondelle. Il est fin prêt!");
}
```

**Notez que vous devez à nouveau utiliser le mot clé `override` pour implémenter les méthodes abstraites.**

Quelques remarques importantes :

- **Vous ne pouvez pas déclarer une méthode comme étant abstraite autre part que dans une classe abstraite;** ceci échouerait :

```
public sealed class Avant : JoueurHockey
{
    0 references
    public abstract void FaireX();
    3 references
}
```

'Chapitre4.Avant.FaireX()' is abstract but it is contained in non-abstract class 'Chapitre4.Avant'

- **Vous ne pouvez pas définir le corps d'une méthode abstraite à même la classe où vous définissez la méthode comme étant abstraite;** ceci échouerait :

```
public abstract void FaireRechauffementAvantMatch()
{
    Console.WriteLine(this.Nom + " se préparer à son réchauffement d'avant-match!");
}
```

❌ 1 'Chapitre4.JoueurHockey.FaireRechauffementAvantMatch()' cannot declare a body because it is marked abstract JoueurHockey.cs

- **Un champ ne peut pas être abstrait;** ceci échouerait :

```
public abstract int age;
```

The modifier 'abstract' is not valid on fields. Try using a property instead.

- **Vous pouvez définir des propriétés abstraites.** À ce moment-là, vous pouvez utiliser ces syntaxes :

```
public abstract int Age { get; }

public abstract int Sexe { get; set; }

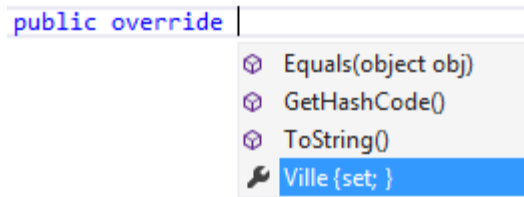
public abstract int Ville { set; }
```

Et vous devrez obligatoirement définir l'accesseur et/ou le mutateur conformément à ce qui est dicté dans votre définition de la propriété abstraite pour tous les descendants.

Par exemple, pour `Ville`, on devrait retrouver ceci dans `Gardien` :

```
public override int Ville
{
    set { this.Ville = value; }
}
```

L'Intelli-Sense vous assiste également à cette tâche :



Si la signature de la propriété abstraite le permet, vous pouvez utiliser une propriété abstraite dans la redéfinition :

```
public override int Sexe { get; set; }
```

- **Vous pouvez redéfinir la méthode abstraite à nouveau pour un sous-descendant si vous le souhaitez.**

Imaginons que la classe `Avant` n'est plus scellée et que nous avons la sous-classe `Attaquant` à nouveau. Il vous est possible d'avoir le code suivant :

```
public class Attaquant : Avant
{
    public override void FaireRechauffementAvantMatch()
    {
        (...)
    }
}
```

Mais cela devient facultatif puisqu'une définition dans `Avant` a déjà été fournie.

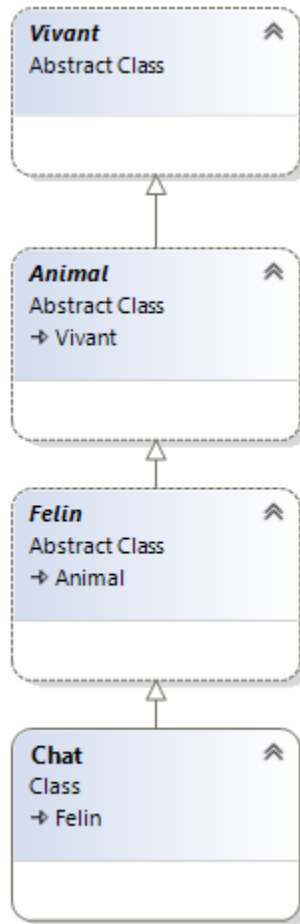
Vous pouvez tout de même bloquer toute redéfinition ultérieure en utilisant **sealed** dans la première définition dans la classe `Avant` :

