Programmation Orientée Objet en C#

Progression pour découvrir les concepts objet et les implémenter en C#

Sommaire

Préambule

Ce support est constitué d'une progression de TP permettant de découvrir les concepts de la Programmation Orientée Objet et de les mettre en œuvre en construisant pas à pas une classe en C#. Il s'agit du même cas qui va évoluer, étape par étape, pour aboutir à un ensemble de classes finalisé dans l'environnement .net.

Chaque exercice est structuré de la façon suivante :

- Description des objectifs visés
- Explications des techniques à utiliser (Ce qu'il faut savoir)
- Énoncé du problème à résoudre (Travail à réaliser)

Les bases de la programmation procédurale n C# sont un prérequis.

1. Classe d'objet

1.1. Objectifs

- Classe, propriété, méthode, objet
- Encapsulation
- Définition d'une nouvelle classe d'objet
- Instanciation d'un objet

1.2. Ce qu'il faut savoir

1.2.1. Notion de classe, de propriété, de méthode

Une ***classe*** est un ***moule,*** un ***patron*** à partir duquel des objets pourront être créés.

Une classe regroupe :

- des ***propriétés ou attributs*** ou données membres,
- des ***méthodes*** ou fonctions membres.

A partir d'une classe (le modèle), des objets sont créées (instanciés) pour être utilisés.

Chaque objet ***connait ses caractéristiques*** (les valeurs affectées à ses propriétés) et **sait faire des opérations** (ses méthodes).

 Créer une ***classe,*** c'est créer un nouveau ***type de données.***

Pour définir une nouvelle classe, il faut donc énumérer toutes les propriétés de cet objet et toutes les fonctions qui vont permettre de définir son comportement. Ces dernières peuvent être classées de la manière suivante :

- Fonctions d'entrées/sorties pour lire et écrire des données sur les périphériques (clavier, écran, fichier)
- Fonctions de calcul
- Opérateurs relationnels pour comparer 2 objets
- Fonctions de conversion vers d'autres types
- Fonctions pour contrôler l'intégrité de l'objet.

Le Framework .net contient de nombreuses classes qui obéissent aux concepts de la programmation objet. Les classes que nous allons créer obéissent aux mêmes règles.

1.2.2. Encapsulation

L'***encapsulation*** est un concept de la POO[1] qui permet de rassembler les propriétés et les méthodes d'un objet pour les manipuler dans une seule entité appelée classe d'objet. Le code des méthodes et la définition des propriétés sont enfermés dans la classe qui est une sorte de ***boite noire***.

Les applications clientes ne connaissent que l'interface des objets qu'elles utilisent et n'ont pas à se soucier du fonctionnement interne de la classe. Si le code de la classe est modifié mais pas l'interface, les applications clientes ne seront pas modifiées.

[1] POO = Programmation Orientée Objet

1.2.3. Définition d'une nouvelle classe

```
public class NomDeLaClasse
{

    //Déclaration des propriétés
     public typePropriete Propr1;
     public typePropriete Propr2;

     //Déclaration des méthodes

     //Une procédure qui ne renvoie pas de résultat
     public void Methode1()
     {
        //Code de la procédure
     }

     //Une fonction qui renvoie un résultat
     public typeRetour Methode2()
     {
         //Code de la fonction
     }

     //Une fonction avec des paramètres
     public typeRetour Methode3(typeParametre param1)
     {
         //Code de la fonction
     }

    //La même fonction avec d'autres paramètres (surcharge)
     public typeRetour Methode3(typeParametre param1, typeParametre param2)
     {
          //Code de la fonction
     }

} //Fin de la classe
```

Par convention, les noms de classes commencent toujours par une majuscule.

Autres directives de nommage :
https://docs.microsoft.com/fr-fr/dotnet/standard/design-guidelines/capitalization-conventions

Lors de sa déclaration, une propriété (variable) peut être initialisée.

Les méthodes sont implémentées soit par des ***procédures*** ou par des ***fonctions*** qui peuvent avoir aucun, un ou plusieurs paramètres.

La ***surcharge*** d'une procédure ou d'une fonction consiste à définir celle-ci en ***plusieurs versions : même nom, même type de retour*** mais avec des ***paramètres différents***. Une même méthode peut avoir plusieurs ***signatures***.

1.2.4. Exemple de classe Personne

```
public class Personne
{

    //Déclaration des propriétés
    public  string   Prenom;
    public  string   Nom;
    public  short    Age;

    //Déclaration des méthodes

    //Une procédure qui ne renvoie pas de résultat
    public void Afficher()
    {
        Console.WriteLine($"Nom = {Nom}, Prenom = {Prenom}, Age = {Age}" );
    }

    //Une fonction qui renvoie un résultat de type chaine de caractère
    public string Identite()
    {
        return Nom.ToUpper() + " " + Prenom.ToLower() ;
    }

} //Fin de la classe Personne
```

1.2.5. Instanciation d'objets

La classe est créée, comment l'utiliser ? Il faut créer (instancier) des objets.

Pour qu'un objet ait une existence, il faut qu'il soit ***instancié***. Une même classe peut être instanciée plusieurs fois : chaque instance pouvant une valeur particulière pour chacune de ses propriétés.

Dans le cas général, la création d'une instance se fait en 2 temps :

- Déclaration d'une variable du type de la classe d'objet
- Instanciation de cette variable par l'instruction ***New***

Ces 2 étapes peuvent être regroupées en une seule.

Dans l'exemple ci-dessous, des objets sont instanciés dans la méthode main d'une classe de Test.

```
public static void Main(string[] args)
{
     Personne p1; //2 étapes séparées : déclaration puis instanciation
     p1 = new Personne();
     p1.Nom = "GREGOIRE";
     p1.Prenom = "Ghislaine";
     p1.Afficher();
     Console.Writeline(p1.Identite());

     Personne p2 = new Personne(); //2 étapes en 1 : déclaration + instanciation
     p2.Nom = "BOISSEAU";
     p2.Prenom = "Serge";
     p2.Afficher();

     //Une autre personne à laquelle est affectée la référence p2
     Personne p3 = p2;
     p3.Afficher();
  }
```

La déclaration d'une variable objet p1 n'est pas suffisante. En effet, p1 ne contient pas une donnée de type Personne mais une ***référence (une adresse en mémoire)*** sur un objet de type Personne. Tant que l'opérateur new n'a pas été appelé, p1 est égal à ***null***.

Une fois un objet instancié, l'accès aux propriétés et aux méthodes de cet objet se fait par ***l'opérateur .*** .

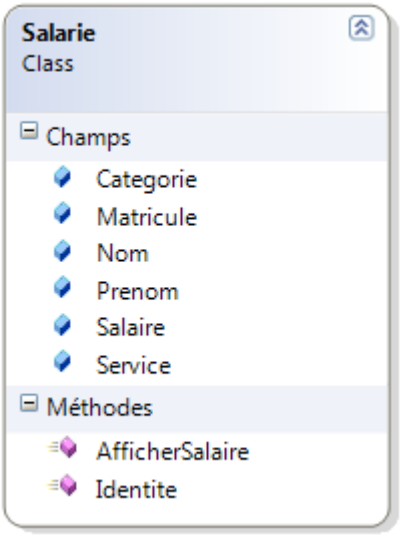
L'accès direct aux propriétés d'un objet ne correspond pas au concept d'encapsulation, nous verrons dans le chapitre suivant comment protéger les données membres en interdisant l'accès direct à ses propriétés.

Quand on considère une méthode par rapport à l'objet à laquelle elle s'applique, il faut voir l'objet comme étant sollicité de l'extérieur par un message. Ce message peut comporter ou non des paramètres. L'objet réagit à ce message en exécutant la méthode.

1.3. TP1 - Travail à réaliser

→ Créer une classe Salarie avec 6 propriétés publiques (Dans cette 1ère version, l'encapsulation ne sera pas encore mise en œuvre) :

- Matricule entier
- Catégorie entier
- Service entier
- Nom chaine de caractères
- Prénom chaine de caractères
- Salaire decimal



→ Coder une méthode *AfficherSalaire* pour afficher sur la console le message suivant : Le salaire de *nom et prénom du salarié* est de *salaire*.

→ Coder une méthode *Identite* qui renvoie une chaine de caractère avec toutes les caractéristiques du salarié.

→ Créer une classe de test comprenant une méthode main, instanciez au moins 2 salariés et appelez les différentes méthodes pour les tester.

2. Encapsulation – Protection et accès aux données membres

2.1. Objectifs

- Protection des données
- Méthodes d'accès aux propriétés : ***Getter*** et ***Setter***
- ***Property C#***

2.2. Ce qu'il faut savoir

2.2.1. Protection des données

En POO, on évite d'accéder directement aux propriétés par l'opérateur **.** . En effet, cette possibilité ne correspond pas au concept d'encapsulation. Certains langages l'interdisent carrément. En .net, c'est le programmeur qui choisit si une donnée ou une fonction membre est accessible directement ou pas.

