THEME 6
CLASSES ABSTRAITES
ET
INTERFACES

CONCEPTS

Les classes abstraites et les interfaces définissent des types de données volontairement incomplets. Cela se traduit par exemple par des corps de méthodes vides.
En conséquence, les classes abstraites et les interfaces ne sont pas directement instanciables. L'instruction suivante est illicite :

```
var test = new ClasseAbstraite (); // KO
```

Quel est l'intérêt :

- Les classes abstraites et les interfaces servent de modèles de classes qui sont ensuite dérivées et complétées (pour une interface on parle plutôt d'implémenter).
- Elles sont polymorphes :les classes dérivées ou qui les implémentent seront manipulées comme avec le type de base, à savoir la classe abstraite mère ou l'interface implémentée)
- Elles sont utilisées de manière intensive dans les design patterns.
- elle permette satisfaire aux principes SOLID
- De manière plus générale elles améliorent la maintenance de vos programmes, la testabilité (fakes) et la robustesse face aux changements.

En résumé, dans le cadre d'une programmation efficace , SOLID et professionnelle ces notions sont fondamentales.

CLASSE ABSTRAITE

DEFINITION

Une classe est dite abstraite lorsque le préfixe abstract est rajouté devant le mot clé class :

```
abstract class AFiche
  {
      Protected int valeur=10;
      public void test() { Console.WriteLine("test"); }
      abstract  public int AccessValeur { get; set; }
  }
```

Une classe abstraite permet de déclarer des membres abstraits (sans corps de fonction).
Les classes dérivées héritant de la classe abstraite SONT OBLIGES d'implémenter le corps de la méthode.

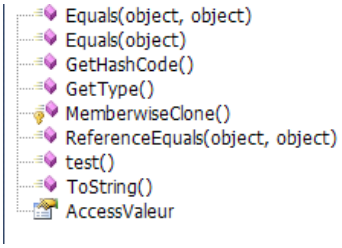
```
abstract class AFiche
{
protected int valeur=10;
public void test()
{ Console.WriteLine("test"); }
abstract public int AccessValeur
 { get; set; } // PAS DE CORPS!!!!
}

class Fiche1 : AFiche1
 {
//OBLIGATION d'IMPLENTER l'ACCESSEUR!
 public override int AccessValeur
 {
 get { return (valeur); }
 set { valeur = value; }
 }
 }
```

Une classe abstraite ne peut être instantiée directement :

```
var test = new A_Fiche1(); // KO
```

SOLUTION

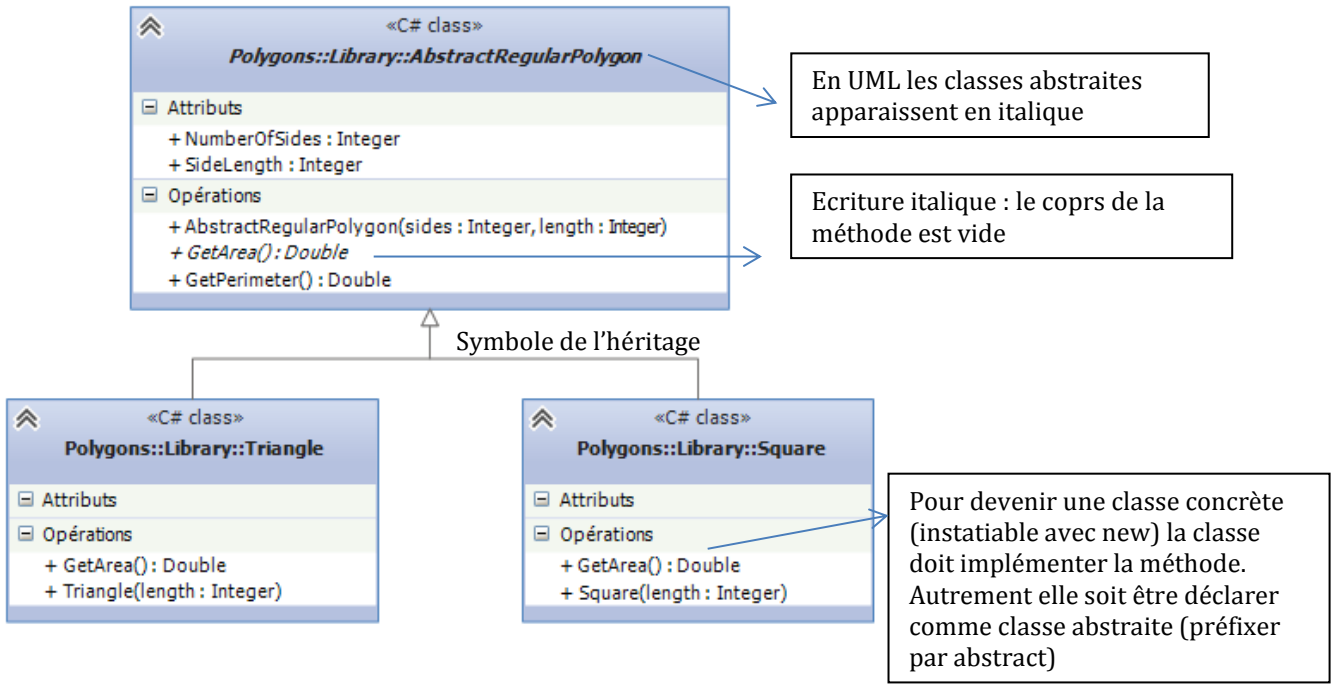
```
var ref_Fiche = new Fiche1();
//ref_Fiche hérite de A_Fiche (et de
Object aussi)
```



