Pour commencer, créez un répertoire **c:\Exos-C#Net** qui contiendra l'ensemble de tous vos exercices du cours.

Exercices (chapitres I) : Exemples

Pour les premiers exercices, créez un sous-répertoire **Exemples** dans **\Exos-C#Net** qui contiendra vos premiers exercices.

Exercice BIENVENUE :

Premier exemple, à compiler sur ligne de commande (premier contact avec le compilateur).

Fichiers : dans \Exos-C#Net\Exemples\Bienvenue
bienvenue.cs
bienvenue.exe

Dans un sous-répertoire **Bienvenue**, écrire un programme qui affiche "Bienvenue" (compilation avec *csc*) et le tester.

Puis, désassembler ce programme **bienvenue.exe** à l'aide de *ildasm* et examiner le contenu des différents éléments de l'arborescence affichée. Sauvegarder, sous le nom **exemple.il**, le résultat du désassemblage (menu Fichier\Dump), ensuite éditer-le pour modifier le texte affiché. Recompiler le fichier modifié **exemple.il** à l'aide de *ilasm* et tester la modification.

A l'aide de l'environnement Visual Studio .Net, préparez une nouvelle solution **Exemples** dans le répertoire \Exos-C#Net.

Exercice ACCUEIL :

Premier exemple, à préparer comme projet dans l'environnement Visual Studio .Net (premier contact avec l'environnement de développement).

Fichiers : dans la solution **Exemples**, nouveau projet "Application Console" **Accueil**
accueil.cs

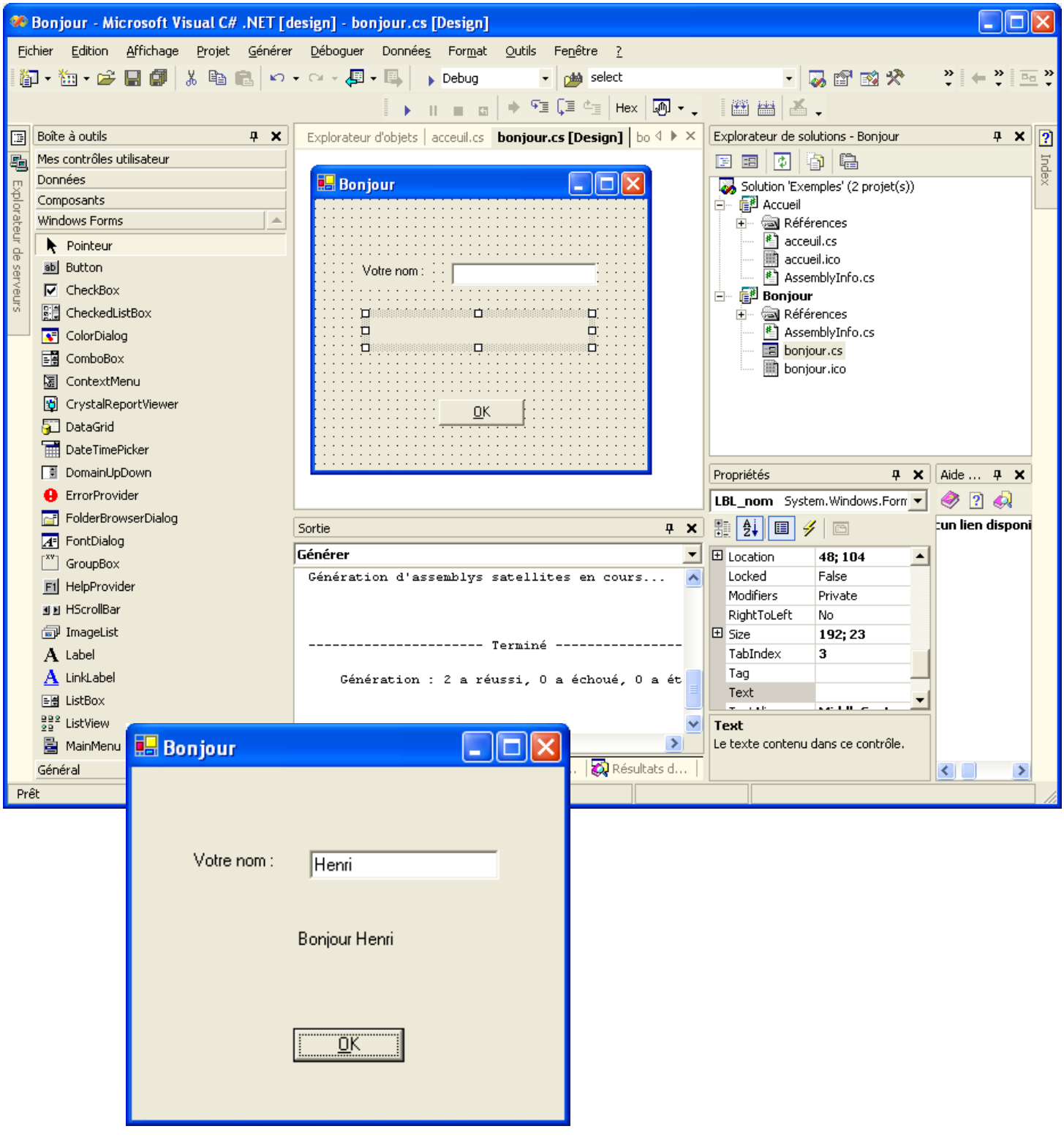
Refaire le premier exercice "Accueil", mais dans l'environnement de développement Visual Studio .Net. Pensez à bien nommer ou renommer vos différents éléments, en particulier les fichiers.

Exercice BONJOUR :

Premier exemple d'application fenêtrée, à préparer comme projet dans l'environnement Visual Studio .Net (premier contact avec l'environnement de développement graphique).

Fichiers : dans la solution **Exemples**, nouveau projet "Application Windows" **Bonjour**
bonjour.cs

Réaliser la fenêtre d'application suivante, en prenant soin de bien nommer ou renommer les objets graphiques et les fichiers.



