PROGRAMMATION OBJET AVANCEE
& LANGAGE C#
Résumé de cours

Héritage & Polymorphisme

# 0. Objectifs

Comprendre les principes de l'héritage
Comprendre le polymorphisme d'héritage
Comprendre le chainage des constructeurs
Comprendre les interfaces
Savoir mette en œuvre ces notions en C#

# 1. Rappel sur les classes & instances

Une classe est une description, un modèle, un plan, un moule ou encore un schéma de construction pour les objets à créer. Une classe n'existe pas physiquement en mémoire (exception faite des statics). Seuls les objets allouent des ressources à chacune de leur création. Une classe décrit l'ensemble des paramètres et des méthodes détenus par chaque instance.

## 1.1 Encapsulation

Un objet est vu de l'extérieur comme une boîte noire fournissant des propriétés et des méthodes. Pour faciliter la mise en œuvre d'un programme, il est utile de savoir comment on utilise un objet sans connaître toute sa mécanique interne. Par analogie avec la vie courante, vous savez comment vous servir d'un téléphone portable et il vous semblerait étrange d'avoir à maîtriser toutes les technologies utilisées à l'intérieur du téléphone pour passer un simple coup de fil. On parle d'encapsulation au sens où les objets que vous concevez doivent masquer leur mécanique interne à la fois pour être facile à utiliser ou pour se protéger d'utilisation maladroite.

Dans une librairie, par exemple .net, Microsoft fournit l'accès aux membres externes (public et protected) de ses classes. Les membres internes des différentes classes restent masqués et leur code peut évoluer sans que cela change le comportement de la classe.

Chaque membre de la classe (champs, méthodes, propriétés) peut être qualifié par un des modificateurs suivant :

**public** : accessible par tout le monde : parents, enfants, autres
**private** : accessible uniquement depuis la classe. Les classes dérivées n'y ont pas accès.
**protected** : accessible uniquement depuis la classe et ses classes dérivées.

Remarque : si aucun qualificatif n'est donné, c'est implicitement un private.

Il n'est pas toujours évident de choisir le bon niveau de visibilité. La prudence suggère de toujours utiliser le niveau de visibilité le plus bas possible. Pour le prototypage où l'on privilégie la rapidité de développement à la sécurité du code, on préfèrera le niveau public. Lors de vos travaux pratiques et exercices en classe, sauf avis contraire, vous pouvez rendre tous les membres publics.

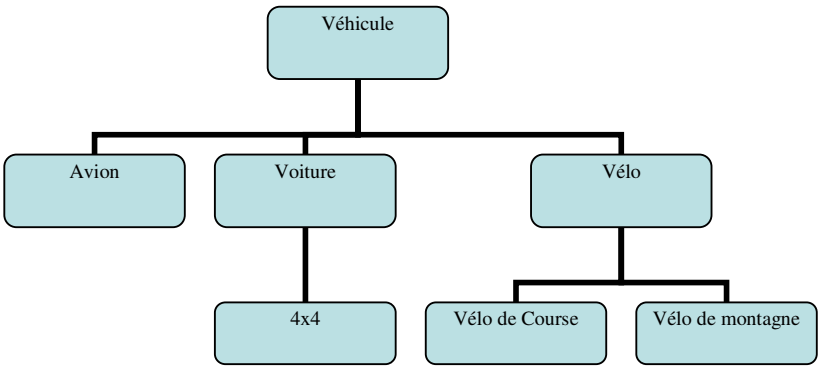
# 2. Héritage

L'héritage est une approche puissante pour la réutilisation du code. Comme toute technique de réutilisation, elle facilite la maintenance, améliore la productivité et permet de mieux structurer votre programme.

L'héritage permet de construire une classe B à partir d'une classe existante A. Il s'agit d'une sorte d'héritage biologique. La classe B va ainsi hériter des variables, des méthodes, bref de tous les membres de A puis, on va ajouter des éléments supplémentaires propres à la classe B.