```
public override sealed void FaireRechauffementAvantMatch()
{
    Console.WriteLine(this.Nom + " pratique ses lancers et son maniement  
de la rondelle. Il est fin prêt!");
}
```

Dans ce cas vous aurez un message d'erreur du côté d'`Attaquant` :

❌ 1 'Chapitre4.Attaquant.FaireRechauffementAvantMatch()': cannot override inherited member 'Chapitre4.Avant.FaireRechauffementAvantMatch()' because it is sealed Attaquant.cs

- Si vous avez plusieurs niveaux de classes abstraites, comme ceci :



Et que vous définissez une méthode abstraite à un certain niveau, **vous n'êtes pas tenu de définir la méthode abstraite avant d'avoir atteint une première classe concrète** dans la hiérarchie.

Par exemple, ceci serait convenable :

```
public abstract class Vivant
{
    public abstract void SeReproduire();
}

public abstract class Animal : Vivant { }
```

```
public abstract class Felin : Animal { }

public class Chat : Felin
{
    public override void SeReproduire()
    {
        Console.WriteLine("Le chat se reproduit");
    }
}
```

Idem pour cela :

```
public abstract class Vivant
{
    public abstract void SeReproduire();
}

public abstract class Animal : Vivant
{
    public override void SeReproduire()
    {
        Console.WriteLine("Le chat se reproduit");
    }
}

public abstract class Felin : Animal { }

public class Chat : Felin { }
```

- Enfin, **vous pouvez avoir une classe abstraite qui hérite d'une classe concrète**; ceci fonctionnerait :

```
public class Vivant { }

public abstract class Animal : Vivant { }

public abstract class Felin : Animal { }

public class Chat : Felin { }
```

Un peu grâce à toutes ces notions – en particulier les classes abstraites – on peut observer toute la puissance du polymorphisme du fait qu'on peut alors se créer une collection de type abstrait qui contient en réalité des objets de dérivation du type de la collection en question.

Par exemple, on pourrait se créer un tableau de joueurs de hockey et appeler les membres en communs sans se soucier du sous-type, et l'implémentation propre du membre de chaque sous-type – si elle existe – sera appelée :



```
class Program
{
    static void Main(string[] args)
    {
        JoueurHockey[] joueurs =
        {
            new Avant(new JoueurHockey.Contrat(2000000, 4, "Tobby McGuire"),
                "Pinocchio", 41, 45, 39, 22, 45, 72, AvantPosition.Attaquant),
            new Avant(new JoueurHockey.Contrat(3500000, 2, "Tobby McGuire"),
                "Figaro", 63, 29, 43, 12, 45, 34, AvantPosition.Defenseur,
                JoueurStatut.Suspendu),
            new Gardien(new JoueurHockey.Contrat(5750000, 3, "Ken Thomas"),
                "Oswald", 11, 39, 43, 44, 45, statut: JoueurStatut.PretAJouer)
        };

        foreach(JoueurHockey j in joueurs)
        {
            j.SEntraîner();
            j.FaireRechauffementAvantMatch();
        }

        Console.ReadLine();
    }
}
```

On aurait en sortie après exécution :

On peut observer que ce sont les méthodes redéfinies qui sont exécutées et non les méthodes de la classe `JoueurHockey`. C'est là du polymorphisme à l'état pur.

#### 4.2.5 Occultation (*shadowing*)

C# fournit le nécessaire pour faire l'opposée logique de la redéfinition de méthodes, qu'on appelle de l'occultation, ou en jargon du *shadowing* :

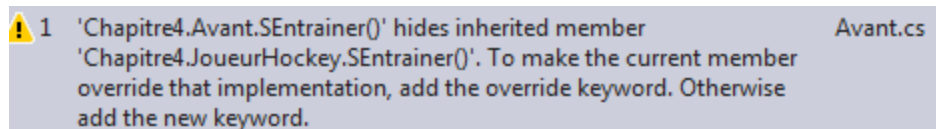
- *Si une classe dérivée définit un membre qui est identique à un membre défini dans la classe de base, on dira que la classe dérivée a occulté la version du membre de la classe parent.*

Par exemple, si on n'utilise pas le mot clé `override` devant la méthode `SEntreiner()` :

