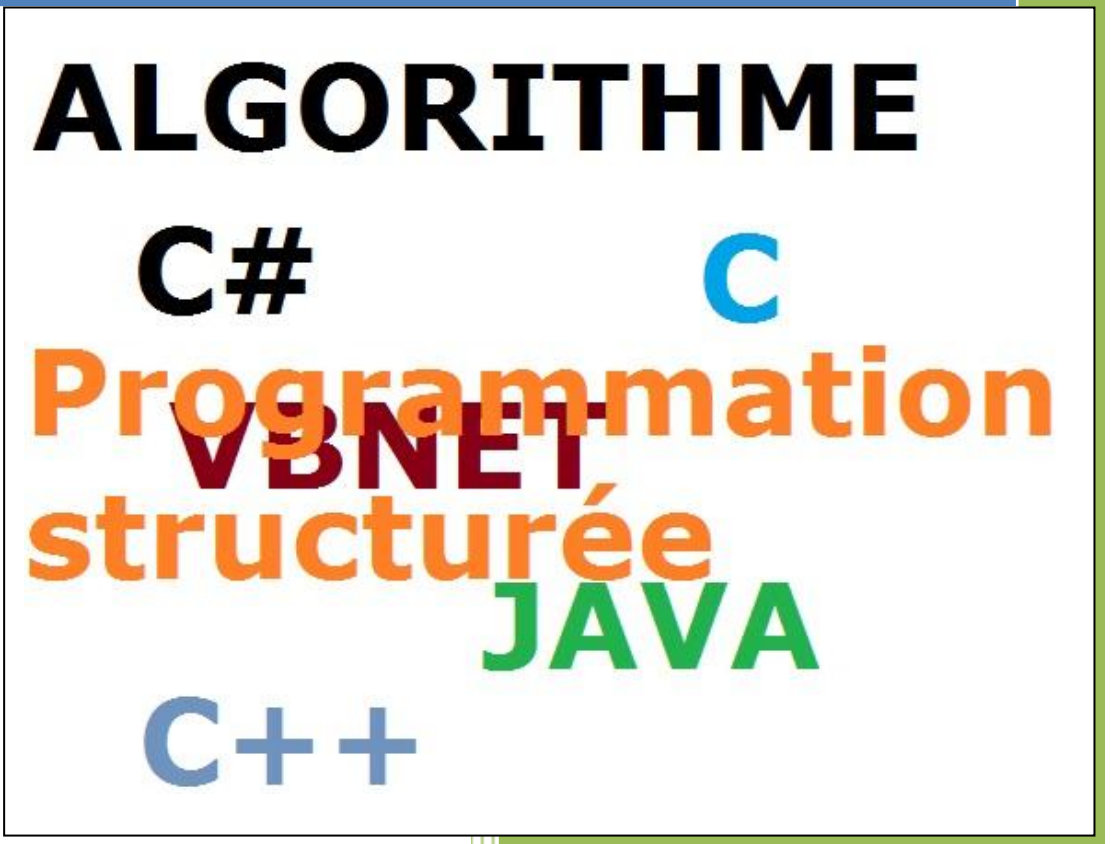
TDI 1ère Année

Apprentissage d'un langage de Programmation Orientée Objet : C# (objet)



HARCHI Abdellah

Formateur NTIC ISTA

MAAMORA KENITRA

2011 - 2012

Sommaire

INTRODUCTION **3**

T.P. N°1 - CLASSE D'OBJET - ENCAPSULATION **4**

1.1 OBJECTIFS 4
1.2 CE QU'IL FAUT SAVOIR 4
1.2.1 Notion de classe 4
1.2.2 Encapsulation 4
1.2.3 Déclaration des propriétés d'un objet 4
1.2.4 Implantation des méthodes 5
1.2.5 Instanciation 5
1.2.6 Accès aux propriétés et aux méthodes 6
1.2.7 Conventions d'écriture 6
1.3 TRAVAIL A REALISER 6

T.P. N° 2 - ENCAPSULATION PROTECTION ET ACCES AUX DONNEES MEMBRES **7**

2.1 OBJECTIFS 7
2.2 CE QU'IL FAUT SAVOIR 7
2.2.1 Protection des propriétés 7
2.2.2 Fonctions d'accès aux propriétés 7
2.3 TRAVAIL A REALISER 8

T.P. N° 3 - CONSTRUCTION ET DESTRUCTION **9**

3.1 OBJECTIFS 9
3.2 CE QU'IL FAUT SAVOIR 9
3.2.1 Constructeurs 9
3.2.2 Surcharge des constructeurs 9
3.2.3 Propriétés de classe 10
3.2.4 Méthodes de classe 10
3.2.5 Destructeur 11
3.3 TRAVAIL A REALISER 11

T.P. N°4 - L'HERITAGE **12**

4.1 OBJECTIFS 12
4.2 CE QU'IL FAUT SAVOIR 12
4.2.1 L'héritage 12
4.2.2 Protection des propriétés et des méthodes 12
4.2.3 Compatibilité et affectation 13
4.2.4 Mode de représentation 14
4.2.5 Insertion d'une classe dans une hiérarchie 14
4.2.6 Insertion d'une nouvelle classe à partir de Object 14

4.2.7 Constructeur et héritage 15
4.2.8 Appel aux méthodes de la classe de base 16
4.3 TRAVAIL A REALISER 16

T.P. N°5 – LES COLLECTIONS 17

5.1 OBJECTIFS 17
5.2 CE QU'IL FAUT SAVOIR 17
5.2.1 Les tableaux statiques 17
5.2.2 Classe ArraysList 17
5.2.3 La classe Hashtable 18
5.2.4 SortedList 18
SortedList est un dictionnaire qui garantit que les clés sont rangées de façon ascendante. 18
5.3 TRAVAIL A REALISER 19

T.P. N° 6 - POLYMORPHISME 20

6.1 OBJECTIFS 20
6.2 CE QU'IL FAUT SAVOIR 20
6.2.1 Polymorphisme 20
6.2.2 Méthodes virtuelles 20
6.2.3 Classes génériques 20
6.3 TRAVAIL A REALISER 20

INTRODUCTION

Ce support traite des concepts de Programmation Orientée Objet en langage C#. Il est constitué d'une liste d'exercices permettant de construire pas à pas une classe d'objet. Les corrigés types de chaque étape sont présentés dans des répertoires séparés, il s'agit en fait du même cas qui va évoluer, étape par étape, pour aboutir à un fonctionnement correct de la classe dans l'univers C#.

Chaque exercice est structuré de la façon suivante :

- Description des objectifs visés.
- Explications des techniques à utiliser (Ce qu'il faut savoir).
- Enoncé du problème à résoudre (Travail à réaliser).
- Renvois bibliographiques éventuels dans les ouvrages traitant de ces techniques (Lectures).

T.P. N°1 - CLASSE D'OBJET - ENCAPSULATION

1.1 OBJECTIFS

- Notion de classe d'objet
- Définition d'une nouvelle classe d'objet en C#.
- Encapsulation des propriétés et des méthodes de cet objet.
- Instanciation de l'objet.