On l'indique au moment de la déclaration d'un attribut ou d'une méthode. Pour cela, il existe principalement 4 mots-clés (**modificateurs d'accès**) :

public	Les données ou fonctions membres sont accessibles à l'extérieur de la classe (cf TP1)
private	Les données ou fonctions membres sont privées à la classe, elles sont accessibles uniquement par les méthodes internes à la classe. L'utilisation du mot private permet de masquer les données ou des fonctions membres, elles ne seront pas accessibles à l'extérieur de la classe.
protected	Les données ou fonctions membres sont privées à la classe mais elles sont néanmoins accessibles dans les classes dérivées (cf TP4).
internal	Les données ou fonctions membres sont accessibles à l'intérieur du même Assembly.

La distinction entre ***private*** et ***protected*** n'est visible que dans le cas de la définition de nouvelles classes par héritage. Ce concept est abordé plus loin, dans le TP4.

Modifions la classe Personne pour qu'elle réponde au concept d'encapsulation :

```
public class Personne
{
   //Déclaration des propriétés
   private string  Prenom;
   private string  Nom;
   private short   Age =18; // valeur initiale

   //Une procédure qui ne renvoie pas de résultat
   public void Afficher()
   {
          Console.WriteLine($"Nom = {Nom}, Prenom = {Prenom}, Age = {Age}" );
   }
…
    } //Fin de la classe Personne
```

Conséquences pour l'utilisation à l'extérieur de la classe :

```
Personne p1 = new Personne();
p1.Nom = "GREGOIRE";
Console.WriteLine(p1.Nom);
```

2.2.2. Fonctions accesseurs

Si les propriétés sont ***privées***, on ne peut plus y avoir accès directement à l'extérieur de la classe par l'opérateur ., il faut donc créer des méthodes ***publiques*** pour accéder aux propriétés qui doivent être visibles à l'extérieur de la classe.

- Méthode pour renvoyer la valeur de la propriété : ***Get***

```
public string GetNom()
{
  return Nom;
}
```

Les fonctions Get peuvent être considérées comme un message envoyé à l'objet : "*Quelle est la valeur de ta propriété ?*"

- Méthode pour modifier la valeur d'une propriété : ***Set***

```
public void SetNom(string nom)
{
  this.Nom = nom;
}
```

Les fonctions Set peuvent être considérées comme un message envoyé à l'objet : "*Maintenant la valeur de ta propriété est … ".*

Le mot-clé ***this***, utilisable à l'intérieur de toute classe, désigne l'instance courante.

Les fonctions accesseurs pourront être utilisées de la manière suivante :

```
Personne p1 = new Personne();
p1.SetNom("GREGOIRE");
p1.SetPrenom("Ghislaine");
p1.SetAge(25);
Console.Writeline(p1.GetNom());
```

L'intérêt de passer par des fonctions S***et*** est de pouvoir y localiser des ***contrôles de validité*** des valeurs passées en paramètres pour assurer la cohérence de l'objet en y déclenchant des exceptions par exemple. Nous aborderons ce point dans la suite de ce support.

2.2.3. Property

En .net, il existe une autre façon de gérer l'encapsulation : c'est le concept de ***property.*** Depuis l'extérieur de la classe, les property permettent de manipuler les propriétés privées de la classe comme si elles étaient publiques.

Une property permet de lire et/ou mettre à jour une donnée membre privée. Les champs privés correspondants ne peuvent pas avoir le même nom que les property, en général, ils sont préfixés par le caractère _.

La classe Personne avec des "property" :

```
public class Personne
{
    //Déclaration des propriétés
    private string _Prenom;

    public string Prenom
    {
      get { return _Prenom; }
      set { _Prenom = value; }
    }

    private string _Nom;

    public string Nom
    {
      get { return _Nom; }
      set { _Nom = value; }
    }

    private short _Age = 18; // valeur initiale

    public short Age
    {
      get { return _Age; }
      set { _Age = value; }
    }

    public void Afficher()
    {
          Console.WriteLine($"Nom = {Nom}, Prenom = {Prenom}, Age = {Age}" );
    }

     public string Identite()
     {
        return Nom.ToUpper() + " " + Prenom.ToLower() ;
     }

} //Fin de la classe Personne
```

Possibilité d'écriture condensée : attribut privé implicite

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

Si certaines données membres doivent être en **lecture seule**, on ne fournit pas la méthode **set**, ainsi la donnée ne pourra pas être assignée depuis l'extérieur de la classe**.**

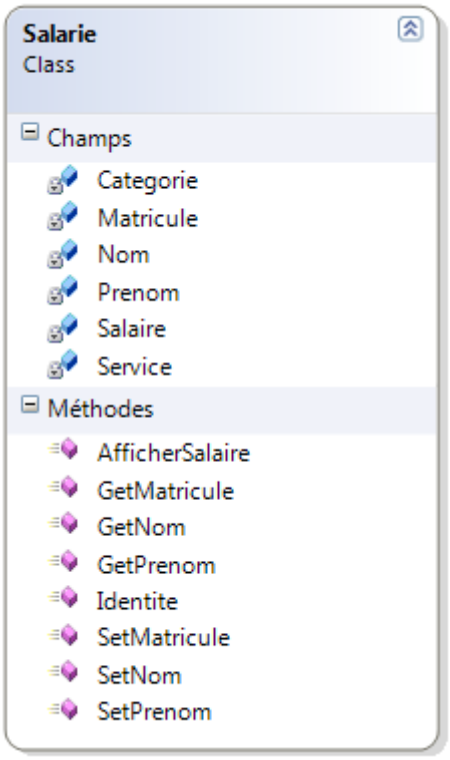
```
public short Age {get;}
public short Age {get; private set;}
```

Dans le main :

```
Personne p1 = new Personne();
p1.Nom = "GREGOIRE";
p1.Prenom = "Ghislaine";
p1.Age = 26;
Console.Writeline(p1.Nom);
```

2.3. TP2 - Travail à réaliser

Reprendre le TP1, modifier la classe Salarie pour :



- → Protéger toutes les données pour interdire l'accès direct depuis l'extérieur de la classe Salarie.
- → Créer les fonctions Get et Set d'accès aux propriétés pour chacune des propriétés.
- → Modifier la classe de test pour tester ces nouvelles méthodes.

Dans un deuxième temps, faire une nouvelle version du TP en utilisant les property pour encapsuler les données.

3. Constructeurs, propriétés et méthodes de classe

3.1. Objectifs

- Constructeurs d'objet
- Propriétés et méthodes de classes
- Destructeur

3.2. Ce qu'il faut savoir

3.2.1. Constructeurs

Lors de l'instanciation d'une variable objet : `p1 = new Personne()`, une méthode particulière est appelée, c'est le ***constructeur***. Elle permet d'affecter une valeur à chaque propriété pour que l'instance soit initialisée avec un contenu cohérent.

Jusqu'à présent :

- C'est le ***constructeur par défaut*** qui a été invoqué, il initialise chaque propriété à une valeur par défaut selon son type.
- Chaque propriété a été initialisée ensuite par un appel à la fonction ***Set*** correspondante.
- Notez que rien n'oblige l'utilisateur à appeler ces méthodes et donc à initialiser toutes les propriétés de l'instance.
- La valeur de chaque propriété peut être contrôlée par la fonction set correspondante mais aucune fonction ne contrôle la cohérence de l'instance.

Les constructeurs permettent de résoudre ce type de problème. Un constructeur est déclaré comme les autres fonctions membres de la classe à une différence près : c'est une ***procédure*** dont ***le nom est toujours identique à celui de la classe. Il peut avoir ou non des paramètres. Il n'a pas de retour.***

```
public class Personne
{
   //Déclaration des propriétés d'instance
   private string _Nom;

   public string Nom
   {
     get { return _Nom; }
     set { _Nom = value; }
   }

   //Declaration des constructeurs
   public Personne(string nom, string prenom, short age)
   {
      this.Nom = nom.ToUpper();
      this.Prenom = prenom.ToUpper();
      this.Age = age;
   }

… etc
}
```

Le constructeur ci-dessus est appelé ***constructeur d'initialisation*** ou ***constructeur complet*** car il affecte une valeur à chacune des propriétés de l'instance.

