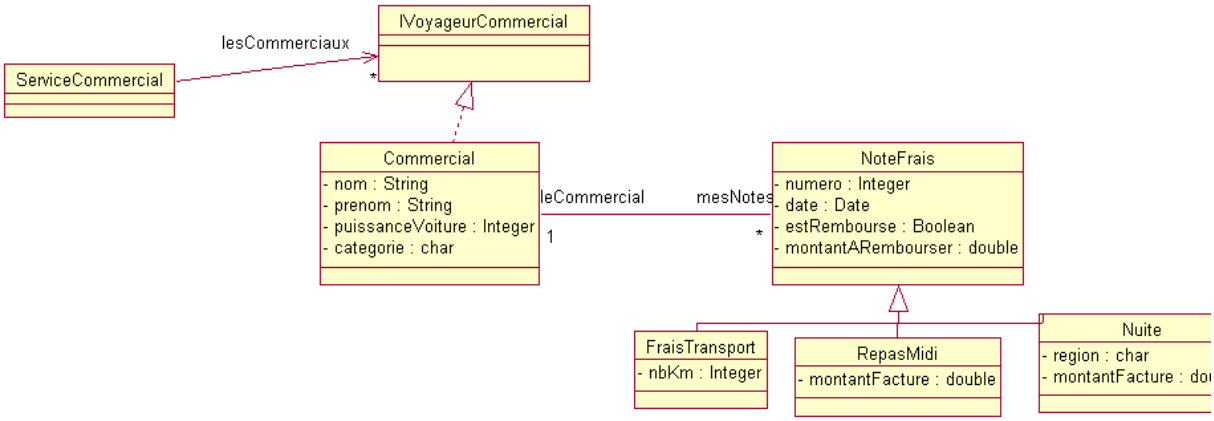
Les classes interfaces

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 dans un premier temps mettre en oeuvre cette notion à partir du cas *commerciaux*.

1- Mise en oeuvre d'interfaces dans le cas Commercial

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 :

```
public interface IVoyageurCommercial
{
    void ajouterNote(DateTime date, int nbKm);
    void ajouterNote(DateTime date, double montantFacture);
    void ajouterNote(DateTime date, double montantFacture, char region);
    NoteFrais getNoteFrais(int i);
    string getNom();
    string getPrenom();
    int nbNotes();
}
```

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

```
public   class Commercial : IVoyageurCommercial
```

Cette déclaration est un engagement -qui sera vérifié à la compilation- **d'écrire le code des 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 :

```
public class ServiceCommercial
{
    public ServiceCommercial()
    {
        this.lesCommerciaux = new ArrayList();
    }
    public void ajouterCommercial(IVoyageurCommercial unCommercial)
    {

        this.lesCommerciaux.Add(unCommercial);
    }
    public IVoyageurCommercial getCommercial(string nom, string prenom)
    {
        bool trouve = false;
        IVoyageurCommercial commercialCourant, leCommercial = null;
        int i = 0;
        while (!trouve && i < this.lesCommerciaux.Count)
        {
            commercialCourant = (IVoyageurCommercial)this.lesCommerciaux[i];
            if (commercialCourant.getNom() == nom && commercialCourant.getPrenom() == prenom )
            {
                leCommercial = commercialCourant;
                trouve = true;
            }
            else
                i++;
        }
        return leCommercial;
    }
```

La classe Commercial a été remplacée par son interface.

Utilisation : dans le Main, on peut écrire :

```
IVoyageurCommercial c1;
ServiceCommercial sc;
sc = new ServiceCommercial();
c1 = new Commercial("Dupond", "Jean", 7, 'B');
sc.ajouterCommercial(c1);
```