1.2 CE QU'IL FAUT SAVOIR

1.2.1 Notion de classe

Créer un nouveau type de données, c'est **modéliser** de la manière la plus juste un objet, à partir des possibilités offertes par un langage de programmation.

Il faudra 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 façon suivante :

- **Les fonctions d'entrée/sortie**. Comme les données de base du langage C#, ces fonctions devront permettre de lire et d'écrire les nouvelles données sur les périphériques (clavier, écran, fichier, etc.).
- **Les opérateurs de calcul**. S'il est possible de calculer sur ces données, il faudra créer les fonctions de calcul.
- **Les opérateurs relationnels**. Il faut au moins pouvoir tester si deux données sont égales. S'il existe une relation d'ordre pour le nouveau type de donnée, il faut pouvoir aussi tester si une donnée est *inférieure* à une autre.
- **Les fonctions de conversion**. Si des conversions vers les autres types de données sont nécessaires, il faudra implémenter également les fonctions correspondantes.
- **Intégrité de l'objet**. La manière dont est modélisé le nouveau type de données n'est probablement pas suffisant pour représenter l'objet de façon exacte. Par exemple, si l'on représente une fraction par un couple de deux entiers, il faut également vérifier que le dénominateur d'une fraction n'est pas nul et que la représentation d'une fraction est unique (simplification).

La déclaration d'un nouveau type de données et les fonctions qui permettent de gérer les objets associés constituent une ***classe*** de l'objet.

Les propriétés de l'objet seront implantées sous la forme de ***données membres*** de la classe.

Les différentes fonctions ou méthodes seront implémentées sous la forme de ***fonctions membres*** de la classe.

De façon courante, dans le *patois* de la programmation orientée objet, ***données membres*** et ***fonctions membres*** de la classe sont considérées respectivement comme synonymes de ***propriétés*** et ***méthodes*** de l'objet.

1.2.2 Encapsulation

L'encapsulation est un concept important de la Programmation Orientée Objet.

L'encapsulation permet de rassembler les ***propriétés*** composant un objet et les ***méthodes*** pour les manipuler dans une seule entité appelée ***classe*** de l'objet.

Une classe, en C# se déclare par le mot clé **class** suivi d'un identificateur de classe choisi par le programmeur de la façon suivante :

```
public class NomDeLaClasse
{
    // Déclaration des propriétés et
    // des méthodes de l'objet
}
```

Le fichier contenant le code source portera le nom **NomdeLaClasse.cs.**

Les identifiants de classe obéissent aux mêmes règles que les identifiants de variables et de fonctions. Par convention, ils commencent par une Majuscule (C# considère les majuscules comme des caractères différents des minuscules).

Les propriétés et les méthodes de l'objet seront déclarées et implémentées dans le bloc **{ }** contrôlé par le mot clé **class**. Cela constitue l'implémentation du concept d'encapsulation en C#.

1.2.3 Déclaration des propriétés d'un objet

Les propriétés d'un objet sont déclarées, comme des variables, à l'intérieur du bloc **{ }** contrôlé par le mot clé **class**.

```
public class NomDeLaClasse
```

```
{
    TypeDeLaProprieté      NomDeLaPropriete;


    // Déclaration des méthodes de l'objet
}
```

- Les propriétés peuvent être déclarées à tout moment à l'intérieur du corps de la classe.
- Chaque déclaration de propriété est construite sur le modèle suivant :

```
TypeDeLaProprieté      NomDeLaPropriete;
```

- Une propriété peut être initialisée lors de sa déclaration :

```
TypeDeLaProprieté      NomDeLaPropriete = valeurInitiale;
```

- Les identifiants de propriété par convention commencent par une majuscule. Comme toutes les instructions du langage C#, chaque déclaration de propriété DOIT absolument être terminée par un point-virgule **;** Le point-virgule ne constitue pas un séparateur, mais un *terminateur* d'instructions.

1.2.4 Implantation des méthodes

Les méthodes d'une classe sont implémentées, à l'intérieur du bloc **{ }** contrôlé par le mot clé **class**.

Quand on considère une méthode par rapport à l'objet à laquelle elle est appliquée, il faut voir l'objet comme étant sollicité de l'extérieur par un ***message***. Ce ***message*** peut comporter des paramètres. L'objet doit alors réagir à ce ***message*** en exécutant cette fonction ou méthode.

```
public class NomDeLaClasse
{
        // Déclaration des propriétés de l'objet
        int Vitesse = 30 ;

        TypeResultat NomDeMethode(Type1 par1, Type2 par2)
        {
            // Instructions de la méthode
            return expression ;
        }
}
```

- Les identifiants de méthodes commencent par une majuscule.
- Comme toutes les instructions du langage C#, les instructions définissant le fonctionnement de la méthode DOIVENT absolument être terminées par un point-virgule **;** Le point-virgule ne constitue pas un séparateur, mais plutôt un *terminateur* d'instructions.
- Si aucun paramètre n'est désigné explicitement entre les parenthèses, le compilateur considère la méthode comme étant sans paramètre. Dans ce cas, les **parenthèses sont néanmoins obligatoires**.
- Sauf si le résultat de la méthode est de type **void**, l'une des instructions doit être **return** suivi d'une expression dont le résultat est de même type. Ce résultat constitue le retour de la méthode et peut être exploité par le programme envoyant le message.

*Les fonctions développées, dans le cadre des T.P. du support précédent, peuvent être considérées comme des méthodes de la classe application. L'une de ces méthodes est la fonction **Main**. Cette fonction constituant le point d'entrée de l'application peut être considérée comme la méthode exécutée en réponse d'un message envoyé par le système (la machine virtuelle C#) qui signifierait "démarre".*

1.2.5 Instanciation

Pour qu'un objet ait une existence, il faut qu'il soit ***instancié***. Une même classe peut être instanciée **plusieurs fois**, chaque instance ayant des propriétés ayant des valeurs spécifiques.

En C#, il n'existe qu'une seule manière de créer une ***instance*** d'une classe. Cette création d'instance peut se faire en deux temps :

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

Par exemple, l'instanciation de la classe **String** dans la fonction **main** d'une classe application se passerait de la façon suivante :

```
public static void Main(String[] args)
{
    String s;
    s = new String("ISTA");
```

```
}
```

La déclaration d'une variable **s** de classe **Salarie** n'est pas suffisante. En effet, **s** ne *contient* pas une donnée de type **Salarie**. **s** est une variable qui contient une référence sur un objet. Par défaut, la valeur de cette référence est **null**, mot clé C# signifiant que la variable n'est pas référencée. La référence d'un objet doit être affectée à cette variable. Cette référence est calculée par l'instruction **new** au moment de l'instanciation.