REMARQUE : le fait de déclarer une méthode abstraite ouvre droit **d'office** au polymorphisme (équivalent à écrire virtual qui devient optionnel). Toutse les fiches peuvent être manipulées comme des AFiche)

QUAND UTILISER UNE CLASSE ABSTRAITE ?

Un exemple typique est l'exercice 3 du thème précédent : calculs effectués sur des formes géométriques de type Polygone Régulier.
Tous les polygones ont des côtés d'une certaine longueur. Nous souhaitons calculer le périmètre et la surface. Le calcul de périmètre est commun à tous les polygones MAIS le calcul de l'aire dépend de la forme effective.
CONCLUSION : l'analyse nous amène à factoriser une partie au sein d'une classe AbstractRegularPolygons qui sera abstraite car la méthode getArea est laissée volontairement vide. A charge pour Triangle et Square de définir le contenu spécifique à chaque forme



COMMENT EXPLOITER UNE CLASSE ABSTRAITE

Les classes abstraites trouvent leur pleine puissance en association avec le polymorphisme. Il est rappelé que le préfixe virtual devient inutile dans une classe abstraite.
Ci-après un exemple illustre le propos :

```
static void Main(string[] args)
        {
            List<AbstractRegularPolygon> listeShape = new List<AbstractRegularPolygon>();
            listeShape.Add(new Square(1)); ;
            listeShape.Add(new Triangle(1));
            //liste.Add(new AbstractRegularPolygon(5));  KO pas
possible
            foreach (AbstractRegularPolygon shape in listeShape)
            {
                DisplayPolygon(shape);
            }
            Console.ReadKey();
        }

        public static void DisplayPolygon(AbstractRegularPolygon polygon)
        {

            Console.WriteLine("Number of Sides:{0}", polygon.NumberOfSides);
            Console.WriteLine("Side Length:{0}", polygon.SideLength);
            Console.WriteLine("Perimeter:{0}", polygon.GetPerimeter());
            Console.WriteLine("Area:{0}", Math.Round(polygon.GetArea(), 2));
            Console.WriteLine();
        }
```

Les polygones sont stockés et manipulés au travers de la classe de base : AbstractRegularPolygon

Manipulation par le biais d'une référence de type AbstractRegularPolygon

Polymorphisme en action : la méthode correspond au vrai type est retrouvé et effectuée

LIMITE DE L'HERITAGE

Les notions de dérivation et d'héritage permettent de construire plus aisément de nouvelles classes en s'inspirant d'un modèle de base (la classe que l'on dérive). Ainsi, plutôt que de partir de rien, la nouvelle classe dérivée hérite d'un certain nombre de fonctionnalités.
Le concepteur pour structurer et factoriser son code est tenté de s'appuyer sur la relation "EST un" :un chien EST UN mammifère. Ce raisonnement, quoique juste conceptuellement, aboutit parfois à des disfonctionnements fonctionnels (voir thème précédent).
C'est la raison pour laquelle la communauté des concepteurs informatiques recommande de manière générale l'utilisation de la composition oar rapport à l'héritage.

Pour illustrer ce cas reprenons l'exemple des des mammifères précédents, pour lequelle nous rajoutons des fonctionnalités à nos mammifères : nombres de pattes, vole, crier etc….
Comment traiter alors une Baleine qui est un mammifère aussi mais qui n'a pas de patte?
Comment traiter la capacité des mammifères à voler? Faut t'il mettre voler dans la classe de base?
Voici un exemple de modifications apportés à l'exemple:

```
  public abstract class Mammifere
    {
        //methodes
        abstract public void Crier();
        abstract public void Sous_Ordre_Especes();
        abstract public void NbreDePattes();
    }

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

    class Baleine : Mammifere
    {
        public override void NbreDePattes()
        {
            throw new NotImplementedException();
        }

        public override void Crier() { Console.WriteLine("Mouuuuu"); }
        public override void Sous_Ordre_Especes() { Console.WriteLine("Cétacés"); }
    }

    class ChauveSouris : Mammifere
    {
        public override void NbreDePattes() { Console.WriteLine("2 papattes"); }
        public override void Crier() { Console.WriteLine("Mriiiiii"); }
        public override void Sous_Ordre_Especes() { Console.WriteLine("chiroptères"); }
        public  void Vole() { Console.WriteLine("Flap flap"); }
    }

static void Main(string[] args)
        {
            ChauveSouris uneChauveSouris = new ChauveSouris();
            uneChauveSouris.Vole(); //OK car uneChauveSouris du type uneChauveSouris
            // On regroupe tout le monde sous le type de base
            List<Mammifere> maMenagerie = new List<Mammifere>();
            // Je peuple la ménagerie
            maMenagerie.Add(new ChauveSouris());
            // maMenagerie[0].Vole();  KO maMenagerie[0] fait référence à un mammifere
            // Vole() n'est pas dans la classe de base!
            maMenagerie.Add(new Baleine());
            // maMenagerie[1].NbreDePattes(); KO la baleine n'a pas de pattes
            //Qui vole et qui a des pattes dans ma ménagerie????
            foreach (Mammifere mam in maMenagerie)
            {
                // mam.Volant();  KO: volant n'est pas dans la classe de base!
                mam.NbreDePattes();//  KO: arrivé à la baleine BOOM
            }
            Console.ReadKey();
        }
```

