Notions de programmation Orientée Objet

1. Concept d'abstraction

› Consiste à identifier une entité en mettant en évidence ses caractéristiques pertinentes de points de vue de son utilisation.

➔ Le concepteur doit focaliser sa modélisation uniquement sur les aspects importants des objets.

› Exemple: La voiture est considérée de point de vue de:

- Sa ***mécanique*** pour un mécanicien
- Son ***équipement*** et de son ***prix*** pour un concessionnaire
- etc.

2. Concept d'encapsulation

L'encapsulation est un principe qui consiste à protéger un objet de manière à:

› Cacher ses détails d'implémentation aux entités externes

➔ L'ensemble est considéré comme une boîte noire ayant un comportement et des propriétés spécifiques.

› Cacher ses données des modifications externes.

➔ Utilisation des accesseurs: les propriétés (méthodes « get » et « set »).

› L'accès aux propriétés de l'objet n'est possible qu'à travers les méthodes de la classe.

› Faciliter et centraliser la MAJ du code des méthodes.

› Rendre les objets indépendants les uns des autres.

3. Notion de classe

› Une classe regroupe un ensemble d'objets semblables ayant:

- Les mêmes propriétés structurelles
- Le même comportement

- Les mêmes liens avec les autres objets
- Un intérêt pour l'application

› Une classe ***encapsule*** les données et les traitements.

› Exemple: La classe **Facture**

3.1. Attributs et méthodes

› Ils décrivent une classe.

› Attribut: Caractéristiques de la classe.

- Exemples:

  La classe: **Facture**

  Attributs: numFacture, dateFacture, …

  La classe: **Chat**

  Attributs: Race, couleurPoil, couleurYeux, …

› Méthodes: Ce que la classe peut faire (fonctions, opérations). Doit coller avec la réalité de la classe.

- Exemples:

  La classe: **Facture**

  Méthodes: Payer(), Chercher(), Modifier(), …

  La classe: **Chat**

  Méthodes: Miauler(), FaireGriffe(), Dormir(), Manger(), Achaler(), …

4. Notion d'objet

› Chaque instance créée de la classe est appelée un objet.

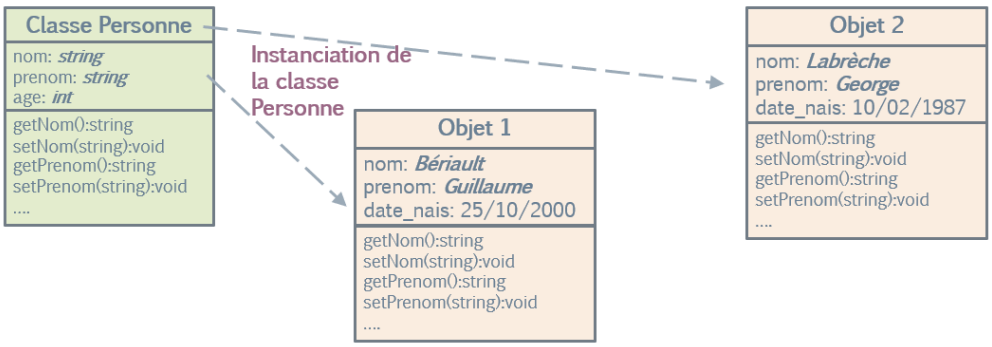
› Un objet est un élément du domaine à modéliser qui satisfait aux principes de:

- **Distinction**: Il a une ***identité***, on peut lui attribuer un nom lui permettant d'être repéré et distingué d'autres objets
- **Permanence**: Il a une certaine durée de vie. À un moment donné, il a un seul état. Au cours de son cycle de vie, certaines de ses caractéristiques peuvent changer.
- **Activité**: Il peut accomplir certaines actions ou subir des traitements. Il agit et réagit aux divers messages qui lui parviennent de son environnement.

› Exemple: La facture **100567** : {100567, 10-01-2022, …} est une instance de la classe Facture qui est:

- Identifiable avec son numéro (100567)
- Caractérisée par sa date (10-01-2022)
- Les traitements qu'elle peut subir: Payer(), Chercher(), Modifier(), …

Classe Vs Objet



5. Les modificateurs d'accès dans C#

› ***public*** : l'accès n'est pas limité

› ***protected*** : l'accès est limité à la classe conteneur et aux types dérivés de la classe conteneur

› ***internal*** : l'accès est limité à l'assembly actuel

- Modificateur d'accès par défaut d'une classe ou d'une interface.
- Ses attributs et méthodes sont invisibles de l'extérieur de l'assembly où elle se trouve.