```
public void SEntreiner()
{
    base.SEntreiner();

    if (new Random().Next(1, 100) > this.Endurance)
    {
        Console.WriteLine("Vous avez maintenant plus d'endurance grâce à vos efforts à l'entraînement.");
        this.Endurance++;
    }
}
```

Rappelez-vous que vous obtenez le message d'erreur suivant :



Si vous lisez attentivement, vous remarquez qu'à la toute fin on vous propose une solution autre que l'utilisation d'`override` : on vous suggère d'utiliser le mot clé `new`.

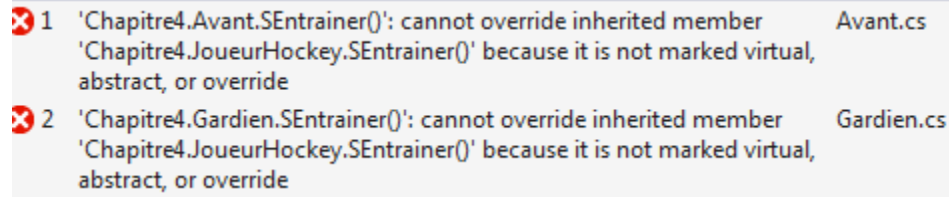
**L'utilisation de `new` plutôt qu'`override` sert exactement à faire du *shadowing*.**

L'idée derrière tout cela est que si vous étendez une classe provenant d'une librairie extérieure au projet que vous ne pouvez pas modifier, puis que vous désirez redéfinir un membre non virtuel, même si vous utilisez `override`, cela ne vous sera d'aucun recours. Entre autres, vous ne pourrez pas modifier le code de la classe parent pour ajouter le modificateur `virtual`.

**L'occultation grâce au mot clé `new` vous permettra de complètement ignorer l'implémentation parente du membre et de faire votre propre version.**

Imaginons un instant que `SEntreiner()` n'était plus une méthode virtuelle et que `JoueurHockey` provenant d'une librairie importée.

Visual Studio vous donne alors les messages d'erreur suivant :



Pour régler le problème, vous pouvez faire ceci :

```
public new void SEntrainer()
{
    base.SEntrainer();

    if (new Random().Next(1, 100) > this.Endurance)
    {
        Console.WriteLine("Vous avez maintenant plus d'endurance grâce à vos efforts à l'entraînement.");
        this.Endurance++;
    }
}
```

Le tout compilera et vous pouvez même appeler la méthode `SEntrainer()` de la classe de base comme auparavant si vous le souhaitez.

**Vous pouvez même, si vous le souhaitez, utiliser `new` devant tout membre hérité de la classe de base, comme un champ, une constante, un membre statique ou une propriété.** C'est une des différences avec `override`, d'ailleurs.

Par exemple (cela ne fera aucun sens dans le jeu du hockey, mais supposons!), si les numéros de chandail des gardiens de but devaient être entre 0 et 9, plutôt que 0 et 99, on pourrait utiliser l'occultation de cette façon :

```
private new int numero = 4;

public new int Numero
{
    get { return this.numero; }
    set { this.numero = value > 9 || this.numero < 0 ? 0 : value; }
}
```

Remarquez qu'il est rarement utile d'occultier un champ d'instance, sauf lorsqu'on veut remplacer ou retirer une valeur par défaut personnalisée.

Enfin, remarquez que le modificateur d'accès pour `numero` est passé de `protected` vers `private`. Cela est dû au fait que la classe `Gardien` est scellée : l'utilisation de `protected` ne fait alors plus de sens. Le *shadowing* permet de changer le modificateur d'accès! Cela constitue une autre différence avec `override`!

## 4.3 RÈGLES DE CONVERSION DE TYPE ENTRE CLASSES DÉRIVÉES

### 4.3.1 Principe général

Avec l'héritage et le polymorphisme viennent également les opérations de conversion entre différents types d'une même hiérarchie.

Sachant que la classe de base de toutes les classes (dont toutes les classes dérivent implicitement) est la classe `System.Object`, on peut considérer que « tout est un `Object` ».

Il est donc tout à fait acceptable par polymorphisme de stocker toute classe dans une variable déclarée de type `Object` :

```
Object joueur = new Avant(new JoueurHockey.Contrat(2000000, 4, "Tobby McGuire"), "Pinocchio", 41, 45, 39, 22, 45, 72, AvantPosition.Attaquant);
```

De façon analogue, on aurait pu utiliser `JoueurHockey` pour stocker un objet de classe `Avant` (comme pour le tableau plus tôt):