Exercices (chapitre I) : Les Bases

Pour ces exercices, créez une solution **LesBases** dans **\Exos-C#Net** qui contiendra les exercices sous forme de projets.

Exercice ARTICLE :

Créer une structure comprenant des champs et des méthodes.

Fichiers : dans la solution **LesBases**, nouveau projet "Application Console" **Article**
article.cs

Créer la structure **Article**, un article étant composé d'un nom, d'un prix et d'une quantité.

Un article comporte les méthodes *Afficher*, *Ajouter* et *Retirer* qui permettent respectivement d'afficher dans la console les champs d'un article, d'ajouter un nombre entier positif à la quantité disponible ou de le retirer.

Un article dispose d'un constructeur prenant en argument toutes les informations permettant d'initialiser cet article.

Pour tester votre structure *Article*, créez deux ou trois articles, affichez-les, puis modifiez leurs quantités avant de les afficher de nouveau. Enfin, écrivez un test permettant à l'utilisateur de saisir au clavier les informations nécessaires à la création d'un nouvel article que vous afficherez également. Vous aurez vraisemblablement besoin de la classe **Convert** ou de méthodes de classes **Parse** (en cas d'erreur de saisie, laissez le programme partir en état d'erreur).

Exercice ARTICLE TYPÉ :

Utilisation d'une énumération.

Fichiers : dans la solution **LesBases**, nouveau projet "Application Console" **ArticleTypé**
articleTypé.cs

Par rapport à l'exercice *Article*, ajouter un champ indiquant le type de l'article. Ce champ doit correspondre à un type énuméré, avec différentes valeurs d'énumération possibles comme alimentaire, droguerie, habillement, loisir,...

Modifier le constructeur et la méthode d'affichage pour qu'ils gèrent le type de l'article.

Pour tester votre structure *ArticleTypé*, créez deux ou trois articles, affichez-les, puis modifiez leurs quantités avant de les afficher de nouveau. Pour le test avec saisie, l'utilisateur doit pouvoir entrer le type de l'article sous forme d'une chaînes de caractères (en cas d'erreur de saisie, laissez le programme partir en état d'erreur).

Exercice TABLEAU D'ARTICLE :

Création d'un tableau d'articles.

Fichiers : dans la solution **LesBases**, nouveau projet "Application Console" **ArticleTableau**
articleTableau.cs

En reprenant l'exercice ArticleTypé, créer un tableau de trois articles. Les initialiser dans le programme, puis manipulez et affichez ce tableau.

Exercice FACTORIELLE :

Utilisation de différents types de boucles.

Fichiers : dans la solution **LesBases**, nouveau projet "Application Console" **Factorielle**
factorielle.cs

Calculer la valeur de factorielle 10 (1x2x3x4x...X10) à l'aide d'une boucle **while**, puis d'une boucle **do...while** et enfin d'une boucle **for**.

Ecrire une méthode de classe (statique) prenant en argument un entier et retournant sa factorielle. Utiliser un mode de calcul récursif.

Exercice JEU :

Mise en œuvre d'entrées/sorties et utilisation des structures de contrôle *if* et *while*.

Fichiers : dans la solution **LesBases**, nouveau projet "Application Console" **Jeu**
jeu.cs

Le but de cet exercice est de trouver interactivement, par saisie au clavier, un nombre cible tiré au hasard, compris entre 1 et 100.

Demander au "joueur" de saisir un nombre au clavier, a priori entre 1 et 100. Lui indiquer si le nombre saisi est trop grand ou trop petit par rapport à la valeur cible visée. Répéter la saisie tant que le "joueur" n'a pas trouvé le nombre cible.

Pour choisir une valeur cible au hasard, utilisé l'instruction suivante :

```
int cible=(new System.Random()).Next(1,100);
```

Exercice INCRÉMENTER UN TABLEAU :

Utilisation de tableaux, de méthodes et de boucles.

Fichiers : dans la solution **LesBases**, nouveau projet "Application Console" **IncrémenterTableau**
incrémenterTableau.cs

Le but de cet exercice est d'écrire trois méthodes pour incrémenter un même tableau, en jouant sur différentes formes de paramétrage.

Vous déclarerez les trois méthodes de classe (statiques) afin de pouvoir les utiliser directement dans la méthode principale, comme s'il s'agissait de simples fonctions.

Méthode **IncTab** : cette méthode incrémente chaque case d'un tableau d'entiers passé en argument. Après l'appel de la méthode, le tableau fourni en argument est modifié. Utilisez une boucle **for**.

Méthode **IncNewTab** : cette méthode prend en paramètre un tableau d'entiers et retourne un nouveau tableau de même taille contenant les valeurs du tableau passé en paramètre, mais incrémentées de 1.

Méthode **IncNewTabOut** : cette méthode prend en paramètre un tableau d'entiers et retourne le résultat dans un autre tableau, également passé en paramètres mais avec l'option **out**. Le deuxième tableau en sortie doit contenir les éléments du premier tableau incrémentés 1.

Dans la méthode principale, déclarez un tableau d'entiers que vous initialiserez et qui vous servira ensuite pour tester les trois méthodes. Utilisez des boucles **foreach** pour afficher les différents résultats après chaque appel.

Exercice TROUVER LE PLUS GRAND :

Utilisation d'une méthode avec un nombre variable d'arguments.

Fichiers : dans la solution **LesBases**, nouveau projet "Application Console" **TrouverLePlusGrand**
trouverLePlusGrand.cs

Ecrire et tester une méthode de classe (statique) **PlusGrand** qui récupère un nombre variable d'arguments entiers et qui retourne le plus grande de ces valeurs.
Vérifiez que cette méthode fonctionne correctement même si toutes les valeurs passées en arguments sont négatives.

Exercices (chapitre II) : Programmation Objet

Pour ces exercices, créez une solution **ProgrammationObjet** dans **\Exos-C#Net** qui contiendra les exercices sous forme de projets.

