Les concepts de la programmation objet : les interfaces

# Définition

Une **interface** est un **modèle de classe** contenant des méthodes et des propriétés **sans fournir l'implémentation des méthodes**.

Chaque interface est vouée à être implémentée par une classe (ou une structure) qui devra implémenter **toutes** les méthodes présentées par l'interface et posséder ses propriétés. Le but d'une interface est de définir les méthodes et propriétés proposées par toute classe (ou structure) qui l'implémente, sans que le développeur ait besoin de savoir comment celles-ci sont codées. En résumé, une interface est un contrat : elle définit des attentes sans prévoir les solutions techniques qui seront utilisées.

Toutes les méthodes d’une interface sont (implicitement) « public » et « abstract ».

Pour déclarer une interface en C#, on utilise la syntaxe :

interface Inom\_interface

**Remarques** :

* Par convention, le nom d'une interface commence toujours par un **'i' majuscule.**
* En termes d'accessibilité, **une interface est toujours publique.**
* Un restricteur d'accès (public, private, protected ou internal) n’est pas utilisé.

**Exemple :**

interface IAnimal

{

// Un objet stocké sous forme de IAnimal aura toujours la propriété Name accessible en lecture et la méthode Move.

string Name { get; }

void Move();

}

# Utiliser une interface

Toute classe implémentant une interface doit implémenter ses méthodes (avec les mêmes signatures) et ses propriétés.

Une classe peut implémenter une ou plusieurs interface(s).

**Reprenons l'exemple précédent.**

class Dog : IAnimal

{

private string m\_name;

// On implémente la propriété Name accessible en lecture.

public string Name

{

get { return m\_name; }

}

public Dog (string name)

{

m\_name = name;

}

// On implémente la méthode Move.

// L'implémentation de Move devra donc aussi se nommer Move, retourner void, et ne // prendre aucun paramètre en entrée.

public void Move()

{

Console.WriteLine("{0} bouge.", m\_name);

}

}

# Utiliser plusieurs interfaces

Il est possible d’utiliser plusieurs interfaces. Pour cela, il faut séparer les interfaces par des virgules, comme dans l'exemple suivant.

class SampleClass : IControl, ISurface

{

    // Code de la classe

}

# Mise en oeuvre d’interfaces dans le cas Commercial

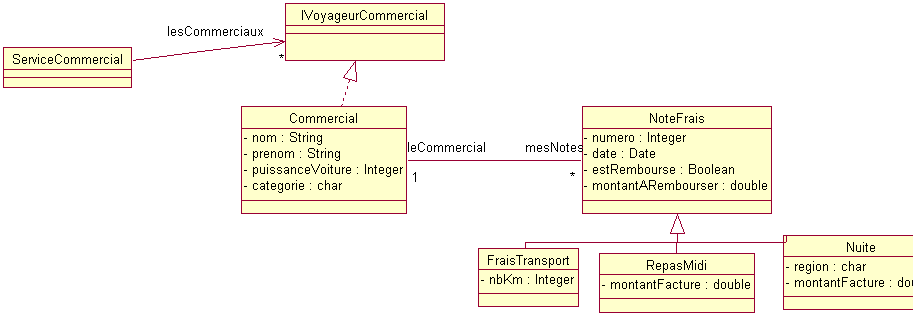
La notion d’interface est très importante en programmation objet. Elle permet de présenter uniquement les  **services utiles à une classe cliente**. On masque ainsi l’implémentation en exposant une interface.

Nous allons mettre en oeuvre cette notion à partir du TP commerciaux.

Actuellement le service commercial utilise les services proposés par la classe Commercial (ajout de notes etc...) ; imaginons que ces mêmes commerciaux participent à un autre processus, par exemple le service comptable responsable des salaires de tous les employés.  De nombreux services disponibles de la classe Commercial ne sont pas utiles pour l’établissement des salaires (ajout de notes de frais par exemple).