```
JoueurHockey joueur = new Avant(new JoueurHockey.Contrat(2000000, 4, "Tobby McGuire"), "Pinocchio", 41, 45, 39, 22, 45, 72, AvantPosition.Attaquant);
```

La règle de base pour la conversion entre des types classe est que lorsque deux classes sont associées par une relation de type « est-un » il est toujours permis de stocker l'objet dérivé dans une référence d'une classe de base.

Il se produit alors une conversion implicite.

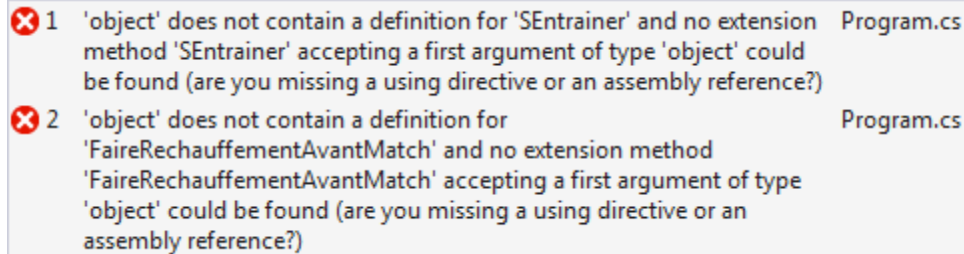
C'était pratique plus tôt pour accéder aux fonctionnalités de `JoueurHockey` accessible à partir d'`Avant` ou `Gardien`, mais si vous aviez plutôt eu un tableau de type `Object`, comme ceci :

```
Object[] joueurs =
{
    new Avant(new JoueurHockey.Contrat(2000000, 4, "Tobby McGuire"),
        "Pinocchio", 41, 45, 39, 22, 45, 72, AvantPosition.Attaquant),
    new Avant(new JoueurHockey.Contrat(3500000, 2, "Tobby McGuire"),
        "Figaro", 63, 29, 43, 12, 45, 34, AvantPosition.Defenseur,
        JoueurStatut.Suspendu),
    new Gardien(new JoueurHockey.Contrat(5750000, 3, "Ken Thomas"),
        "Oswald", 11, 39, 43, 44, 45, statut: JoueurStatut.PretAJouer)
};

foreach(Object j in joueurs)
```

```
{
    j.SEntrainer();
    j.FaireRechauffementAvantMatch();
}
```

Vous auriez généré des erreurs de compilation :



Même chose si vous aviez tenté d'appeler des membres particuliers d'Avant ou Gardien :

```
foreach(JoueurHockey j in joueurs)
{
    j.SEntrainer();
    j.FaireRechauffementAvantMatch();
    j.PratiquerGant();
}
Console.ReadLine();
```

'Chapitre4.JoueurHockey' does not contain a definition for 'PratiquerGant' and no extension method 'PratiquerGant' could be found (are you missing a using directive or an assembly reference?)

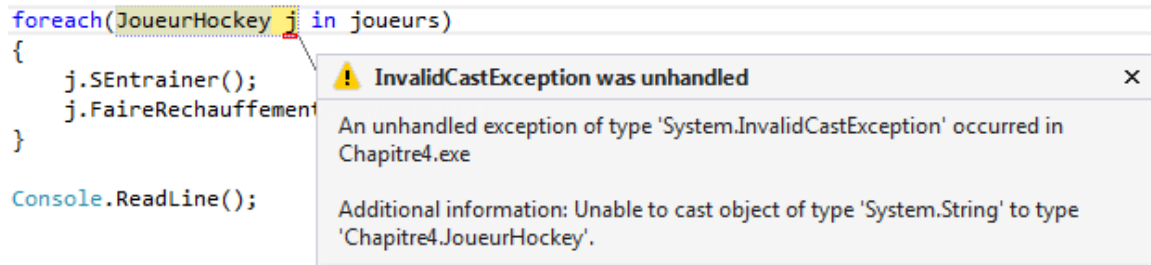
Et même si de prime à bord ce code compile :