Remarques: Nous constatons que la spécialisation rajoutée à la chauve-souris ne permet pas d'appliquer un traitement de base du type: qui vole? (polymorphisme). Une solution consiste à intégrer Vole() dans mammifère en virtual. Ceci nous amènera à surchager toutes les classes dérivées, ce qui n'est pas souhaitable dans notyre cas.
Le code produit n'est donc pas robuste et une modification du cahier des charges implique une retouche de toutes les classes.

La question est alors de savoir comment injecter des comportements à des objets non liés spécifiquement à la classe de base (un animal volant n'est pas une spécifité des mammifères).

Pour répondre à cette questions plusieurs outils sont utiles :

- Les "interfaces" en C#
- Les design pattern: pattern Décorateur par exemple.
- Le refactoring des classes (nouveau modèle) : exemple utiliser une classe AttributMammifere encapsulant un dictionnaire de dans lequels seront rajoutés toutes spécifités propres à l'animal. Ces spécifités seront définis par ailleurs grâce à des enum.

LES INTERFACES

CONCEPT

La notion d'interface rejoint en partie la notion de de classe abstraite. Une classe abstraite est une classe marqué **abstract** qui défini une ensemble de fonctionnalités (méthodes) sans corps associé. Une classe concrète est une classe pour laquelle l'ensemble des fonctionnalités jusqu'alors abstraites ont toutes été concrétisées. Une fonctionnalité concrète est une fonction membre qui possède un corps (c'est à dire du code).

Une **interface**, quant à elle, peut être vue donc comme une classe abstraite PURE. Tous les corps des méthodes sont laissés vide. D'autre part une interface ne continedra aucun attribut (pas de champs).
L'interface défini en fait un ensemble de prototypes que les classes, *qui choisiront d'implémenter l'interface devront respecter et implémenter.*

Une interface définit un contrat. Vos classes choisissent ou non de respecter ce contrat (ie : implémenter l'interface).

Exemple1

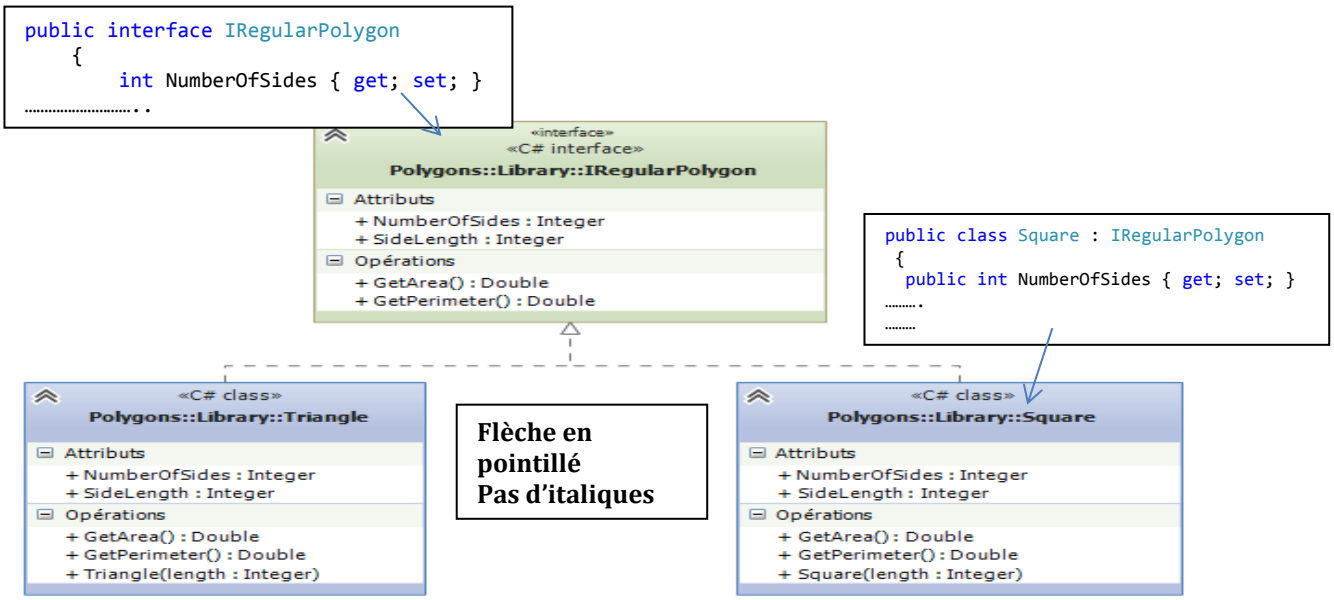
```
    interface IVolant
    {void Vole();}

 // Fonction a des pattes
    interface IPatte
    {
// définition d'une propriété en lecture
     int NbPatte { get; }
    }
```

```
class ChauveSouris : Mammifere,IVolant,IPatte
{   // Implantation de NbPatte défini dans IPatte
    public int NbPatte
         {   get {return(2);}
         }
    // Implantation de Vole défini dans IVolant
    public void Vole() { Console.WriteLine("Flap flap"); }
.....
}
```