› ***protected internal*** : l'accès est limité à l'assembly actuel ou aux types dérivés de la classe conteneur.

› ***private*** : l'accès est limité au type conteneur.

- Modificateur d'accès par défaut des attributs et méthodes d'une classe.

› ***private protected*** : l'accès est limité à la classe conteneur et aux types dérivés de la classe conteneur dans l'assembly actuel.

6. Concept d'héritage

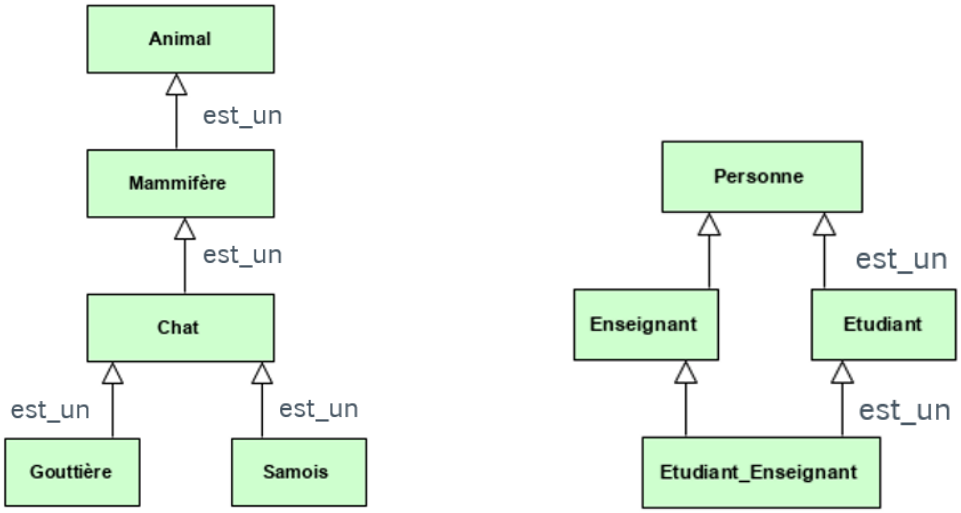
› L'héritage est un mécanisme permettant le partage des caractéristiques ***d'une classe*** (généralisation) par une ou plusieurs sous-classes (spécialisation).

› Exemples:

- Un Chat est un Mammifère
- Un Mammifère est un Animal
- Un ***Gouttière*** est un ***Cha***t, un ***Samois*** est un ***Chat***

› Règles d'héritage:

- Pas de sidecasting « Chat » et « Chien » sont des « Mammifères », mais on ne peut pas transformer « Chat » en « Chien »



6.1. Héritage simple

C'est lorsqu'une classe spécialisée hérite d'une seule classe de base.

Exemple : La classe « Etudiant » hérite seulement de la classe « Personne ».

6.2. Héritage multiple

› C'est lorsqu'une classe spécialisée hérite de deux ou plusieurs classes.

› Certains langages de programmation le permettent comme C++.

› D'autres langages ne le supportent pas comme C# et Java. Dans ce cas, pour simuler l'héritage multiples, on hérite au maximum d'une classe et d'une ou plusieurs interfaces.

6.3. Avantages de l'héritage

› Améliorer la lisibilité des modèles et du code.

› Éviter la redondance des données en les factorisant dans des classes génériques.

› Définir des abstractions de haut niveau

› Permettre la réutilisation de classes existantes simplifiant ainsi la définition de nouvelles classes.

7. Concept du polymorphisme

› Vient du grec et signifie « qui peut prendre plusieurs formes ».

› Le polymorphisme concerne les méthodes des objets.

› L'utilisation du polymorphisme vise à:

- Assurer une programmation plus générique ➔ extensibilité du code
- Renforcer la simplicité
- Maintenir facilement les applications

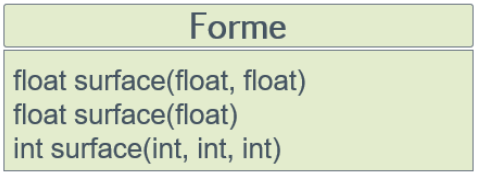
› Trois types de polymorphismes:

- Polymorphisme paramétrique
- Polymorphisme d'héritage (overriding)
- Polymorphisme ad-hoc (overloading)

7.1. Polymorphisme paramétrique

› Représente la possibilité de définir plusieurs fonctions de même nom mais possédant des paramètres différents (en nombre et/ou en type).