Exercice PERSONNES :

Créer une classe comprenant des champs et des méthodes d'instance et de classe.

Fichiers : dans la solution **ProgrammationObjet**, nouveau projet "Application Console" **Personnes**
personne.cs

Écrire une classe **Personne** qui comporte trois champs privés, nom, prénom et âge.

Cette classe comporte un constructeur pour permettre d'initialiser les données. Elle comporte également une méthode pour afficher les données de chaque personne.

Définir également une méthode de classe *Combien()* qui permette de connaître le nombre de personnes créées depuis le lancement du programme.

Écrire une classe **TestPersonne** afin de tester le fonctionnement de la classe *Personne*.

Modifiez la classe *Personne* pour que la méthode *Combien()* ne retourne plus le nombre d'instances créées, mais le nombre d'instances en cours d'utilisation. Pourquoi cela ne marche-t-il pas ?

Exercice SOCIÉTÉ AVEC TABLEAU :

Mise en œuvre des notions d'héritage et de polymorphisme à travers plusieurs dérivations.

Fichiers : dans la solution **ProgrammationObjet**,
nouveau projet "Application Console" **SociétéTableau**
personne.cs, **employé.cs**, **chef.cs**, **directeur.cs** et **sociétéTableau.cs**

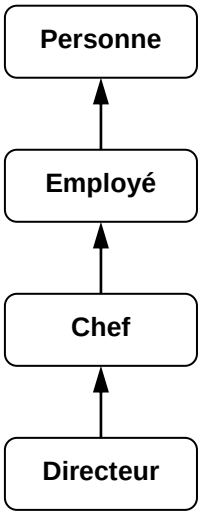
Créer, par ajout à votre projet, quatre fichiers de classes pour les classes suivantes :

classe **Personne** : champs _Nom, _Prénom et _Age, avec les propriétés correspondantes, deux constructeurs, une surcharge de la méthode **ToString**, une méthode polymorphe **Afficher** et surcharge de l'**opérateur ++** pour incrémenter l'âge d'une personne.

classe **Employé** : dérive de la classe *Personne*, avec en plus un champ _Salaire accompagné de sa propriété, avec un constructeur et surcharge des méthodes *ToString* et *Afficher*.

classe **Chef** : dérive de la classe *Employé*, avec en plus un champ _Service accompagné de sa propriété, avec un constructeur et surcharge des méthodes *ToString* et *Afficher*.

classe **Directeur** : dérive de la classe *Chef*, avec en plus un champ _Société accompagné de sa propriété, avec un constructeur et surcharge des méthodes *ToString* et *Afficher*.



Employé : Personne

```
- _Salaire  decimal
+ Salaire  decimal [get/set]
+ Employé(nom,prénom,age,
          salaire)
+ ToString()
+ Afficher()
```

Chef : Employé

```
- _Service  string
+ Service  string [get/set]
+ Chef(nom,prénom,age,
       salaire,service)
+ ToString()
+ Afficher()
```

Personne

```
- _Nom      string
- _Prénom   string
- _Age      int
+ Nom      string [get]
+ Prénom   string [get]
+ Age      int    [get/set]
+ Personne(nom,prénom)
+ Personne(nom,prénom,age)
+ ToString()
+ Afficher()
+ operator ++(Personne)
```

Directeur : Chef

```
- _Société  string
+ Société  string [get]
+ Directeur(nom,prénom,age,
             salaire,service,
             société)
+ ToString()
+ Afficher()
```

Dans la méthode principale de l'application (fichier *sociétéTableau.cs*), créez un tableau de huit personnes avec cinq employés, deux chefs et un directeur (8 références de la classe *Personne* dans lesquelles ranger 5 instances de la classe *Employé*, 2 de la classe *Chef* et 1 de la classe *Directeur*).

Affichez l'ensemble des éléments du tableau à l'aide de **for**.
Incrémentez l'âge d'un employé, modifiez un salaire et l'intitulé du service d'un chef.
Affichez l'ensemble des éléments du tableau à l'aide de **foreach**.

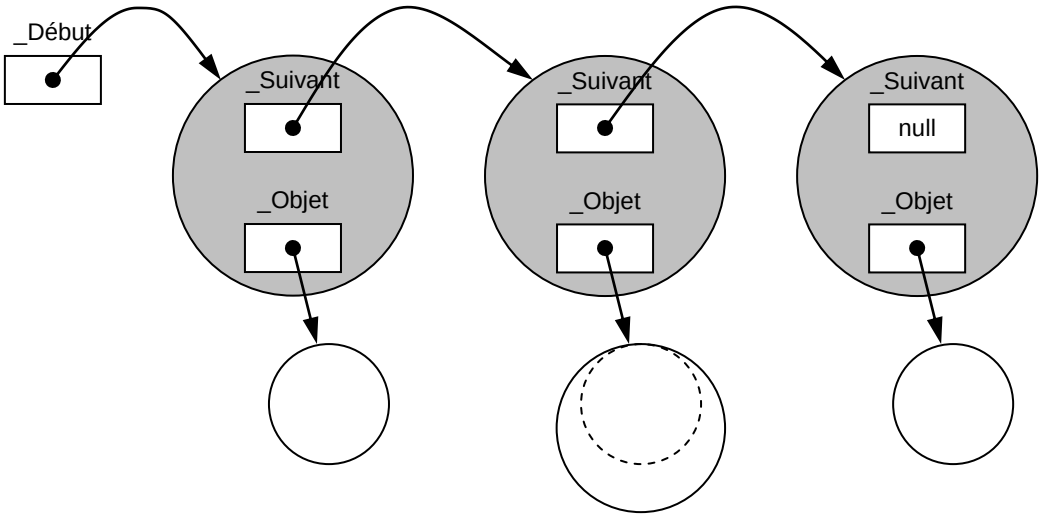
Exercice SOCIÉTÉ AVEC LISTE :

Mise en place d'une liste chaînée avant simple, à l'image d'une classe containeur.