```
static void Main(string[] args)
{
    Object[] joueurs =
    {
        "Bonjour",
        true,
        14,
        new Avant(new JoueurHockey.Contrat(2000000, 4, "Tobby McGuire"),
        "Pinocchio", 41, 45, 39, 22, 45, 72, AvantPosition.Attaquant),
        new Avant(new JoueurHockey.Contrat(3500000, 2, "Tobby McGuire"),
        "Figaro", 63, 29, 43, 12, 45, 34, AvantPosition.Defenseur,
        JoueurStatut.Suspendu),
        new Gardien(new JoueurHockey.Contrat(5750000, 3, "Ken Thomas"),
        "Oswald", 11, 39, 43, 44, 45, statut: JoueurStatut.PretAJouer)
    };

    foreach(JoueurHockey j in joueurs)
    {
        j.SEntrainer();
        j.FaireRechauffementAvantMatch();
    }
}
```

```
        Console.ReadLine();  
    }  
}
```

Vous aurez des erreurs à l'exécution :



Le problème évidemment est qu'avant l'exécution, il n'est pas toujours possible pour le compilateur de savoir si le contenu du tableau sera uniquement des `JoueursHockey` ou des descendants.

Une possibilité serait d'utiliser une conversion explicite tel que :

```
JoueurHockey joueur = new Avant(new JoueurHockey.Contrat(2000000, 4, "Tobby  
McGuire"), "Pinocchio", 41, 45, 39, 22, 45, 72, AvantPosition.Attaquant);  
  
((Gardien)joueur).PratiquerBaton();
```

Cela compile, mais ce genre de stratégie n'est pas toujours approprié (cela planterait dans tous les cas, le joueur étant un avant et non un gardien).

En fait, cette conversion explicite est déjà ce qui est tenté dans le `foreach` précédent : .NET tente de convertir explicitement chaque instance dans `joueurs` en un `JoueurHockey`.

### 4.3.2 As

Pour éviter de lever une exception d'invalidité de conversion à l'exécution, vous avez la possibilité d'utiliser le mot clé **as**, qui vérifie la compatibilité à la conversion. S'il s'avère que la conversion est impossible, alors `null` est retourné plutôt qu'une erreur.

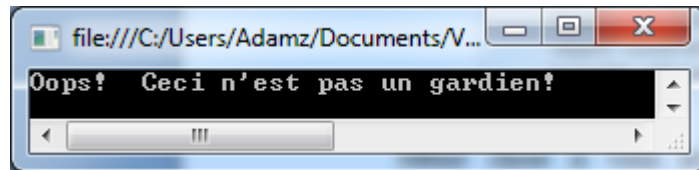
Par exemple :

```
JoueurHockey joueur = new Avant(new JoueurHockey.Contrat(2000000, 4,  
"Tobby McGuire"), "Pinocchio", 41, 45, 39, 22, 45, 72,  
AvantPosition.Attaquant);  
  
//...
```

```
Gardien gardien = joueur as Gardien;

if (gardien == null)
    Console.WriteLine("Oops! Ceci n'est pas un gardien!");
else
    gardien.PratiquerBaton();
```

Alors à l'exécution on aurait ceci :



Pour le tableau de joueur, on pourrait adopter une technique similaire :

```
static void Main(string[] args)
{
    JoueurHockey[] joueurs =
    {
        new Avant(new JoueurHockey.Contrat(2000000, 4, "Tobby McGuire"),
            "Pinocchio", 41, 45, 39, 22, 45, 72, AvantPosition.Attaquant),
        new Avant(new JoueurHockey.Contrat(3500000, 2, "Tobby McGuire"),
            "Figaro", 63, 29, 43, 12, 45, 34, AvantPosition.Defenseur,
            JoueurStatut.Suspendu),
        new Gardien(new JoueurHockey.Contrat(5750000, 3, "Ken Thomas"),
            "Oswald", 11, 39, 43, 44, 45, statut: JoueurStatut.PretAJouer)
    };