Un héritage entre plusieurs classes fait naître une notion de **hiérarchie**. Si B hérite de A, on dit que B est une **classe fille**, une **sous-classe**, une **classe enfant** ou encore une **classe dérivée** de A. D'une autre manière, A se nomme la **classe mère**, la **classe de base**, la **classe parent** ou encore la **super classe** de B.

On peut examiner une hiérarchie de classes de deux façons différentes. On peut choisir une vision par spécialisation/redéfinition. Dans ce cas, la classe B est vue comme un cas particulier de la classe A. Par exemple, un 4x4 est un cas particulier d'une Voiture. Dans cette vision, l'ensemble des Voitures est l'ensemble englobant et les 4x4 représentent un sous-ensemble des Voitures. Un autre point de vue est celui de l'extension. La classe B est vue comme une version étendue de la classe A. Par exemple, un 4x4 est une voiture disposant de 2 roues motrices supplémentaires. Du point de vue des fonctionnalités, la classe 4x4 englobe les fonctionnalités de la classe Voiture, la classe Voiture a moins de paramètres internes que la classe 4x4.




Règles :

- Il ne peut y avoir qu'une seule classe parent. L'héritage multiple est interdit.
- Les cycles sont interdits : A ne peut hériter de B si B hérite déjà de A.

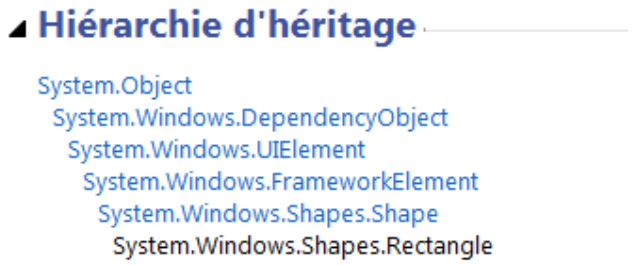
Syntaxe :

```
class Vehicule
{
        public string   nom;
        public void     AfficheNom() { WriteLine(nom);  }
}

class Voiture : Vehicule
{
        public int      puissance;
        public void     Info() {  WriteLine(nom); WriteLine(puissance); }
}
```

| | | | |
|---|---|---|---|
| Vehicule V = new Vehicule(); | | Voiture K = new Voiture(); | |
| V.nom = ''Proto''; | => OK | K.nom = ''Buggy''; | => OK |
| V.AfficheNom(); | => Proto | K.puissance = 4 ; | => OK |
| V.puissance = 4; | => ERREUR | K.Info() ; | => Buggy 4 |
| V.Info(); | => ERREUR | K.AfficheNom() ; | => Buggy |

La MSDN permet de voir la hiérarchie de classe pour chaque classe du framework .net : Par exemple pour la classe System.Windows.Shapes.Rectangle on obtient :




Ce qui signifie que la classe Rectangle hérite de Shape, qui elle-même hérite de FrameworkElement et ainsi de suite. Vous constaterez que les librairies modernes utilisent abondamment l'héritage et que les hiérarchies de classe peuvent être relativement complexes.

Remarque : le mot clef sealed précédant le mot clef class permet d'empêcher l'extension d'une classe.

Remarque : en C#, les hiérarchies des classes étant fixes durant l'exécution du programme, il faut que la relation entre les classes restent valide dans le temps. Par exemple, la Kangoo ne pas être fille de la classe Voiture Essence, bien qu'il existe essentiellement des Kangoo dotées d'un moteur essence. En effet, il suffit qu'il existe également des motorisations électriques (ou diesel) pour que Kangoo ne puisse plus hériter de Voiture Essence.

HP : public class et internal.

# 3. Constructeurs et chaînage

Prenons une classe B enfant d'une classe A. Lors de l'écriture de ses constructeurs paramétriques ou sans arguments de la classe B, on doit se poser la question de comment va-t-on gérer la construction des éléments provenant de A.