 Quand un constructeur avec paramètres est déclaré dans une classe, **le constructeur par défaut ne peut plus être utilisé**. Pour créer une instance, il faut appeler le constructeur et fournir une valeur pour chaque propriété au moment de l'instanciation : on pourra donc être sûr que toutes les propriétés seront initialisées.

```
Personne p1 = new Personne("GREGOIRE","Ghislaine", 26);
Personne p2 = new Personne();
```

 Notez d'autres façons d'appeler le constructeur :

```
Personne p1 = new Personne{Nom="GREGOIRE",Prenom="Ghislaine", Age=26};
Personne p1 = new Personne{Prenom = "Ghislaine", Nom = "GREGOIRE", Age = 26};
```

3.2.2. Surcharge des constructeurs

En revanche, dans une classe, il existe souvent plusieurs constructeurs. Chacun d'eux correspond à une intialisation particulière des objets. Les différents constructeurs ont le même nom, ils se distinguent par le nombre et le type des paramètres passés : cette possibilité s'appelle la ***surcharge***. On appelle ***signature*** chaque séquence distincte de paramètres, c'est la signature qui identifie quel constructeur sera appelé.

2 constructeurs sont remarquables :

- Le constructeur d'intialisation
- Le constructeur par défaut (sans paramètre)

Quand on crée une nouvelle classe, il est indispensable de prévoir tous les constructeurs nécessaires. Si on ne déclare aucun constructeur, un constructeur par défaut est disponible. Dès qu'un constructeur est déclaré, si on souhaite aussi disposer du constructeur par défaut, il faut le déclarer explicitement comme ici dans la classe Personne .

```
//Constructeur complet
public Personne(string nom, string prenom, short age)
{
    this.Nom = nom.ToUpper();
    this.Prenom = prenom.ToUpper();
    this.Age = age;
}

//Constructeur par défaut
public Personne()
{
    this.Nom = "BIDON";
    this.Prenom = "BIDON";
    this.Age = 0;
}
```

Il existe maintenant 2 possibilités de créer un objet Personne :

```
Personne p1 = new Personne("GREGOIRE","Ghislaine", 26);
Personne p2 = new Personne();
```

On peut simplifier l'écriture du code en chainant les constructeurs, c'est-à-dire en appelant le constructeur complet dans le constructeur par défaut.

```
public Personne() : this("Bidon", "Bidon", 0) {}
```

En conclusion :

- Si une classe n'a aucun constructeur explicitement déclaré, elle en dispose toujours d'un, sans paramètre, qui initialise chaque propriété à une valeur par défaut.
- Dès qu'un constructeur est déclaré, le constructeur par défaut doit aussi être déclaré si on souhaite l'utiliser (s'il a un sens).
- Quand une classe a un ou plusieurs constructeurs, l'un de ces constructeurs doit être obligatoirement appelé pour créer un objet de cette classe.

3.2.3. Propriété de classe

Jusqu'ici les propriétés qui ont été déclarées sont des ***propriétés d'instances***. C'est-à-dire que ***chaque objet***, instancié à partir de la même classe, possède ***sa propre valeur*** pour chaque propriété.

Supposons maintenant que l'on souhaite compter le nombre d'instances créées pour la classe Personne. Il semblerait judicieux que le constructeur puisse incrémenter un compteur à chaque fois qu'il est appelé.

Il faut donc définir dans la classe Personne, une ***propriété de classe***, c'est-à-dire une propriété qui possède ***une seule valeur*** qui est ***partagée entre toutes les instances de la classe***.

Une ***propriété de classe*** est déclarée par le mot-clé ***static.***

```
public class Personne
{
   //Déclaration de propriétés de classe
   private static int NbPersonnes = 0;

   public Personne(string nom, string prenom, short age)
   {
      this.Nom = nom.ToUpper();
      this.Prenom = prenom.ToUpper();
      this.Age = age;
      NbPersonnes = NbPersonnes + 1;
   }
…
}
```

3.2.4. Méthode de classe

Comme pour les autres propriétés privées, il est nécessaire de prévoir des fonctions d'accès associées. La propriété NbPersonnes est en lecture seule, elle est incrémentée par le constructeur et ne peut pas être modifiée depuis l'extérieur de la classe Personne.

Le message NbPersonnes ne doit pas être envoyé à une instance donnée mais à la classe elle-même, c'est donc une propriété **de classe**.

Déclaration d'une propriété de classe dans la classe Personne :

```
public static int NBPersonnes
{
    get { return _NBPersonnes; }
    private set { _NBPersonnes = value; }
}
```

Le mot-clé ***static*** indique qu'il s'agit d'une propriété ou une méthode ***de classe*** et non d'instance.

L'appel d'une propriété ou d'une méthode de classe diffère : en effet ce n'est pas à une instance particulière que le message est envoyé mais à la classe elle-même. En d'autres termes, il n'est pas nécessaire d'instancier la classe pour avoir accès à une méthode de classe.

Dans le bloc main, on pourrait écrire indifféremment :

```
int nb = Personne.NBPersonnes;
Console.WriteLine("Nb de personnes " + Personne.NBPersonnes);
```

3.2.5. Destructeur

Le destructeur est une méthode qui est appelée quand un objet est détruit. La destruction intervient ***automatiquement*** lorsqu'un objet est hors de portée ou en fin de programme, qu'il n'est plus référencé. C'est le ***Garbage Collector*** ou le ramasse-miettes qui se charge de repérer les objets candidats à la destruction et de les détruire ensuite, il libère ainsi la place mémoire associée.

Si un objet devient inutile, il est possible pour le programmeur d'indiquer qu'il est candidat à la destruction par : **`p1 = null`**

Attention, la destruction de l'objet n'est pas forcément synchrone.

Dans une classe, on ne déclare un destructeur uniquement si on a besoin d'écrire un traitement spécifique. C'est le cas de notre classe Personne : le nombre de personnes doit être décrémenté lors de la destruction d'une Personne.

Le destructeur est obligatoirement une ***méthode d'instance*** avec la signature ci-dessous :

```
~Personne()
{
    Console.WriteLine("Une personne est supprimée");
    NbPersonnes = NbPersonnes - 1;
}
```

3.2.6. Paramètre et retour de méthode de type Objet

En C#, les paramètres peuvent être passés ***par valeur*** ou ***par référence***.

Les paramètres de ***type valeur*** (int, …) ne se comportent pas de la même manière que les paramètres de ***type référence*** (objet).

Par défaut, les ***paramètres*** d'une méthode sont passés ***par valeur*** : c'est-à-dire que la méthode recopie les valeurs de paramètres passés dans des variables, si la méthode modifie le paramètre, la valeur d'origine ne sera pas modifiée dans le programme appelant.

Le passage de paramètres par référence permet aux méthodes de changer la valeur des paramètres et de conserver ces modifications au retour dans le programme appelant. Pour passer un paramètre par référence, on fait précéder le paramètre par le mot clé **ref** ou **out.**

Qu'en est-il des paramètres de type objet ?

Dans le cas d'un paramètre de type objet, il faut rappeler que ce n'est pas l'objet lui-même qui est transmis mais une référence sur cet objet. C'est donc la valeur de la référence qui sera copiée et non l'objet lui-même, il n'y a donc pas construction d'un nouvel objet, si la méthode modifie l'objet, c'est bien sur l'objet initial qu'elle travaille, l'objet sera modifié !

Retour de type object

Quand une fonction renvoie comme résultat un objet, elle renvoie une référence sur une nouvelle instance.

Exemple de la classe Fraction :

```
public class Fraction
{
   //Propriétes d'instance
   private int _Numerateur;
   public int Numerateur
   {
     get { return _Numerateur; }
     set { _Numerateur = value; }
   }
   private int _Denominateur;
   public int Denominateur
   {
     get { return _Denominateur; }
     set { _Denominateur = value; }
   }
   //Constructeurs
   public Fraction(int numerateur, int denominateur)
   {
     this.Numerateur = numerateur;
     this.Denominateur = denominateur;
    }
    //La somme de 2 fractions est une nouvelle fraction
    public Fraction Somme(Fraction f)
    {
       Fraction fResultat;
       fResultat = new Fraction(
                     Numerateur * f.Denominateur + f.Numerateur * Denominateur,
                     Denominateur * f.Denominateur);
       return fResultat;
    }
    //Le produit de 2 fractions est une nouvelle fraction égale au produit en croix
    public Fraction Produit(Fraction f)
    {
      return new Fraction(Numerateur * f.Numerateur,
                          Denominateur * f.Denominateur);
    }
}
```

3.3. TP3 - Travail à réaliser

Reprendre la classe Salarie réalisée dans le TP2 :

- → Coder au moins 2 constructeurs. Pour mettre en évidence le constructeur qui sera appelé lors des futures instanciations, vous afficherez un message à la console.
- → Ajouter les attributs et méthodes nécessaires pour compter les salariés créés.
- → Tester dans une classe de test l'ensemble de ces nouvelles fonctions.
- → En bonus, ajouter un constructeur qui permet de créer une instance de salarié à partir d'une autre instance de salarié passée en paramètre.

4. Héritage

4.1. Objectifs

- Héritage
- Classe Object

4.2. Ce qu'il faut savoir

4.2.1. L'héritage

L'héritage est un des principes fondamentaux de la POO. L'héritage consiste en la création d'une nouvelle classe dite ***classe dérivée*** ou ***sous-classe*** à partir d'une classe existante appelée ***super-classe*** ou ***classe de base***. On parle aussi de ***généralisation-spécialisation***.