- Exemple:

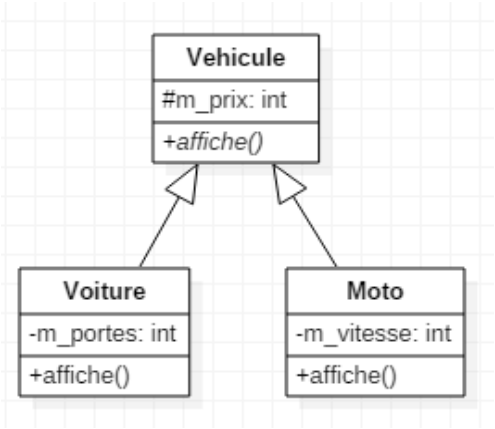


7.2. Polymorphisme d'héritage (overriding)

› La ***redéfinition d'une méthode*** dans des sous-classes héritant d'une classe de base s'appelle la ***spécialisation***.

› Le polymorphisme d'héritage permet de faire abstraction des détails des classes spécialisées d'une famille d'objets, en les masquant par la classe de base.

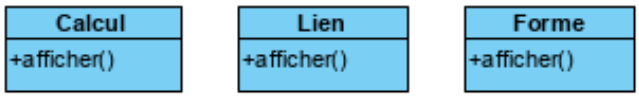
- Exemple:



7.3. *Polymorphisme ad-hoc (overloading)*

› Surcharge de méthodes: définir ***des méthodes de même nom*** et avec des fonctionnalités similaires dans des classes différentes (et sans aucun rapport entre elles).

- Exemple:



› Surcharge d'opérateurs: leur utilisation sera différente selon le type des paramètres qui leur sont passés.

- Exemples: surcharges des opérateurs +, -, ++, ==

Exemple (surcharge d'opérateurs) :

```
class Point
{
    private int x=0;
    private int y=0;

    public Point()
    {
    }

    public Point(int i, int j)
    {
        x = i;
        y = j;
    }

    public void Afficher()
    {
        Console.WriteLine(x + " " + y);
    }
```

```
    public static Point operator +(Point p1, Point p2)
    {
        Point point = new Point();
        point.x = p1.x + p2.x;
        point.y = p1.y + p2.y;
        return point;
    }
}
```

```
public class Program
{
    static void Main()
    {
        Point p1 = new Point(2, 4);
        p1.Afficher();
        Point p2 = new Point(1, 2);
        p2.Afficher();
        Point p3 = new Point();
        p3.Afficher();

        p3 = p1 + p2;
        Console.WriteLine("\nAprès l'addition des deux points:");
        p3.Afficher();

        Console.ReadKey();
    }
}
```

8. Constructeurs (notions avancées)

› Dans un nouveau projet, créez la classe « Employe » :

```
public class Employe
{
    public int SemainesVacances { get; set; }

    public Employe(int nbSemaines)
    {
        SemainesVacances = nbSemaines;
    }
}
```

› Ajoutez la classe « Secretaire » qui hérite de la classe « Employe » :

```
public class Secretaire : Employe
{
    public bool FormationStenographie { get; set; }

    public Secretaire(bool formSteno)
    {
```

```
            FormationStenographie = formSteno;
        }
    }
```

Ne compile pas :

➔ C'est parce que la super classe possède un constructeur à 1 paramètre (nbSemaines) et aucun constructeur à 0 paramètres.

Solutions :

1- Créer un constructeur vide avec 0 paramètre dans la super classe « Employe ».

```
public Employe()
{
}
```

2- Étant donné que le nombre de semaine de vacances d'une secrétaire est 3, ajoutez dans le constructeur de la sous-classe (Secretaire) un appel avec une valeur par défaut « 3 » en utilisant le mot clé « base ».

```
public Secretaire(bool formSteno, int nbSemaines): base(3)
{
    FormationStenographie = formSteno;
}
```

› Constructeurs multiples :

- Pour avoir l'option de spécifier le nombre de semaines, on peut remplacer la valeur par défaut par le paramètre « nbSemaines » :

```
public Secretaire(bool formSteno, int nbSemaines): base(nbSemaines)
{
    FormationStenographie = formSteno;
}
```