Fichiers : dans la solution **ProgrammationObjet**,
nouveau projet "Application Console" **SociétéListe**
personne.cs, **employé.cs**, **chef.cs**, **directeur.cs, listeChaînée.cs** et **sociétéListe.cs**

Reprendre les quatre classes **Personne**, **Employé**, **Chef** et **Directeur,** de l'exercice précédent, toujours sous forme de quatre fichiers de classes séparés, *personne.cs*, *employé.cs*, *chef.cs* et *directeur.cs*. Les définitions de ces quatre classes doivent appartenir au même espace de nommage **Société**.

Créez, dans un fichier de classe *listeChainée.cs*, deux classes **Elément** et **Liste** pour mettre en place le mécanisme de liste chaînée. Ces deux classes doivent appartenir au même espace de nommage **ListeChainée**. Le principe est de chaîner, par un chaînage simple avant, des objets de type *Elément*, chaque objet de type *Elément* pointant lui-même sur un objet quelconque, entité que l'on cherche en fait à chaîner. Un objet *Liste* contient un ensemble d'objets de type *Elément* constitués en un chaînage unique. Notre système *Liste*/*Elément* peut être considéré comme un mécanisme de type "containeur";



classe **Elément** : permet de décrire un élément, la liste chaînée étant constituée d'un chaînage de ces éléments

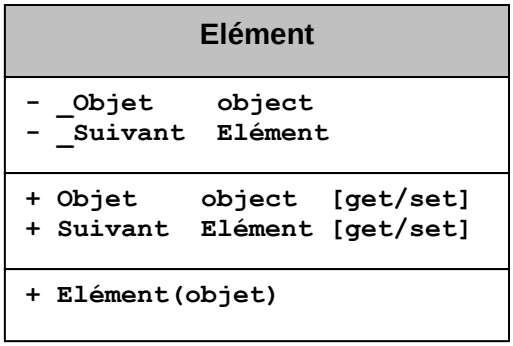
- champ **_Objet** : référence vers l'objet qui doit en être considéré comme l'entité cible du chaînage
- champ **_Suivant** : référence l'élément suivant dans la liste (*null* pour le dernier élément de la liste)
- propriété **Objet** : référence sur l'objet "entité à chaîner" (en lecture et en modification)
- propriété **Suivant** : référence sur l'élément suivant du chaînage (en lecture et modification)
- constructeur **Elément** : constructeur avec un argument de type objet, pointant sur l'entité à référencer dans l'élément

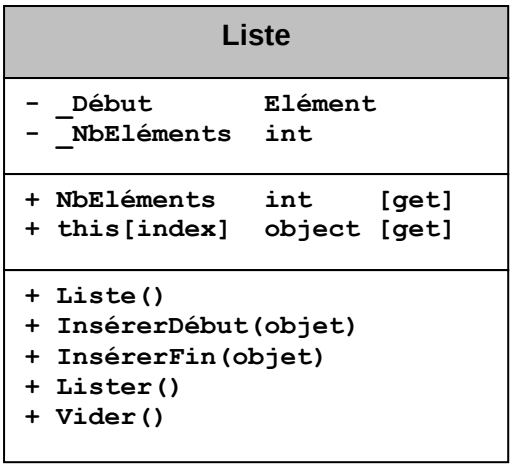
classe **Liste** : permet de gérer une liste chaînée d'éléments

- champ **_Début** : référence sur le premier élément de la liste (*null* si la liste est vide)
- champ **_NbEléments** : nombre actuel d'éléments dans le liste (0 si de liste vide)
- propriété **NbEléments** : nombre d'éléments (en lecture seule)
- constructeur **Liste** : constructeur sans argument pour initialiser _Début et _NbEléments
- méthode **InsérerDébut** : insère un nouvel élément en début de liste (prend comme paramètre un référence sur l'objet "entité à chaîner" que doit référencer ce nouvel élément)
- méthode **InsérerFin** : insère un nouvel élément en fin de liste (prend comme paramètre un référence sur l'objet "entité à chaîner" que doit référencer ce nouvel élément)
- méthode **Lister** : parcourt l'ensemble de la liste chaînée pour afficher chaque objet "entité à chaîner" référencé par chacun des éléments de la liste; l'affichage se fait à l'aide de la méthode *ToString* que les objets référencés dans la liste doivent avoir redéfinie
- méthode **Vider** : vidage de la liste en parcourant tous les éléments afin de les référencer à *null* (ne pas oublier de remettre _Début à *null* et le nombre d'éléments à zéro)

Le fichier *sociétéListe.cs* contient la classe **GérerSociétéListe**, dans l'espace de nommage **SociétéListe**, avec le point d'entrée principal de l'application. Pour tester le système de liste chaînée, construisez une liste chaînée *Liste* en y insérant, en début de liste, dans n'importe quel ordre, les entités suivantes : 5 *Employés*, 2 *Chefs* et 1 *Directeur*. Listez le contenu de la liste chaînée, puis videz cette liste pour y insérer de nouveau les mêmes éléments, mais avec insertion en fin de liste. Affichez la nouvelle liste.

Pour ce deuxième affichage, modifiez la classe *Liste* de façon à y introduire un indexeur de type entier permettant de parcourir la liste comme un tableau afin de récupérer chacun des objets référencés par les éléments de la liste (c'est là que la propriété *NbEléments* va devenir intéressante). Affichez la liste, avec insertion en fin, en ayant recours à cet indexeur.





Exercice SOCIÉTÉ AVEC ÉNUMÉRATEUR :

Mise en place d'un énumérateur pour parcourir une liste chaînée avant simple.

