THEME 4
Héritage

CONCEPT

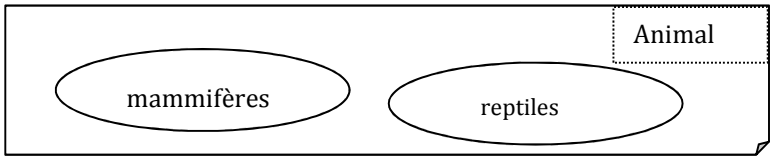
L'héritage permet de construire de nouvelles classes (classe fille) dérivant d'une classe existante (classe mère).La classe dérivée(fille) hérite automatiquement des données et du savoir-faire de la classe mère. Elle peut bien sûr ajouter de nouvelles données et de nouvelles fonctionnalités qui lui sont propres. Ce mécanisme est appelé spécialisation. Un ensemble de classes obtenues par dérivation successive forme une hiérarchie de classe.

La notion d'héritage et de dérivation est à rapprocher de la notion d'ensemble. Cette vision ensembliste permet de définir des sur-ensembles et des sous-ensembles. La classe fille est un sous-ensemble de la classe mère.

Quelques exemples pour fixer les idées :

Exemple de classe de base
Classe Individu
Classe Véhicule
Classe Animal

En prenant l'exemple de la classe animal graphiquement nous pourrions représenter sous forme d'ensemble :



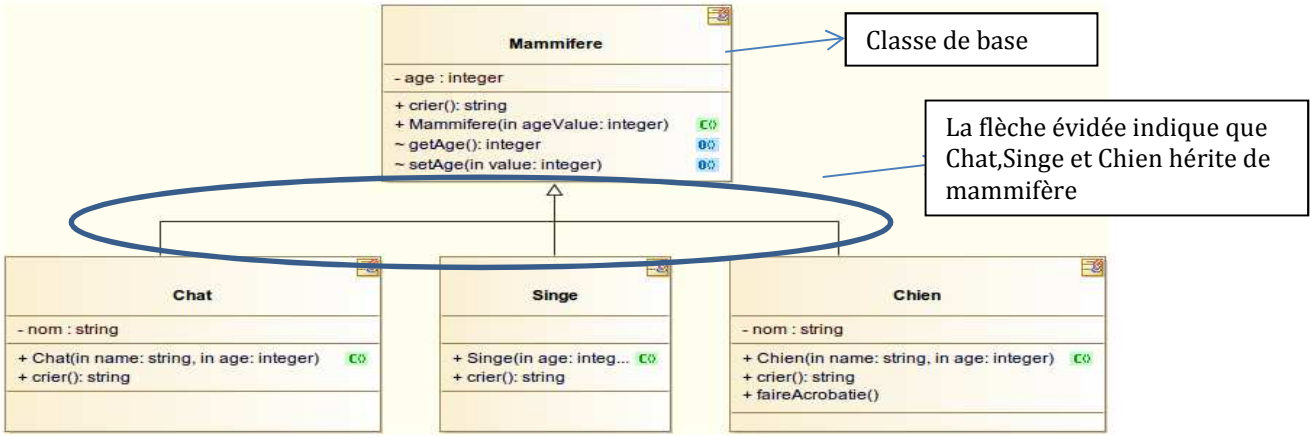
Que peut-on en conclure :

un mammifère est un animal
un animal n'est pas forcément un mammifère
un reptile est un animal
les animaux possèdent des caractéristiques communes (poids, âge ….) et des fonctions communes (manger, crier etc…)

Les mammifères, en plus des caractéristiques communes à tous les animaux, possèdent leurs propres caractéristiques ainsi que des méthodes propres.

L'héritage exprime donc une relation du type « est un ».
Par exemple : un chien « est un » mammifère. Une voiture « est un » véhicule.

Représentation graphique en UML:



L'inclusion, c'est-à-dire lorsqu'une classe réutilise pour sa définition une autre classe (exemple d'une classe rectangle utilisant une classe Point), exprime une *relation « possède un » ou « à un ».*
Par exemple : Un rectangle « possède des » points. Une voiture « à un » châssis.

MIS EN ŒUVRE

En C#, comme en Java d'ailleurs, une classe ne peut hériter que **d'une autre classe**. L'héritage multiple est interdit (autorisé en C++).
La syntaxe pour exprimer une dérivation est la suivante **: class B : A**

où A est apellée classe de base, "superclasse",classe mère…
où B est appellée classe dérivée, classe spécialisée, classe fille….

Un exemple commenté (lisez bien !!) pour comprendre :

```
enum RaceChien { INCONNU, GOLDEN, CAIRN, DANDIE, SHETLAND, DOBERMAN, LAB };
public class Mammifere
{
    protected int poids;
    private int age
    // propriété
    public int Poids {
        get
        {
            return poids;
        }
        set
        {
            poids = value;
        }
    }
    // constructeur de la classe de base
    public Mammifere(int age, int poids)
    {
        // Cette affichage est à but pédagogique uniquement
        // Objectif: voir la séquence d'appel des ctor
        Console.WriteLine("Ctor Mammifere");
        this.age = age; // accès direct au champs (le concepteur n'a pas voulu de modification après crétaion de l'objet)
        Poids = poids;// accès via propriété
    }
    public void Crier() { Console.WriteLine("Raaaa"); }
    public void Dormir() { Console.WriteLine("Zzzzz"); }
    public override string ToString()
    {   // un mammifere est capable d'afficher son age son poids
        return ("age " + age+ ", poids " + Poids);
    }
}
// un chien EST un mammifere
class Chien : Mammifere
{
    //
    public RaceChien Race { get; set; }
    public string Nom { get; set; }
    // ctor1
    public Chien(int age, int poids)
        : base(age, poids)
    {   // Cette affichage est à but pédagogique uniquement
        // Objectif: voir la séquence d'appel des ctor
        Console.WriteLine("Ctor Chien age,poids");
        Race = RaceChien.INCONNU;
    }
    // ctor2
    public Chien(RaceChien race, int age, int poids)
        : base(age, poids)
    {
        Console.WriteLine("Ctor Chien race,age,poids");
        Race = race;
    }
    // ctor3
    public Chien(RaceChien race,String nom, int age, int poids)
        : this(race,age, poids)
    {
        Console.WriteLine("Ctor Chien race,nom,age,poids");
        Nom = nom;
    }
    public override string ToString()
    {
        // base.ToString() appelle le ToString de la classe de base (Mammifere donc)
        // et je complete avec les infos spécifiques du Chien
        string retourToStringdeLaClasseMammifere = base.ToString();
        return ("Chien " + retourToStringdeLaClasseMammifere + " ,Race " + Race);
    }

    public void changeAttributsPoidsEtAge(int ageChien, int poidsChien)
    {
        poids = poidsChien; // autorisé car protected mais déconseillé : on préfère Poids=poidsChien;
        // KO age = ageChien;   car Age est private
    }    };
```

private : Chien n'aura pas d'accès direct à Age
protected : pouvoir de filiation accordée => Chien aura accès direct car *fils de*