La programmation objet permet à une classe de présenter différentes interfaces suivant les *consommateurs* de la classe. Ainsi,  la même classe Commercial proposera une interface au service commercial en fournissant les services dont il a besoin mais proposera une autre interface à une autre classe correspondant à ses seuls besoins.

 La notion d’interface est proche de la notion de spécialisation si ce n’est qu’une classe peut exposer plusieurs interfaces distinctes (ce qui n’est pas le cas de la spécialisation).

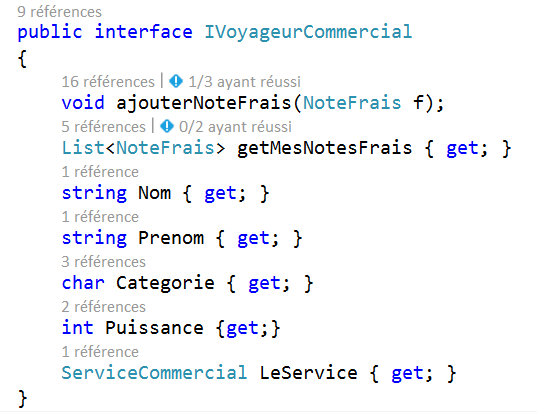


Ainsi, c’est la nouvelle classe (Interface) qui va collaborer avec le service commercial ; cette dernière ne connaîtra pas l’implémentation du commercial mais seulement ses services utiles.

**Remarque** : la relation *implémente* (entre une interface et une classe) se représente avec une fléche pointillée.

**Code** :

 On déclare une classe interface :



**Rappel** : on n’indique pas le niveau de visibilité (toujours public) des méthodes d’une interface.

**Elle ne contient que les signatures des méthodes (sans le code)** utiles au service commercial. On appelle signature d'une méthode (ou prototype) la déclaration d'une méthode de cette forme :    
    typeDeRetour nomMéthode(liste des paramètres)**;**

Il n'y a pas de corps, mais à la place un ";".

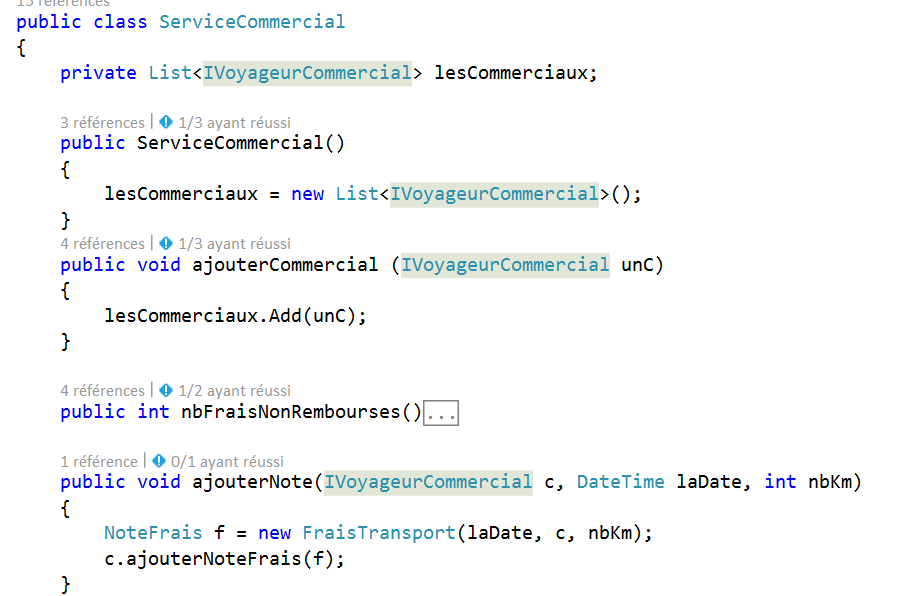
La classe Commercial est inchangée, si ce n’est qu’elle déclare *implémenter*  l’interface IvoyageurCommercial  :

http://www.grand-patrice.fr/drupal-6.19/images/interface/image003.png

**Cette déclaration est un engagement**-qui sera vérifié à la compilation- **d’écrire le code de toutes les méthodes de l’interface**. La classe commercial peut bien sûr (et c’est le cas ici) posséder des méthodes autres que celles de son interface.