Le choix maladroit :

| | | |
|---|---|---|
| Constructeur de Vehicule : | Vehicule(string _nom) | { nom = _nom ; } |
| Constructeur de Voiture : | Voiture(string _nom, int p) | { nom = _nom ; puissance = p ; } |

Cette solution est correcte et fonctionnelle. Cependant elle constitue une source de futurs problèmes. Dans le cas d'une classe plus complexe, pour certains paramètres vous allez effectuer des tests pour vérifier l'exactitude des valeurs passées. Par exemple, pour un

objet Rectangle, vous vérifiez que sa longueur est positive. Si vous créez une classe dérivée Carré, il faudra soit dupliquer ces tests dans le nouveau constructeur – et il est absolument interdit de copier coller du code à l'identique – soit oublier de faire ces vérifications, ce qui est fâcheux. Pour éviter ces problèmes, il faut utiliser le chaînage de constructeurs.

Le choix correct :

```
class Voiture
{
        …
        public Voiture(string _nom, int p) : base(_nom)
        { puissance = p ; }
}
```

Le mot clef « base » sert à appeler explicitement le constructeur de la classe mère de Voiture. Le constructeur de la classe parent fait ce qu'il doit faire : vérification / affectation / initialisation et nous faisons dans le constructeur de Voiture uniquement ce qui concerne la classe Voiture.

Question : dans quel ordre sont lancés les constructeurs ?

```
class A
{
   public A()                   { WriteLine("construction de A"); }
}

class B : A
{
   public B() : base()          { WriteLine("construction de B"); }
}

B toto = new B() ;      =>  construction de A
                        =>  construction de B
```

Le constructeur de la classe parent est lancé AVANT les instructions du constructeur de la classe B. Si la classe A possède un parent, son constructeur sera appelé avant que celui de A soit exécuté. Au final, c'est le constructeur du parent initial dans la hiérarchie qui sera le premier lancé, puis ceux de ses enfants successivement. Ce comportement logique provient du fait que pour fonctionner, la classe B peut nécessiter des ressources venant de sa classe mère, le parent doit donc être créé avant.

Remarque 1 : l'écriture de « : base() » est optionnel. Dans tous les cas, cela veut dire que si vous ne chaînez pas explicitement le constructeur parent, PAR DEFAUT LE CONSTRUCTEUR SANS ARGUMENT DU PARENT SERA APPELE. Les raisons sont identiques aux précédentes, pour fonctionner B peut avoir besoin des ressources provenant de A, il faut donc qu'elles soient initialisées correctement.

Remarque 2 : si vous faîtes un héritage en laissant l'appel par défaut au constructeur sans arguments, le système exigera que le constructeur sans arguments soit codé, il n'en fournira plus un par défaut. Inutile de retenir ce détail, les messages du compilateur vous le rappelleront.

Remarque 3 : Lors de la construction de B, le fait que le constructeur de A soit appelé ne veut pas dire qu'un objet de type A est créé mais que les membres hérités de A dans B sont initialisés. Au final, il n'y a bien qu'un seul objet de type B de créé.

# 4 Polymorphisme

## 4.1 Principe

Le terme polymorphisme désigne la capacité d'une méthode à pouvoir prendre un comportement différent suivant le contexte (les types utilisés). Ainsi nous pouvons obtenir des implémentations plus générales, mieux structurées et plus simples à comprendre.

Par exemple, si l'on veut créer une fonction calculant le maximum de deux nombres, il serait pratique qu'elle puisse porter le même nom quels que soient les paramètres utilisés. On voudrait créer ces fonctions de la manière suivante :

```
int     max(int a, int b)              { … }
float   max(float a, float b)          { … }
double max(double a,double b)          { … }
…
k = max(u,v) ;
```

Le compilateur connaît les types des variables u et v, il va donc chercher la fonction max prenant ces types là en arguments. Les différentes fonctions max ne traitent pas les mêmes éléments, pourtant conceptuellement, elles effectuent le même calcul. C'est un des aspects du **polymorphisme**.