Ces deux étapes peuvent être réduites à une seule instruction :

```
String s = new String("ISTA");
```

1.2.6 Accès aux propriétés et aux méthodes

Bien qu'un accès direct aux propriétés d'un objet ne corresponde pas exactement au concept d'encapsulation de la programmation orientée objet, il est possible de le faire en C#. On verra, au chapitre suivant, comment protéger les données membres en en interdisant l'accès.

L'accès aux propriétés et aux méthodes d'une classe se fait par l'opérateur..

Opérateur.

```
Fraction fr1 = new Fraction();
fr1.Numerateur = 3;
fr1.Denominateur = 4;
fr1.Afficher();
```

1.2.7 Conventions d'écriture

Quand on débute la Programmation Orientée Objet, l'un des aspects le plus rebutant est l'impression d'éparpillement du code des applications. En respectant quelques conventions dans l'écriture du code et en organisant celui-ci de façon rationnelle, il est possible de remédier facilement à cet inconvénient.

Ces conventions ne sont pas normalisées et peuvent différer en fonctions des ouvrages consultés.

Les classes

Tous les identificateurs de classe commencent par une Majuscule. Par exemple : **Fraction**.

Les propriétés

Tous les identificateurs public de propriétés commencent par une majuscule, les propriétés private ou protected par une minuscule.

Par exemple :

private **denominateur**, **numerateur**. (sans accent)

Méthode

Elle commence par une lettre en majuscule.

e**xemple** : AjouterVehicule() ; Demarrer() ; InsererFichier () ; le premier mot est un verbe.

Les noms de fichiersConvention d'écriture:

On codera chaque classe d'objet dans des fichiers différents dont l'extension sera **.cs** Le nom de fichier sera identique au nom de la classe.

Exemple pour la classe **Fraction** : **Fraction.cs**

Les noms de fichiersConvention d'écriture:

Les noms de vairiables de travail commencent par une minuscule

1.3 TRAVAIL A REALISER

- Créer la classe **clsSalarie**. Cette classe aura 5 propriétés de type public:
  - matricule **Matricule** **int**
  - catégorie **Categorie** **int**
  - service **Service** **int**
  - nom **Nom** **String**
  - salaire **Salaire** **double**
- Créer une méthode public **CalculerSalaire** pour afficher la mention "Le salaire de " suivie du nom du salarié, suivi de " est de ", suivi de la valeur du salaire.

Implanter une classe application, avec une méthode **Main** dans laquelle la classe **Salarie** sera instanciée pour en tester les propriétés et les méthodes.

T.P. N° 2 - ENCAPSULATION PROTECTION ET ACCES AUX DONNEES MEMBRES

2.1 OBJECTIFS

- Protection des propriétés (données membres).
- Fonctions de type ***Get*** et ***Set***, d'accès aux propriétés.

2.2 CE QU'IL FAUT SAVOIR

2.2.1 Protection des propriétés

En Programmation Orientée Objet, 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 C#, c'est le programmeur qui choisit si une donnée membre ou une fonction membre est accessible directement ou pas.

Par défaut, en C#, toutes les propriétés et méthodes sont accessibles directement. Il faut donc préciser explicitement les conditions d'accès pour chaque propriété et chaque méthode. Pour cela, il existe trois mots-clés :

- **public** - Après ce mot clé, toutes les données ou fonctions membres sont accessibles.
- **private** - Après ce mot clé, toutes les données ou fonctions membres sont verrouillées et ne seront pas accessibles dans les classes dérivées.
- **protected** - Après ce mot clé, toutes les données ou fonctions membres sont verrouillées mais sont néanmoins accessibles dans les classes dérivées.

*La distinction entre **private** et **protected** n'est visible que dans le cas de la déclaration de nouvelles classes par héritage. Ce concept sera abordé ultérieurement dans ce cours.*

Afin d'implanter **correctement** le concept d'encapsulation, il convient de **verrouiller** l'accès aux propriétés et de les déclarer **private**, tout en maintenant l'accès aux méthodes en les déclarant **public**.

Exemple :

```
public class Client
{
        private int numeroClient;          // numéro client
        private String  nomClient;         // nom du client
        private double caClient;           // chiffre d'affaire client

        public void AugmenterCA(double montant)
        {
            caClient = montant;
        }
}
```

2.2.2 Fonctions d'accès aux propriétés

Si les propriétés sont verrouillées, on ne peut plus y avoir accès de l'extérieur de la classe. Il faut donc créer des méthodes dédiées à l'accès aux propriétés pour chacune d'elles. Ces méthodes doivent permettre un accès dans les deux sens :

- **pour connaître la valeur de la propriété**. Ces méthodes sont appelées méthodes de type ***Get***.*?*". La réponse de l'objet, donc la valeur retournée par la méthode ***Get,*** doit être cette valeur.

  Par exemple, pour la propriété **numeroClient**, déclarée **int**, la fonction ***Get*** serait déclarée de la façon suivante :

```
public int Numero()                    ou public GetNumero()
{
    return numeroClient;
}
```

  Cette fonction pourra être utilisée dans la fonction **Main**, par exemple :

```
Client cli = new Client();
int numero = cli.numeroClient;
int numero = cli. Numero ();      ou cli.GetNumero();
```

- **pour modifier la valeur d'une propriété**. Ces méthodes sont appelées méthodes ***Set***. Cette méthode ne retourne aucune réponse. Par contre, un paramètre de même nature que la propriété doit lui être indiquée.

  Par exemple, pour la propriété **numeroClient**, déclarée **int**, la fonction ***Set*** sera déclarée de la façon suivante :

```
public void Numero(int numero) ou SetNumero(int numero)
{
    numeroClient = numero;
}
```

Cette fonction pourra être utilisée dans la fonction **main**, par exemple :

```
Client sal = new Client();
~~cli.numeroClient = 5;~~
cli.Numero(5);
```

L'intérêt de passer par des fonctions ***Set*** est de pouvoir y localiser des contrôles de validité des paramètres passés pour assurer la cohérence de l'objet, en y déclenchant des exceptions par exemple. La sécurisation des classes sera abordée ultérieurement dans ce cours.

2.3 TRAVAIL A REALISER

A partir du travail réalisé au T.P. N° 1, modifier la classe **clsSalarie** pour :

- protéger les propriétés et en interdire l'accès de l'extérieur de l'objet (l'accès aux fonctions membres doit toujours être possible), créer les méthodes d'accès aux attributs Get et Set.
- Modifier la fonction **Main** de la classe Application pour tester.

T.P. N° 3 - CONSTRUCTION ET DESTRUCTION

3.1 OBJECTIFS

- Constructeurs et destructeur des objets
- Propriétés et méthodes de classe

3.2 CE QU'IL FAUT SAVOIR

3.2.1 Constructeurs

Quand une instance d'une classe d'objet est créée au moment de l'instanciation d'une variable avec **new**, une fonction particulière est exécutée. Cette fonction s'appelle le ***constructeur***. Elle permet, entre autre, d'initialiser chaque instance pour que ses propriétés aient un contenu cohérent.