- Créez un objet de type « Secretaire » en utilisant le constructeur ci-dessus :

```
Secretaire Secretaire1 = new Secretaire(true, 4);

nbSemVac = Secretaire1.SemainesVacances;

Console.WriteLine("Sécrétaire1:\nNombre de semaines de vacances: " + nbSemVac);
```

- Dans ce cas, on peut créer un autre constructeur dans lequel on conserve notre valeur par défaut sans dupliquer le code.

```
public Secretaire(bool formSteno) : this(formSteno, 3)
{
}
```

- Créez un objet de type « Secretaire » en utilisant le constructeur ci-dessus :

```
Secretaire secretaire2 = new Secretaire(true);

nbSemVac = secretaire2.SemainesVacances;
```

```
Console.WriteLine("Sécrétaire2:\nNombre de semaines de vacances: " + nbSemVac);
```

- Ajoutez un constructeur non paramétré :

```
public Secretaire() : base(6)
{
}
```

- Ajoutez dans la méthode « Main » les instructions suivantes pour instancier un objet « Secretaire » avec le constructeur non paramétré, puis afficher le nombre de semaines de vacances :

```
Secretaire Secretaire3 = new Secretaire();
int nbSemVac = Secretaire3.SemainesVacances;
Console.WriteLine("Nombre de semaines de vacances: " + nbSemVac);
```

9. Abstract, virtual et override

- **Classe abstraite (abstract) :**
  - Une classe qui ne peut pas être implémentée (interdit d'utiliser new).
  - Il est possible d'implémenter ces enfants (par héritage).
- **Méthode abstraite (abstract) :**
  - L'enfant (par héritage) de la classe qui contient cette méthode doit l'implémenter (obligatoire).
- **Virtual :**
  - Jamais sur une classe
  - Plutôt sur un élément de la classe
  - Il est possible aux enfants de surcharger la méthode (n'est pas obligatoire)
  - Sans « virtual », la déclaration de la même méthode par l'enfant fera 2 méthodes distinctes, pas une surcharge.
- **Override :**
  - Mot clé utilisé pour signifier qu'on fait une surcharge.

Exemple: Jeux Warcraft

› Créez un nouveau projet, puis en créez la classe « Unite ».

```
public abstract class Unite
{
    private int pointsVie;

    public Unite()
    {
    }
```

```
        /// <summary>
        /// Quand on clique sur une unité déjà sélectionnée, on la fait parler.
        /// </summary>
        public virtual void Cliquer()
        {
            Parler();
        }

        public abstract void Parler();
}
```

Remarquez qu'on ne peut pas ajouter de code pour la méthode « Parler » (méthode abstraite), mais on peut le faire pour la méthode cliquer (méthode virtuelle).

› Ajoutez la classe abstraite « Horde » qui hérite de la classe abstraite « Unite » et qui permettra plus tard de distinguer les types d'unités.

```
public abstract class Horde: Unite
{
    public Horde()
    {
    }
}
```

› Ajoutez la classe « Peon » qui hérite de la classe abstraite « Horde ».

```
public class Peon: Horde
{
    public Peon()
    {
    }
}
```

› La classe « Peon » doit implémenter la méthode abstraite « Parler » (déclarée dans la classe abstraite « Unite ».

```
public override void Parler()
{
    Console.WriteLine("Le Péon parle!");
}
```

Notez que nous avons utilisé le mot clé « override » dans l'implémentation de cette méthode abstraite.

› Pour que ça soit possible d'assigner des points de vies dans le constructeur de « Peon », il faut que l'attribut « pointsVie » (se trouve dans la classe générique « Unite ») soit protected.

Faire le changement dans la classe « Unite ».

```
protected int pointsVie;
```

Dans le constructeur de « Peon », ajoutez :

```
pointsVie = 250;
```

› Créez la classe « Paysan » dans l'alliance :

```
public class Paysan: Horde
{
    public Paysan()
    {
        pointsVie = 220;
    }

    public override void Parler()
    {
        Console.WriteLine("Paysan parle!");
    }
}
```

› Créez la classe « Mouton » qui hérite de la classe « Unite » :

```
public class Mouton:Unite
{
    public Mouton()
    {

    }

    public override void Parler()
    {
        Console.WriteLine("Le mouton bêle!");
    }

}
```

› Pour le « Mouton », on va changer le fonctionnement du clic :

```
public override void Cliquer()
{
    base.Cliquer();
    Exploser();
}
```

Ici, « base » sert à utiliser la méthode de la classe parent, en plus de nos ajouts dans cette sous-classe.