L'héritage permet de :

- Récupérer le comportement standard d'une classe et de réutiliser les propriétés et des méthodes définies dans celle-ci,
- Ajouter des fonctionnalités supplémentaires en créant de nouvelles propriétés et de nouvelles méthodes dans la classe dérivée,
- Modifier le comportement standard d'une classe en remplaçant certaines méthodes de la classe parente dans la classe dérivée.

C'est un gain de temps dans les tâches de codage :

- Le programmeur dispose d'une bibliothèque d'objets standards qu'il peut adapter pour les besoins de son application.
- Pour la maintenance des applications, il est possible de corriger un comportement anormal d'une classe en dérivant cette classe et en subsistant les méthodes boguées. Il n'est même pas nécessaire d'avoir les sources.

Syntaxe pour indiquer qu'une classe hérite d'une classe :

```
class ClasseFille : ClasseParente
{
}
```

En C#, une classe hérite directement ***au plus d'une autre***, **pas d'héritage multiple.**

Notez qu'une classe déclarée ***sealed*** est une ***classe finale,*** elle ne pourra pas être dérivée. Dans le framework .net, c'est le cas de la classe String.

Reprenons notre exemple de la classe Personne : Un étudiant ***est une Personne.*** La classe Etudiant hérite des caractéristiques et du comportement de la classe Personne. Un étudiant possède une propriété supplémentaire : sa section.

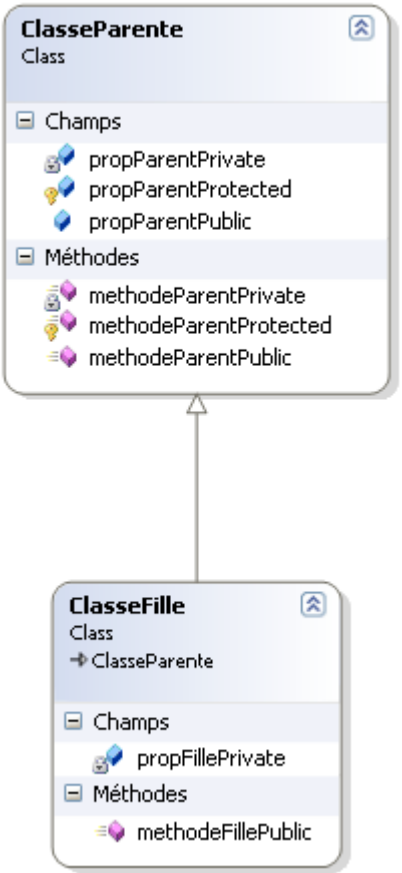
4.2.2. Visibilité des propriétés et des méthodes dans la classe Fille

En plus des mots-clés ***public*** et ***private*** utilisés dans les TP précédents, il est possible d'utiliser un niveau de protection intermédiaire : ***protected***.

Rappelons que :

- Les propriétés ou les méthodes privées d'une classe sont accessibles qu'à l'intérieur de cette classe donc pas dans les classes dérivées.
- Les propriétés ou les méthodes publiques d'une classe sont accessibles partout.
- Les propriétés ou les méthodes déclarées avec le mot-clé *protected* sont accessibles dans les classes dérivées par héritage.

Pour comprendre les différences entre ***private***, public et ***protected***




```
public class ClasseParente
{
    private     int PropParentPrivate;
    protected   int PropParentProtected;
    public      int PropParentPublic;

    private void MethodeParentPrivate()
    {
       Console.WriteLine ("Méthode privée parent");
    }

    protected void MethodeParentProtected()
    {
       Console.WriteLine("Méthode protected parent");
    }

    public void MethodeParentPublic()
    {
       Console.WriteLine("Méthode publique parent");
    }
}
```

```
public class ClasseFille : ClasseParente
{
    private int PropFillePrivate;

    public void MethodeFillePublic()
    {
       Console.WriteLine("Methode spécifique ClasseFille");
       Console.WriteLine(PropFillePrivate);

       Console.WriteLine(PropParentProtected);
       Console.WriteLine(PropParentPublic);

       MethodeParentProtected();
       MethodeParentPublic();
     }
}
```

A l'extérieur des classes :

```
class Testeur
{
   public static void main(string[] args)
   {
      ClasseParente   cp = new ClasseParente();
      ClasseFille     cf = new ClasseFille();

      Console.WriteLine(cp.PropParentprivate);
      Console.WriteLine(cp.PropParentProtected);
      Console.WriteLine(cp.PropParentPublic);

      cp.MethodeParentPrivate();
      cp.MethodeParentProtected();
      cp.MethodeParentPublic();

      Console.WriteLine(cf.PropParentprivate);
      Console.WriteLine(cf.PropParentProtected);
      Console.WriteLine(cf.PropParentPublic);

      cf.MethodeParentPrivate();
      cf.MethodeParentProtected();
      cf.MethodeParentPublic();

      cf.MethodeFillePublic();
   }
}
```

4.2.3. Constructeurs dans une classe dérivée

Un Etudiant est une Personne. Pour créer une instance d'Etudiant, il faut d'abord créer une instance de Personne et affecter ensuite des valeurs aux propriétés spécifiques de l'étudiant.

En conséquence, le constructeur de la classe dérivée doit obligatoirement appeler un constructeur de la classe Parente. : ***base()*** appelle le constructeur correspondant de la classe Parente en lui passant en paramètres les valeurs nécessaires.

```
class Etudiant : Personne
{
  private string _Section;

  public string Section
   {
     get { return _Section;}
     set { _Section = value;}
   }

  public Etudiant(string nom, string prenom, short age, string section)
                : base(nom, prenom, age)
  {
    this.Section = section;
  }
}
```

En résumé, le constructeur d'une classe dérivée :

- Appelle un des constructeurs de la classe de base en lui passant tous les paramètres dont il a besoin
- Initialise les propriétés spécifiques de sa classe.

4.2.4. Redéfinition des méthodes de la classe Parente

Les méthodes de la classe de base qui ont été déclarées avec le mot clé `virtual` peuvent être redéfinies dans la classe Dérivée. Pour redéfinir une méthode dans la classe dérivée, il faut conserver la même signature et préciser le mot clé `override`.

Dans la classe de base Personne :

```
public virtual void Afficher()
{
    Console.WriteLine($"Nom = {Nom}, Prenom = {Prenom}, Age {Age}" );
}

public virtual string Identite()
{
   return Nom.ToUpper() + " " + Prenom.ToLower() + " " + Age + "ans";
}
```

Dans la classe Dérivée Etudiant:

```
public override void Afficher()
{
    base.Afficher();
    Console.WriteLine("Section = " + Section);
}

public override string Identite()
{
    return base.Identite() + " section : " + Section;
}
```

Notez que on peut faire appel à la méthode de la classe Parente `base.Methode()`pour réutiliser le code de la classe de base (dont on ne dispose pas toujours).

De façon générale, si O est un objet et M une méthode, pour exécuter la méthode O.M(), le système cherche la méthode M de la manière suivante :

- Dans la classe de l'objet O
- Dans sa super classe s'il en a une
- Dans la super classe de sa classe parente si elle existe …

4.2.5. Compatibilité et affectation, casting

Quelle sont les règles à respecter lors des affectations ?

Soit 2 instances :

```
ClasseParente cp = new ClasseParente();
ClasseFille cf = new ClasseFille();
```

L'affectation **`cp = cf`** est autorisée car une instance de la classe Fille est aussi une instance de la classe Parente. Il n'y a pas de conversion, l'objet référencé par cp reste une instance de la classe Fille.

L'affectation inverse **`cf = cp`** génére une erreur de compilation. Mais si cp référence une instance de la classe Fille alors on peut effectuer une conversion vers la ClasseFille :

Il est possible de forcer la conversion en faisant un **casting** :

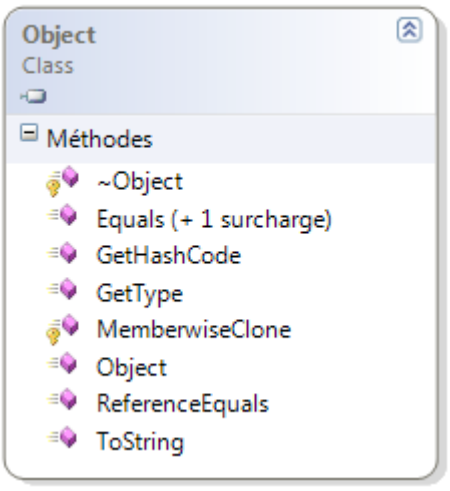
```
cf = (ClasseFille) cp;
```

L'instruction ci-dessus peut provoquer une InvalidCastException à l'éxécution si la conversion n'est pas possible (si cp n'est pas une instance de ClasseFille) .