Remarques:

- par convention le nom donné aux interfaces est préfixé par un I : `IVolant, IPatte, Ixxxxx`
- les interfaces n'ont pas de constructeurs
- les interfaces ne définissent pas de code des méthodes (concept abstraction)
- les interfaces n'ont pas de champs
- Les méthodes sont toutes publiques par défaut (normal car le contrat est public)
- Une classe peut choisir d'implémenter plusieurs interfaces

Exemple 2 : exemple des polygones



Le programme principal ne change pratiquement pas:

```
static void Main(string[] args)
        {
            var listeShape = new List<IRegularPolygon>();
            listeShape.Add(new Square(1)); ;
            listeShape.Add(new Triangle(1));
            //liste.Add(new IRegularPolygon(5));  KO pas possible
            foreach (IRegularPolygon shape in listeShape)
            {
                DisplayPolygon(shape);
            }
            Console.ReadKey();
        }

        public static void DisplayPolygon(IRegularPolygon polygon)
        {

            Console.WriteLine("Number of Sides:{0}", polygon.NumberOfSides);
            Console.WriteLine("Side Length:{0}", polygon.SideLength);
            Console.WriteLine("Perimeter:{0}", polygon.GetPerimeter());
            Console.WriteLine("Area:{0}", Math.Round(polygon.GetArea(), 2));
            Console.WriteLine();
        }
```

INTERET DES INTERFACES

Les interfaces , lors de l'étude d'un projet, permettent de définir en amont les fonctionnalités attendues des objets sans pour autant figer ou imposer de solutions techniques.
Une fois définies, elles obligent toutes les classes clientes de ces interfaces à respecter le contrat : dans un travail en équipe par exemple l'architecte logiciel ayant défini les interfaces, les objets et leurs interactions les programmeurs devront respecter les choix imposés (conventions, méthodes, fonctionnalités ….)

L'héritage multiple étant interdit (choix des concepteurs du C#), les interfaces permettent de décrire des fonctionnalités multiples et non spécifiques à la classe de base .

- ✓ Tous les mammifères ne sont pas volants: fonction spécifique des chauve-souris (interfaces Ivolant)

- ✓ Les ouvrages (livres, DVD) d'une bibliothèque ne sont pas tous empruntables (interfaces IEmpruntable qui contiendra sans doute un historique des emprunts). Par contre tous les ouvrages ont un genre, possède un identifiant unique, une date de création etc (=> invariant dans le temps=> classe de base)
- ✓ Des formes en 2D ont toutes la faculté de renvoyer un périmètre, une surface (=> classe de base). Certaines formes sont définies à l'aide d'une liste de point (rectangle , carré, losange, triangle), d'autre non (cercle) => interface Ipoint

Exemple

```
//Fonction Vole
interface IVolant
{ void Vole();}

interface IPatte
{
    // définition d'une propriété en lecture
    int NbPatte { get; }
}

// classe de base bastraite
public abstract class Mammifere
{
    private string name;
    public string Nom
    {
        get { return name; }
        set { name = value; }
    }
    public abstract void Crier();
}

// Chien est un mammifère qui a aussi des pattes
class Chien : Mammifere, IPatte
{  // Implantation de NbPatte défini dans IPatte
    public int NbPatte
    {
        get { return (4); }
    }
    // implémente Crier de Mammifere
    public override void Crier() { Console.WriteLine("Ouarfff"); }
}

class Baleine : Mammifere
{
    public override void Crier() { Console.WriteLine("Mouuuuu"); }
}

class ChauveSouris : Mammifere, IVolant, IPatte
{   // Implémente  NbPatte défini dans IPatte
    public int NbPatte
    {
        get { return (2); }
    }
    // Implémente  Vole défini dans IVolant
    public void Vole() { Console.WriteLine("Flap flap"); }
    // surcharge polymorphe des classes de bases
    public override void Crier() { Console.WriteLine("Mriiiiii"); }

}
```

```
class Program
    {
        static void Main(string[] args)
        {
            // On regroupe tout le monde sous le type de base
            List<Mammifere> maMenagerie = new List<Mammifere>();
            // Je peuple la ménagerie
            maMenagerie.Add(new ChauveSouris());
            maMenagerie.Add(new Baleine());
            maMenagerie.Add(new ChauveSouris());
            maMenagerie.Add(new Chien());
            maMenagerie[3].Nom = "medor";

            //Allez tous le monde crie
            foreach (Mammifere mam in maMenagerie)
                mam.Crier();

            //Qui vole dans ma ménagerie????
            // Extraction des volants
            var listeVolant = maMenagerie.OfType<IVolant>();
            foreach (IVolant mamVolant in listeVolant)
            {
                mamVolant.Vole();
            }

            Console.ReadKey();
        }
    }
```