› Ajout de la nouvelle méthode « Explose » :

```
public void Exploser()
{
    pointsVie = 0;
    Console.WriteLine("***Boum***");
}
```

› Utilisation de ces classes dans la méthode « Main » :

```
Peon Thrall = new Peon();
Paysan Wrynn = new Paysan();
Mouton Pouf = new Mouton();
```

```
// On ajoute nos trois unités dans une seule liste.
List<Unite> listeUnites = new List<Unite>();

listeUnites.Add(Thrall);
listeUnites.Add(Wrynn);
listeUnites.Add(Pouf);

Console.WriteLine("On clique sur nos trois unités ...");
Console.WriteLine();

foreach (Unite u in listeUnites)
{
    u.Cliquer();
}

Console.ReadKey();
```

10. Les interfaces

La notion d'interface est une notion importante de la POO. Une interface fournit une sorte de contrat et ne fournit pas de code C# associé. C'est aux objets qui implémentent cette interface de coder la fonctionnalité associée au contrat.

Contrairement à l'héritage, un objet est capable d'implémenter plusieurs interfaces.

10.1. Création des interfaces

Comme c'est le cas pour une classe, il est recommandé de créer une interface dans un fichier à part. La convention dit aussi qu'il faut que le nom d'une interface commence avec la lettre « i » en majuscule.

Créez une nouvelle application console « ExempleInterface » puis ajoutez dans un nouveau fichier une interface « IRoulant » possédant une propriété de type « int » et la signature d'une méthode « Rouler ».

```
public interface IRoulant
{
    int NombreRoues { get; set; }
    void Rouler();
}
```

Dans un nouveau fichier, créez l'interface « IVolant ».

```
interface IVolant
{
    int AltitudeMaximale { get; set; }

    void Voler();
}
```

NB : Remarquez que nous n'avons pas définit une visibilité sur les membres des interfaces, que ce soient les propriétés ou les méthodes.

10.2. Implémentation des interfaces

Les objets qui implémenteront une interface doivent avoir ses propriétés et définir ses méthodes. Ces objets doivent aussi définir les visibilités en public sur ces éléments.

Créez la classe « Camion » qui implémente l'interface « IRoulant » :

```
class Camion : IRoulant
{
    public int NombreRoues { get; set; }

    public Camion(int NombreRoues)
    {
        this.NombreRoues = NombreRoues;
    }

    public void Rouler()
    {
        Console.WriteLine("Le camion roule grâce à ses " + NombreRoues + " roues");
    }
}
```

Définissez la classe Oiseau implémentant l'interface IVolant :

```
public class Oiseau : IVolant
{
    public int AltitudeMaximale { get; set; }

    public Oiseau(int AltitudeMaximale)
    {
        this.AltitudeMaximale = AltitudeMaximale;
    }

    public void Voler()
    {
       Console.WriteLine("Un oiseau vole à une altitude maximale de " + AltitudeMaximale
+ " mètres");
    }
}
```

Définissez la classe Avion utilisant l'héritage multiple en implémentant les 2 interfaces « IVolant » et « IRoulant ».

```
public class Avion : IVolant, IRoulant
{
    public int AltitudeMaximale { get; set; }
    public int NombreRoues { get; set; }

    public Avion(int AltitudeMaximale, int NombreRoues)
    {
        this.AltitudeMaximale = AltitudeMaximale;
        this.NombreRoues = NombreRoues;
    }
```

```
        public void Voler()
        {
            Console.WriteLine("L'avion vole à une altitude maximale de " + AltitudeMaximale
+ " mètres");
        }

        public void Rouler()
        {
            Console.WriteLine("L'avion roule grâce à ses " + NombreRoues + " roues");
        }
    }
```

Ajoutez les instructions suivantes dans la méthode « Main » :

```
    Oiseau cigogne = new Oiseau(4000);

    Camion mack = new Camion(10);

    Avion boeing = new Avion(12000, 5);

    cigogne.Voler();

    mack.Rouler();

    boeing.Voler();
    boeing.Rouler();
```