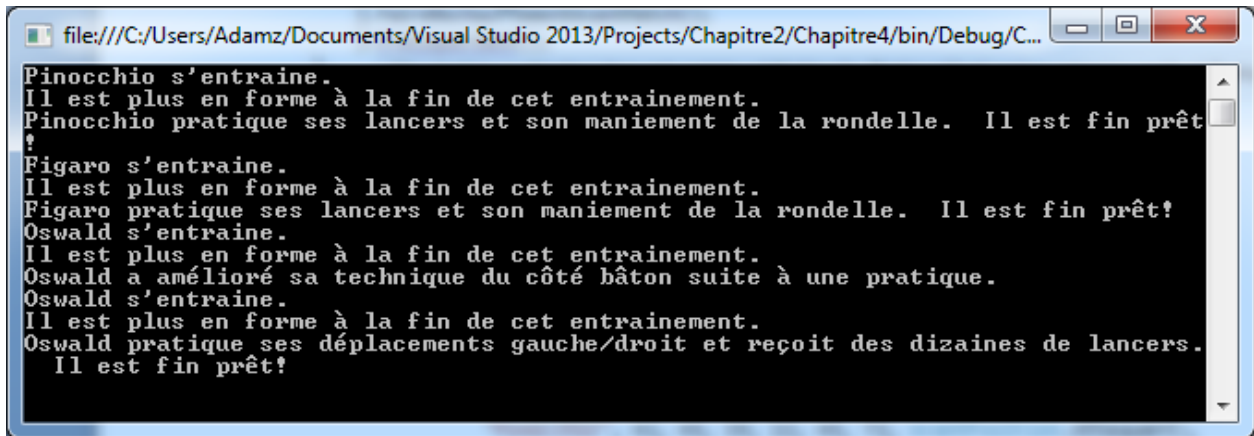
    foreach(JoueurHockey j in joueurs)
    {
        Gardien g = j as Gardien;

        if (g != null)
            g.PratiquerBaton();

        j.SEntrainer();
        j.FaireRechauffementAvantMatch();
    }

    Console.ReadLine();
}
```

Et à la sortie en console on pourrait observer que seulement si le joueur est un gardien, on fait un traitement supplémentaire :



```
file:///C:/Users/Adamz/Documents/Visual Studio 2013/Projects/Chapitre2/Chapitre4/bin/Debug/C...
Pinocchio s'entraîne.
Il est plus en forme à la fin de cet entraînement.
Pinocchio pratique ses lancers et son maniement de la rondelle. Il est fin prêt!
Figaro s'entraîne.
Il est plus en forme à la fin de cet entraînement.
Figaro pratique ses lancers et son maniement de la rondelle. Il est fin prêt!
Oswald s'entraîne.
Il est plus en forme à la fin de cet entraînement.
Oswald a amélioré sa technique du côté bâton suite à une pratique.
Oswald s'entraîne.
Il est plus en forme à la fin de cet entraînement.
Oswald pratique ses déplacements gauche/droit et reçoit des dizaines de lancers.
Il est fin prêt!
```

### 4.3.3 Is

Une autre possibilité est d'utiliser le mot clé **is**, qui permet de déterminer si vrai ou faux l'instance est d'un type ou non.

**En comparaison, alors que `as` retourne null ou l'objet converti, `is` retourne simplement `true` ou `false`.**

Ainsi, on pourrait avoir le code suivant :

```
foreach(JoueurHockey j in joueurs)
{
    if (j is Gardien)
        ((Gardien)j).PratiquerBaton();

    j.SEntrainer();
    j.FaireRechauffementAvantMatch();
}
```

Qui donnerait le même résultat que précédent, moyennant une affectation de moins.

On peut donc s'en servir pour faire des traitements particuliers à toutes sortes de dérivées du type de base :

```
foreach(JoueurHockey j in joueurs)
{
    if (j is Gardien)
        ((Gardien)j).PratiquerBaton();

    if (j is Avant)
    {
        ((Avant)j).PratiquerLancer();
        ((Avant)j).PratiquerManiementRondelle();
    }
}
```



```

        j.SEntrainer();
        j.FaireRechauffementAvantMatch();
    }

```

C'est également très utile pour traiter des paramètres d'un type de base pour une méthode qui nécessite des traitements particuliers selon le type dérivé. Par exemple :