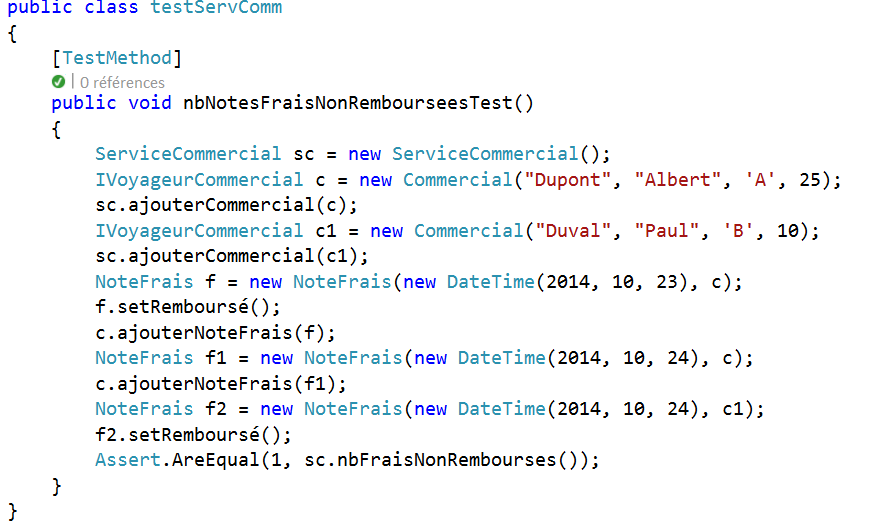
Le code de la classe ServiceCommercial va faire référence, non plus à la classe Commercial mais à son interface.

**Extraits de code** :



La classe Commercial a été remplacée par son interface.

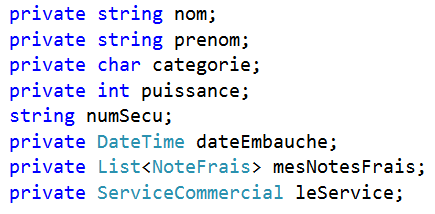
**Utilisation** : dans le test de vérification du nombre de frais non remboursés, on peut écrire :



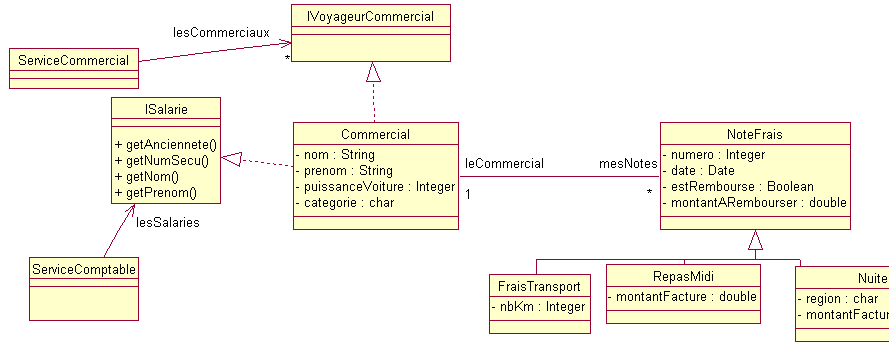
Certes, on crée bien un *Commercial*  (**on ne peut pas instancier une interface**) mais le service commercial  n’a accès qu’aux méthodes de *IVoyageurCommercial*, son interface, ce qui était le but recherché.

Passons maintenant au service comptable, responsable de la paye des commerciaux ; imaginons que pour établir la paye elle ait besoin de son numéro de sécurité sociale et de sa date d’embauche.

Ajoutons dans un premier temps ces champs dans la classe *Commercial*.