LE POLYMORPHISME D'INTERFACE

Les interfaces permettent au polymorphisme d'opérer de manière très simple en manipulant les objets au travers d'une référence de type interface.

Exemple 1:

```
List<Ivolant> maMenagerieVolante = new List< Ivolant>();
// Je peuple la ménagerie
maMenagerie.Add(new ChauveSouris());
maMenagerie.Add(new Canari());
maMenagerie.Add(new Perroquet());
maMenagerie.Add(new pterodactyle ());
foreach (Ivolant animalVolant in maMenagerieVolante)
        animalVolant.Vole();
```

On constate qu'il est possible de regrouper des objets n'appartenant pas à la même classe de base!! (Canari n'est pas un mammifère mais dérive peut-être d'une classe Animal). Il suffit juste que Canari implémente `Ivolant`

Remarque : il faut bien comprendre `List<Ivolant> maMenagerieVolante` maintient une liste de référénce vers des objets en mémoires. Ces objets , lors de leur création , ne sont pas tronqué en mémoire.
`maMenagerie.Add(new ChauveSouris());` réalise 2 opérations :

- Création d'un objet complet en mémoire de type Chauve-souris
- Stockage dans liste d'une référénce vers cet objet en tant que `Ivolant`

Tant que cette chauve souris est manipulée en tant que `Ivolant` seule les propriétés et méthodes des `Ivolant` seront accesibles.
A tout moment il est possible de changer de point de vue par une opération de cast pour accéder pleinement à l'objet : (ChauveSouris)maRéférenceIvolant *à condition bien entendu que maRéférenceIvolant référence à l'instant t une chauve-souris !!!*

Exemple 2:

```
List<Mammifere> maMenagerie = new List<Mammifere>();
// Je peuple la ménagerie
maMenagerie.Add(new ChauveSouris());
maMenagerie.Add(new Baleine());
maMenagerie.Add(new ChauveSouris());
maMenagerie.Add(new Chien());
//Qui vole dans ma ménagerie????
foreach (Mammifere mam in maMenagerie)
{
if (mam is IVolant)
((IVolant)mam).Vole();
else
Console.WriteLine("Non Volant");
}
//Allez tous le monde crie
foreach (Mammifere mam in maMenagerie)
mam.Crier(); // triatement de base crier =>polymorphisme
```

ATTENTION : possible mais constitue une mauvaise pratique car elle nuit à maintance de votre programme (rajout de if en cascade au fur et à mesure de l'évolution de vos classes !)

INTERFACE EXPLICITE ET RESOLUTION DE CONFLIT

Il est possible de résoudre les conflits de noms en explicitant, lors de l'implantation de la méthode, le nom de l'interface.

```
interface I1 { void Foo(); }
interface I2 { int Foo(); }
public class Widget : I1, I2
{
public void Foo ()
{Console.WriteLine ("Widget's implementation of I1.Foo");
}
int I2.Foo()
{
Console.WriteLine ("Widget's implementation of I2.Foo");
return 42;
}
}
```

```
Widget w = new Widget();
w.Foo(); // Widget's implementation of I1.Foo
((I1)w).Foo(); // Widget's implementation of I1.Foo
((I2)w).Foo(); // Widget's implementation of I2.Foo
```

COMPARAISON INTERFACE /CLASSE ABSTRAITE

Le tableau résume la situation:

Classes abstraites	Interfaces
Peut contenir du code	Ne contient aucun code
Un seul héritage à la fois possible (si j'hérite dela classe abstraite il n'est plus possible d'hériter d'une autre classe)	Une classe peut implémenter plusieurs interfaces
Les membres de la classes ont des modificateurs d'accès (public, private..)	Tous les membres de la classes sont public
Les membres peuvent être des: ▪ propriétés ▪ méthodes ▪ evénement (event) ▪ indexeurs ▪ constructeurs/destructeurs ▪ champs	Les membres peuvent être des: ▪ propriétés ▪ méthodes ▪ evénement (event) ▪ indexeurs