Classe de base (ou classe mère)

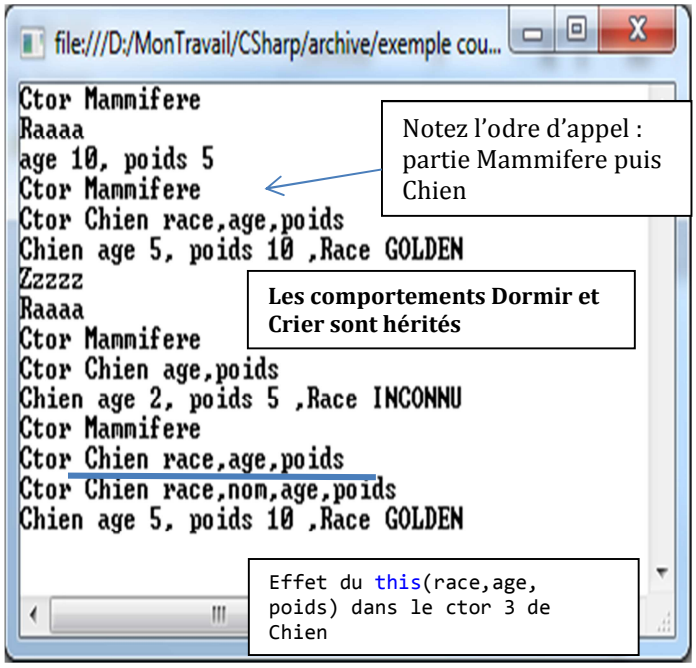
`base`: appelle le ctor de Mammifere ayant la même signature. Permet d'initialiser la partie Mammifère du Chien

`this`: appelle le ctor de Chien ayant la même signature. Evite des réécritures de code inutile (

`base.ToString();`: appelle le ToString de la classe Mere On completera ensuite avec les infos spécifiques de Chien

Accès direct à poids car protected
Age est interdit car private

```
    class Program
    {
        static void Main(string[] args)
        {
            Mammifere unMamifere = new
Mammifere(10, 5);
            unMamifere.Crier();
            Console.WriteLine(unMamifere);
            Chien unChien =
              new Chien(RaceChien.GOLDEN, 5, 10);
            Console.WriteLine(unChien);
            unChien.Dormir();
            unChien.Crier();
            Chien autreChien = new Chien(2, 1);

autreChien.changeAttributsPoidsEtAge(4, 5);
// probleme car l'age de Mammifere est private et
pas  de setter pour l'age
            Console.WriteLine(autreChien);

            Console.ReadKey();
        }
    }
```



Commentaire :

1-la classe chien dérive de la classe mammifère. Ceci implique que les méthodes membres de la classe chien peuvent accéder aux champs et propriétés de la classe mammifère en fonction du niveau de visibilité définit par le programmeur:

- ✓ public : tout le monde
- ✓ protected : RESCTRICTIONS AUX CLASSES DERIVEES
  - o `protected int poids`
- ✓ private: RESTRICTION A LA CLASSE DE BASE
  - o `private int age`

2-L'idée de base de la dérivation est de pouvoir partager certaines propriétés et certaines méthodes pour une hierarchie de classes. La classe de base est le "dénomitateur" commun d'un ensemble de classe dérivée:
`Exemple:  unChien.Dormir(); et unChien.Crier`
Cependant nous remarquons ici que l'action unChien.Crier ne produit pas une action atisfaisante. La méthode Crier peut être "spécialisée" et adaptée pour refléter le cri d"un Chien: "*Ouarfffff*"

3-La création d"un objet dérivée, ici un chien, implique une séquence d"initialisation logique:

- Appel du ctor de chien
- Appel du ctor de mammifère :*implicite => appel de Mammifere() ou explicite => Chien(yyyyy):base(xxxxxx)*
- Exécution du code du ctor de mammifère

4-  **Cette exemple d'héritage est donné à titre pédagogique car il est simple. Cependant il est discutable et met en avant certaines faiblesses de l'héritage**.
En effet on devine en fonction du cahier des charges une explosion des classes dérivées de Mammifères. Il sera compliqué de maintenir de manière efficace le code avec la multiplication des mammifères spécifiques (baleines, guepard, girafe etc…)
Une solution possible serait de rajouter un dictionnaire contenant les particularités de chaque mammifère (pas de classes dérivées donc) avec des couples cles/valeurs comme :
« NomEspeces »/ « Baleine »
« age »/ « 20 »
« poids »/ « 10 »
« nombrePattes »/ « 0 »
De manière générale les problématiques sont récurrentes et au fil des années les informaticiens ont dégagé des précepts de programmation efficace et robuste (exemple : SOLID, GRASP ..) ainsi que des patrons (canevas, guide) de bonnes conceptions à des problématiques récurrentes (design pattern : référence gang of four)

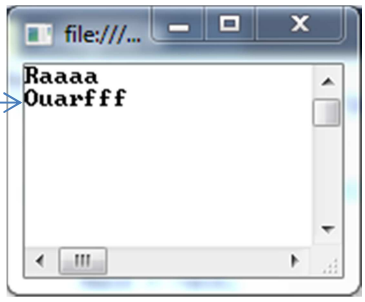
LA SPECIALISATION

Nous avons vu que la classe de base peut intégrer des méthodes communes à toute les classe dérivées. Par exemple la méthode crier de la classe Mammifere peut être vue comme une fonction générique commune à toutes les espèces.
Cependant le cri d'un chat ou d'un chien n'est pas le même évidemment. Il faut donc substituer la fonction crier de base fournie par la classe mère, avec la fonction crier spécifique des classes filles Chat et Chien.
Ce mécanisme est appelé aussi substitution de méthode.