Un constructeur est déclaré comme les autres fonctions membres à deux différences près :

- Le nom de l'identificateur du constructeur est le même nom que l'identificateur de la classe.
- Un constructeur ne renvoie pas de résultat.

Exemple :

```
public Client()
{
        // code du constructeur sans paramètre
}
```

3.2.2 Surcharge des constructeurs

Il peut y avoir plusieurs constructeurs pour une même classe, chacun d'eux correspondant à une initialisation particulière. Tous les constructeurs ont le même nom mais se distinguent par le nombre et le type des paramètres passés (cette propriété s'appelle ***surcharge*** en C#). Quand on crée une nouvelle classe, il est indispensable de prévoir tous les constructeurs nécessaires. Deux sont particulier :

Constructeur d'initialisation

Ce constructeur permet de procéder à une instanciation en initialisant toutes les propriétés, la valeur de celles-ci étant passée dans les paramètres.

Pour la classe **Client**, ce constructeur est déclaré comme ceci :

```
public class Client
{
        private int numeroClient;       // numéro client
        private String  nomClient;            // nom du client
        private double caClient;        // chiffre d'affaire client

        public void AugmenterCA(double montant)
        {
            caClient = montant;
        }
}
```

```
// constructeur

public Client(int numero, String nom , double ca)
{
        numeroClient = numero;
        nomClient = nom;
        caClient = ca;
}
}
```

Cela va permettre d'instancier la classe **Client** de la façon suivante :

```
public static void Main (String[] args)
{
        Salarie sal = new Client(2,"Toto" 2500.00);
        // ...
}
```

Constructeur par défaut

Un constructeur par défaut existe déjà pour chaque classe si aucun autre constructeur n'est déclaré. A partir du moment où le constructeur d'initialisation de la classe **Client** existe, il devient impossible de déclarer une instance comme on l'a fait dans le T.P. précédent :

```
Salarie sal = new Salarie();
```

Pour qu'une telle déclaration, sans paramètre d'initialisation, soit encore possible, il faut créer un ***constructeur par défaut***. En fait ce n'est réellement indispensable que si une instanciation de l'objet, avec des valeurs par défaut pour ses propriétés, a un sens.

Pour la classe **Client**, on peut s'interroger sur le sens des valeurs par défaut des propriétés. A titre d'exemple, ce constructeur serait déclaré comme ceci :

```
public Client()
{
    numeroClient = 0;
    nomClient = "Dupont";
    caClient = 0.00;
}
```

Constructeur de recopie

Le constructeur de recopie permet de recopier les propriétés d'un objet existant dans vers la nouvelle instance

```
public static void Main (String[] args)
{
        Client client1 = new Client();                  // 1
        Client client2 = new Client (client1);     // 2
        Client cli = new Client(2,"Toto",17500.00);   // 3

        // ...
}
```

En 1, client1 est un objet créé via le constructeur par défaut. La propriété nomClien" est initialisé par "Dupont"

En 2, salarie2 est un objet différent de salarie1 mais les propriétés des deux objets ont les mêmes valeurs. Pour cela, le développeur doit écrire un constructeur de recopie.

En 3, salarie3 est instancié via le constructeur d'initialisation

```
public Client (Client unClient)
{
    numeroClient = unClient. numeroClient;
    nomClient = unClient.nomClient ;
    caClient = unClient. caClient;
}
```

3.2.3 Propriétés de classe

Jusqu'à présent, les propriétés déclarées étaient des ***propriétés d'instance***. C'est à dire que les propriétés de chaque objet, instancié à partir de la même classe, peuvent avoir des valeurs différentes (numero, nom, etc).

Supposons donc maintenant, que dans la classe **Client**, il soit nécessaire de disposer d'un compteur d'instance, dont la valeur serait le nombre d'instances en cours à un instant donné.

En C#, il est possible de créer des ***propriétés de classe***. Leur valeur est la même pour toutes les instances d'une même classe. Pour déclarer une telle propriété, on utilise le mot-clé **static**. Par exemple, dans la classe **Salarie**, le compteur d'instance pourrait être déclaré :

```
public class Client
{
        private static int compteur = 0;
}
```

La propriété de classe **compteur**, initialisée à 0 lors de sa déclaration, sera incrémentée de 1 dans tous les constructeurs développés pour la classe **Client**. Sa valeur sera le nombre d'instances valides à un instant donné.

3.2.4 Méthodes de classe

Comme pour les autres propriétés déclarées **private**, il est nécessaire de créer les méthodes d'accès associées. Pour ce compteur, seule la méthode ***Get*** est nécessaire. Cependant, comme les propriétés de classe sont partagées par toutes les instances de la classe, le fait d'envoyer un message ***Get*** pour obtenir leur valeur n'a pas de sens. En reformulant, on pourrait dire que le message ***Get*** à envoyer pour obtenir le nombre d'instances de classe **Salarie** ne doit pas être envoyée à une instance donnée de cette classe, mais plutôt à la classe elle-même. De telles méthodes sont appelées ***méthodes de classe***.

Une méthode de classe est déclarée par le mot-clé **static**. Pour la méthode ***Get*** d'accès au compteur d'instance on déclarerait :

```
public static int Compteur()  ou GetCompteur()
{
    return compteur;
}
```

L'appel à une méthode **static** est sensiblement différent aux appels standard. En effet, ce n'est pas à une instance particulière que le message correspondant doit être envoyé, mais à la classe.

Dans la fonction **Main**, par exemple, si l'on veut affecter à une variable le nombre d'instances de la classe Salarie, cela s'écrirait :

```
public static void Main (String[] args)
{
      Client cli = new Client(2,"Toto",17500.00);
      // ...
      int nInst = cli.compteur();      // private
      int nInst = Client.Compteur();
}
```

3.2.5 Destructeur

En C# on ne peut pas déclencher explicitement la destruction d'un objet. Les instances sont automatiquement détruites lorsqu'elles ne sont plus référencées. Le programme qui se charge de cette tâche s'appelle le ***Garbage Collector*** ou, en français, le ***ramasse-miettes***. Le ***Garbage Collector*** est un système capable de surveiller les objets créés par une application, de déterminer quand ces objets ne sont plus utiles, d'informer sur ces objets et de détruire les objets pour récupérer leurs ressources.

Or si l'on veut que le compteur d'instances soit à jour, il est nécessaire de connaître le moment ou les instances sont détruites pour décrémenter la variable **compteur**.