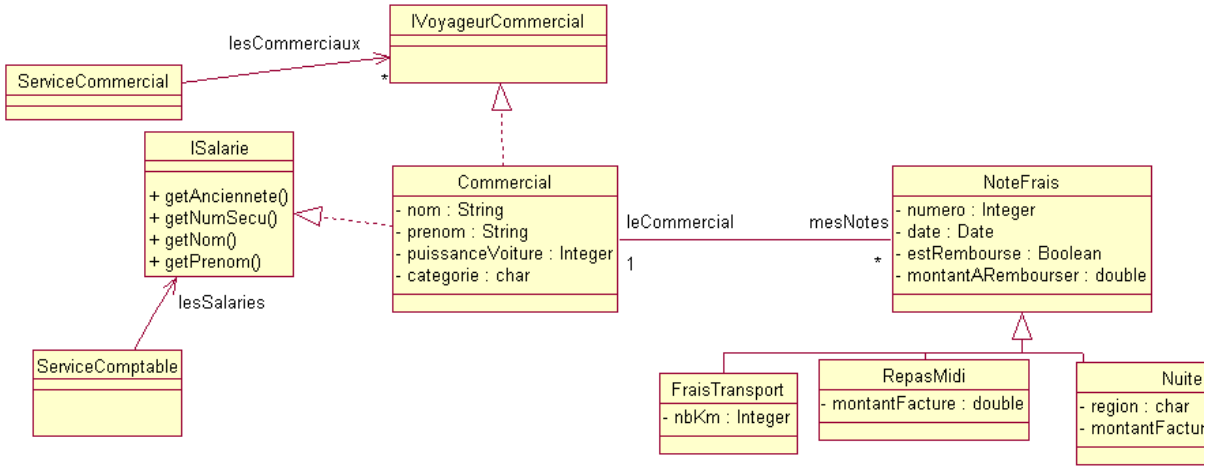
Certes, on crée bien un *Commercial* au service (on ne peut pas instancier une interface) mais le service commercial n'a accès qu'aux méthodes de *IVoyageurCommercial*, son interface, ce qui était le but recherché.

Passons maintenant au service comptable, responsable de la paye des commerciaux ; imaginons que pour établir la paye elle ait besoin de son numéro de sécu et de sa date d'embauche.
Ajoutons dans un premier temps ces champs dans la classe commercial.

```
        private string nom;
        private string prenom;
        private int puissanceVoiture;
        private char categorie;
        private string numSecu;
        private DateTime dateEmbauche;
        private ArrayList mesNotes;

    }
```

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 :

```
interface ISalarie
{
    string getNom();
    string getPrenom();
    string getNumSecu();
    int getAnciennete();
}
```

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

```
        public string getNumSecu()
        {
            return this.numSecu;
        }
        public int getAnciennete()
        {
            return DateTime.Now.Year - this.dateEmbauche.Year;
        }
```

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

```
public   class Commercial : IVoyageurCommercial, ISalarie
    ,
```

Proposons un second constructeur correspondant à l'interface ISalarie :

```
public Commercial(string nom, string prenom, DateTime embauche, string numSecu)
{
    this.nom = nom;
    this.prenom = prenom;
    this.dateEmbauche = embauche;
    this.numSecu = numSecu;
    mesNotes = new ArrayList();
}
```

Ajoutons des méthodes minimum dans la classe ServiceComptable :

```
    class ServiceComptable
    {
        public ServiceComptable()
        {
            lesSalaries = new ArrayList();
        }
        public void ajouteSalarie(ISalarie s)
        {
            lesSalaries.Add(s);
        }
        public ISalarie getSalarie(int i)
        {
            return (ISalarie)lesSalaries[i];
        }

        private ArrayList lesSalaries;
    }
```

Utilisation de la nouvelle interface :

```
IVoyageurCommercial c1;
ISalarie s1;
ServiceCommercial sc;
ServiceComptable scomp;
sc = new ServiceCommercial();
scomp = new ServiceComptable();
DateTime d = new DateTime(2004, 10, 21);
c1 = new Commercial("Dupond", "Jean", 7, 'B');
sc.ajouterCommercial(c1);
s1 = new Commercial("Dupond", "Jean", d, "1740275111068");
scomp.ajouteSalarie(s1);
```

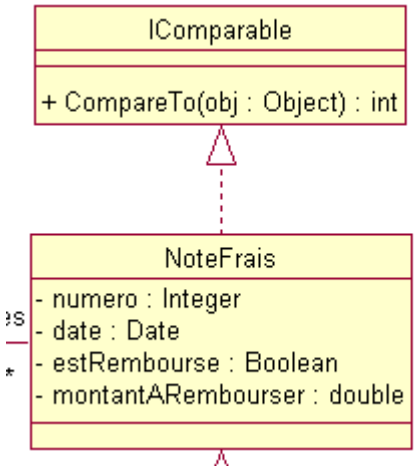
2- Intérêt des interfaces dans un framework

Le framework propose de nombreuses classe *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.