Fichiers : dans la solution **ProgrammationObjet**,
nouveau projet "Application Console" **SociétéEnumération**
personne.cs, **employé.cs**, **chef.cs**, **directeur.cs, listeChaînée.cs**,
listeEnumération et **sociétéEnumération.cs**

Reprendre, de l'exercice précédent, les quatre classes **Personne**, **Employé**, **Chef** et **Directeur** ainsi que les deux classes **Elément** et **Liste,** toujours sous forme de fichiers de classes séparés, *personne.cs*, *employé.cs*, *chef.cs* et *directeur.cs* ainsi que le fichier *listeChainée.cs*. Les définitions de ces quatre premières classes doivent toujours appartenir au même espace de nommage **Société** et les deux dernières à l'espace de nommage **ListeChainée**.

Dans un nouveau fichier de classe *listeEnumération.cs*, créez une classe **ListeEnumération** implémentant l'interface **IEnumerator**. Le but de cette classe est de pouvoir offrir la possibilité de parcourir une liste chaînée de type *Liste* à l'aide d'un **foreach** comme si c'était une "collection". Cette classe devra appartenir au même espace de nommage **ListeChainée** que les classes *Eléments* et *Liste*. Cet énumérateur pourra s'appuyer sur l'indexeur.

Le fichier *sociétéEnumération.cs* contient la classe **GérerSociétéEnumération**, dans l'espace de nommage **SociétéEnumération**, avec le point d'entrée principal de l'application. Pour tester le système de liste énumérée, construisez une liste chaînée *Liste* en y insérant, en début ou en fin de liste, dans n'importe quel ordre, les entités suivantes : 5 *Employés*, 2 *Chefs* et 1 *Directeur*. Listez le contenu de la liste chaînée à l'aide de l'indexeur, puis une seconde fois à l'aide de l'énumérateur.

ListeEnumération
- _Liste Liste - _indiceCourant int
+ Current object [get]
+ ListeEnumération(liste) + MoveNext() + Reset()

Exercices (chapitre III) : Concepts avancés

Pour ces exercices, créez une solution **ConceptsAvances** dans **\Exos-C#Net** qui contiendra les exercices sous forme de projets.

Exercice SOCIÉTÉ ORDONNÉE :

Mise en place d'une liste chaînée avant simple avec insertion en fonction d'un ordre et gestion d'anomalies.

Fichiers : dans la solution **ProgrammationObjet**,
nouveau projet "Application Console" **SociétéOrdonnée**
personne.cs, **employé.cs**, **chef.cs**, **directeur.cs, listeChaînée.cs**, **listeEnumération**, **comparaison.cs** et **sociétéOrdonnée.cs**

Reprendre, de l'exercice précédent, les quatre classes **Personne**, **Employé**, **Chef** et **Directeur** ainsi que les classes **Elément, Liste** et **ListeEnumération** sous forme de six fichiers de classes séparés, *personne.cs*, *employé.cs*, *chef.cs* et *directeur.cs* ainsi que le fichier *listeChainée.cs* et *listeEnumération*. Les définitions de ces quatre premières classes doivent toujours appartenir au même espace de nommage **Société** et les trois dernières à l'espace de nommage **ListeChainée**.

Dans un nouveau fichier de classe *comparaison.cs*, définissez une interface ***IComparaison*** comportant une seule méthode **ComparerAvec**. Cette méthode reçoit en paramètre un objet de type *Object* et retourne un *int*. Le but de cette méthode est de pouvoir introduire une notion d'ordre entre des personnes, ordre basé soit sur l'ordre alphabétique des noms, soit sur l'âge. Cette interface devra appartenir à un nouvel espace de nommage **Comparaison**.

IComparaison
+ ComparerAvec(objet)

Règles pour implémenter la méthode **ComparerAvec** :

si obj1>obj2 alors obj1.ComparerAvec(obj2) doit retourner un entier >0
si obj1=obj2 alors obj1.ComparerAvec(obj2) doit retourner 0
si obj1<obj2 alors obj1.ComparerAvec(obj2) doit retourner un entier <0

Cette interface doit donc être implémentée dans la classe **Personne** afin de permettre la comparaison de deux personnes suivant l'ordre alphabétique des noms (à cette fin, vous pouvez vous aider de la méthode *CompareTo* de la classe *String*).

La classe **Liste** doit comporter une nouvelle méthode supplémentaire **InsérerOrdre** qui permet d'insérer de nouveaux éléments dans la liste chaînées en respectant l'ordre sur les objets défini par la méthode **ComparerAvec**.

Le fichier *sociétéOrdonnée.cs* contient la classe **GérerSociétéOrdonnée**, dans l'espace de nommage **SociétéOrdonnée**, avec le point d'entrée principal de l'application. Pour tester le système de liste ordonnée, construisez une liste chaînée *Liste* en y insérant, à l'aide de la méthode **InsérerOrdre**, les entités suivantes : 5 *Employés*, 2 *Chefs* et 1 *Directeur*. Listez le contenu de la liste chaînée à l'aide de l'indexeur, puis une seconde fois à l'aide de l'énumérateur : les éléments doivent apparaître dans l'ordre alphabétique des noms.

Modifier l'implémentation de l'interface ***IComparaison*** afin d'obtenir à la place un listage par ordre croissant d'âge.

Gestion d'exception :

Toujours dans le même exercice, il s'agit d'introduire une gestion d'exception en cas de débordement d'indice dans l'indexeur de la liste chaînée.

Dans un nouveau fichier de classe *listeIndiceException.cs*, définissez une nouvelle classe **ListeIndiceException** avec l'idée, qu'en cas de débordement d'indice dans l'indexeur on puisse éventuellement récupérer un message, l'indice en cause et le champ des valeurs valides d'indice.

ListeIndiceException
- _Liste Liste **- _IndiceMauvais int**
+ IndiceMauvais int [get]
+ ListeIndiceException(message, **liste,** **indiceMauvais)** **+ ToString()**

Faites un test de débordement à l'aide d'un indice invalide pour l'indexeur et affichez les informations pertinentes à votre disposition concernant cette anomalie.

Exercice PERDU DANS LE DÉSERT :

Mise en œuvre de la notion de délégué.