Un exemple :

```
enum RACE { INCONNU, GOLDEN, CAIRN, DANDIE, SHETLAND, DOBERMAN, LAB };

    public class Mammifere
    {
        // propriétés automatique
        public int Age { get; set; }
        // constructeurs
        public Mammifere(int age)
        {
            Age = age;
        }
        //methodes
        public void Crier() { Console.WriteLine("Raaaa"); }
    }

    class Chien : Mammifere
    {  // propiété auto
        public RACE Race { get; set; }
        // ctor 1
        public Chien(int age, RACE race)
            : base(age)
        {
            Race = race;
        }
        // specialisation
        // new est optionnel => evite warning compilateur
        //new précise l'intention du concepteur de spécialisé la méthode
        public new void Crier()
        { Console.WriteLine("Ouarfff"); }
    };


    class specialise
    {
        static void Main(string[] args)
        {
            Mammifere unMamifere = new Mammifere(10);
            unMamifere.Crier();
            Chien unChien = new Chien(5, RACE.GOLDEN);
            unChien.Crier();
            Console.ReadKey();
        }
    }
}
```

Surchage de méthode



CONVERTION DE TYPE

Convertion implicite et explicite
Le langage prévoit la possibilité de référencer :

- ✓ un objet de type classe dérivée à partir d'une référence de type classe de Base

```
Mammifere mam =new Chien(5, RACE.GOLDEN); //Upcast
```

Cette convertion de type est appelée "UpCast". **Cette convertion réussit toujours à condition de respecter la filiation des classes (convertion dérivée vers Base- ie classe fille vers classe mère-). En effet compte tenu de la définition TOUS les chiens sont des mammifères**

- ✓ un objet de type classe de Base à partir d'une référence de type classe dérivée

```
Mammifere unMammifere2 = new Mammifere(5);
Chien autreChien2 = (Chien)unMammifere2; // Downcast
```

Cette convertion de type est appelée "DownCasting". **Cette convertion NE réussie PAS toujours**. Dans ce cas une exeception est levée à l'éxecution (runtime) car **un mammifère n'est pas forcément un Chien, l'inverse par contre étant toujours vrai!**
Cette convertion n'est donc pas de type "safe-fail".

OPERATEUR IS ET AS

Ces opérateurs sont utilisés pour des convertions de type safe-fail, c'est-à-dire sans levée d'exception au runtime.

L'opérateur as permet de réaliser une convertion de downcasting. La référence résultat est positionnée à null en cas d'échec.

L'opérateur is permet de réaliser un test de conversion. Le resultat, un boolean , est utilisé pour réaliser effectivement la convertion en cas de succcès.

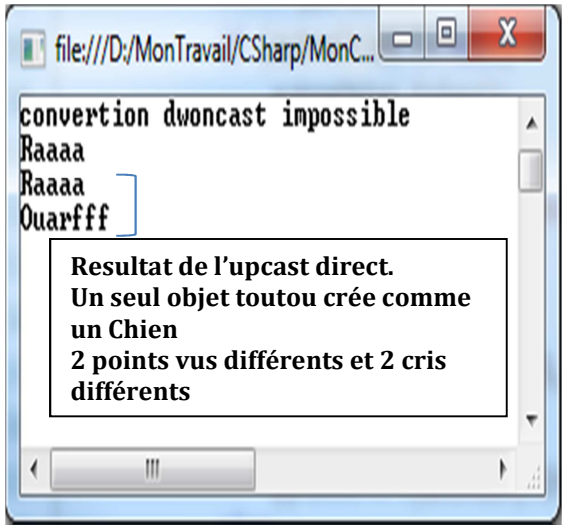
Remarque : si votre code contient beaucoup de test de ce type c'est le signe d'une mauvaise architecture logicielle (violation du L des principes SOLID : Substitution de Liskov ou Liskov Substitution Principle)Il est temps de repenser votre code et de le refactoriser

```
// downcast avec as
Mammifere unMammifere = new Mammifere(5);
Chien autreChien = unMammifere as Chien;
if (autreChien != null)
autreChien.Crier();
else
Console.WriteLine("convertion dwoncast impossible");

// upcast avec is
Chien unChien =new Chien(5, RACE.GOLDEN);
if (unChien is Mammifere)
((Mammifere)unChien).Crier();
else
Console.WriteLine("convertion upcast impossible");
```

L'exemple complet

```
enum RACE { INCONNU, GOLDEN, CAIRN, DANDIE, SHETLAND, DOBERMAN, LAB };

    public class Mammifere
    {
        // propriétés automatique
        public int Age { get; set; }
        // constructeurs
        public Mammifere(int age)
        {
            Age = age;
        }
        //methodes
        public void Crier() { Console.WriteLine("Raaaa"); }
    }

    class Chien : Mammifere
    {  // propiété auto
        public RACE Race { get; set; }
        // ctor 1
        public Chien(int age, RACE race)
            : base(age)
        {
            Race = race;
        }
        // specialisation
        // new est optionnel => evite warning compilateur
        //new précise l'intention du concepteur de spécialisé la méthode
        public new void Crier()
```

```
        { Console.WriteLine("Ouarfff"); }
};


class convert
    {
        static void Main(string[] args)
        {


            // downcast avec as
            Mammifere unMammifere = new Mammifere(5);
            Chien autreChien = unMammifere as Chien;
            if (autreChien != null)
                autreChien.Crier();
            else
                Console.WriteLine("convertion dwoncast
                                  impossible");
            // upcast avec is
            Chien unChien =new Chien(5, RACE.GOLDEN);
            if (unChien is Mammifere)
                ((Mammifere)unChien).Crier();
            else
                Console.WriteLine("convertion upcast
                                  impossible");

            // UPCAST DIRECT
            Chien toutou = new Chien(5, RACE.GOLDEN);
            Mammifere mam=toutou;  //Upcast
            mam.Crier(); // Raaaa .. le chien est vu comme un mammifere
            toutou.Crier(); //Ouarfff ... le chien est est vu comme un chien

            // DOWNCAST DIRECT
            Mammifere unMammifere2 = new Mammifere(5);
            // KO EXCEPTION Chien autreChien2 = (Chien)unMammifere2; // Downcast
            // KO EXCEPTION  autreChien2.Crier();

            Console.ReadKey();
        }
    }
}
```



Remarque : le résultat obtenu lors de l'upcast peut sembler curieux puisque pour un même objet nous obtenons 2 résultats différents.Le chapitre suivant explique comment obtenir le même résultat en manipulant un Chien comme un Mammifère (**concept de polymorphisme**). C'est le concepteur, en fonction de son intenetion, qui décide donc de la stratégie à adopter.

METHODE VIRTUELLE

Une méthode est qualifiée de virtuelle lorsque le mot-clé `virtual` est en **préfixe** de sa déclaration. Le qualificatif virtual est associé au qualificatif `override` qui préfixe la *méthode substituée dans la classe* fille (exemple complet page suivante). L'objectif d'une telle déclaration virtuelle est de pouvoir appeler la méthode de substitution adéquate à partir d'une référence *de base pointant vers des objets de type classe dérivée (opération de Upcast)* : ce mécanisme est apellé polymorphisme (voir paragraphe suivant) :

Exemple :

```
Mammifere mam = new Chien(); // Notre chien est manipulé comme un mammifère
mam.Crier(); // JE VEUX Ouarfff
```

Même si le chien est techniquement manipulé comme un mammifère la vraie nature de l'objet au moment de l'éxécution de la méthode Crier() est retrouvée. Crier s'applique donc comme un Chien et non comme un Mammifère.

