

Conception Objet et Programmation en *C#*

Rémy Malgouyres
LIMOS UMR 6158, IUT, département info
Clermont Université
B.P. 86
63172 AUBIERE cedex
[http ://www.malgouyres.org/](http://www.malgouyres.org/)

Table des matières

1	ABC de la conception et programmation objet	3
1.1	Conception du Package Métier	3
1.1.1	Organisation d'une Application en packages et/ou namespace	3
1.1.2	Dialogue avec l'Expert Métier	3
1.1.3	Déterminer les acteurs et actions simples	4
1.1.4	Première ébauche du package Métier	4
1.2	Créer une classe simple	5
1.2.1	Modélisation Statique (diagramme de classes)	5
1.2.2	Code Source	5
1.2.3	Classe de test	6
1.3	Relation de Composition	8
1.3.1	Modélisation Statique	8
1.3.2	Code source	8
1.3.3	Classe de tests	12
1.4	Pattern <i>Strategy</i>	12
1.4.1	Modélisation statique	13
1.4.2	Code source	13
1.4.3	Classe de Test	15
1.5	Diagrammes de Séquence	17
1.5.1	Notion de diagramme de séquence et exemple simple	17
1.5.2	Exemple de diagramme de séquence pour <i>UrBookLTD</i>	20
2	Delegates et Expressions Lambda	22
2.1	Fonctions de comparaison basiques	22
2.2	Uniformisation de la signature des fonctions	25
2.3	Types <code>delegate</code>	27
2.4	Exemple d'utilisation : méthode de tri générique	30
2.5	Expressions lambda	31
3	Collections	33
3.1	Listes	34
3.2	Files	35
3.3	Dictionnaires	36
3.4	Protocole de comparaison	38
3.5	Protocole d'égalité	39

4	Requêtes <i>LINQ</i>	42
4.1	Types délégués anonymes	42
4.2	Méthodes d'extension	43
4.3	Interface <code>IEnumerable<T></code>	44
4.4	Quelques méthodes de <i>LINQ</i>	45
4.4.1	Filtre <code>Where</code>	45
4.4.2	Méthode de tri <code>OrderBy</code>	45
4.4.3	Filtre <code>Where</code> sur un dictionnaire	46
4.4.4	Méthode de regroupement <code>GroupBy</code>	46
4.4.5	Exemples	47

Chapitre 1

ABC de la conception et programmation objet

1.1 Conception du Package Métier

1.1.1 Organisation d'une Application en packages et/ou namespace

Une application va regrouper plusieurs unités logiques, qui chacune comportent une ou (généralement) plusieurs classes. Une petite application peut comporter, par exemple :

- Le package *Métier* (qui dépend de l'activité du "client" pour qui on développe l'application);
- Un ou plusieurs packages de mise en forme dans une Interface Homme Machine (*IHM*);
- Un package de persistance (permet de gérer l'enregistrement des données de manière permanente, par exemple dans une base de données);
- etc.

Nous développons dans la suite l'exemple d'une application pour un grossiste de livres.

1.1.2 Dialogue avec l'Expert Métier

Supposons que nous développons une application de gestion pour un grossiste de livres *Ur Books LTD*. Le but ici n'est pas de développer complètement cette application mais de donner les grandes lignes de l'organisation de notre application, de la démarche, de la conception et des outils de programmation.

a) Description du métier par l'expert

Nous interrogeons l'*Expert Métier*, c'est à dire la personne de *Ur Books LTD* avec qui nous dialogons pour établir le cahier des charges ou valider le fonctionnement de notre logiciel. Cet expert métier nous dit :

“Nous vendons des **livres**. Un livre possède un ou plusieurs **auteurs** et un **éditeur**, plus éventuellement un numéro d'édition. Il y a deux **formats de livres** : les

formats poches et les formats brochés. Lorsqu'un **client** nous **commande** un (ou plusieurs livres), nous devons, si le livre n'est pas en stock, commander nous même des **exemplaires du livre** chez notre **fournisseur**. Lorsque nous avons reçu tous les livres de la commande du client, nous envoyons les livres à l'**adresse** du client. Le client peut être un **professionnel** (libraire) ou un **particulier**. Si le client est un professionnel, nous lui envoyons régulièrement automatiquement des informations sur les **nouveautés**, qui nous sont données par les éditeurs."

1.1.3 Déterminer les acteurs et actions simples

Toujours dans un dialogue avec l'expert métier et notre client, nous proposons ici une interface de type "console". L'interface comporte différentes vues :

1. La vue client, correspondant à l'acteur *Client*. L'accueil propose les choix suivants :
 - S'enregistrer ou modifier ses données personnelles (saisir son nom et ses coordonnées)
 - Afficher la liste des livres du catalogue (avec pagination) ;
 - Commander un livre en saisissant sa référence.
2. La vue du Back Office, correspondant à l'acteur *Manager*. L'accueil propose les choix suivants :
 - Consulter la liste des commandes et la liste des commandes en attente ;
 - Voir l'état d'une commande et gérer une commande en saisissant son numéro de commande.