Fichiers : dans la solution **ProgrammationObjet**,
nouveau projet "Application Console" **PerduDansLeDésert**
promenadeDansLeDésert.cs et **perduDansLeDésert.cs**

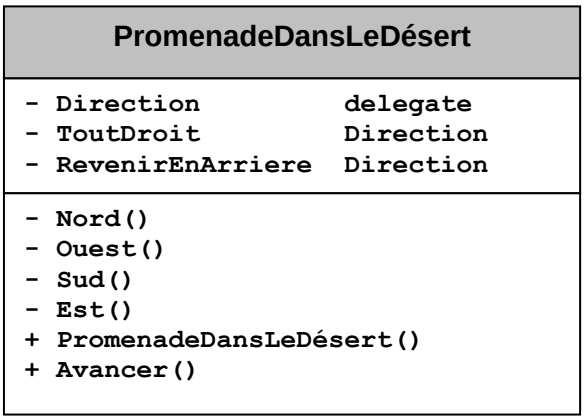
Ecrire une classe **PromenadeDansLeDésert** offrant quatre méthodes d'instance privées, **Nord**, **Ouest**, **Sud** et **Est**, simulant un déplacement dans la direction nommée. Une méthode d'instance publique **Avancer** permet de demander à l'utilisateur la direction dans laquelle il souhaite avancer, nord, ouest, sud, est ou continuer tout droit.

Un délégué **ToutDroit** permet de gérer le déplacement "tout droit". Ce délégué est par défaut, au départ, défini pour aller vers le nord.

Ajouter une option "retour à la case départ" qui permet à l'utilisateur, à l'aide d'un autre délégué **RetourEnArrière**, de revenir sur ses pas jusqu'au point de départ.

A chaque étape d'avancement, vous indiquerez la direction choisie par l'utilisateur. En cas de retour au point de départ, vous afficherez les directions successives empruntées pour chacune des étapes du retour en arrière.

Une classe **PerduDansLeDésert** permet la mise en œuvre d'un parcours à l'aide d'une instance de la classe **PromenadeDansLeDésert**.



Voici un exemple d'exécution :

```
Dans quelle direction voulez-vous aller ?
  (N[ord], S[ud], E[st], O[uest], T[out droit], R[etour], F[in])) : S
Vers le sud...
Dans quelle direction voulez-vous aller ?
  (N[ord], S[ud], E[st], O[uest], T[out droit], R[etour], F[in])) : S
Vers le sud...
Dans quelle direction voulez-vous aller ?
  (N[ord], S[ud], E[st], O[uest], T[out droit], R[etour], F[in])) : E
Vers l'est...
Dans quelle direction voulez-vous aller ?
  (N[ord], S[ud], E[st], O[uest], T[out droit], R[etour], F[in])) : N
Vers le nord...
Dans quelle direction voulez-vous aller ?
  (N[ord], S[ud], E[st], O[uest], T[out droit], R[etour], F[in])) : E
Vers l'est...
Dans quelle direction voulez-vous aller ?
  (N[ord], S[ud], E[st], O[uest], T[out droit], R[etour], F[in])) : T
Vers l'est...
Dans quelle direction voulez-vous aller ?
  (N[ord], S[ud], E[st], O[uest], T[out droit], R[etour], F[in])) : R
Vers l'ouest...
Vers l'ouest...
Vers le sud...
Vers l'ouest...
Vers le nord...
Vers le nord...
Dans quelle direction voulez-vous aller ?
  (N[ord], S[ud], E[st], O[uest], T[out droit], R[etour], F[in])) : F

*** Fin de parcours ***
```

Exercice LISTE DE DIFFUSION :

Mise en œuvre de la notion de délégué et d'événement.

Fichiers : dans la solution **ProgrammationObjet**,
nouveau projet "Application Console" **Diffusion**
Client.cs, **ArchivageMessages.cs**, **EnvoiMessage.cs** et **diffusion.cs**

Le principe de cet exercice est de diffuser des messages à des clients qui s'abonnent à une diffusion et également de réaliser un archivage de tous les messages envoyés.

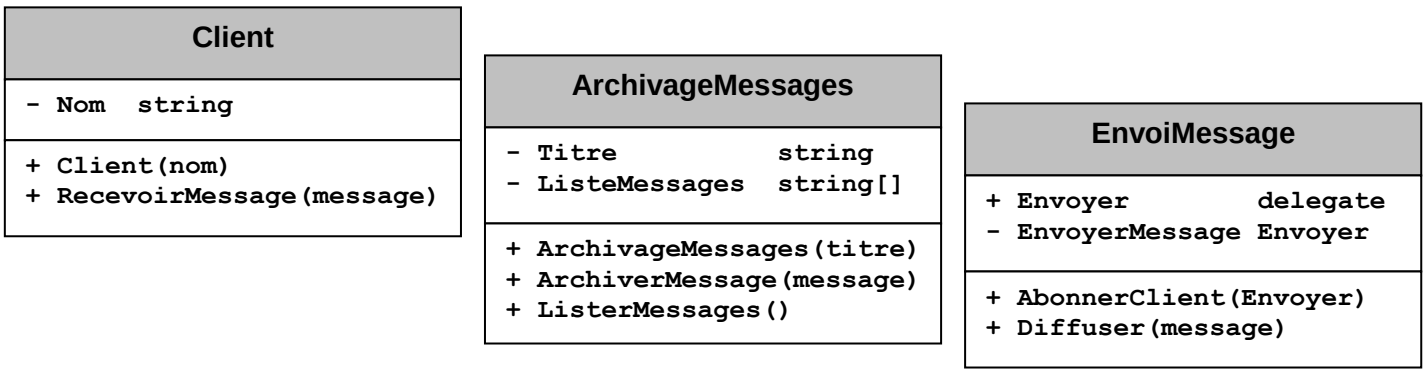
Toutes les classes de cet exercice appartiennent au même espace de nommage **ListeDiffusion**.