C'est pour cela que le ***ramasse –miettes*** envoie un message à chaque instance avant qu'elle soit détruite, et ceci quelle que soit la classe. Il suffit d'écrire la méthode de réponse à ce message, c'est la méthode **Finalize()**. Cette méthode s'appelle ***destructeur***. C'est , bien sûr, une méthode d'instance de la classe que l'on est en train de développer. Dans la classe **Client** par exemple, elle se déclare OBLIGATOIREMENT de la façon suivante :

```
protected void Finalize()
{
      compteur --;          // décrémentation du compteur
}
```

L'exemple suivant permet de visualiser le fonctionnement asynchrone du *Garbage Collector. (si vous avez de la chance …)*

Le programme principal **Main** propose une boucle permettant de créer un grand nombre d'instances de la classe **Detruire**. A chaque instanciation, le programme perd la référence sur l'objet créé auparavant et le destructeur est ainsi invoqué. Le fait de créer un grand nombre d'instance laisse le temps au "thread" du ***garbage collectior*** de faire sont travail**.**

```
public class Gc
{
     public static void Main(String args[])
     {
          Detruire d = new Detruire ();
          for (int i = 0 ; i < 1000000000 ; i++)
                        {
                              d = new Detruire ();
                        }
     }
}

//////////////////////////////////////////////////
public class Detruire
{
     static int count = 0;
     public Detruire ()
            {
           count++;
                }

     protected void Finalize()
     {
           Console.WriteLine ("objet détruit " );
     }}
```

3.3 TRAVAIL A REALISER

- A partir du travail réalisé au T.P. N° 2, implémenter les constructeurs et le destructeur de la classe **Salarie**.
- Afin de mettre en évidence les rôles respectifs des constructeurs et du destructeur, implémenter ceux-ci pour qu'ils affichent un message à chaque fois qu'ils sont exécutés.
- Implémenter un compteur d'instances pour la classe **Salarie**.
- Ajouter une méthode de classe permettant de mettre le compteur à zéro ou à une valeur prédéfinie.
- Modifier le jeu d'essai de l'application Main pour tester ces fonctions.

T.P. N°4 - L'HERITAGE

4.1 OBJECTIFS

- Généralités sur l'héritage
- Dérivation de la classe racine **Object**

4.2 CE QU'IL FAUT SAVOIR

4.2.1 L'héritage

Le concept d'héritage est l'un des trois principaux fondements de la Programmation Orientée Objet, le premier étant l'encapsulation vu précédemment dans les T.P. 1 à 3 et le dernier étant le polymorphisme qui sera abordé plus loin dans ce document.

L'héritage consiste en la création d'une nouvelle classe dite ***classe dérivée*** à partir d'une classe existante dite ***classe de base*** ou ***classe parente***.

L'héritage permet de :

- **récupérer** le comportement standard d'une classe d'objet (classe parente) à partir de propriétés et des méthodes définies dans celles-ci,
- **ajouter** des fonctionnalités supplémentaires en créant de nouvelles propriétés et méthodes dans la classe dérivée,
- **modifier** le comportement standard d'une classe d'objet (classe parente) en surchargeant certaines méthodes de la classe parente dans la classe dérivée.

Exemple :

```
class ClasseA
{
      // Propriétés de la classe A
      public int dataA;
      // Méthodes de la classe A
      public int FonctionA1()
      {
            // Code de la méthode fonctionA1
      }
      public virtual int FonctionA2()        //surchargeable
      {
            // Code de la méthode fonction A2
      }
}
```

```
class ClasseB : ClasseA
{
      // Propriétés de la classe B
      public int dataB;
      // Méthodes de la classe B
      public override int FonctionA2()      // surchargée
      {
            // Code de la méthode fonctionA2
      }
      public int FonctionB1()
      {
            // Code de la méthode fonctionB2
      }
}
```

Dans cet exemple :

- **ClasseA** est la ***classe parente***.
- **ClasseB** est une ***classe dérivée*** de la classe **ClasseA**.
- **dataA** est une ***propriété*** de la classe **ClasseA**. Par héritage, **dataA** est aussi une ***propriété*** de la classe **ClasseB**.
- **dataB** est une ***propriété*** de la classe **ClasseB** (mais pas de **ClasseA**).
- **FonctionA1** est une ***méthode*** de la classe **ClasseA**. Par héritage, c'est aussi une ***méthode*** de la classe **ClasseB**.
- **FonctionB1** est une ***méthode*** de la classe **ClasseB** (mais pas de **ClasseA**).
- **FonctionA2** est une ***méthode*** des classes **ClasseA** et **ClasseB**.
  - dans la classe A, FonctionA2() est déclarée **virtual** car elle est surchargeable dans la classe B
  - dans la classe B, FonctionA2 est déclarée **override** car elle remplace la méthode de la classe A

4.2.2 Protection des propriétés et des méthodes

En plus des mots-clés **public** et **private** décrits dans les chapitres précédents, il est possible d'utiliser un niveau de protection intermédiaire de propriétés et des méthodes par le mot-clé **protected**. Les exemples ci-dessous permettent d'illustrer ces différents niveaux de protection.

Exemple pour la déclaration des classes **ClasseA** et **ClasseB** :

```
public class ClasseA
{
     // Propriétés de la classe A
     private int dataA1;
     protected int dataA2;
     public int dataA3;

     // Méthodes de la classe A
     public int FonctionA()
     {
          dataA1 = 5;          // 1
          dataA2 = 6;
          dataA3 = 7;
          dataB = 8;           // 2
     }
}

public class ClasseB : ClasseA
{
     // Propriétés de la classe B
          public int dataB;
     // Méthodes de la classe B
     public int FonctionB()
     {
          dataA1 = 15;         // 3
          dataA2 = 16;
          dataA3 = 17;
          dataB = 18;
     }};
```

Exemple pour l'utilisation d'instances des classes **ClasseA** et **ClasseB** :

```
public static void Main (String[] args)
{
     ClasseA ca = new ClasseA();
     ClasseB cb = new ClasseB();

     ca.dataA1 = 5; // 4
     ca.dataA2 = 6; // 5
     ca.dataA3 = 7; // 6

     cb.dataA1 = 5; // 7
     cb.dataA2 = 6; // 8
     cb.dataA3 = 7; // 9
     cb.dataB = 8;
}
```

- **dataA1**, **dataA2** et **dataA3** sont des propriétés de la classe **ClasseA** respectivement **private**, **protected** et **public**.
- **FonctionA** est une méthode de **ClasseA** et **FonctionB** une méthode de **ClasseB**.

1. **dataA1**, **dataA2** et **dataA3** sont des propriétés de **ClasseA**. Elles sont donc directement accessibles dans toutes les méthodes de **ClasseA**, donc dans **FonctionA**.
2. **dataB** est propriété de **ClasseB** (classe dérivée de **ClasseA**). Une propriété d'une classe dérivée est inaccessible dans la classe de base.
3. **dataA1** est déclarée **private** dans la classe **ClasseA**. Elle fait bien partie des composantes de la classe **ClasseB** mais est inaccessible dans toutes les méthodes de cette classe.
4. **dataA1** est déclarée **private** dans la classe **ClasseA**. Elle est inaccessible dans toutes les fonctions qui utilisent des instances de la classe **ClasseA**. Il n'y a pas ici de distinction entre **private** et **protected**.
5. **dataA2** est déclarée **protected** dans la classe **ClasseA**. Elle est inaccessible dans toutes les fonctions qui utilisent des instances de la classe **ClasseA**. Il n'y a pas ici de distinction entre **private** et **protected**.
6. **dataA3** est déclarée **public** dans la classe **ClasseA**. Elle est donc directement accessible dans toutes les fonctions qui utilisent des instances de la classe **ClasseA**.
7. **dataA1** est déclarée **private** dans la classe **ClasseA**. Elle est inaccessible dans toutes les fonctions qui utilisent des instances de la classe **ClasseB**.
8. **dataA2** est déclarée **protected** dans la classe **ClasseA**. Elle est accessible dans toutes les fonctions qui utilisent des instances de la classe **ClasseB**.
9. **dataA3** est déclarée **public** dans la classe **ClasseA**. Elle est donc directement accessible dans toutes les fonctions qui utilisent des instances de la classe **ClasseB**.