Remarque : en C strict, un identificateur de fonction ne peut être associé qu'à une liste unique de paramètres. Le langage d'origine n'implémentait pas le polymorphisme et il fallait alors créer des fonctions différentes : max_int max_float et max_double.

Ce concept de polymorphisme s'étend à l'héritage. Par exemple si vous faîtes un jeu de simulation affichant plusieurs types de véhicules à l'écran, il serait agréable d'avoir une unique fonction AfficheToi(). Ainsi pour dessiner l'écran du jeu, je demande à chaque objet de s'afficher en utilisant le même nom de fonction. Cette action est similaire pour une voiture, un avion, un vélo, chacun va s'afficher à l'écran, mais avec un comportement différent. Un objet A320 dessinera spécifiquement un Airbus A320, chaque voiture Peugeot 205 dessinera une Peugeot 205… Il est possible, mais plus lourd à gérer, de créer des fonctions différentes comme AfficheA320(), AfficheC3(), AfficheP eugeot205()…

Ainsi une fonction peut exister sous le même nom dans une hiérarchie mais avoir un comportement différent suivant le type de l'objet instancié. On parle alors de **polymorphisme d'héritage**.

## 4.2 Mise en place

L'approche à éviter :

A première vue, il n'y a pas de syntaxe particulière à mettre en place. La règle qui s'applique est la suivante : la redéclaration d'une méthode d'instance dans une hiérarchie a pour effet de **masquer** la méthode du parent et de **propager** cette nouvelle méthode à l'ensemble de ses enfants.

```
class A         { public void Aff()   { WriteLine("Bonjour"); } }
class B : A     { }
class C : B     { public void Aff()   { WriteLine("Coucou"); } }
```

```
class D : C        { }
```

Pour créer une collection d'objets, on stocke leurs références dans une structure de données. Prenons un tableau par exemple. En créant ce tableau, la syntaxe nous impose de choisir un type unique pour toutes les références qu'il va stocker. Par bon sens, nous choisissons comme type une référence vers l'ancêtre de la hiérarchie qui nous intéresse :

```
A[] Liste = new A[4] ;
Liste[0] = new A();
Liste[1] = new B() ;
Liste[2] = new C() ;
Liste[3] = new D() ;

foreach(A m in Liste)
        m.Aff();          // résultats : « Bonjour »  « Bonjour »  « Bonjour »  « Bonjour »
```

Cette approche pose problème. Les objets sont gérés comme s'ils étaient TOUS de type A alors que ce n'est pas le cas. En l'absence de toute syntaxe particulière, le langage considère que le type de l'objet référencé est donné par le type de la référence. Cette règle est la source de ce comportement pathogène.

L'approche correcte :

Le polymorphisme d'héritage est garanti si l'on utilise une syntaxe particulière. La première classe où la méthode est déclarée pour la première fois doit porter le qualificatif **virtual** et toutes les classes enfants modifiant son comportement doivent précéder sa déclaration par le qualificatif **override**.

```
class A         { public virtual     void Aff()      { WriteLine("Bonjour");  } }
class B : A     { }
class C : B     { public override    void Aff()      { WriteLine("Coucou");  } }
class D : C     { }

A[] Liste = new A[4] ;
Liste[0] = new A();
Liste[1] = new B() ;
Liste[2] = new C() ;
Liste[3] = new D() ;

foreach(A m in Liste)
        m.Aff();          // Résultats : « Bonjour» « Bonjour »  « Coucou »  « Coucou »
```

Rem 1 : le mot clef override est obligatoire pour redéfinir une fonction virtuelle.
Rem 2 : on peut utiliser le mot clef final pour empêcher toute redéfinition.

Le polymorphisme d'héritage est une facilité pour le programmeur mais elle a un coût pour le programme. En effet, le type de l'objet n'est pas connu à la compilation mais déterminé à l'exécution. En effet, suivant les actions de votre joueur, la liste d'objets peut contenir uniquement des voitures ou seulement des avions… Le langage doit donc effectuer un test pendant l'exécution pour connaître le type exact de l'objet référencé. Pour des fonctions effectuant beaucoup de calculs, comme un affichage graphique, ce coût est insignifiant. Par contre, faire du polymorphisme d'héritage sur des fonctions effectuant très peu de calculs peut engendrer un surcoût d'opérations.