Une classe **Client** permet de définir un client pouvant s'abonner à une liste de diffusion de messages. Un client est identifié par un champ _Nom. Cette classe doit comporter une méthode **RecevoirMessage** qui ne retourne rien et qui reçoit en argument un message sous forme d'une chaîne de caractères; cette méthode affiche le nom du destinataire et le message reçu.

Une classe **ArchivageMessages** permet de recevoir tous les messages afin de les archiver dans un tableau. Chaque archive comporte un titre et un tableau des messages reçus. La méthode **ArchiverMessage** ne retourne rien et reçoit en argument un message sous forme d'une chaîne de caractères; cette méthode archive le message reçu dans le tableau d'archives. Une méthode **ListerMessages** permet d'afficher le titre de l'archive et liste tout le contenu du tableau d'archives.

Une classe **EnvoiMessage** contient la définition d'un type délégué **Envoyer** pour des méthodes avec un argument de type chaîne de caractères et sans valeur retournée. Cette classe contient également un champ **EnvoyerMessage** qui est une référence de type "*event*" **Envoyer**. Une méthode **Abonner** permet d'enregistrer une méthode dans le champ *event* **EnvoyerMessage** afin de simuler l'inscription de clients, ainsi que l'archivage, comme destinataires de la diffusion de messages. Une méthode **Diffuser** reçoit un message pour le diffuser via l'événement **EnvoyerMessage**.

Une classe de test **Diffusion** permet de mettre en œuvre des listes de diffusion et de diffuser des messages à ces listes. Par exemple, créez deux listes "réunion" et "week-end" et abonnez à la première cinq clients et en plus de l'archive; à la seconde abonnez seulement trois de ces clients ainsi que l'archive. Envoyez un ou deux messages à chacune des deux listes, puis lister le contenu de l'archivage.



Exercices (chapitre III) : Assembly

Pour ces exercices, créez une solution **Assemblages** dans **\Exos-C#Net** qui contiendra les exercices sous forme de projets.

Exercice ASSEMBLY :

Créer une classe sous forme de librairie dynamique "privée".

Fichiers : dans la solution **Assemblages** nouveau projet "Bibliothèque de classes" **Asm**
asm.cs

Dans l'espace de nommage **Asm**, écrire une classe **Point** permettant de décrire un point, avec comme champs privés les coordonnées entières du point _X et _Y, ainsi qu'un booléen _IsVisible indiquant si le point est visible ou non.
Cette classe dispose d'un constructeur avec deux arguments entiers pour initialiser les coordonnées d'un point, un point n'étant pas visible lors de sa création.
Une propriété *IsVisible* [set], permet de modifier l'état visible ou non d'un point et également d'afficher un message indiquant ses coordonnées et son état visible ou non.

Point
- _X int **- _Y int** **- _IsVisible bool**
+ IsVisible [set] bool
+ Point(int,int)

Exercice ASSEMBLY CLIENT :

Utilisation d'une classe rangée dans une libraire "privée".

Fichiers : dans la solution **Assemblages** nouveau projet "Application Console" **AsmClient**
asmClient.cs

Dans l'espace de nommage **AsmClient**, écrire une classe **GérerPoint** pour créer une série de 4 points sous forme d'un tableau et manipuler ces points afin de les rendre visible ou invisible. Ces points sont des instances de la classe **Point** définie dans l'exercice précédent **Assembly** sous forme d'une bibliothèque de classes.

Exercice ASSEMBLY PARTAGÉ :

Créer une classe sous forme de librairie dynamique "publique".

Fichiers : dans la solution **Assemblages** nouveau projet "Bibliothèque de classes" **AsmPartagé**
asmPartagé.cs, **asmPartagé.snk**

Dans cet exercice, on reprend la classe **Point** décrite dans l'exercice précédent **Assembly**.

Dans l'espace de nommage **AsmPartagé**, écrire une classe **Point** permettant de décrire un point, avec comme champs privés les coordonnées entières du point _X et _Y, ainsi qu'un booléen _IsVisible indiquant si le point est visible ou non.
Cette classe dispose d'un constructeur avec deux arguments entiers pour initialiser les coordonnées d'un point, un point n'étant pas visible lors de sa création. Une propriété *IsVisible* [set], permet de modifier l'état visible ou non d'un point et également d'afficher un message indiquant ses coordonnées et son état visible ou non.

Point
- _X int - _Y int - _IsVisible bool
+ IsVisible [set] bool
+ Point(int,int)

Modifiez le fichier **AssembyInfo.cs** de ce projet en renseignant divers rubriques (Produit, Auteur, etc.). Fixez le numéro de version à 1.0.0.0 et créez un fichier *.snk de façon que cet *assembly* soit publiable dans le GAC (*Global Assembly Cache*) et publiez-le.

Exercice ASSEMBLY PARTAGÉ CLIENT :

Utilisation d'une classe rangée dans une libraire "publique".

Fichiers : dans la solution **Assemblages** nouveau projet "Application Console" **AsmPartagéClient**
asmPartagéClient.cs

Dans l'espace de nommage **AsmPartagéClient**, écrire une classe **GérerPoint** pour créer un tableau de 4 points et manipuler ces points afin de les rendre visible ou invisible. Ces points sont des instances de la classe **Point** définie dans l'exercice précédent **AssemblyPartagé** sous forme d'une bibliothèque de classes. Vérifiez que cet *assembly* s'exécute.