IEQUATABLE ET IEQUALITYCOMPARER

Le framework .NET fournit un nombre important d'interface répondant à des problèmes classqiues de l'algorithmique:

- Comment cloner des objets *ICloneable<T>*
- Comment parcourir une liste : *IEnumerable<T>*
- Comment tester l'égalité des objets: *IEquatable<T>* et *IEqualityComparer<T>*

Comment comparer des objets: *IComparable<T>* et *IComparer<T>*

IEquatable<T>

Le comportement par défaut d'un test d'égalité x==y ou x.Equals(y) est:

- Pour les types valeurs: vrai si les données sont indentiques
- Pour les types références: vrai si les références sont indentiques! (et non pas le contenu des objets)

L'interface IEquatable définit la méthode Equals, qui détermine l'égalité des instances du type d'implémentation.

```
public interface IEquatable<T>
{
bool Equals (T other);
}
```

Exemple: Classe Température incluant deux mesures.Définition de l'égalité de 2 températures sur la valeur moyenne des mesures

```
public class Temperature : IEquatable<Temperature>
{
    //Implémentation de IEquatable
    //paramètre <T> de type Temperature
    public bool Equals(Temperature other)
    {   // appel de méthode par défaut Equals des types par valeurs double.Equals
        double moyenneOther = (other.mesure1+ other.mesure2) / 2;
        double moyenne = (this.mesure1+ this.mesure2) / 2;
        return moyenne.Equals(moyenneOther);
    }

    // CHAMPS
    private double mesure1 = 0.0;
    private double mesure2 = 0.0;

    // CTOR
    public Temperature(double capteur1, double capteur2)
    {
        mesure1 = capteur1;
        mesure2 = capteur2;
    }
}

public static void Main()
    {
        var listeTemp = new List<Temperature>();

        listeTemp.Add(new Temperature(2017.15, 1000));
        listeTemp.Add(new Temperature(0, 100));
        listeTemp.Add(new Temperature(50, 50));

        var tempReference = new Temperature(25, 75);
        foreach (Temperature temperature in listeTemp)
            if (temperature.Equals(tempReference))
                Console.WriteLine("température moyenne égale");
            else
                Console.WriteLine("température moyenne différente");

        Console.ReadKey();

    }
```

On rappelle que Equals est une méthode d'instance :
temperature1.Equals(temperature2)

Remarquez bien la syntaxe :
other : variable locale transmise de type Temperature
this : l'objet en cours d'utilisation

On itère la liste et toute les températures sont comparées à la tempéraure de référence

```
température moyenne différente
température moyenne égale
température moyenne égale
```

TRAVAIL PERSONNEL

Ex1 : Premier pas avec les abstraits

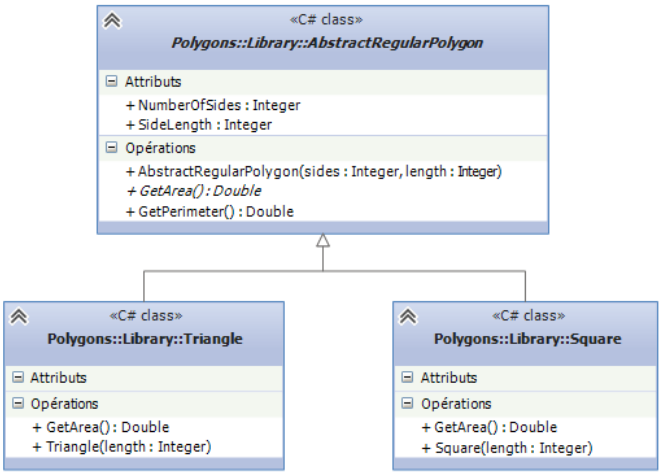
Etudier attentivement le programme ci-dessous. Prédire et compléter les affichages. Créer un projet dans VS2013 et tester le programme. Vérifier.

Si vous avez commis une erreur de prédiction exécutez le programme en mode pas à pas et vérifier par où passe le programme !

```
namespace TestAbstrait
{
   abstract class AFiche1
    {
        protected int valeur=10;
        public void Test() { Console.WriteLine("test"); }
        abstract public int AccessValeur { get; set; }
    }

   class Fiche1 : AFiche1
   {
       public override int AccessValeur
       {
           get { return (valeur); }
           set { valeur = value; }
       }

       public override string ToString()
       {

           return valeur.ToString();
       }

   }
   class Fiche2 : Fiche1
   {
       public new int AccessValeur
       {
           get { return (valeur); }
           set { valeur = 2*value; }
       }
   }

   class Fiche3 : Fiche1
   {
       public override int AccessValeur
       {
           get { return (valeur); }
           set { valeur = 2 * value; }
       }
   }

    class abstrait1
    {
        static void Main(string[] args)
        {
            //AFiche1 test = new AFiche1();
            Fiche1 refTypeFiche1 = new Fiche1(); // valeur=10;
            Console.WriteLine(refTypeFiche1);
            AFiche1 refTypeAFiche1 = refTypeFiche1;//
            refTypeAFiche1.AccessValeur = 20; //
            Console.WriteLine(refTypeAFiche1);

            Fiche2 refTypeFiche2 = new Fiche2(); // valeur Fiche2=10;
            refTypeFiche1 = refTypeFiche2;  // Fiche2 vu comme une Fiche1
            refTypeFiche1.AccessValeur = 60; //
            Console.WriteLine(refTypeFiche1);

            Fiche3 refTypeFiche3 = new Fiche3(); // valeur Fiche2=10;
            refTypeFiche1 = refTypeFiche3;  // Fiche3 vu comme une Fiche1
            refTypeAFiche1 = refTypeFiche3;
            refTypeAFiche1.AccessValeur = 60; //
            Console.WriteLine(refTypeFiche1);
            Console.ReadKey();
        }
    }
}
```