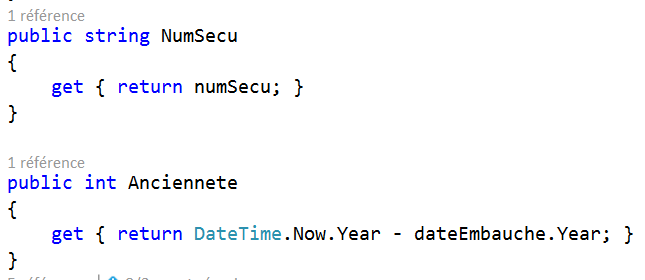


Créons le service comptable et ajoutons une nouvelle interface ISalarie à la classe Commercial.



|  |  |
| --- | --- |
| Nous n’avons (pour simplifier) pas fait figurer les méthodes de l’interface IVoyageurCommercial. La classe ISalarie se présente ainsi : |  |

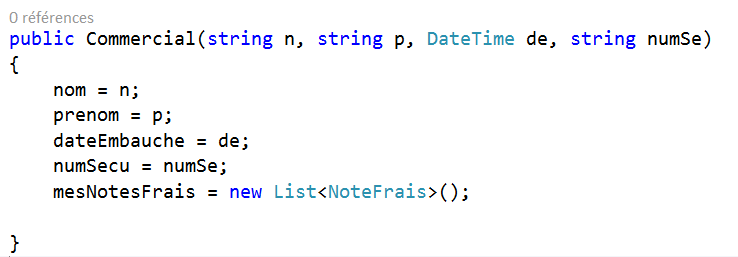
C’est à la classe Commercial d’écrire (implémenter le code) ces méthodes (les deux premières sont déjà écrites).



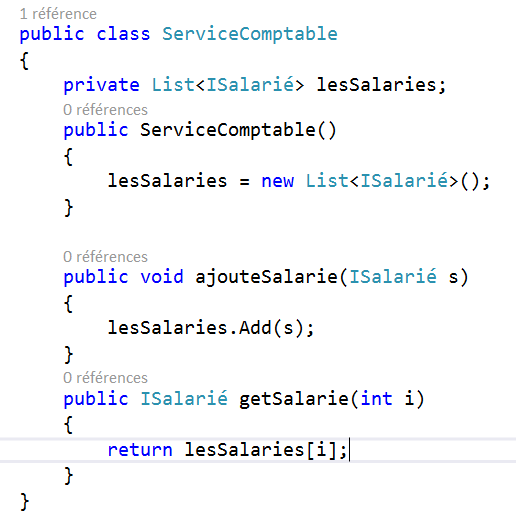
On peut maintenant indiquer que la classe Commercial implémente la nouvelle interface :

http://www.grand-patrice.fr/drupal-6.19/images/interface/image010.png

Proposons un second constructeur correspondant à l’interface ISalarie :



Ajoutons des méthodes minimum dans la classe ServiceComptable :



|  |  |
| --- | --- |
| Utilisation de la nouvelle interface : |  |

# Intérêt des interfaces dans un framework

Un framework propose de nombreuses classes *Interface ;* le grand bénéfice de cette organisation est de faire bénéficier ses propres classes des services prévus pour les interfaces. Il suffit ainsi de déclarer que sa classe implémente une interface pour bénéficier de ces services ; à condition, bien sûr, d’implémenter la ou les méthodes exigées par l’interface.

## L’interface IComparable

Nous allons mettre en oeuvre cette situation dans l’exemple de la méthode Sort de la classe List.

La classe List est capable de trier (Sort) les éléments de sa liste à condition que la classe implémente une interface du framework : IComparable.

|  |  |
| --- | --- |
| Nous allons, par exemple, trier les notes de frais par date croissante. Ceci n’est possible que si la classe NoteFrais implémente IComparable dont la seule méthode est CompareTo : | http://www.grand-patrice.fr/drupal-6.19/images/interface/image014.png |

|  |  |
| --- | --- |
| Dans la classe NoteFrais, on déclare l’interface : | http://www.grand-patrice.fr/drupal-6.19/images/interface/image015.png |

|  |  |
| --- | --- |
| On s’engage ainsi à implémenter CompareTo : |  |

Dans ce cas, le tri se fait sur les dates des notes de frais.