La classe Convert fournit des méthodes statiques pour les conversions usuelles.
Voir aussi Opérateurs is et as

4.2.6. Object comme classe de base

Toute classe hérite de la classe **Object** qui est la ***classe racine de toutes les classes .net.***



Object définit le comportement commun à toutes les classes.

Le constructeur par défaut de toute classe est hérité de la classe Objet.

La méthode ~Object, destructeur, protected, est également héritée par toute classe. Il faut la redéfinir si on souhaite l'adapter.

4.2.6.1. Méthode ToString

La méthode ToString garantit que tout objet peut être converti en chaine de caractères (pour afficher ses caractéristiques par exemple).

La méthode ToString définie dans la classe Object affiche le type de l'instance (sa classe). C'est le comportement par défaut pour tout objet.

La méthode ToString est virtuelle, il est conseillé de la redéfinir dans chaque classe créée pour avoir un comportement spécifique.

Un exemple ci-dessous pour la classe Personne :

```
public override string ToString()
{
   return base.ToString() + $"Nom = {Nom}, Prenom = {Prenom}, Age {Age}";
}
```

4.2.6.2. Méthode Equals

Quand 2 objets sont comparés **p1==p2**, ce sont leurs références qui sont comparées et non les contenus. (ceci ne s'applique pas aux string pour lesquelles on compare les contenus)

Or 2 références peuvent correspondre à la même personne et donc avoir des caractéristiques identiques : nom, prenom, age.

La méthode Equals permet de comparer une instance de la classe avec un autre objet quelconque puisque le paramètre fourni est du type object. Par défaut, la méthode Equals définie dans la classe Object renvoie vrai si les références des objets sont égales, faux sinon. Pour modifier ce comportement dans une classe donnée, il faut redéfinir la méthode Equals.

Pour la classe Personne, par exemple, on peut considérer que 2 instances sont identiques si :

- Elles sont du même type
- Elles ont même nom, même prénom et même age.

```
public override bool Equals(object obj)
{
   if (obj is Personne) //2 personnes : on compare alors chacune de leurs
caractéristiques
   {
     Personne p = (Personne) obj;
     return Nom == p.Nom && Prenom == p.Prenom && Age == p.Age;
   }
   else
   {
     return false;
   }
}
```

Utilisation

```
if (p2.Equals(p3)) { ….}
```

A noter, la méthode statique ReferenceEquals(Object, Object) de la classe Object permet de comparer l'égalité de 2 références.

```
if (Personne.ReferenceEquals(p1, p3)) { ….}
```

Il est conseillé de redéfinir aussi les opérateurs `==` et `!=`, ce qui permettra de comparer les objets aussi avec == ou !=.

En conclusion :

- L'héritage garantit que les classes dérivées d'une classe Parente possèdent toutes un certain nombre de méthodes. En d'autres termes, la **superclasse** définit un **comportement commun** pour ses **sous-classes**.
- Centraliser du code commun dans une superclasse permet à toutes les sous-classes d'en hériter : quand vous voulez changer ce comportement, il suffit de modifier la superclasse et toutes les sous-classes verront le changement.

Attention à ne pas surutiliser l'héritage. L'héritage doit être utilisé pour traduire une relation « **est une sorte de** » entre une sous-classe et une superclasse.

TP4 – Travail à effectuer

→ Reprendre la classe Salarie écrite dans le TP3 :
- Redéfinir les méthodes ToString héritée de la classe Object pour renvoyer toutes les propriétés d'un salarié séparées par une virgule.
- Redéfinir la méthode Equals héritée de la classe Object : on considèrera que 2 salariés sont égaux, s'ils ont même matricule, même nom et même prénom.

→ Créer ensuite une classe Commercial dérivée de la classe Salarie
- L'encapsulation doit être mise en œuvre
- Un Commercial a 2 propriétés supplémentaires : le chiffre d'affaire réalisé (double) et le pourcentage de commission (entier)
- Redéfinir la méthode Afficher pour afficher le salaire (fixe et commission)
- Redéfinir d'autres méthodes si nécessaire.

→ Créer un main pour tester les classes Salarie et Commercial.

5. Polymorphisme

5.1. Objectifs

- Comprendre le polymorphisme d'héritage
- Méthodes virtuelles

5.2. Ce qu'il faut savoir

Dans le chapitre précédent, nous avons vu qu'il est possible d'écrire :

```
Personne p1 = new Etudiant("Durand", "Anthony", 18, "Anglais");
```

La référence est du type Personne, l'objet est du type Etudiant. Une telle affectation est correcte et ne génère pas d'erreur de compilation car tout objet qui satisfait au test "*Est une Personne"* peut être affecté à une référence de type Personne.

Que se passe-t-il quand le message `Identite()` est envoyé à `p1` ?

```
string str = p1.Identite();
```

Est-ce la méthode `Identite` de la classe Personne ou celle redéfinie dans la sous-classe Etudiant qui sera appelée ? Un simple test permet de mettre en évidence que c'est bien la méthode Identite de la classe Etudiant qui est appelée. En effet, le système sait et se souvient comment les objets ont été créés pour pouvoir appeler la bonne méthode. C'est au moment de l'exécution que l'ambigüité est levée en fonction de la classe qui a été utilisée pour créer l'instance. On dit que la méthode Identite est une ***méthode virtuelle***et qu'elle a un ***comportement polymorphique***.

En d'autres termes, c'est la méthode la moins haut placée dans la hiérarchie des classes qui l'emporte.

Pour qu'une méthode ait un comportement polymorphique, il faut réunir les 2 conditions :

- ***héritage***
- ***redéfinition*** de la méthode.

Pour redéfinir une méthode de la classe de base, il faut ***respecter le contrat***, c'est-à-dire respecter la signature exacte de la méthode : nom de la méthode, liste des paramètres (même nombre, même ordre, même type) et le type du retour.

Les méthodes ***ToString*** et ***Equals*** de la classe Object et redéfinies dans les classes Filles sont des méthodes virtuelles, elles ont un comportement polymorphique.

5.2.1. TP5 – Travail à réaliser

A partir des classes Salarie et Commercial écrites dans les précédents TP, mettre en évidence le comportement polymorphique des méthodes Afficher, ToString et Equals.

6. Classe Abstraite et interfaces

6.1. Objectifs

- Classe abstraite
- Interface

6.2. Ce qu'il faut savoir

6.2.1. Classe abstraite

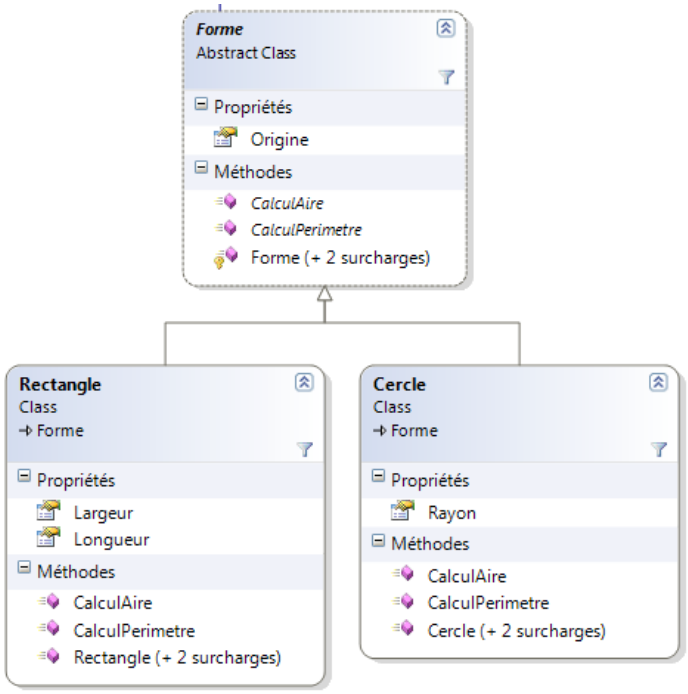
Dans certains cas, lors de l'élaboration d'une hiérarchie de classes, il peut être intéressant de ***mettre en facteur*** dans une superclasse un ***comportement commun à plusieurs classes dérivées***, il peut arriver aussi que cette superclasse ne soit pas suffisamment complète pour que des objets puissent être instanciés à partir de celle-ci. C'est le cas de la classe Forme définie ci-après.

La classe Forme est une ***classe abstraite*** est déclarée avec le mot-clé ***abstract, elle ne peut pas être instanciée, elle est destinée à être dérivée.***

```
public abstract class Forme
{
…
}
```

En effet, instancier une Forme n'a pas de sens, on ne peut instancier qu'un cercle ou un rectangle (les classes concrètes dérivées).

```
Forme f1, f2;
f1 = new Forme();
f1 = new Rectangle();
f2 = new Cercle();
```



6.2.2. Méthode abstraite

Quand une méthode ne peut pas être écrite dans une superclasse (parce que c'est trop tôt), elle peut être déclarée abstraite avec le mot clé ***abstract.*** Une méthode abstraite n'a pas de corps.

```
public abstract class Forme
{
    public abstract double CalculPerimetre();
    public abstract double CalculAire();
}
```

Une méthode abstraite devra ***obligatoirement*** être redéfinie dans toutes les classes dérivées.
Une classe abstraite définit ***un contrat***.