2.1 L'interface IComparable

Nous allons mettre en oeuvre cette situation dans l'exemple de la méthode **Sort** de la classe ArrayList.
La classe ArrayList 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 :



Dans la classe NoteFrais
On déclare l'interface :

```
public   class NoteFrais : IComparable
```

On s'engage ainsi à implémenter **CompareTo** :

```
public int CompareTo(object obj)
{
    return this.date.CompareTo(((NoteFrais)obj).date);

}
```

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

On peut envisager un autre tri, plus élaboré, sur les montants à rembourser :

```
public int CompareTo(object obj)
{
    if (this.montantARembourser > ((NoteFrais)obj).montantARembourser)
        return 1;
    else return - 1;
}
```

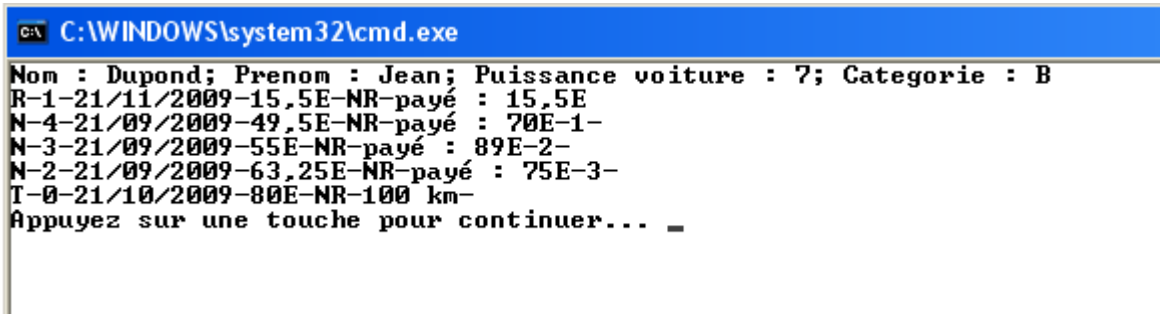
La classe Commercial peut maintenant trier son ArrayList :

```
public void trierNotes()
    {
        this.mesNotes.Sort();
    }
```

On peut tester ainsi :

```
DateTime dEmbauche = new DateTime(1998, 10, 21);
Commercial c = new Commercial("Dupond", "Jean", 7, 'B',dEmbauche,"014532654578");
DateTime d = new DateTime(2009, 10, 21);
DateTime d1 = new DateTime(2009, 11, 21);
DateTime d2 = new DateTime(2009, 9, 21);
c.ajouterNote(d, 100);
c.ajouterNote(d1, 15.5);
c.ajouterNote(d2, 75, '3');
c.ajouterNote(d2, 89, '2');
c.ajouterNote(d2, 70, '1');
c.trierNotes();
Ecran.affiche(c);
```

Ce qui produit :



2.2 L'interface IEnumerable

Dans la 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));
        }
```

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 :

- `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 l'ArrayList . Les trois méthodes font références 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 le 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);
```

2.3 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 type 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 :

```
DateTime d = new DateTime(2004, 10, 21);
            Commercial c = new Commercial("Durand", "Dominique", 11, 'C',
d, "0132457898");
            foreach (object o in c)
                Console.WriteLine(o.ToString());
```

Ce qui produit bien :

```
C:\WINDOWS\system32\cmd.exe
Durand
Dominique
C
11
0132457898
5
Appuyez sur une touche pour continuer... _
```

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