Suite de l'exercice :

(1) - Déplacez cet *assembly* exécutable dans un autre répertoire c:\asm0
(2) - Supprimez l'*assembly* AsmPartage.dll de l'environnement de développement
(3) - Vérifiez que dans c:\asm0 l'*assembly* exécutable fonctionne toujours correctement
(4) - Générez un nouvel *assembly* AsmPartage.dll avec 1.0.0.1 comme numéro de version
(5) - Publiez dans le GAC ce nouvel *assembly* et vérifiez la présence des deux versions
(6) - Générez de nouveau l'*assembly* exécutable AsmPartageClient.exe
(7) - Déplacez cet *assembly* exécutable dans un autre répertoire c:\asm1
(8) - Supprimez l'*assembly* AsmPartage.dll de l'environnement de développement
(9) - Vérifiez dans c:\asm0 et c:\asm1 que les deux *assemblies* exécutables fonctionnent toujours
(10) - Supprimez du GAC l'une des deux versions de la bibliothèque AsmPartage.dll
(11) - Tentez d'exécuter chacun des deux *assemblies* exécutables rangés dans c:\asm0 et c:\asm1

Exercices (chapitre IV) : Classes utilitaires

Pour ces exercices, créez une solution **ClassesFramework** dans **\Exos-C#Net** qui contiendra les exercices sous forme de projets.

Exercice DATES :

Utilisation des classes permettant de gérer date et heure.

Fichiers : dans la solution **ClassesFramework** nouveau projet "Application Console" **Dates**
dates.cs

Affichez le jour de la semaine où vous êtes né(e), le nombre jours que vous avez déjà vécus, ainsi les nombres de minutes et de secondes.

Exercice LISTE ORDONNÉE :

Utilisation de la classe *ArrayList* et de l'interface *IComparable*.

Fichiers : dans la solution **ClassesFramework** nouveau projet "Application Console" **ListeOrdonnée**
personne.cs, **employé.cs**, **chef.cs**, **directeur.cs** et **listeOrdonnée.cs**

Reprendre les quatre classes **Personne**, **Employé**, **Chef** et **Directeur,** de l'exercice précédent **SociétéListe**, toujours sous forme de quatre fichiers de classes séparés, *personne.cs*, *employé.cs*, *chef.cs* et *directeur.cs*. Les définitions de ces quatre classes appartiennent au même espace de nommage **Société**.

L'interface ***IComparable*** doit être implémentée dans la classe **Personne** afin de permettre la comparaison de deux personnes suivant l'ordre alphabétique des noms (à cette fin, vous pouvez vous aider de la méthode *CompareTo* de la classe *String*).

Le fichier *listeOrdonnée.cs* contient la classe **GérerListeOrdonnée**, dans l'espace de nommage **ListeOrdonnée**, avec le point d'entrée principal de l'application. Utilisez un tableau de type ***ArrayList*** pour y insérer, comme précédemment dans n'importe quel ordre, les entités suivantes : 5 *Employés*, 2 *Chefs* et 1 *Directeur*.

Affichez un première les éléments du tableau à l'aide d'une boucle ***for***.

Triez le tableau par ordre alphabétique croissant sur les noms.

Affichez de nouveau les éléments du tableau trié à l'aide d'une boucle ***foreach***.

Exercice LISTE PLUSIEURS TRIS :

Utilisation de la classe *ArrayList* et de l'interface *IComparer*.

Fichiers : dans la solution **ClassesFramework** nouveau projet "Application Console" **ListeTris**
personne.cs, **employé.cs**, **chef.cs**, **directeur.cs**, **comparateurs.cs** et **listeTris.cs**

Reprendre les quatre classes **Personne**, **Employé**, **Chef** et **Directeur,** de l'exercice précédent **SociétéListe**, toujours sous forme de quatre fichiers de classes séparés, *personne.cs*, *employé.cs*, *chef.cs* et *directeur.cs*. Les définitions de ces quatre classes appartiennent au même espace de nommage **Société**.

Dans un nouveau fichier de classe *comparateurs.cs*, définissez deux classes **ComparateurPersonneNom** et **ComparateurPersonneAge** qui implémentent l'une et l'autre l'interface ***IComparer***, la première pour permettre la comparaison de deux personnes par rapport à l'ordre alphabétique de leurs noms, et la seconde par rapport à leurs âges. Ces deux classes devront appartenir au même espace de nommage **ComparateursPersonnes**.

Le fichier *listeTris.cs* contient la classe **GérerListeTris**, dans l'espace de nommage **ListeTris**, avec le point d'entrée principal de l'application. Utilisez un tableau de type ***ArrayList*** pour y insérer, comme précédemment dans n'importe quel ordre, les entités suivantes : 5 *Employés*, 2 *Chefs* et 1 *Directeur*.

Affichez une première fois les éléments du tableau après insertion, puis opérer un premier tri pour lister ces éléments par ordre alphabétique croissant sur les noms, puis opérer un deuxième tri pour lister les mêmes éléments mais en ordre croissant par rapport aux âges.

Exercice LISTE TRIÉE :

Utilisation de la classe *SortedList*.

Fichiers : dans la solution **ClassesFramework** nouveau projet "Application Console" **ListeTrieée**
personne.cs, **employé.cs**, **chef.cs**, **directeur.cs** et **listeTriée.cs**

Reprendre les quatre classes **Personne**, **Employé**, **Chef** et **Directeur,** de l'exercice précédent **SociétéListe**, toujours sous forme de quatre fichiers de classes séparés, *personne.cs*, *employé.cs*, *chef.cs* et *directeur.cs*. Les définitions de ces quatre classes appartiennent au même espace de nommage **Société**.

Le fichier *listeTriée.cs* contient la classe **GérerListeTriée**, dans l'espace de nommage **ListeTriée**, avec le point d'entrée principal de l'application. Utilisez une liste de type ***SortedList*** pour y insérer, comme précédemment dans n'importe quel ordre, les entités suivantes : 5 *Employés*, 2 *Chefs* et 1 *Directeur*, en vous servant comme clé unique du nom ou du prénom de chaque personne. Opérer un premier affichage à l'aide de la méthode ***GetKey***, puis un second en parcourant le liste avec un ***foreach*** à l'aide la classe ***DictionaryEntry***.

Exercice GENERIQUES :

Refaire les **3 exercices précédents** (sur les listes) en proposant une version utilisant les génériques (cf. chapitre II).