Si une sous-classe hérite de Forme alors elle s'engage à écrire les méthodes CalculPerimetre et CalculAire.

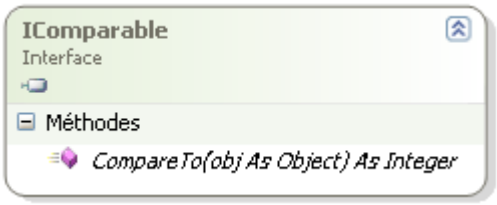
```
public class Rectangle : Forme
{
…
    public override double CalculAire()
    {
       return Longueur * Largeur;
    }

    public override double CalculPerimetre()
    {
       return (Longueur + Largeur) * 2;
    }
}
```

6.2.3. Interface

Les interfaces permettent d'uniformiser le comportement d'objets.

Une ***interface*** est un ensemble de ***prototypes*** de méthodes et/ou de propriétés qui forment un **contrat**. Une interface peut être vue comme une classe abstraite "pure" dans laquelle aucune méthode n'est implémentée.

Il existe de nombreuses interfaces dans le Framework .net.
Par exemple, l'interface ***IComparable*** a été décrite de la manière suivante :




```
public interface IComparable
{
      int CompareTo (Object obj);
}
```

Par convention, en .net, le nom des interfaces commence par un I.

Une classe qui décide d'***implémenter*** une interface s'***engage à fournir une implémentation pour toutes les méthodes définies dans l'interface***. Cette vérification est faite à la compilation.

Dans la classe Personne :

```
class Personne : IComparable {
      …
      public int CompareTo(Object obj)
      {
            Personne other = obj as Personne;
            if (other != null)
               return this.Nom.CompareTo(other.Nom);
            else
               throw new ArgumentException("L'objet n'est pas une Personne");
      }
}
```

Utilisation :

```
Console.WriteLine(p1.CompareTo(p2));
```

On peut donc être sûr qu'un objet Personne a donc "***la capacité à se comparer à un autre objet***".

Une **classe ne peut hériter que d'une seule classe**, en revanche, **une classe peut implémenter plusieurs interfaces** : c'est un biais pour implémenter l'héritage multiple.

Une instance de Personne est du type Personne mais aussi du type IComparable.
Les interfaces sont très utilisées pour typer des paramètres de méthodes.

La classe String implémente notamment les interfaces IComparable, IConvertible.

6.3. TP6 - Travail à réaliser

Reprendre la classe Salarie écrite dans les précédents TP et implémenter l'interface System.IComparable qui permettra à toute instance de Salarie de se comparer à une instance d'Object :

- Un salarié est supérieur à un objet quelconque ou à un objet qui est null
- Comparer 2 salariés revient à comparer leur salaire de base.

7. Collectionner des objets

7.1. Objectifs

- Utiliser des tableaux
- Utiliser des collections

7.2. Ce qu'il faut savoir

Les **tableaux** comme les **collections ne contiennent pas directement les objets, ils conservent des références sur les objets**, ils permettent de les manipuler comme un ensemble.

7.2.1. Les tableaux

- Un tableau stocke des éléments d'un type défini.
- Le nombre d'éléments d'un tableau doit être défini au moment de sa déclaration.
- Pour placer un objet dans un tableau, vous devez connaitre son emplacement (indice compris entre 0 et la taille du tableau)

```
//Tableau de constantes
string[] tabNoms = {"AA", "AB", "AC"};
int[] tabEntiers; //tableau pas encore alloué
tabEntiers = new int[3];

//Tableau de personnes
Personne[] tabPersonnes = new Personne[10];

//Le tableau est vide : les références de chaque élément sont égales à null
tabPersonnes[0] = new Personne("GREGOIRE", "Ghislaine", 26);
tabPersonnes[1] = new Etudiant("GREGOIRE", "Laurie", 20, "COMMERCE");
tabPersonnes[2] = new Personne("BOISSEAU", "Serge", 25);

//Parcours avec une boucle for
for (int i = 0; i < tabPersonnes.Length; i++)
{
    if (tabPersonnes[i] != null)
        Console.WriteLine (tabPersonnes[i]);
}

//Parcours avec une boucle for each
foreach (Personne pers in tabPersonnes)
{
    if (pers != null)
       Console.WriteLine (pers);
}
```

Un **tableau est un objet de la classe *Array*.**
La classe Array possède des propriétés d'instance : Length et méthodes d'instance Initialize, Clear, CopyTo, …
Cette classe possède notamment une méthode statique **Sort** qui trie les objets du tableau à condition que leur classe implémente l'**interface IComparable**.

7.2.2. Les collections

Une alternative aux tableaux est l'usage de collections. Les collections sont des classes du Framework .net. (Voir l'espace de noms System.Collections)

Le nombre d'éléments d'une collection n'est pas défini à l'avance, la ***taille*** de la collection ***augmente dynamiquement*** au fur et à mesure des besoins.

Les ***collections*** permettent :

- de stocker ***un nombre quelconque d'éléments***
- d'ajouter un élément à n'importe quel endroit
- d'enlever un élément sans avoir à déplacer les autres

Il existe plusieurs implémentations des collections, on peut les classer de la manière suivante :

→ Les ***listes*** permettent d'utiliser un **indice**.
- ArrayList
- List ou List<T>

→ Les ***dictionnaires*** gèrent des **clés** pour accéder aux objets.
- Hashtable
- Dictionary<T>

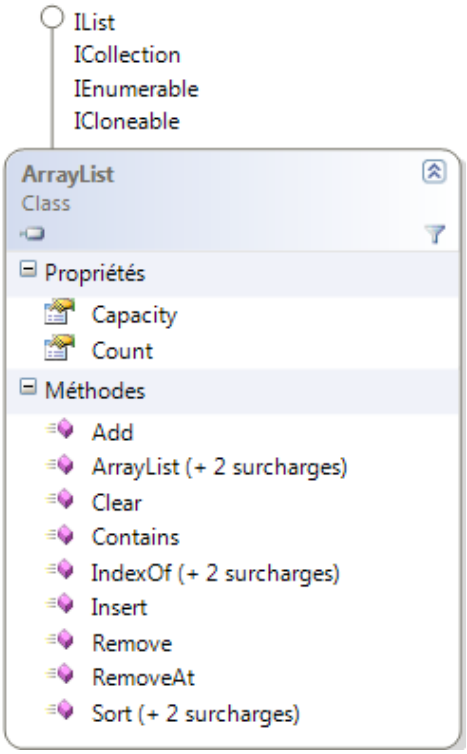
→ Les ***listes-dictionnaires*** utilisent **une clé** ou **un indice**, la collection est ***triée automatiquement***
- SortedList

→ Les ***files*** gèrent les éléments en ***FIFO*** (First In First Out)
- Queue ou Queue<T>

→ Les ***piles*** gèrent les éléments en ***LIFO*** (Last In First Out)
- Stack ou Stack<T>

→ Les ***listes chaînées*** dans lesquelles chaque élément sait trouver ***le précédent, le suivant***
- LinkedList<T>

Choisir le type de collection adaptée à vos besoins :
https://msdn.microsoft.com/fr-fr/library/6tc79sx1(v=vs.110).aspx



7.2.2.1. Une collection simple : Arraylist

→ Créer une liste (il n'est pas nécessaire d'indiquer sa capacité)

```
ArrayList liste = new ArrayList();
```

→ Ajouter des éléments à la liste

```
Personne p = new Personne("DURAND", "marie", 47);
liste.Add(p);
liste.Add(new Etudiant("DURAND", "Paul", 20, "COMMERCE"));
liste.Add(new Personne("BOISSEAU", "Serge", 42));
```

Une ArrayList peut contenir des objets de types différents.

→ Connaitre le nombre d'éléments de la liste

```
int nbElements = liste.Count;
```

→ Savoir un objet est présent dans la liste

```
if (liste.Contains(p)) //Utilise Equals
   Console.WriteLine("La liste contient l'objet " + p);
```

→ Trouver la place d'un objet (son indice dans la liste)
```
int indice = liste.IndexOf(p);
```

→ Enlever un objet dans la liste
```
liste.Remove(p);
liste.RemoveAt(indice);
```

→ Effacer tous les éléments de la liste
```
liste.clear();
```

→ Parcourir la liste avec un indice (idem tableau)
```
for (int i = 0; i < liste.Count - 1; i++)
{
    Console.WriteLine($"Item {i}= {liste[i]}");
}
```
→ Parcourir la liste avec une boucle for … each
```
foreach (Object obj in liste)
{
    Console.WriteLine(obj);
}
```

7.2.2.2. Une collection fortement typée : List<T>

Il existe des ***collections génériques*** : ***fortement typées***, qui interdisent de stocker des éléments d'un autre type que celui qui a été défini au moment de la création et dont les méthodes retournent des objets du type défini.