|  |  |
| --- | --- |
| On pourrait envisager un autre tri, plus élaboré, sur les montants à rembourser : | http://www.grand-patrice.fr/drupal-6.19/images/interface/image017.png |

|  |  |
| --- | --- |
| La classe Commercial peut maintenant trier sa List : | public void trierNotes()  {  mesNotesFrais.Sort();  } |
| Dans le test, nous utiliserons un objet IvoyageurCommercial. Ajoutons la méthode trierNotes() à cette interface |  |
| On peut tester ainsi dans le programme principal : |  |
| Ce qui produit : |  |

## L’interface Ienumerable

|  |  |
| --- | --- |
| Ajouter dans la classe ServiceCommercial, les méthodes getCommercial(i) et nbCommerciaux() | public int nbCommerciaux()  {  return lesCommerciaux.Count;  }  public IVoyageurCommercial getCommercial(int i)  {  return lesCommerciaux[i];  } |
| Créer dans une classe Ecran, une méthode *Affiche* permet d’afficher les commerciaux d’un service commercial : | public static void affiche(ServiceCommercial sc)     {        for(int i = 0; i < sc.nbCommerciaux();i++)                  affiche(sc.getCommercial(i));     } |
| Pour tester cette méthode, ajouter également la méthode affiche(IVoyageurCommercial voyageurCommercial)dans la classe écran. | private static void affiche(IVoyageurCommercial voyageurCommercial)  {  Console.WriteLine(voyageurCommercial.ToString());  } |
| Comme vous pouvez le constater, il faut ajouter également la méthode ToString dans l'interface IVoyageurCommercial |  |
|  |  |

Dans la méthode affiche(ServiceCommercial sc,nous parcourons tous les commerciaux avec une boucle for ; il n’est pas possible d’utiliser un *foreach* car la classe *ServiceCommercial* ne peut *itérer* comme le fait – par exemple - la classe ArrayList. Il est néanmoins  possible de fournir à la classe *ServiceCommercial* ce service, en utilisant le mécanisme des classes Interfaces.

Pour pouvoir être *itéré* (c’est à dire pouvoir utiliser *foreach*), la classe doit implémenter *IEnumerable*.

Commençons par ajouter à la classe *ServiceCommercial*, son engagement à implémenter les méthodes de *IEnumerable*.

**public class ServiceCommercial : IEnumerable**

|  |  |
| --- | --- |
| L’environnement propose alors d’ajouter la seule méthode à implémenter : | Public IEnumerator GetEnumerator()          {          } |
| Cette méthode retourne une Interface ; on doit créer une classe implémentant cette interface : | class EnumereCommerciaux : IEnumerator |
| L’environnement  nous précise alors les méthodes à implémenter (clic sur l'interface puis implémenter explicitement l'interface) : | public object Current  public bool MoveNext()  public void Reset() |
| Remarque : Current est une propriété et non une méthode classique |  |

Le rôle de la classe est de parcourir les éléments d’un service commercial ; ces élément sont les commerciaux de la List . Les trois méthodes font référence à ce parcours des commerciaux. C’est pourquoi cette classe *EnumereCommerciaux* doit avoir une référence sur un service commercial.

Les champs peuvent être :

private ServiceCommercial leService;

private int index ;

On ajoute le code des méthodes :

class EnumereCommerciaux : IEnumerator

    {

      public EnumereCommerciaux(ServiceCommercial leService)

        {

            this.leService = leService;

            this.index = -1;

        }

       public object Current

        {

            get { return this.leService.getCommercial(this.index); }

        }

        public bool MoveNext()

        {

            this.index++;

            return this.index < this.leService.nbCommerciaux();

        }

        public void Reset()

        {

            index = -1;

        }

        private ServiceCommercial leService;

        private int index ;