### 4.3 Chainage des redéfinitions

Il est toujours possible d'appeler la fonction qui a été redéfinie à partir de la redéfinition avec le mot clé base. Cela permet de construire des chaines de fonctions dans le même principe que le chainage des constructeurs. Par exemple, supposons que la classe Voiture dispose d'une fonction TestAllumage() qui vérifie que tous les composants électriques (ordinateur de bord, ABS, ESP, air bag, ...) de la voiture fonctionnent bien. Dans le cas d'un Cabriolet qui ajoute un toit amovible, il faudra contrôler l'ensemble des systèmes d'une voiture simple et le système qui rentre/sort le toit amovible.

```
class Cabriolet : Voiture
{
        public override void TestAllumage()
        {
                base() ;                // test des éléments de la classe Voiture
                                        // par appel a TestAllumage() de Voiture
                testToitAmovible() ;    // test des éléments spécifiques au Cabriolet
                ...
        }
}
```

# 5 Méthodes de modélisation objet

L'agrégation consiste à dire qu'un objet intègre dans ses paramètres d'autres objets. Par exemple, une voiture a 4 roues, nous pouvons donc modéliser cette situation en mettant un tableau de 4 objets de type Roue dans un objet Voiture.

L'héritage, lui permet de dire qu'une classe est un cas particulier ou une extension d'une autre classe. La classe enfant possède alors tous les attributs de la classe mère. La question de savoir comment utiliser l'héritage et l'agrégation est délicate et mérite un cours à elle seule. Il s'agit d'une problématique centrale des cours de POO et de Génie Logiciel. Néanmoins, on peut tirer quelques principes généraux :

- l'agrégation correspond à la relation "... a ...". Par exemple "Une voiture a 4 roues".
- l'héritage correspond à la relation "... est ...". Par exemple "Une voiture est un véhicule".

Néanmoins cette approche ne permet pas rigide car une relation "... est ..." peut facilement être transformée en une relation "... a ...". Par exemple "La Renault Fluence est une voiture électrique" devient "La Renault Fluence a un moteur électrique".

Par ailleurs, comment faire lorsqu'une classe semble hériter de plusieurs autres classes. Par exemple : un smartphone est à la fois un téléphone, un lecteur mp3 et un microordinateur.

Une première solution serait de créer une fusion des deux classes Téléphone et Lecteur MP3. On parle alors d'**héritage multiple**. Ce choix fut longtemps plébiscité par le langage C++. Ainsi, la nouvelle classe rassemble tous les membres des classes parentes. Une contrainte apparaît au niveau des conflits entre les membres de même nom. Il faudra alors les discerner explicitement. Tous les membres des classes parents sont conservés. Un problème apparaît réellement si des membres ont le même rôle. Ils deviennent alors redondants ce qui peut être source de confusion : par exemple, un téléphone et un lecteur

MP3 possèdent respectivement une batterie, un objet SmartPhone possède-t-il deux batteries ? Si l'on veut conserver une seule batterie, laquelle faut-il conserver ?

L'héritage multiple peut poser des problèmes délicats, il y a donc controverse sur le fait de savoir si ses avantages surpassent ses inconvénients. Pour cette raison, il a été retiré des langages modernes (Java, C#). Face à cette situation, il faut essayer d'utiliser une agrégation ou éventuellement si cela est possible des interfaces.