Nous verrons sous peu **l'intérêt de ce mécanisme très puissant.**

```
using System;
using System.Text;

namespace virtualexmple
```

```
{
```

```
 public class Mammifere
 {
//methodes
public virtual void Crier() { Console.WriteLine("Raaaa"); }
public virtual void Sous_Ordre_Especes() { Console.WriteLine("Mammifere"); }
public void NbreDePattes() { Console.WriteLine("Inconnu"); }
}

class Chien : Mammifere
 {
    public override void Crier() { Console.WriteLine("Ouarfff"); }
    public void Sous_Ordre_Especes() { Console.WriteLine("Canidés"); }
    public void NbreDePattes() { Console.WriteLine("Chien: 4 pattes"); }
 };

class Chat : Mammifere
{
    public override void Crier() { Console.WriteLine("Miaouu"); }
    public override void Sous_Ordre_Especes() { Console.WriteLine("Félidés"); }
    public  void NbreDePattes() { Console.WriteLine("Chat: 4 pattes"); }
};

class virtualExemple
{
static void Main(string[] args)
{


Chien unChat = new Chat(); KO impossible
Mammifere mam1 = new Chat(); // Upcast OK
mam1.Crier();
mam1.Sous_Ordre_Especes();
mam1.NbreDePattes();

mam1 = new Chien(); // mam1 pointe vers un chien maintenant Upcast OK
mam1.Crier();
mam1.Sous_Ordre_Especes();
mam1.NbreDePattes();

// On regroupe tout le monde sous le type de base
List<Mammifere> maMenagerie=new  List<Mammifere>() ;
// Je peuple la ménagerie
maMenagerie.Add(new Chat());
maMenagerie.Add(new Chien());
maMenagerie.Add(new Mammifere());
// Je fais crier tous le monde
foreach (Mammifere mam in maMenagerie)
    mam.Crier();
Console.ReadKey();}
}}
```

Miaouu - **résultat de mam1.Crier(); mam1 bien qu'étant de type mammifère"comprend" grâce au- couple virtual/override que l'objet pointée est un Chat**

Félidés- **résultat de mam1.Sous_Ordre_Especes(); même remarque**

Inconnu - **résultat de mam1.NbreDePattes();c'est la méthode de la classe mammifère qui a été appelé!virtual ABSENT sur NbreDePattes (classe Mammifere)**

Ouarfff- **résultat de mam1.Crier(); effet de virtuarl/override en place**

Mammifere- **résultat** `mam1.Sous_Ordre_Especes();` **override absent dans la classe dérivée**

Inconnu -**résultat de mam1.NbreDePattes();c'est la méthode de la classe mammifère qui a été appelé!virtual ABSENT sur NbreDePattes (classe Mammifere)**

POLYMORPHISME EN ACTION.
Un tableau d'objets de type différent mais ayant un ancêtre commun.
La « bonne » méthode est appelée grâce au couple VIRTUAL/OVERRIDE

Plusieurs objets de natures différentes peuvent avoir des fonctionnalités similaires mais implémentées différemment au sein de chaque objet (c'est la spécialisation).
Le cas précédent en est un exemple : tous les mammifères ont un cri mais qui diffèrent finalement en fonction de chaque espèce.
Le polymorphisme (« qui possède plusieurs formes »), permet de donner un même nom à chacune de ces actions. La bonne action sera sélectionnée contextuellement en fonction de la classe d'objets à laquelle elle s'applique.

L'intérêt de cette approche est de pouvoir unifier un traitement appliqué à des objets de nature différente:

En reprenant l'exemple précédent des mammifères:

```
// On regroupe tout le monde sous le type de base
List<Mammifere> maMenagerie=new  List<Mammifere>() ;
// Je peuple la ménagerie
maMenagerie.Add(new Chat());
maMenagerie.Add(new Chien());
maMenagerie.Add(new Mammifere()
// Je fais crier tous le monde
foreach (Mammifere mam in maMenagerie)
    mam.Crier(); ⇦ TRAITEMENT DE BASE APPLIQUE A DES OBJETS DIFFERENTS( MAIS
                 APPARTENANT A LA MÊME CLASSE DE BASE)
```

```
Miaouu
Ouarfff
Raaaa
```

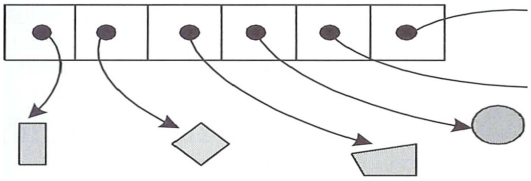
Exemple d'application polymorphique

Prenons l'exemple d'une application manipulant un ensemble d'objets graphiques.Chacun de ces objets possède une méthode de calcul de surface et de périmètre.
La généralisation du concept d'objets graphiques aboutit à une classe *ObjetGraphique* contenant les fonctions virtuelles surfaces et périmètres.

Il est possible alors de stocker tous les objets manipulés par l'application dans un tableau du type ObjetGraphique.

```
ObjetGraphique [] monTableau =……
monTableau[0]=new Rectangle(…..)
monTableau[1]=new Cercle(…..)
monTableau[2]=new Cercle(…..)
foreach (ObjetGraphique obj in monTableau)
       obj.CalculSurface();
```

Chaque case du tableau est une référence sur des objets différents mais que appartiennent à la même famille.



Grâce à la virtualisation les méthodes de surface et périmètre sont polymorphiques (codes différents en fonction de la nature de l'objet): la fonction adéquate est automatiquement appelée.

Le polymorphisme fournit un niveau d'abstraction élevée : il est possible de demander un objet graphique de calculer son périmètre ou sa surface sans avoir besoin de connaître précisément sa classe réelle.
Les avantages :

-possibilité de définir un corps différent pour une méthodes dans des des classes différentes
- manipulation d'un ensemble d'objets de façon générique sans connaître leur type exact
- possibilité d'ajouter des classes dérivées supplémentaires sans retoucher au code.

LA CLASSE OBJECT

Il s'agit de la classe de base fondamentale parmi toutes les classes du .NET Framework. La plupart des classes que vous manipulez héritent de « object ». Cette classe constitue la racine de la hiérarchie des types.

```
public class Object
{
public Object();
public extern Type GetType();
public virtual bool Equals (object obj);
public static bool Equals (object objA, object
objB);
public static bool ReferenceEquals (object objA,
object objB);
public virtual int GetHashCode();
public virtual string ToString();
protected override void Finalize();
protected extern object MemberwiseClone();
}
```

Commme toutes les classes du .NET Framework sont dérivées de **Object**, les méthodes définies dans la classe **Object** sont disponibles dans la totalité des objets de vos programmes.