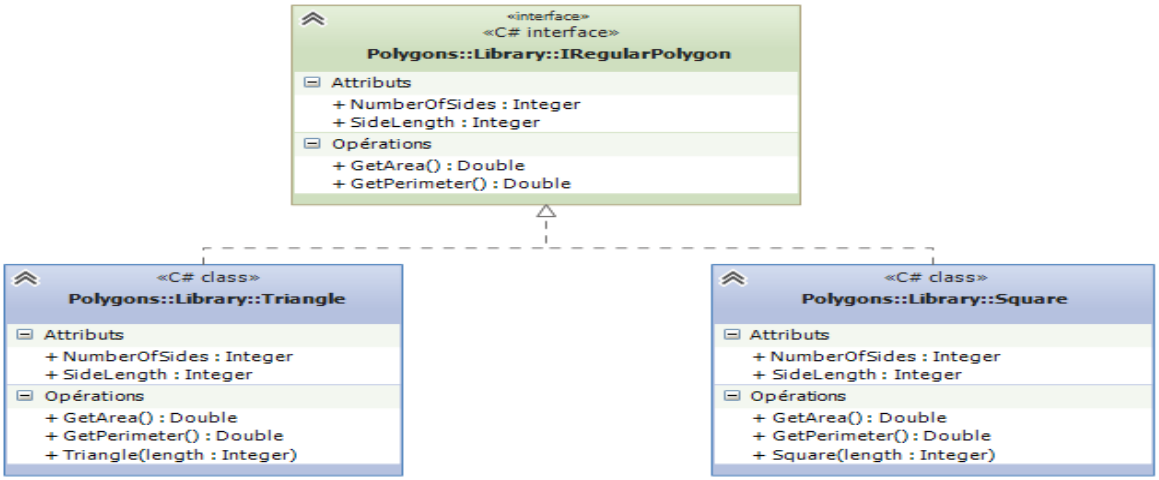
Ex2 : les formes géométriques avec les classes abstraites

Copier et coller le travail déjà effectué au thème précédent.
Recodez à l'aide schéma des classes suivants et reprendre par exemple le code de ce thème comme programme principal.



Ex3 : les formes géométriques avec des interfaces

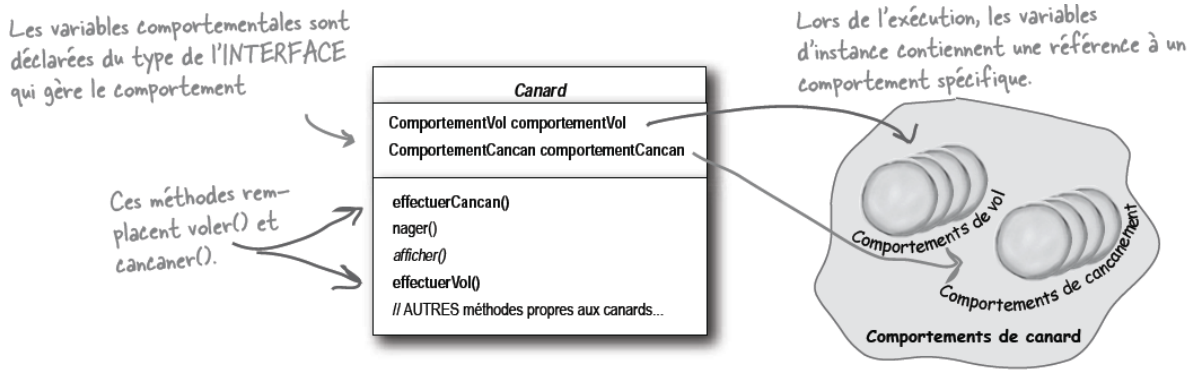
Même exercice que le précédent avec un interface cette fois ci