## 5.1 Interface

Dans certains cas, on cherche seulement à hériter d'un ensemble de fonctionnalités, d'un type de service ou d'un comportement. Par exemple, la climatisation de votre voiture SKUDA peut être vue comme un service fournissant obligatoirement l'ensemble des fonctions suivantes :

```
int   Temperature_actuelle() ;
void Activer() ;
void Arreter() ;
void Vitesse(int v) ;
```

En fait, nous remarquons que le service « climatisation » est indépendant de la SKUDA. Il peut exister dans toute voiture, voir dans une chambre ou dans un avion. Dans cet exemple, la climatisation n'a pas de variables propres et dans ce cas précis, elle peut être décrite à partir d'une **interface** en C#. Si vous voulez rajouter un paramètre interne à la climatisation comme l'état de son filtre, sa puissance de refroidissement… il faudra alors créer une classe et passer par une agrégation.

Une interface définit un comportement devant être implémenté par les classes déclarant cette interface. On peut voir une interface comme une norme servant à décrire tous les services devant être proposés par une classe pour adhérer à cette norme.

Un des exemples les plus courants dans .net est le comportement « énumérable », c'est-à-dire la capacité à accéder aux éléments d'un objet l'un après l'autre. On peut considérer qu'un Triangle est « énumérable » en fournissant chacun de ses trois sommets l'un après l'autre. Le C# définit justement l'instruction foreach pour parcourir ce type d'objet. Cette interface est définie dans System.Collections.IEnumerable.

Une interface est définie de manière similaire à une classe mais elle ne contient que des prototypes de fonctions sans qualificateur de visibilité. C'est la classe implémentant l'interface qui choisit leurs visibilités. Une classe peut hériter de plusieurs interfaces. Elle doit pour cela implémenter le code de chacune des fonctions proposées par les différentes interfaces.

Quelques repères :

- Interface et classe abstraite ne sont pas équivalentes. Une interface n'implémente aucune méthode. Une classe abstraite peut ne pas implémenter certaines de ces méthodes.
- Une classe peut implémenter plusieurs interfaces.
- Une classe a au plus une super-classe.
- Des classes non liées hiérarchiquement peuvent implémenter une même interface.

Les interfaces en C# disposent aussi des facilités de l'héritage et comme une interface n'a pas de champs interne, l'héritage multiples est cette fois autorisé. L'interface fille n'implémente pas ces interfaces car ce serait contraire au principe des interfaces. La classe qui implémente l'interface fille devra par contre implémenter toutes les fonctions décrite par la hiérarchie des interfaces.

```
class Enfant { ... }
```

```
interface     IBonjour        {  void DireBonjour() ;  }
interface     IAurevoir       {  void DireAurevoir() ; }
interface     IPolitesse : IBonjour, IAurevoir        {}

class EnfantPoli : IPolitesse
{
        public void DireBonjour() {...}
        public void DireAurevoir() {...}
}
```

Remarque : comme les interfaces sont proches des classes, on peut aussi par le même mécanisme déclarer une référence sur une interface :

```
EnfantPoli Paul = new EnfantPoli() ;
IPolitesse  iP = Paul ;                         // correct Paul implémente IPolitesse
iP.DitBonjour() ;
```

# 6. Cas particuliers

## 6.1 Classe abstraite

Considérons un logiciel de dessin vectoriel. Ce logiciel de dessin doit être capable de manipuler différentes objets : des ellipses, des rectangles... Il va donc être utile de les regrouper dans une même hiérarchie avec un parent commun : ObjetVectoriel. Il sera naturel d'utiliser une méthode polymorphe Draw() pour ces objets. Cependant la classe ObjetVectoriel ne contient pas d'information géométrique et elle ne sait pas se dessiner. Deux options s'offrent à vous, lorsque vous créez la méthode virtuelle Draw de ObjetVectoriel :

- Vous déclarez la fonction et vous lui associez un corps vide : {}. Cette solution « fonctionne » mais amène quelques risques. En effet, l'utilisateur (un autre programmeur que vous), peut alors instancier à tort un ObjetVectoriel alors que cet objet est une coquille vide. Cela n'apportera pas vraiment de bugs, par contre, l'utilisateur va s'énerver pour faire marcher cet objet et il n'y arrivera pas. Pensez au temps perdu lorsque vous avez essayé d'instancier la classe stream.
- Vous être un programmeur sympathique et pensez au bien être de vos collaborateurs. Vous ajoutez un qualificatif indiquant que cette classe n'est pas utilisable et vous empêchez toute tentative d'instanciation. En bonus, vous n'êtes pas obligé de programmer des méthodes vides. Une telle classe est qualifiée d'**abstraite**.