Les classes dérivées peuvent substituer certaines de ces méthodes, notamment :

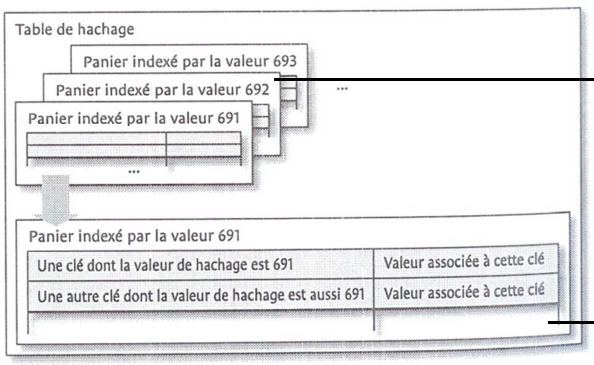
- Equals - Prend en charge les comparaisons entre objets.
- Finalize - Effectue des opérations de nettoyage avant qu'un objet soit automatiquement récupéré.
- GetHashCode - Génère un nombre correspondant à la valeur de l'objet pour prendre en charge l'utilisation d'une table de hachage.
- ToString - Fabrique une chaîne de texte explicite qui décrit une instance de la classe.

Compléments :

ToString : Par défaut la méthode ToString appliquée à une de vos propres classes renverra le type sous forme de chaîne de caractère. Vous comprenez maintenant mieux la syntaxe utilisée depuis le debut pour personnaliser l'affichage de vos classes :

```
public override string ToString()  // spécialisation de la classe
      {     // base.ToString() ici appelle ToString de la classe mère
            // +" Auteur: "+Auteur; et on ajoute les infos spécifiques à la classe fille
          string chaine=base.ToString()+" Auteur: "+Auteur;
           return (chaine);
      }
```

GetHashCode : certains conteneurs (Dictonnary, SortedList ..) utilise une clé dite de hachage, en fait tout simplement un entier, pour indexer les éléments du conteneur. Tous les éléments ayant une même clé sous regroupé dans un même de panier (bucket). La clé de hachage, qui n'est pas unique mais suffisant « distincte » ne doit pas être confondu avec la clé d'identification d'un dictionnaire, qui elle est unique et qui permet de retrouver l'élément dans un panier.



Hash code : identifiant « panier »
Possible d'avoir 2 paniers indexés 691 !!!

Couple <Key,Data> :
Key :clé **unique** d'identification de la valeur
Data : valeur que l'on souhaite stocké

Comment construire le « hash code ».
Afin d'optimiser les algorithmes de recherches et d'indexation, votre méthode GetHashCode doit renvoyer un entier « suffisament unique ». Il n'y pas de règles absolues mais l'examen de vos classes permet de vous donner des pistes.
Exemples divers pour vous aider :

```
public override int GetHashCode()
{
return _unChampInt.GetHashCode();
}
```

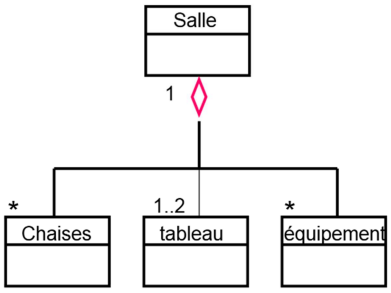
```
-------------------------------------------------------
public override int GetHashCode (Customer obj)
{
return (obj.prenom + ";" + obj.nom).GetHashCode();
}
-------------------------------------------------------
public override int GetHashCode() {
      return coordx ^ coordy; // OU exclusive
public override int GetHashCode () {
return a.GetHashCode() ^ b.GetHashCode() ^ c.GetHashCode();
-------------------------------------------------------
public override int GetHashCode()
{
return Measure2 * 31 + Measure1; // 31 = un nombre premier
}

-------------------------------------------------------
-------------------------------------------------------
```

COMPOSITION ET AGREGATION

L'héritage n'est pas la seule relation liant les objets entre eux. Nous verrons que la composition et l'aggrégation d'objets sont des techniques très couramment utlisées pour modéliser un problème donné. Il est à noter que les prinicipes de bonne programmation SOLID préconisent l'utilisation de la composition plutôt que l'héritage. Nous verrons plus en détail ces points mais nous pouvons remarquer dès à présent que l'héritage constitue un couplage fort , car non modifiable en cours d'éxécution, entre la classe parent et la classe fille. La classe fille hérite de l'intégralité du comportement de la classe parent et ne peut décider de supprimer tel ou tel attributs ou une des méthodes héritées.

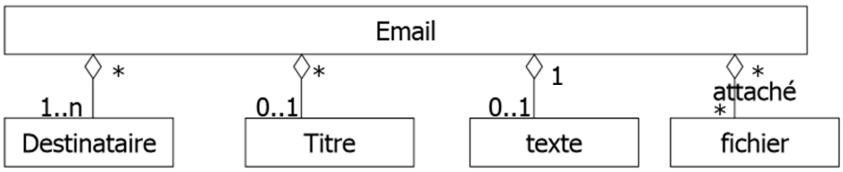
AGREGATION

L'agrégation et la composition expriment un couplage fort et une relation de subordination netre deux types d'objets. Elles représentent une relation de type "ensemble / élément"ou encore « A un ».(par opposition a « EST un »).
La durée de vie de l'objet aggrégé est independant de l'agrégat
Représentation UML
Exemple 1



Une salle est composée de plusieurs chaises, de plusieurs équipements et de 1 ou 2 tableaux.
Le losange évivé indique une aggrégation.
En effet, si la salle est condamné, tous le matériel continue d'exister et sera réaffecté à une autre salle.

Exemple 2

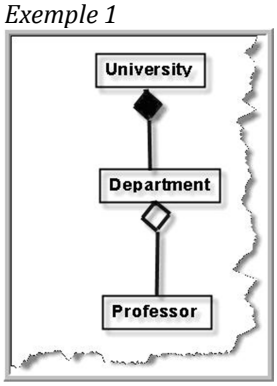


Les différentes parties du mail peuvent vivre sans l'objet mail

COMPOSITION

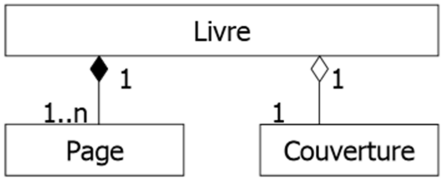
La composition exprime une agrégation forteen rajoutant une contrainte supplémentaire :si l'agrégat est détruit, les composants aussi.
La composition est représentée en UML avec un losange plein.