    }

**Remarques** :

* Le constructeur doit récupérer le service commercial
* Comme *Current* est une propriété, on doit écrire le *geter –la méthode get-*.
* La méthode *MoveNext* fait avancer l’index et indique si on est en fin de parcours.

|  |  |
| --- | --- |
| Terminons en écrivant le code de la méthode à implémenter dans la classe ServiceCommercial : | public IEnumerator GetEnumerator()          {              EnumereCommerciaux en = new EnumereCommerciaux(this);              return en;          } |
| On peut maintenant  modifier la méthode Affiche vue plus haut : | public static void affiche(ServiceCommercial sc)    {       //for(int i = 0; i <sc.nbCommerciaux();i++)      //    affiche(sc.getCommercial(i));       foreach (Commercial c in sc)              affiche (c);    } |
| Dans le Main on peut faire le test suivant : | static void Main(string[] args)    {        Commercial c1, c2, c3;        ServiceCommercial sc;        sc = new ServiceCommercial();        c1 = new Commercial("Dupond", "Jean", 7, 'B');        c2 = new Commercial("Durand", "Dominique", 11, 'C');        c3 = new Commercial("Chamir", "Jéremy", 15, 'A');        sc.ajouterCommercial(c1);        sc.ajouterCommercial(c2);        sc.ajouterCommercial(c3);        Ecran.affiche(sc); |

## Une variation inattendue du foreach

Pour terminer, on peut s’autoriser à mettre en oeuvre une utilisation un peu exotique du *foreach*. Utilisons-le pour parcourir les **champs** de la classe Commercial !! Ils sont de types différents et non contenus dans une structure de donnée linéaire (tableau ou ArrayList). Cependant rien n’empêche d’itérer sur ces champs si nous construisons l’itérateur correctement.

 Code de la classe implémentant IEnumerator :

 class EnumereChampsCommercial : IEnumerator

    {

        public EnumereChampsCommercial(Commercial leCommercial)

        {

            this.leCommercial = leCommercial;

            this.index = -1;

        }

        public object Current

        {

            get {

                object o=null ;

                switch (this.index)

                {

                    case 0:

                        {

                            o = this.leCommercial.getNom(); break;

                        }

                    case 1:

                        {

                            o = this.leCommercial.getPrenom(); break;

                        }

                    case 2 :

                        {

                            o = this.leCommercial.getCategorie(); break;

                        }

                    case 3:

                        {

                            o = this.leCommercial.getPuissance(); break;

                        }

                    case 4:

                        {

                            o = this.leCommercial.getNumSecu(); break;

                        }

                    case 5:

                        {

                            o = this.leCommercial.getAnciennete(); break;

                        }

                }

                return o;

              }

        }

        public bool MoveNext()

        {

            this.index++;

            return this.index<6;

        }

        public void Reset()

        {

            this.index = -1;

        }

        private Commercial leCommercial;

        private int index;

    }

Code de la méthode retournant un objet *EnumereChampsCommercial* :

public   class Commercial : IVoyageurCommercial, ISalarie, IEnumerable

...

Public IEnumerator GetEnumerator()

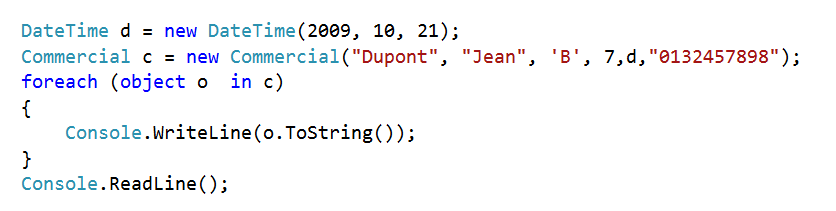
        {

            EnumereChampsCommercial en = new EnumereChampsCommercial(this);

            return en;

        }

Test dans le Main :



|  |  |
| --- | --- |
| Ce qui produit bien : |  |

**Remarque** : la valeur 5 correspond à l’ancienneté calculée par la méthode *getAnciennete()*  et non au champ *dateEmbauche*.