4.2.3 Compatibilité et affectation

Lors des affectations (opérateurs =), il est possible de mélanger des instances de classes différentes. Cependant, il y a quelques règles à observer :

```
ClasseA caObj;
ClasseB cbObj;
```

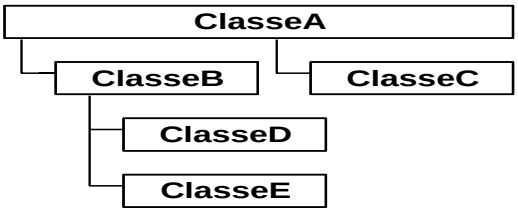
- **ClasseA** est une classe de base.
- **ClasseB** est une classe dérivée de **ClasseA**.
- **caObj** est une instance de la classe **ClasseA**.
- **cbObj** est une instance de la classe **ClasseB**.

- L'instruction **caObj = cbObj;** est légale. Elle affecte à **caObj** l'objet référencé par **cbObj**. Ceci est possible car une instance de **ClasseB** est aussi une instance de **ClasseA**. Il n'y pas de conversion ici au sens propre. L'objet référencé par **caObj** reste une instance de **ClasseB**. Une variable d'une ***classe de base*** est compatible avec les variables de toutes ses ***classes dérivées***.
- L'instruction **cbObj = caObj;** génère une erreur de compilation. Cependant, si **caObj** contient une référence sur une instance de **ClasseB**, une telle affectation peut être forcée par un ***casting*** (changement de type) : **cbObj = (ClassB)pcaObj;**

4.2.4 Mode de représentation

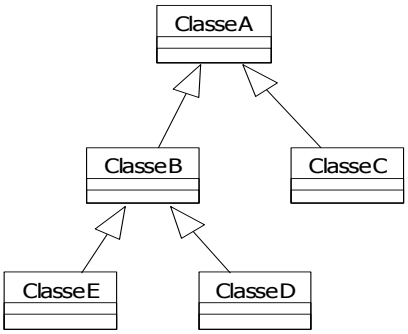
Le concept d'héritage peut être utilisé pratiquement à l'infini. On peut créer des classes dérivées à partir de n'importe quelle autre classe, y compris celles qui sont déjà des classes dérivées.

Supposons que **ClasseB** et **ClasseC** soient des classes dérivées de **ClasseA** et que **ClasseD** et **ClasseE** soient des classes dérivées de **ClasseB**. Les instances de la classe **ClasseE** auront des données et des fonctions membres communes avec les instances de la classe **ClasseB**, voire de la classe **ClasseA**. Si les dérivations sont effectuées sur plusieurs niveaux, une représentation graphique de l'organisation de ces classes devient indispensable. Voici la représentation graphique de l'organisation de ces classes :



Le diagramme ci-dessus constitue la représentation graphique de la ***hiérarchie de classes*** construite à partir de **ClasseA**.

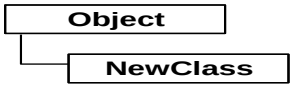
Dans le cadre de la conception orientée objet, la méthode UML (United Modeling Language) propose une autre représentation graphique d'une telle hiérarchie :



4.2.5 Insertion d'une classe dans une hiérarchie

Bien que l'on ait opéré comme cela depuis le début de ce cours, la création d'une nouvelle classe indépendante et isolée n'a pratiquement aucun intérêt en Programmation Orientée Objet. Afin de rendre homogène le comportement des instances d'une nouvelle classe il est important de l'insérer dans une bibliothèque de classe existante. Cette bibliothèque est fournie sous la forme d'une ***hiérarchie de classes*** construite à partir d'une ***classe racine***.

En C#, il est impossible de créer une classe isolée. En effet, lorsqu'on crée une nouvelle classe sans mentionner de classe de base, c'est la classe **Object** , la classe racine de toutes les classes C#, qui est utilisée.



NewClass est une nouvelle classe insérée dans la hiérarchie de classes construite à partir de **Object**.

Le Framework .NET propose une hiérarchie de classes normalisées prêtes à l'emploi, ou à dériver.

4.2.6 Insertion d'une nouvelle classe à partir de Object

La dérivation d'une classe par rapport à une classe prédéfinie doit obéir à un certain nombre de règles définies dans la documentation de la ***classe de base*** à dériver. L'une de ces contraintes est la surcharge OBLIGATOIRE de certaines méthodes de la classe de base avec la même signature. Pour dériver la classe **Object**, cas le plus courant, cela se résume à ne réécrire que quelques méthodes :

Méthode Finalize

Il s'agit du destructeur dont on déjà décrit le rôle dans le chapitre précédent.

Méthode ToString

Cette méthode doit créer une chaîne de caractères (instance de la classe **String**) qui représente les propriétés des instances de la classe. Par exemple, pour la classe **Fraction**, la chaîne de caractères représentant une fraction pourrait être **"nnnn/dddd"** où **nnnn** et **dddd** correspondraient respectivement aux chiffres composant le numérateur et le dénominateur de la fraction.

```
class Fraction
{
      private long numerateur;
      private long denominateur;

      public String ToString()
      {
            return numerateur + "/" + denominateur;
      }
}
```

La méthode **ToString** est appelée lorsqu'une conversion implicite d'un objet en chaîne de caractères est nécessaire comme c'est le cas pour la fonction **WriteLine de l'objet Console**, par exemple :

```
Fraction fr = new Fraction(1,2);
Console.WriteLine("fraction = " + fr);
```

Méthode Equals

Cette méthode doit répondre VRAI si deux instances sont rigoureusement égales. Deux conditions sont à vérifier :

- Il faut que les deux objets soient de la même classe. Le paramètre de la méthode **Equals** étant de type **Object**, notre instance peut donc être comparée à une instance d'une classe quelconque dérivée de **Object**.
- Il faut qu'une règle d'égalité soit appliquée : par exemple deux objets client sont égaux si leurs numéros sont égaux .; pour la classe **Fraction**, on peut dire que deux fractions sont égales si le produit des *extrêmes* est égal au produit des mo*y*ens. Ce qui peut s'écrire :

```
class Fraction
{
      private long numerateur;
      private long denominateur;

      public boolean equals(Fraction fr)
      {
            return    numerateur * fr). denominateur ==
                      denominateur * fr ). numerateur;
      }
}
```

4.2.7 Constructeur et héritage

Chaque constructeur d'une classe dérivée doit obligatoirement appeler le constructeur équivalent de la classe de base.