Exemple 1



Plusiers enseignants sont affectés au département GEII.
Autrement dit le département GEII a plusieurs enseignants.
L'université a plusieurs départements.
Que se passe t'il si l'université de Toulon ferme ses portes.
Les départements cessent d'exister (Composition) et les enseignants seront réaffectés (la vie continue=> agrégation)

Exemple 2



Le livre contient physiquement les pages. Si le livre est détruit, les pages le sont aussi (COMPOSITION)
La couverture de protection peut être récupérée (AGREGATION)

MISE EN OEUVRE

L'agrégation ou la composition sont mises en œuvre par le biais de référénces.
L'agrégat contient un champ attribut qui est une référence vers l'objet agrégé.

Reprise de l'exemple 1 :

```
    public class Professor
    {
        public string FirstName { get; set; }
        public string LastName { get; set; }
    }

class Department
    {
        private List<Professor> employees = new List<Professor>();
        public string Name { get; set; }

        public void Add(Professor employee)
        {
            employees.Add(employee);
        }
        public void Remove(Professor employee)
        {
            employees.Remove(employee);
        }
        public List<Professor> GetProfessorsByDepartement()
        {
            return (employees);
        }


        public Professor this[int index]
```

Mise en place de la relation 1 à plusieurs
Référence de type List

```
        {
            get { return employees[index]; }
        }
    }
```

Mise en place de la relation 1 à plusieurs
Référence de type List

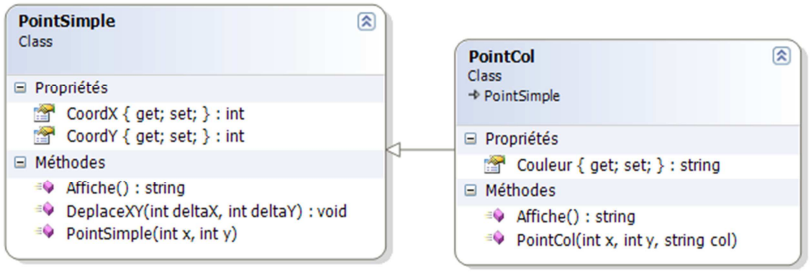
```
class University
    {
     //departments is a composition.
    //departments lifetime controlled by lifetime of Company
    private Dictionary<string, Department> departments = new Dictionary<string,
Department>();

    public University(string[] departmentsToCreate)
    {
        Console.WriteLine("Creating the company and its departments");
        foreach(string departmentToCreate in departmentsToCreate)
        {
            departments.Add(departmentToCreate, new Department{Name = departmentToCreate});
        }
    }

    ~University()  // destructor
    {
        // cleanup statements...
        departments.Clear();
        departments = null;
    }

    public Department this[string departmentName]
    {
        get { return departments[departmentName]; }
    }

    public void ReassignAndCloseDepartmentEmployees(string departmentToCloseName, string
reassignmentDepartmentName)
    {
        Department fromDepartment = this[departmentToCloseName];
        Department toDepartment = this[reassignmentDepartmentName];
        foreach (Professor employee in fromDepartment.GetProfessorsByDepartement())
        {
            toDepartment.Add(employee);
        }
        departments.Remove(departmentToCloseName);
    }
    }
}
```

Destructeur de University
Les départements sont détruits.
Les enseignants restent en vie..Ouff.

Mise en place de la relation 1 à plusieurs
Mise en évidence de l'agrégation. Un département est fermé et les enseignants réaffectés

TRAVAIL PERSONNEL

Ex1 : Une héritage simple qui semble fonctionner….
créer une classe pointcol, dérivé de PointSimple, comme suit :



Les méthodes de la classe de base ne sont pas virtuelles.
Plutôt que d'utiliser de la méthode ToString comme nous faissions habituellement, la méthode Affiche est crée afin d'afficher l'état d'un point. Affiche est spécialisée (surchage sans virtual/override) dans la classe dérivée PointCol de telle manière à renvoyé une chaine avec X,Y + la couleur.
. Vous utiliserez le mot clé *base* pour faire appel à des méthodes de la classe de base (constructeur de PointCol et Affiche)

Tester vos classes en créant un point , un point couleur . Mettre en oeuvre les méthodes.

Faire une opération de downcast : PointSimple unPtcast = unPtCol; et tester Console.WriteLine(unPtcast.Affiche());
Que se passe t'il ?

Créer un tableau de PointSimple où seront stockés des PointSimple mais aussi des PointCol. Balayer le tableau et appliquer à tous les objets la méthode Affiche.
Apporter les modifications nécessaires à votre programme pour que l'affichage garde son sens en fonction de l'objet en cours. (*si vous ne trouvez pas relisez la p8 et 9)*
Comment s'appelle le mécanisme que vous avez misen œuvre ?

Conclusion : Un point de couleur est un point et pour l'instant tout s'est bien passé pour notre modélisation : PointCol hérite de PointSimple.

Ex2: Un héritage qui finalement pose problème
Nous reprenons les classes précédentents et mettons en place une méthode pour comparer des points entre eux.
Rajouter dans PointSimple la méthode `public virtual bool Compare(PointSimple point).`
(rappelez vous que Compare est une méthode d'instance et s'appelle comme ceci unPt.Compare(autrePt) … revoir au besoin l'exercice sur les Villes du thèmes objets)
Pour tester la méthode Compare rajouter la méthode statique

```
  static void Display(bool result)
        {
            if (result == true)
            {
                Console.WriteLine("points identiques");
            }
            else
            {
                Console.WriteLine("points différents");
            }
        }
```

Cette méthode Display pourra être appelée comme suit :
`Display(unPt.Compare(autrePt));`

On souhaite adapter la méthode Compare dans la classe PointCol :

```
public override bool Compare(PointSimple point)
        {
            PointCol ptCol = point as PointCol;
            if (base.Compare(ptCol) && (this.Couleur == ptCol.Couleur))
            {
```

```
            return true;
        }
        else
        {
            return false;
        }
    }
```

Expliquer pourquoi l'opération d'upcast est nécessaire `PointCol ptCol = point as PointCol; ?` Si vous ne voyez pas supprimez cette ligne ? Ou est l'erreur de compilation ?

Testez la comparaison avec des points de couleurs.

Le début des problèmes……..

Maintenant tester de comparer un point de couleur avec un point : pointCol.Compare(point) . Que se passe t'il (ne charchez pas à corriger le problème avec des if)?
Maintenant tester de comparer un point avec un point de couleur : point.Compare(pointCol) . Testez avec des coordonnées différentes et égales. Quelle autre stratégie de comparaison aurait-on pu dans ce cas(non implémenté ici) adopté.
Rajouter des if dans chaque classe pour corriger les problèmes et faire en sorte qu'une comparaison en points de nature différentes renvoie false dans tous les cas.

Conclusion : La modélisation Point de Couleur héritant d'un Point semble naturelles car elle exprime la relation EST-UN. Cependant elle aboutit à des anomalies de fonctionnements. Ces anomalies pourraient être contournées avec de if dans chaque classe.
Que ce passerait-il pour nos if si nous rajoutons deux autres classes : point3D et point3DCouleur.