Lorsque vous êtes surs que le parent commun ne doit pas être instancié, il est fortement recommandé de construire une classe abstraite et voici la syntaxe :

```
abstract class Shape
{
        protected string nom ;
        public abstract void Draw() ;          // méthode abstraite : pas de block {}
}

class Triangle : Shape
{
```

```
        protected Point p1, p2, p3 ;
        public override void Draw()  { ... }
}
```

Remarque : une méthode abstraite est automatiquement virtuelle. Par contre, il faut utiliser le qualificatif override lors de sa définition dans les sous-classes.

Remarque 2 : une classe abstraite ne peut pas être instanciée, ce qui ne l'empêche pas de posséder un ou plusieurs constructeurs appelés par le mécanisme de chaînage des constructeurs.

Remarque 3 : une classe non abstraite fille d'une classe abstraite doit implémenter toutes les méthodes abstraites de sa classe mère.

Remarque 4 : vous pouvez créer une sous-classe abstraite d'une classe abstraite. Il faudra alors qualifier explicitement cette sous classe d'abstraite.

## 6.2 Une classe mère commune

En C#, toute classe sans parent explicite hérite implicitement de la classe mère Object. La class Object est ainsi la racine de la hiérarchie de toutes les classes, les vôtres comme celles de .net. La classe Object déclare cinq méthodes de base accessibles à partir de toutes les instances. Nous détaillerons leurs rôles dans une autre partie du cours.

| Méthode | Action |
|---|---|
| public bool Equals(Object) | Détermine si l'objet spécifié est égal à l'objet en cours |
| public void Finalize() | Libère les ressources avant que l'objet soit détruit par le ramasse miettes |
| public int GetHashCode() | Retourne le code de hashage de l'objet |
| public Type GetType() | Retourne le type de l'objet |
| public Object MemberWiseClone() | Créé une copie superficielle de l'objet |
| public string ToString() | Retourne une chaîne qui représente l'objet |

## 6.3 Identification dynamique des types de classes

On peut trouver des collections dans lesquelles sont stockés toutes sortes d'objets : des véhicules, des téléviseurs, des personnes… Cela est techniquement possible puisque nous venons de voir que tout objet en C# appartient implicitement à la même famille dont la racine est Object. Si certains objets appartiennent à une même hiérarchie (Voiture), d'autres sont sans rapport entre eux. Dans cette situation, il peut être utile de déterminer le type d'un objet à la volée, en C# le mot clef **is** fournit cette facilité.

```
class A { ... }
class B : A { ... }
class C { ... }

A a = new B();
if ( a is A )     WriteLine("Je suis de type A");     // => Je suis de type A
if ( a is B )     WriteLine("Je suis de type B");     // => Je suis de type B
if ( a is C )     WriteLine("Je suis de type C");     // - rien
```

Une instance de la classe B est considérée comme étant de type A car elle est une descendante de A. Si nous prenons un exemple d'utilisation, nous obtenons :

```
foreach (Object o in Liste)
```

```
{
        if ( o is Voiture )
        {
                Voiture v = (Voiture) o ;
                v. Demarre() ;
        }
}
```

Quelle que soit le modèle de voiture (C3, Corsa, SCUDA), nous déterminons si o est un objet héritant de la classe Voiture. Nous effectuons une conversion explicite et nous avons ainsi accès à toutes les méthodes polymorphes des Voitures.

On peut utiliser une syntaxe équivalente avec le mot clef **as**. Ce mot clef effectue une tentative de transtypage et si celle-ci échoue le résultat est null.

```
Voiture v = o as B ;
if ( v != null )    v.Demarre() ;
```