Si **BaseClass** est une classe de base appartenant à la hiérarchie de classe construite à partir de **Object**, si **NewClass** est une classe dérivée de **BaseClass**, cela se fait de la façon suivante en utilisant le mot-clé **base** :

Constructeur par défaut

```
public NewClass() : BaseClass
{
      public NewClass() : base()  // constructeur
      // les données membres de la classe de base sont
      // initialisées par le constructeur par défaut
      // de cette classe de base
      // Initialisation des nouvelles données membres
      // de la classe dérivée
}
```

Constructeur d'initialisation

```
public NewClass (Valeurs des propriétés de NewClass)
      : base(Valeur des propriétés de la classe BaseClass);
{

      // Initialisation des nouvelles données membres
      // de la classe dérivée de BaseClass
}
```

L'appel de **: base(valeurs)** initialise la partie de l'objet créé avec BaseClass en utilisant le constructeur d'initialisation celle-ci . Puis on complète l'objet en initialisant les attributs spécifiques à NewClass

4.2.8 Appel aux méthodes de la classe de base

Lors de la surcharge d'une méthode de la classe de base dans une classe dérivée, il peut être utile de reprendre la fonctionnalité standard pour y ajouter les nouvelles fonctionnalités. Pour ne pas avoir à réécrire le code de la classe de base (dont on ne dispose pas forcément), il est plus simple de faire appel à la méthode de la classe de base. Pour cela on utilise le mot-clé **base** avec une syntaxe différente à celle utilisée pour le constructeur :

```
public override String Methode1(String s)
{
      // Calcul du résultat standard par appel
      // à la méthode de la classe de base
      String resultat = base.Methode1(s);
      // Calcul du nouveau résultat en modifiant
      // le résultat standard
      resultat = resultat + "," + data2;
      return resultat;
}
```

Attention : la méthode Methode1 de la classe de base devra avoir la signature suivante

```
public virtual String Methode1(String s)
```

4.3 TRAVAIL A REALISER

Pour ce travail, à réaliser à partir de ce qui a été produit au T.P. N° 3, on procédera en deux temps :

- Ajouter à la classe **Salarie** les méthodes **Equals** et **ToString**. La règle d'égalité pour la classe **Salarie** peut s'énoncer de la façon suivante : deux salariés sont égaux s'ils ont le même numéro de matricule et le même nom. **ToString** doit renvoyer toutes les propriétés séparées par des virgules.

- Créer une classe **Commercial** en dérivant la classe **Salarie**. Cette classe aura 2 propriétés supplémentaires pour calculer la commission :
  - chiffre d'affaire **chiffreAffaire** **double**
  - commission en % **commission** **int**

    - Créer les deux constructeurs standards de la classe **Commercial**. Ne pas oublier d'appeler les constructeurs équivalents de la classe de base.
    - Créer les méthodes d'accès aux propriétés supplémentaires.
    - Surcharger la méthode **CalculerSalaire** pour calculer le salaire réel (fixe + commission).
    - Surcharger les autres méthodes de la classe de base pour lesquelles on jugera nécessaire de faire ainsi.

Tester les classes Salarie et Commercial

T.P. N°5 – LES COLLECTIONS

5.1 OBJECTIFS

- Tableaux statiques
- ArrayList
- Dictionnaire Hashtable
- Dictionnaire trié SortedList

5.2 CE QU'IL FAUT SAVOIR

5.2.1 Les tableaux statiques

Les tableaux contiennent des éléments, chacun d'eux étant repéré par son indice. En C#, il existe une manière très simple de créer des tableaux "classiques" sans faire référence aux classes ***collections*** :

```
int[] aTab = new int[20];
for (int i = 0; i < 20; i++)
{
          aTab[i] = i + 1900;
}
String[] aStr = new String[5];
aStr[0] = "André";
aStr[1] = "Mohamed";
aStr[2] = "Marc";
aStr[3] = "Francis";
aStr[4] = "Paul";
Salarie[] aSal = new Salarie[5];
aSal[0] = new Salarie(16, 1, 10, "CAUJOL", 10900.00);
aSal[1] = new Salarie(5, 1, 10, "DUMOULIN", 15600.00);
aSal[2] = new Salarie(29, 3, 20, "AMBERT", 5800.00);
aSal[3] = new Salarie(20, 2, 20, "CITEAUX", 8000.00);
aSal[4] = new Salarie(34, 2, 30, "CHARTIER", 7800.00);
```

Le problème de ce type de tableaux réside en deux points :

- Tous les éléments du tableau doivent être de **même type**.
- Le nombre d'éléments que peut contenir un tableau est limité au moment de la déclaration. Cela sous-entend que le programmeur connaît, au moment de l'écriture du programme, le nombre maximum d'éléments que doit contenir le tableau.

Itération dans le tableau

```
foreach ( intVal in aTab)
{
        Console.WriteLine(intVal);
}
```

5.2.2 Classe ArraysList

C'est un tableau dynamique auquel on peut rajouter ou insérer des éléments, en supprimer.

Il faut utiliser le namespace System.Collections :

```
using System.Collections;
ArrayList al = new ArrayList();
al.Add(1) ;
al.Add(45) ;
al.Add(87) ;
al.Remove(1) ;
etc etc
```

Quelques méthodes de la classe ArrayList

Méthodes ou propriétés	But
Capacity	Détermine le nombre d'éléments que le tableau peut contenir
Count	Donne le nombre d'éléments actuelleùent dans le tableau
Add(object)	Ajoute un élément au tableau
Remove(object)	Enlève un élément du tableau
RemoveAt(int)	Enlève un élément à l'indice fourni
Insert(int, object)	Insère un élément à l'indice fourni
Clear()	Vide le tableau
Contains(object)	Renvoie un booléen vrai si l'objet fourni est présent dans le tableau
al[index]	Fournit l'objet situé à la valeur de index

Explorer le tableau

```
foreach ( int obj in al)
{
        Console.WriteLine(obj);
}
```

Accès à un élément du tableau

```
Console.WriteLine("deuxième élément " + al[1]);
```

5.2.3 La classe Hashtable

Hashtable a un comportement analogue à un tableau associatif. C'est un dictionnaire.

Chaque objet dans C# possède une valeur permettant de l'identifier de façon unique; c'est son **hashcode**. Une mission de la classe **Object** est de s'assurer que tout Objet a un **hashcode** différent. A chaque instanciation, C# calcule un hashcode unique pour l'objet qu'il vient de créer.