La situation devient inextricable (des ifs de partout qui seraient à reprendre en cas de rajout d'une nouvelle classe où d'un changement de stratégie de comparaison !!!!).

Cette modélisation est donc erronée. Utiliser sans réfléxion un héritage pour une relation EST-UN est une erreur.
Nous verrons plus tard que notre modélisation viole deux principes des précepts de bonne conception S**OL**ID :

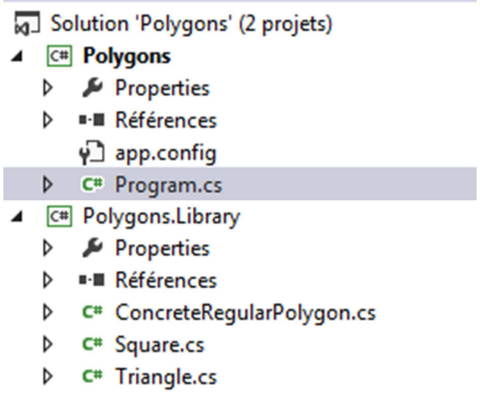
- Principe de substitution de Liskov (LSP)
- Open/closed principle (OCP)

Ex3: Structurer vos solutions

Nous allons crée une application permettant de calculer l'aire de polygones régulers (carre, triangle, octogone etc..)

Etape 1 :.

La solution sera structurée en 2 projets :



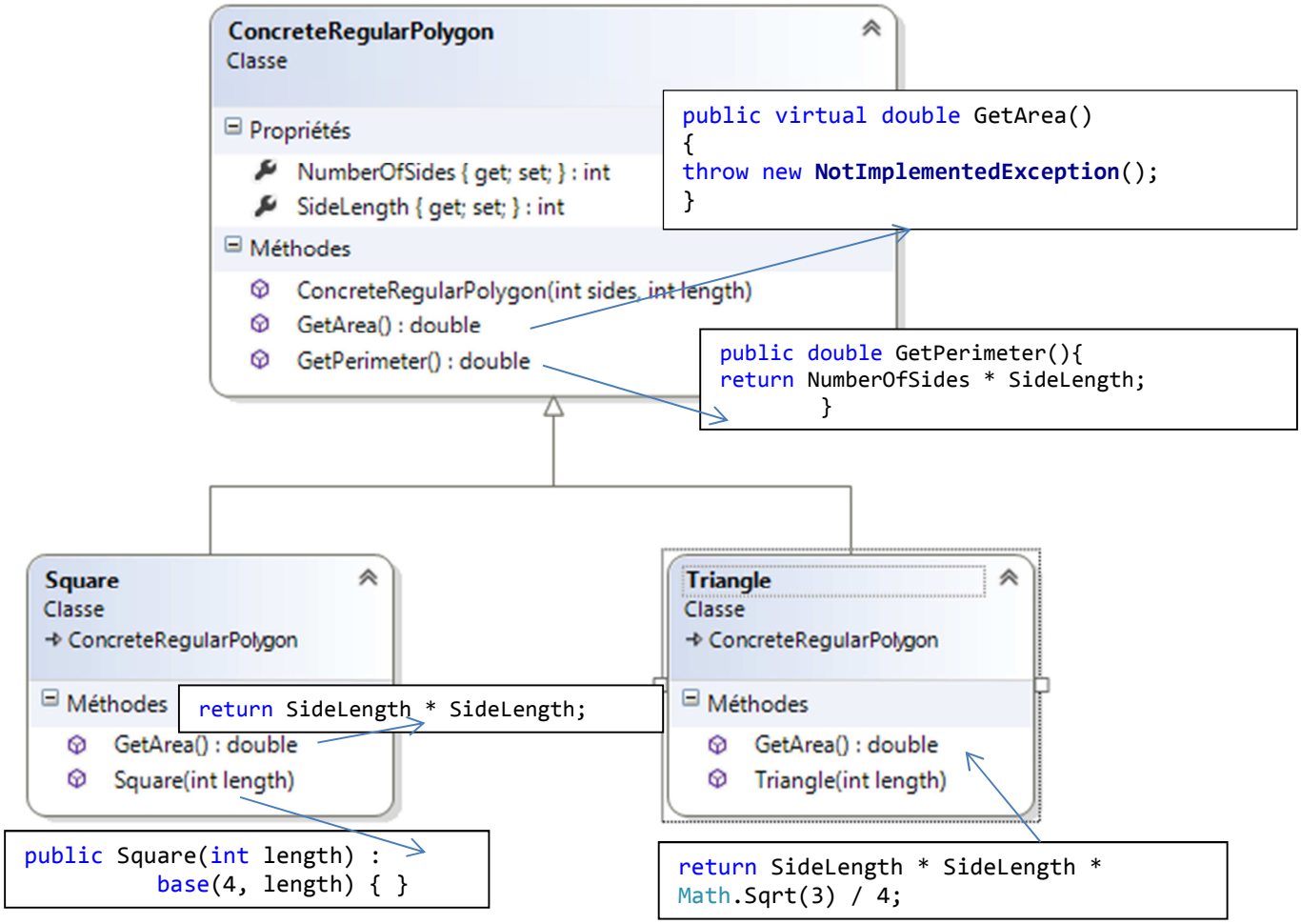
Etape A : créer une solution avec un projet application Console
C'est notre main : la classe CLIENTE program qui utilise les services proposées par ConcreteRegularPolygon , Square et Triangle.

Ne tapez pas de code pour l'instant

Etape B : rajouter un nouveau projet Polygons.Library de type CLASSE.
Renommez le fichier : ConcreteRegularPolygon
Ajoutez 2 autres fichiers de type CLASSES (Square et Triangle)

Etape 2: Mofications de nos classes modèles

Implémenter le schéma d'héritage suivant :



Etape 3: Mofications de la classe cliente (le main)

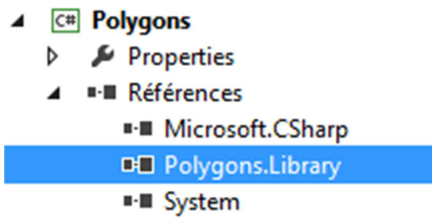
Copiez ce code

```
static void Main(string[] args)
{
    Square square = new Square(5);
    DisplayPolygon("Square", square);

    Console.Read();
}

public static void DisplayPolygon(string polygonType, ConcreteRegularPolygon polygon)
{
Try
{
Console.WriteLine("{0} Number of Sides: {1}", polygonType, polygon.NumberOfSides);
Console.WriteLine("{0} Side Length: {1}", polygonType, polygon.SideLength);
Console.WriteLine("{0} Perimeter: {1}", polygonType, polygon.GetPerimeter());
Console.WriteLine("{0} Area: {1}", polygonType, Math.Round(polygon.GetArea(), 2));
Console.WriteLine();
}
catch (Exception ex)
{
Console.WriteLine("Exception thrown while trying to process {0}:\n   {1}",
    polygonType, ex.GetType().Name);
Console.WriteLine();
}
}
```

Etape 4: Rajout des références (dll +using)

A ce stade vous constatez que la classe cliente (Program +main) ne connaît pas la classe Square par exemple.