```

static void Main(string[] args)
{
    JoueurHockey[] joueurs =
    {
        new Avant(new JoueurHockey.Contrat(2000000, 4, "Tobby McGuire"),
            "Pinocchio", 41, 45, 39, 22, 45, 72, AvantPosition.Attaquant),
        new Avant(new JoueurHockey.Contrat(3500000, 2, "Tobby McGuire"),
            "Figaro", 63, 29, 43, 12, 45, 34, AvantPosition.Defenseur,
            JoueurStatut.Suspendu),
        new Gardien(new JoueurHockey.Contrat(5750000, 3, "Ken Thomas"),
            "Oswald", 11, 39, 43, 44, 45, statut: JoueurStatut.PretAJouer)
    };

    foreach(JoueurHockey j in joueurs)
    {
        TraiterJoueur(j);
    }

    Console.ReadLine();
}

public static void TraiterJoueur(JoueurHockey joueur)
{
    if (joueur is Gardien)
        ((Gardien)joueur).PratiquerBaton();

    if (joueur is Avant)
    {
        ((Avant)joueur).PratiquerLancer();
        ((Avant)joueur).PratiquerManiementRondelle();
    }

    joueur.SEntrainer();
    joueur.FaireRechauffementAvantMatch();
}

```

## 4.4 OBJECT.TOSTRING()

Enfin, il est difficile de passer à côté de la redéfinition de la méthode `ToString()`. Les classes et les structures ont toutes une méthode `ToString()` qui est héritée de la classe `System.Object`.

Par défaut, elle retourne par réflexion le type complet (c'est-à-dire l'espace de nom complet et le type) de l'instance en question.

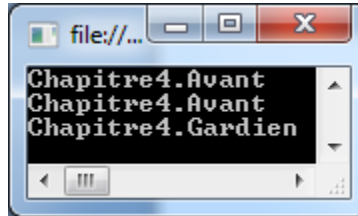
Par exemple :

```
static void Main(string[] args)
{
    JoueurHockey[] joueurs =
    {
        new Avant(new JoueurHockey.Contrat(2000000, 4, "Tobby McGuire"),
            "Pinocchio", 41, 45, 39, 22, 45, 72, AvantPosition.Attaquant),
        new Avant(new JoueurHockey.Contrat(3500000, 2, "Tobby McGuire"),
            "Figaro", 63, 29, 43, 12, 45, 34, AvantPosition.Defenseur,
            JoueurStatut.Suspendu),
        new Gardien(new JoueurHockey.Contrat(5750000, 3, "Ken Thomas"),
            "Oswald", 11, 39, 43, 44, 45, statut: JoueurStatut.PretAJouer)
    };

    foreach(JoueurHockey j in joueurs)
        Console.WriteLine(j.ToString());

    Console.ReadLine();
}
```

Donnera ceci :



Écrire `.ToString()` est facultatif : si vous ne faites qu'écrire `j`, implicitement .NET fera appel à la méthode `ToString()`.

Ainsi, ceci donne le même résultat :

```
foreach(JoueurHockey j in joueurs)
    Console.WriteLine(j);
```

L'intérêt de la méthode `ToString()` est surtout d'être redéfinie à travers ses classes dérivées, pour donner une représentation textuelle de l'instance et son état.

Par exemple, vous pourriez rajouter ceci dans `JoueurHockey` pour forcer `Avant` et `Gardien` à redéfinir `ToString()`, sans que `JoueurHockey` ne la redéfinisse lui-même :

```
public abstract override string ToString();
```

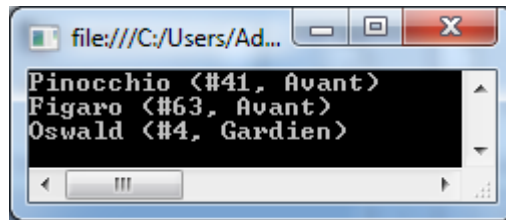
Puis, dans `Gardien` et `Avant`, on pourrait retrouver ceci pour `Gardien` :

```
public override string ToString()
{
    return String.Format("{0} ({1}, Gardien)", this.Nom, this.Numero);
}
```

Et pour `Avant`:

```
public override string ToString()
{
    return String.Format("{0} ({1}, Avant)", this.Nom, this.Numero);
}
```

Alors à l'affichage on aurait maintenant plutôt ceci :



### 4.5 SHADOWING (BIS)

Maintenant que nous avons présenté la conversion, nous allons pouvoir aller au-delà des différences vues jusqu'ici entre la redéfinition et l'occultation.

Le *shadowing* agit pour véritablement **cacher** l'implémentation de base.

Si on avait le code suivant :

```
public class ClasseBase
{
    public virtual void FaireX()
    {
        Console.WriteLine("FaireX() ClasseBase");
    }
}
```

```
public class ClasseOverride : ClasseBase
{
    public override void FaireX()
    {
        Console.WriteLine("FaireX() ClasseOverride");
    }
}

public class ClasseShadowing : ClasseBase
{
    public new void FaireX()
    {
        Console.WriteLine("FaireX() ClasseShadowing");
    }
}
```

On aurait donc une classe de base, avec deux classes dérivées :

- Une utilisant la redéfinition;
- L'autre utilisant l'occultation.

Voyons voir comment se comporte .NET si on a le code suivant dans le Main () :

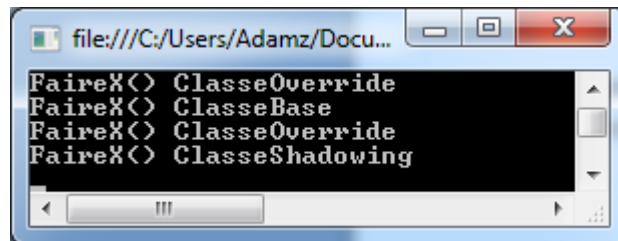
```
static void Main(string[] args)
{
    ClasseBase cA = new ClasseOverride();
    ClasseBase cB = new ClasseShadowing();

    //-- à partir de la classe de base par polymorphisme --
    cA.FaireX();
    cB.FaireX();

    //-- directement à partir de la classe dérivée --
    ((ClasseOverride)cA).FaireX();
    ((ClasseShadowing)cB).FaireX();

    Console.ReadLine();
}
```

En sortie, on obtient ceci :



Que s'est-il passé?

- En fait, dans le cas d'une méthode redéfinie, si on invoque la méthode du même nom à partir de `ClasseBase`, c'est la méthode redéfinie par la classe dérivée qui est appelée et non celle de la classe de base, conformément au principe du polymorphisme;
- Dans le second cas, la méthode occultée est ignorée et c'est tout de même celle de la classe de base qui est appelée. La seule façon d'y faire véritablement appel est d'avoir une variable (convertie) de type `ClasseShadowing` et d'appeler la méthode.

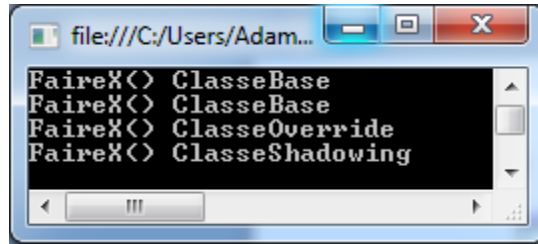
Si on pousse le test un peu plus loin, en faisant en sorte que `ClasseOverride` dérive non plus de `ClasseBase` mais de `ClasseShadowing`, comme ceci (remarquez qu'on doit ajouter `virtual` à côté de `new` dans `ClasseShadowing` pour permettre la redéfinition) :

```
public class ClasseBase
{
    public virtual void FaireX()
    {
        Console.WriteLine("FaireX() ClasseBase");
    }
}

public class ClasseOverride : ClasseShadowing
{
    public override void FaireX()
    {
        Console.WriteLine("FaireX() ClasseOverride");
    }
}

public class ClasseShadowing : ClasseBase
{
    public new virtual void FaireX()
    {
        Console.WriteLine("FaireX() ClasseShadowing");
    }
}
```

Et qu'on réexécute le même jeu de tests, on obtiendra :



On voit que l'ombrage se poursuit pour la descendance du point de vue de `ClasseBase`.

Toutefois, si on avait ceci dans le `Main()` :

```
ClasseShadowing cA = new ClasseOverride();  
ClasseShadowing cB = new ClasseShadowing();
```

Alors en sortie tout se comporterait conformément à la redéfinition de méthodes, comme si la méthode cachée virtuelle constituait un nouveau point de départ :