Créer une liste de Personnes, y ajouter des personnes
```
List<Personne> listeDePersonnes = new List<Personne>();
listeDePersonnes.Add(new Personne("GREGOIRE", "Ghislaine", 25));
listeDePersonnes.Add(new Etudiant("gregoire", "Laurie", 20, "Commerce"));
~~listeDePersonnes.Add(new Object())~~ ;//interdit par les listes fortement typées
```

Quand on parcourt la liste de Personne, on récupère un objet de type Personne et non de type Object, on a ainsi accès aux méthodes de la classe Personne.

```
foreach (Personne pers in listeDePersonnes)
{
        Console.WriteLine(pers.Identite());
}

for (int i = 0; i < listeDePersonnes.Count - 1; i++)
{
    Console.WriteLine(listeDePersonnes[i].Identite());
}
```

7.2.2.3. Une collection de type dictionnaire : SortedDictionary <T>)

Les collections de type Dictionnaire permettent d'associer une ***clé*** à chaque objet pour le retrouver plus facilement. De plus, la collection ***SortedDictionary*** ***trie*** les éléments automatiquement dans l'ordre de leurs clés.

→ Créer une liste de départements
```
SortedDictionary<int, string> ListeDepartements = new SortedDictionary<int, string>();
```

→ Ajouter des départements (clé, valeur)
```
ListeDepartements.Add(12, "AVEYRON");
ListeDepartements.Add(81, "TARN");
ListeDepartements.Add(34, "HERAULT");
```

→ Rechercher un élément à partir de sa clé
```
if (ListeDepartements.ContainsKey(48))
    Console.WriteLine(ListeDepartements[48]);
```

→ Parcourir les valeurs du dictionnaire
```
foreach(Object dept in ListeDepartements.Values)
{
    Console.WriteLine(dept);
}
```

7.2.2.4. Utiliser un itérateur pour parcourir les éléments

Les collections ont un comportement commun : elles implémentent l'interface IEnumerable. Elles possèdent toutes une méthode GetEnumerator qui renvoie un itérateur (type IEnumerator).

Cet objet permet d'énumérer les éléments de tous les types de collections de la même manière, sans avoir à connaitre comment la collection stocke ces éléments :

- Lors de sa création, l'énumérateur se place **avant le premier élément**
- La méthode **MoveNext** déplace le curseur sur l'élément suivant et renvoie faux quand il n'y a plus d'éléments à traiter
- La propriété **Current** désigne l'objet courant (type Object ou type T si la collection est fortement typée).

La procédure ci-dessous affiche le contenu de toute structure de données passée en paramètre tout autant qu'elle implémente l'interface IEnumerable.

```
public static void PrintValues (IEnumerable objEnumerable)
{
   IEnumerator ie = objEnumerable.GetEnumerator();
   while (ie.MoveNext())
   {
      Console.WriteLine(ie.Current);
   }
}
```

Ici les éléments retournés sont de type Object, on devra caster ces objets pour accèder à leurs méthodes propres (si on est sûr de leur type).

7.2.2.5. Une collection comme propriété d'un objet

Une propriété d'un objet peut être une collection.

Par exemple, la classe ListBox possède une propriété publique nommée items qui est une collection qui gèrent les éléments à afficher.

```
listBox1.Items.add("Un item")
listBox1.Items.clear()
listBox1.Items[0]
```

7.3. TP7 - Travail à réaliser

→ Créer un tableau de Salariés (Salarié et Commercial) :
- Y ranger au moins 3 instances de la classe Salarie
- Afficher son contenu
- Trier le contenu du tableau, constater que la méthode sort utilise la méthode CompareTo de IComparable

→ Créer une liste de type ArrayList pour collectionner des de salariés (Salarié et Commercial) :
- Y ranger au moins 3 instances de la classe Salarie
- Appeler la méthode Contains et vérifier en débogage qu'elle appelle la méthode Equals
- Afficher le contenu de la liste
- Afficher le contenu de la liste en utilisant un itérateur (GetEnumerator)

→ Créer une Liste fortement typée pour collectionner des salariés (Salarié et Commercial) :
- Y ranger au moins 3 instances de la classe Salarie
- Tenter d'y ranger un objet d'un autre type que salarié
- Essayer d'y ranger 2 objets identiques
- Afficher son contenu

→ Créer un Dictionnaire de Salariés (Salarié et Commercial) :
- La clé pourra être le matricule
- Y ranger au moins 3 instances de la classe Personne (la clé peut être le matricule)
- Rechercher un salarié à partir de sa clé

→ Créer une SortedList ou un SortedDictionnary
- Y ranger au moins 3 instances de la classe Personne (la clé peut être le matricule)
- Vérifier que les éléments sont automatiquement triés

Faire un tableau de synthèse pour noter les avantages et inconvénients de ces différents types de collections.

Pour vous aider à choisir la collection adaptée à vos besoins :
https://msdn.microsoft.com/fr-fr/library/6tc79sx1(v=vs.110).aspx

8. Exceptions

8.1. Objectifs

- Créer ses propres classes d'exception
- Lever une exception
- Gérer les exceptions

8.2. Ce qu'il faut savoir

8.2.1. La robustesse des classes

Une classe robuste est une classe exempte de bogues. Pour ce faire, il est indispensable d'intégrer la gestion des exceptions dans les classes développées et éventuellement de créer des classes d'exception pour gérer les cas critiques propres à la classe. La génération des exceptions et leur gestion mettent en œuvre les concepts objet : héritage et polymorphisme.

8.2.2. La classe Exception et ses filles

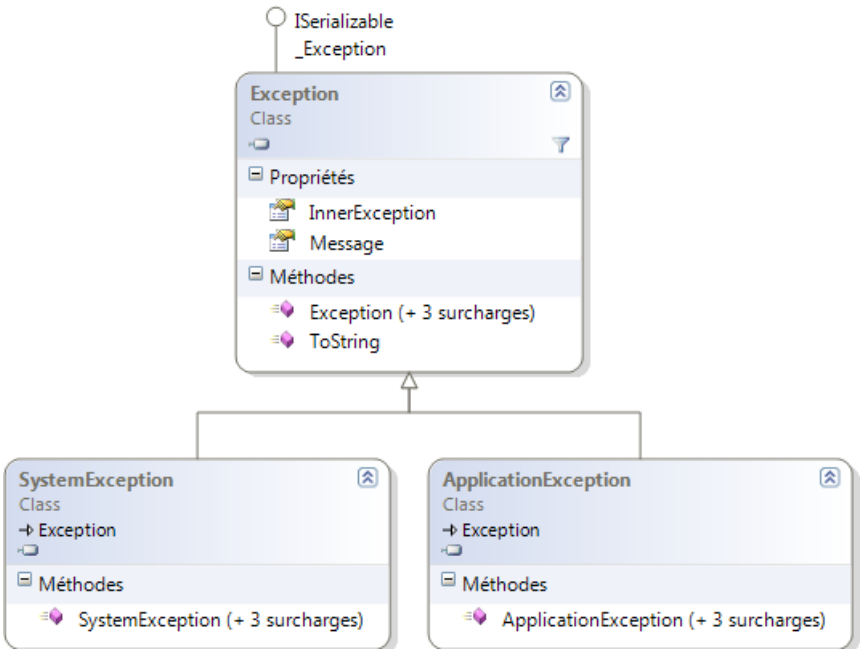
Le framework .net dispose de la classe ***Exception*** qui est la classe de base pour toutes les exceptions. Lorsqu'une erreur se produit, le système ou l'application en cours d'exécution la signale en levant une exception qui contient des informations sur l'erreur. Une fois levée, une exception est gérée par l'application ou par le gestionnaire d'exceptions par défaut.

La classe Exception contient une propriété remarquable : le ***message***, c'est lui qui est passé en paramètre au constructeur lors de la création d'une instance et qui sera affiché par la méthode ***ToString().***

Il existe plusieurs types d'exceptions.

La classe Exception est dérivée en :

- Les **SystemException** levées par le Common Language Runtime en cas d'erreurs récupérables par des programmes utilisateur.
  Elles sont à leur tour dérivées en : FormatException, IoException, IndexOutOfBoundException, ….etc.
- Les **ApplicationException** levées par les applications utilisateurs.



8.2.3. Création d'une nouvelle classe Exception

La création d'une nouvelle classe d'exception s'impose quand on veut filtrer, identifier les exceptions propres à une classe donnée.

3 étapes :

- Dériver la classe Exception pour créer une nouvelle classe par héritage.
- Créer un constructeur pour la nouvelle classe compatible avec les arguments que l'on souhaite afficher dans le message d'erreur
- Redéfinir la méthode ToString() : cette méthode est utilisée pour afficher le message d'erreur.