Pour corriger le problème :
Dans l'explorateur de solution cliquez droit sur référénce puis ajouter la dll Polygons.Library



Rajourter la ligne suivante dans le fichier Program.cs :

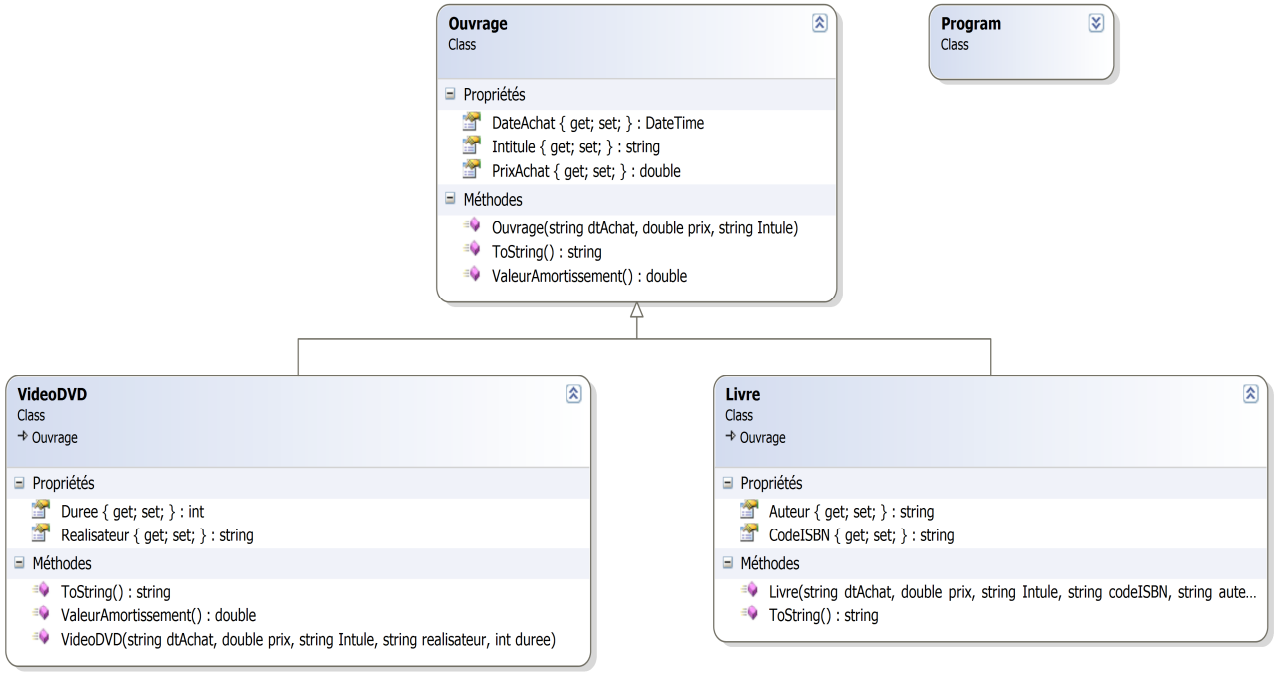
```
using Polygons.Library;
```

Votre solution, hors erreurs de syntaxe, doit compiler.

Etape 5 : Les tests
Tester un carre et un triangle afin de vérifier l'exactitude des calculs de périmètre et d'aire.
Créer une liste de `ConcreteRegularPolygon : List < ConcreteRegularPolygon >`
Peupler cette liste avec quelques triangles et carrés.
Parcourir la liste et calculer pour tous les objets le périmètre et l'aire. Quels est le mécanisme en action ici ?
Rajouter à la liste un objet de type `ConcreteRegularPolygon.` Exécuter de nouveau votre appli. Que ce passe t'il ? Normal ?

`ConcreteRegularPolygon,` servant de classe de base à l'ensemble de nos polygones, ne peut proposer de calcul par défaut de l'aire (calcul qui dépend de la forme effective d'un polygone). L'instruction `throw new NotImplementedException` génère donc une exception. C'est normal.
La solution : elle sera vue au thème suivant en utilisant une classe abstraite ou une interface.

Ex4 : Polymorphisme suite et fin
Cet exemple est donné à titre pédagogique uniquement (Nous verrons plus tard comment faire mieux)



La méthode VIRTUELLE ValeurAmortissement() d'Ouvrage permet de calculer la valeur d'un ouvrage en fonction de son age selon la formule $(1\text{-}x\%)^d$ avec d= jour écoulé entre date actuelle- date d'achat et x: pourcent de dépréciasation de l'ouvrage chaque jour (par exemple 0,1%)

```
TimeSpan delta = DateTime.Now - DateAchat;
double valeur=PrixAchat*Math.Pow((1-0.0005),delta.Days);
if (valeur<0) valeur=0;
return (valeur);
```

La méthode ToString de la classe de base, surchagée, renvoie une chaine avec la date d'achat .

Les méthodes ToString des classes dérivées complète m'information avec leurs champs spécifiques en réutilise ToString de Ouvrage (`string chaine = base.ToString() + " Réalisateur: "…..`)

Faire un programme qui permette de créer une liste d'ouvrage. Peupler cette liste.
Balayer la liste ainsi crée et :

- Afficher l'ouvrage en cours : `Console.WriteLine(ouvrage); //appel de ToString`
- Calculer la valeur de l'ouvrage
- Mettre à jour la valeur globale du stock

Afficher la valeur totale de votre stock.

Pourquoi cet exemple est mauvais :

- Nous risquons de voir apparaitre une multiplication de classe :
  DVDFilm, DVDMusicaux, DVDEnfant, Livre, BandeDessinées,BluRayFilm, BluRayMusicaux
- Il existe sans douite un problème de modélisation :
  Une classe ŒUVRE serait plus adaptée. La notion de DVD, BluRay, CD, livre est en relation plutôt avec un type de SUPPORT
- Ouvrage, VideoDVD contiennent des attributs qui ne sont pas de leurs responsabilités : Date d'achat, Calcul d'armortissement.
- Une classe ItemMediatheque pourrait être utilisée. Un item a effectivement une date d'achat, un prix d'achat, fait référence à une œuvre, et délègue le calcul de l'amortissement à une classe ActifFinancier.
- Une classe InventaireMediatheque a pour charge de repertorier et administrer les items de la médiathèques.