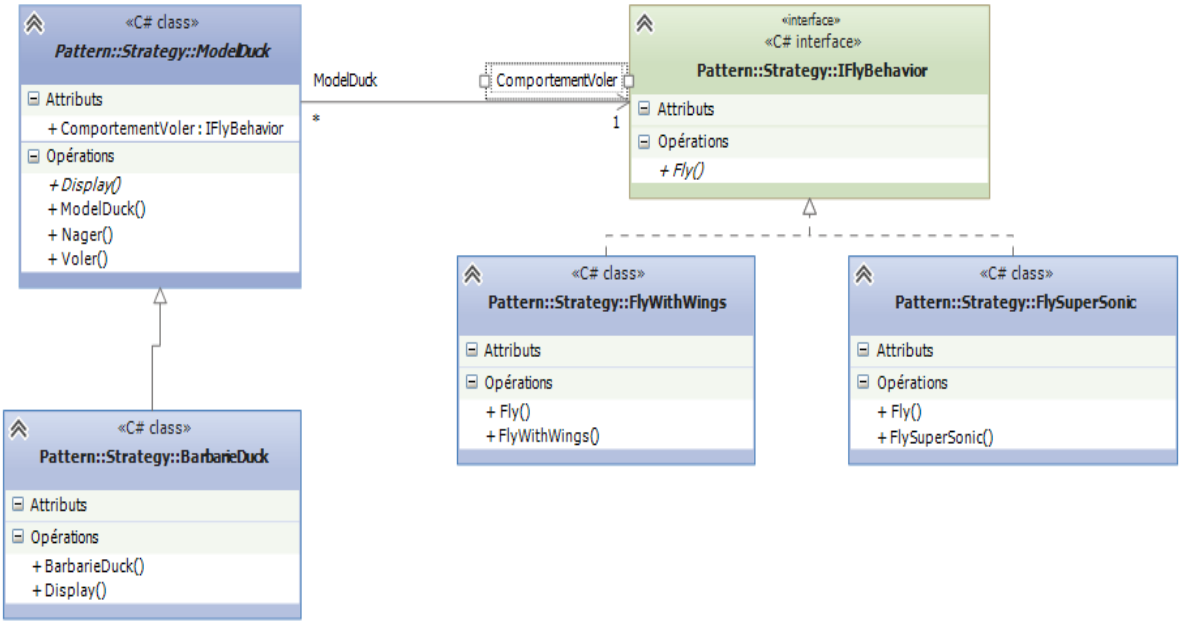
Ex4 : Le pattern Strategy

Si après plusieurs lecture de l'exercice vous êtes bloqués voici une liste de liens :
http://fr.wikipedia.org/wiki/Strat%C3%A9gie_%28patron_de_conception%29
http://www.informatix.fr/tutoriels/conception/le-design-pattern-strategie-169
http://www.u-picardie.fr/~furst/docs/Patterns.pdf

L'objectif est de modéliser une classe Canard de telle manière à pouvoir définir de façon flexible des comportements spécifiques à chaque canard.
Le schéma de principe est le suivant :



Le modèle de classe de notre exercice est le suivant :



Etape1 : Créer l'interface `IFlyBehavior` et deux classes `FlyWithWings` et `FlySuperSonic`.
On se contentera de faire dans la méthode fly un affichage :

```
public void Fly()
        {
            Console.WriteLine("I'm flying!!");
        }
```

Etape2 : créer la classe abstraite `ModelDuck`
Le comportement de vol utilisera une référence de type `IFlyBehavior`.
Nous utiliserons une propriété automatique :

```
public IFlyBehavior ComportementVoler { get; set; }
```

Vous mettrez un affichage de votre choix dans Nager et Voler
L'appel du comportement Voler s'effectue comme suit :

```
public void Voler()
        {
            ComportementVoler.Fly();
        }
```

Etape3 : créer un modèle particulier de Canard. Ici `BarbarieDuck`
Le comportement de vol sera crée dans le constructeur de `BarbarieDuck`

```
public BarbarieDuck()
        {
            ComportementVoler = new FlyWithWings();
        }
```

La methode Display sera implémenté avec un affichage de votre choix

Etape4 : Saisir le programme suivant pour tester.

```
            ModelDuck anatole = new BarbarieDuck();
            anatole.Display();
            anatole.Voler();

            anatole.ComportementVoler=new FlySuperSonic();
            anatole.Display();
            Console.ReadKey();
```

Etape 5 :
Créer un nouveau type de canard.Tester
Créer un nouveau type de vol.
Créer un nouveau type de comportement IQuackBehavior référéncé dans ModelDuck avec une propriété ComportementCancanner de type IQuackBehavior.
Créer différents façon de cancanner et modifier vos canards existants.

Conclusion : le pattern Sratgy permet d'encapsuler une famille d'algorithme interchangeable (comportement). Ce pattern est basé sur une composition (relation a un) qui s'appuie sur une abstraction (Icomportement). L'extension des comportements se résume à la création d'une nouvelle classe implémentant Icomportement puis au référencement par le biais de notre constructeur et 'en dur'(aspect statique) ou de notre propriété automatique (aspect dynamique). Cette technique s'appelle une injection de dépendance (nous approndirons plus tard).