Par convention, le nom des classes dérivant de la classe Exception se termine par le mot Exception.

```
class PersonneAgeException : Exception
{
   public PersonneAgeException(Personne p): base(p.ToString()) {}

   public override string Message
   {
      get
      {
        return base.Message + " L'age est hors limites" ;
      }
    }
}
```


Il existe un extrait de code bien pratique pour insérer tous les constructeurs d'une Exception.

8.2.4. Génération d'exceptions

Pour commencer, il faut recenser toutes les méthodes de la classe susceptibles de générer une exception : méthodes effectuant un contrôle d'intégrité de l'objet par exemple.

Si on considère la classe Personne, la valeur attribuée à la propriété âge devrait être contrôlée et se situer dans une fourchette de 0 à 120 par exemple. Il faut donc revoir le constructeur et la méthode setAge pour qu'elle puisse contrôler cette donnée.

Dans la classe Personne :

La génération d'une nouvelle exception s'effectue par l'instruction ***throw*** :

```
public short Age
{
    get { return _Age; }
    set
    {
      _Age = value;
      if (value < 0)
      {
         throw new PersonneAgeException(this);
      }
    }
}
```

L'instruction ***throw*** effectue un branchement inconditionnel vers le bloc catch correspondant au traitement de cette exception.

Lors de l'utilisation :

```
try
{
    Personne p = new Personne("ESSAI", "Test", -1);
}
catch (PersonneAgeException ex)
{
    Console.WriteLine(ex.Message); //ou Console.Writeline(ex);
}
```

8.3. TP8 - Travail à réaliser

→ Ajouter des contrôles d'intégrité aux classes Salarie et Commercial écrites dans les TP précédents :
- Le salaire est toujours positif
- La catégorie ne peut être que 1(Cadre), 2(Technicien), 3(Employé)

→ Créer les classes d'exception nécessaires pour permettre de distinguer les erreurs

→ Modifier les constructeurs et les méthodes set concernées pour générer les exceptions appropriées

→ Dans un bloc main, tester ces méthodes et récupérer les erreurs.

9. Un peu de conception : Héritage vs agrégation

La technique d'héritage a été largement abordée dans ce support. A l'***héritage***, on peut opposer l'***agrégation***, L'agrégation consiste à ***utiliser comme membres d'une classe des instances d'autres classes***.

La tentation des débutants est de surutiliser l'héritage parce que cela fait plus objet.

Quand doit-on utiliser l'agrégation plutôt que l'héritage ? Il existe un moyen pour faire le choix entre héritage et agrégation :

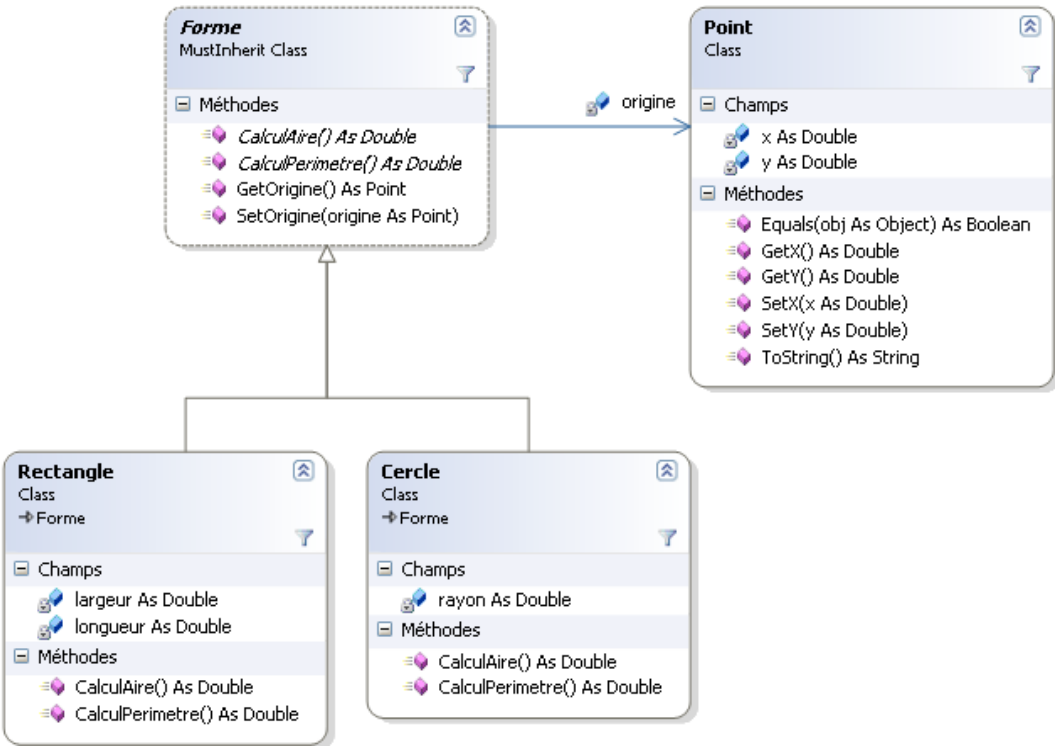
- Si la relation entre deux classes est exprimée sous la forme "***est une sorte de …"*** alors l'héritage s'impose.
- Si la relation entre 2 classes est exprimée sous la forme "***a un …*** " ou "***est composé de …***" alors on est en présence d'une ***agrégation***.

Exemple : On veut construire un modèle pour représenter des formes géométriques planes telles que des cercles, des rectangles, des carrés.

Toutes ces formes géométriques ont un comportement commun :

- Une forme ***a*** un point d'origine défini par ses coordonnées x et y (Cette relation est une agrégation)
- Une forme sait calculer son périmètre et son aire.

Un ***cercle est une forme géométrique***, il est caractérisé par son ***rayon***.
Un ***rectangle est une forme géométrique***, il est caractérisé par une ***longueur*** et une ***largeur***.
Un ***carré est un rectangle*** particulier pour lequel la largeur est égale à la longueur.



10. Bonnes pratiques pour créer une nouvelle classe

La démarche à suivre pour créer une nouvelle classe :

→ Recenser ses données membres
→ Protéger les données membres et créer les fonctions d'accès nécessaires à l'encapsulation
→ Créer des constructeurs en envisageant les cas d'usage.
→ Énumérer les opérations à opérer sur l'objet (entrées/sorties, calculs, conversion, comparaison, …)
→ Redéfinir afin de les substituer les méthodes héritées de la classe Object (ToString et Equals)
→ Repérer les exceptions générées par les fonctions utilisées dans ces méthodes et les traiter
Identifier les contrôles nécessaires pour assurer l'intégrité de l'objet et créer les classes d'exceptions correspondants.

11. Documenter les classes

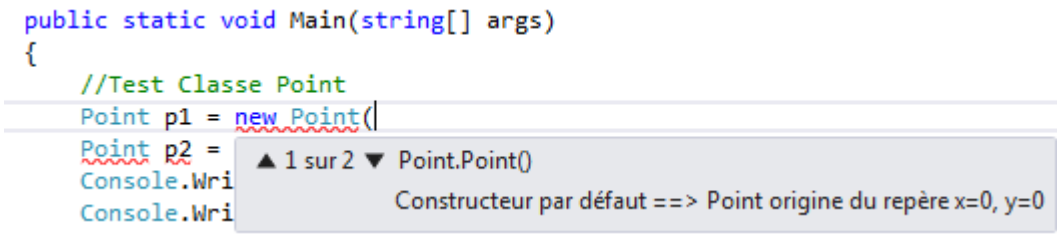
La documentation des classes se base sur des commentaires XML.

Pour documenter une classe, il suffit de placer le curseur devant chaque item à documenter (classe, méthode, property, …) et de taper ///. Un modèle de commentaire XML est alors généré à partir de la signature de la méthode, il reste à le compléter.

Ci-dessous un aperçu de quelques commentaires XML :

```
namespace FormesGeometriques
{
    /// <summary>
    /// Classe qui représente un point du repère
    /// </summary>
    public class Point
    {

    #region Constructeurs

        /// <summary>
        /// Constructeur par défaut ==> Point origine du repère x=0, y=0
        /// </summary>
        public Point() : this(0,0) { }

        /// <summary>
        /// Construit un point du repère
        /// </summary>
        /// <param name="abscisse">Abscisse du point</param>
        /// <param name="ordonnee">Ordonnée du point</param>
        public Point(int abscisse , int ordonnee)
        {
            this.Abscisse = abscisse;
            this.Ordonnee = ordonnee;
        }
    #endregion
….
```

Les commentaires sont affichés par l'IntelliSense.



En savoir plus sur les balises recommandées : http://msdn.microsoft.com/fr-fr/library/b2s063f7.aspx

Index