Exemple un dictionnaire contenant des objets de la classe Client :

Objet Clef	Objet valeur
Numero client 1	Objet client 1
Numero client 2	Objet client 2

Les dictionnaires (ou Map) contiennent des éléments, chacun d'eux étant repéré par une clef. En C#, les dictionnaires peuvent être implémentés par la classe **Hashtable**. ou la classe SortedList

Ne pas oublier de faire référence au namspace System.Collections

Instanciation d'un objet Hashtable

```
Hashtable dict = new Hashtable();
```

Insérer par Add

```
        dict.Add(1,"Toto") ;
        dict.Add(2,"Titi") ;.
```

Nombre d'éléments du dictionnaire propriété Count

```
         dict.Count
```

Sélectionner un élément

```
        variable = dict[1]
```

Supprimer un élément

```
        dict.Remove(clé)
```

vider le dictionnaire

```
        dict.Clear();
```

explorer les clés et les valeurs

```
        foreach ( int i int dict.Keys)

foreach ( string str  in dict.Values)
```

Le dictionnaire Hashtable ne trie pas les éléments et les restitue dans l'ordre inverse du stockage

5.2.4 SortedList

SortedList est un dictionnaire qui garantit que **les clés sont rangées de façon ascendante**.

Exemple:

```
static void Main()
{
        SortedList sl = new SortedList();
        sl.Add(32, "Java");
        sl.Add(21, "C#");
        sl.Add(7, "VB.Net");
        sl.Add(49, "C++");
        Console.WriteLine("Les éléments triés sont...");
        Console.WriteLine("\t Clé \t\t Valeur");
        Console.WriteLine("\t === \t\t =====");
        for(int i=0; i<sl.Count; i++)
        {
        Console.WriteLine("\t {0} \t\t {1}", sl.GetKey(i), sl.GetByIndex(i));
        }
 }
```

Méthodes ou propriétés	But
Keys	Collection des clés de la liste triée
Values	Collection des valeurs (objets) de la liste
Count	Donne le nombre d'éléments actuelleùent dans la liste
Add(object key, object value)	Ajoute un élément à la liste (paire clé-valeur)
Remove(object key)	Enlève un élément e la liste dont la valeur correspond à la clé
RemoveAt(int)	Enlève un élément à l'indice fourni

Clear()	Vide la liste
ContainsKey(object key)	Renvoie un booléen vrai si la clé fournie est présent dans la liste
ContainsValue(object value)	Renvoie un booléen vrai si la valeur fournie est présent dans la liste
GetKey(int index)	Renvoie la clé correspondant l'indice spécifié
GetByIndex(int index)	Renvoie la valeur correspondant à l'indice spécifié
IndexOfKey(object key)	Renvoie l'indice correspondant à une clé
IndexOfValue(object value)	Renvoie l'indice correspondant à une valeur

5.3 TRAVAIL A REALISER

- Tester quelques méthodes sur les différentes collections : ArrayList, Hashtable, SortedList

A partir de la classe **Salarie** implémentée dans les T.P. précédents, dans la fonction **Main** de l'application,

- Créer une instance d'une collection **SortedList** dans laquelle on va ranger des instances de la classe **Salarie** repérées par leur numéro de matricule.
- Créer au moins cinq instances de la classe **Salarie** et les insérer dans la collection.
- Faire l'itération de la collection pour afficher son contenu par **ordre croissant** des numéros de matricules.
- Chercher un salarié dans la collection en fournissant son matricule

T.P. N° 6 - POLYMORPHISME

6.1 OBJECTIFS

- Polymorphisme
- Fonctions virtuelles
- Classes génériques

6.2 CE QU'IL FAUT SAVOIR

6.2.1 Polymorphisme

Considérons l'exemple suivant.

```
Salarie sal = new Commercial(7, 2, 10, "PEDROL Jean", 8000.00,45000.00, 12);
string str = sal.CalculerSalaire();
```

La variable **sal** est déclarée de type **Salarie**, mais elle est instanciée à partir de la classe **Commercial**. Une telle affectation est correcte et ne génère pas d'erreur de compilation. La question que l'on doit se poser maintenant est la suivante : lors de l'envoi du message **calculSalaire** à **sal**, quelle est la méthode qui va être exécutée ? La méthode **calculSalaire** de la classe **Salarie** qui calcul le salaire uniquement à partir du salaire de base, ou la méthode **calculSalaire** de la classe **Commercial** qui prend en compte les propriétés commission et chiffre d'affaire de la classe **Commercial** ? Un simple test va permettre de mettre en évidence que c'est bien la méthode **calculSalaire** de la classe **Commercial** qui est exécutée. C# *sait* et *se souvient* comment les objets ont été instancié pour pouvoir appeler la bonne méthode.

En Programmation Orientée Objet, c'est ce que l'on appelle le concept de ***Polymorphisme***. Pour toutes les instances de salarié, quelles soient instanciées à partir de la classe de base **Salarie** ou d'une classe dérivée comme **Commercial**, il faut pouvoir calculer le salaire. C'est le ***comportement polymorphique***. Par contre le calcul ne se fait pas de façon identique pour les salariés normaux et les commerciaux.

6.2.2 Méthodes virtuelles

L'ambiguïté est levée du fait que la résolution du lien (entre le programme appelant et la méthode) ne se fait pas au moment de la compilation, mais pendant l'exécution en fonction de la classe qui a été utilisée pour l'instanciation. On parle de ***méthode virtuelle***. **calculSalaire** est une méthode virtuelle qui défini un comportement polymorphique sur la hiérarchie de classe construite sur **Salarie**.

Pour associer un comportement polymorphique à une méthode, il suffit de "surcharger" la méthode la classe de base, avec exactement la même signature, c'est-à-dire le même nom, le même type de résultat, le même nombre de paramètres, de mêmes types, dans le même ordre.

La classe racine **Object** est elle-même un polymorphisme. Les méthodes **Finalize**, **ToString** et **Equals**, surchargées dans les T.P. précédents pour les classes **Salarie** et **Commercial**, sont des fonctions virtuelles qui définissent un comportement polymorphique pour toutes les classes dérivées de **Object**.

6.2.3 Classes génériques

Dans certains cas, lors de l'élaboration d'une hiérarchie de classe ayant un comportement polymorphique, il peut être intéressant de mettre en facteur le comportement commun à plusieurs classes, c'est-à-dire créer une ***classe générique***. Il est probable que le fait de créer une instance de cette classe ne correspond pas à une réalité au niveau conceptuel. C'est le cas de la classe racine **Object** : Il est absurde de créer une instance de la classe **Object** qui, par ailleurs, définit le comportement commun à toutes les classes C#. C'est le cas également de la classe **Dictionary** qui définit le comportement commun à toutes les classes dictionnaire, mais qui ne peut être instanciée. Une telle classe est déclarée **abstract** :

```
abstract public class Dictionary
{
        // Définitions des membres de la classe
}
```

Essayer d'instancier une classe **abstract** avec l'instruction **new** va générer une erreur de compilation.

6.3 TRAVAIL A REALISER

- Modifier la fonction **Main** développée dans le T.P. précédent en insérant quelques instances de la classe **Commercial** dans la liste triée.
- Vérifier le fonctionnement du *Polymorphisme* lié à la fonction **CalculerSalaire**.