1.1.4 Première ébauche du package Métier

Nous proposons de faire un package Métier qui contient :

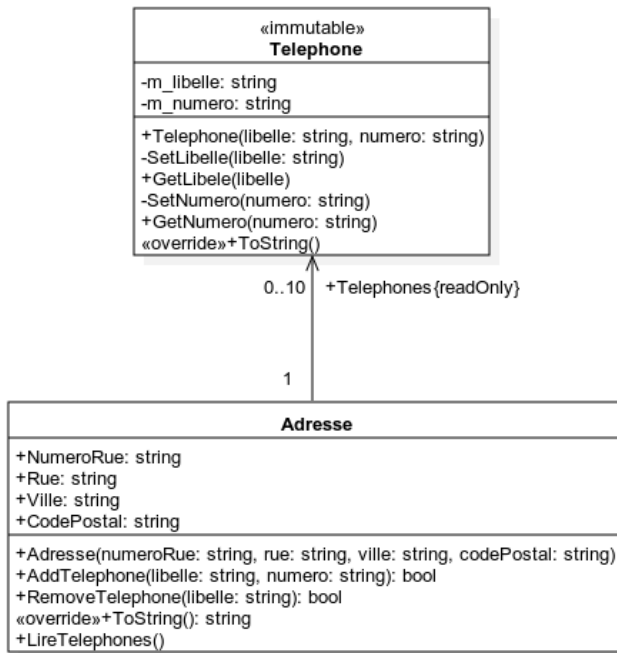
- Une classe Livre ;
- Une classe Editeur ;
- Une classe Fournisseur ;
- Une classe Client ;
- Une classe Adresse pour les adresses des clients, des éditeurs et des fournisseurs ;
- Une classe Commande.

Ces classes sont destinées à représenter les entités manipulées dans le métier de notre client. On commence par mettre en évidence les plus grosses classes. Nous pourrons toujours rajouter des classes plus tard.

1.2 Créer une classe simple

1.2.1 Modélisation Statique (diagramme de classes)

Voyons tout d'abord comment créer une classe pour représenter un numéro de téléphone. Nous verrons ensuite comment les information concernant l'adresse d'une personne ou d'une institution (entreprise ou administration) pourra "contenir" des numéros de téléphone.



Diag 1. Diagramme de Classes avec relation sous forme d'attribut.

La classe `Telephone` est immuable. Cela signifie qu'une fois l'instance du téléphone créée, on ne peut plus la modifier. Le seul moyen de changer le numéro de téléphone serait de détruire (supprimer toute référence à) l'instance de la classe `Telephone` correspondante, et de construire une nouvelle instance.

Pour cela, comme tous les attributs sont eux même immuables, il suffit d'interdire l'écriture sur les attributs (attributs et leurs *Setters* privés).

Dans les *setters*, nous réaliserons des tests sur la forme des chaînes de caractères pour assurer la cohérence des données (et la sécurité de l'application). On utilise pour cela des expressions régulières (*Regex*).

Dans la classe `Adresse`, nous avons cette fois défini des *propriétés*, ce qui permet d'écrire les *Getters* et les *Setters* de manière plus compacte. La propriété `Telephones`, qui est de type `Array` de `Telephone`, est en lecture seule.



Cela ne signifie pas qu'une autre classe ne puisse pas modifier les numéros de téléphone. En effet, l'accès à la référence de l'`Array` permet de modifier les éléments de cet `Array`, et donc par exemple d'ajouter ou supprimer un numéro de téléphone. On voit donc que les *Getters* peuvent parfois permettre de modifier les objets référencés. Ça ne serait pas le cas avec une structure, qui serait intégralement recopiée lors du *return* du *Getter*, interdisant la modification de ses données internes.

1.2.2 Code Source

UrBooksLTD/Metier/Telephone.cs

```

1
2 using System;
3 using System.Text;
4 using System.Text.RegularExpressions;
5
6 namespace Metier
7 {
8     public class Telephone
9     {
    
```

```
10     private string m_libelle;
11     private string m_numero;
12
13     public string GetLibelle()
14     {
15         return m_libelle;
16     }
17
18     public string GetNumero()
19     {
20         return m_numero;
21     }
22
23     private void SetLibelle(string libelle)
24     {
25         Regex myRegex = new Regex("^[a-zA-Z]{4,16}$");
26         if (myRegex.IsMatch(libelle))
27         {
28             m_libelle = libelle;
29         }
30         else
31         {
32             throw new ArgumentException("Le libellé est invalide");
33         }
34     }
35
36     private void SetNumero(string numero)
37     {
38         Regex myRegex = new Regex("[0-9]{10}$");
39         if (myRegex.IsMatch(numero))
40         {
41             m_numero = numero;
42         }
43         else
44         {
45             throw new ArgumentException("Le numéro est invalide");
46         }
47     }
48
49     public Telephone(string libelle, string numero)
50     {
51         SetLibelle(libelle);
52         SetNumero(numero);
53     }
54
55     public override string ToString()
56     {
57         StringBuilder retour = new StringBuilder();
58         retour.Append(m_libelle + " : ");
59         retour.AppendLine(" " + m_numero);
60         return retour.ToString();
61     }
62 }
63 }
```

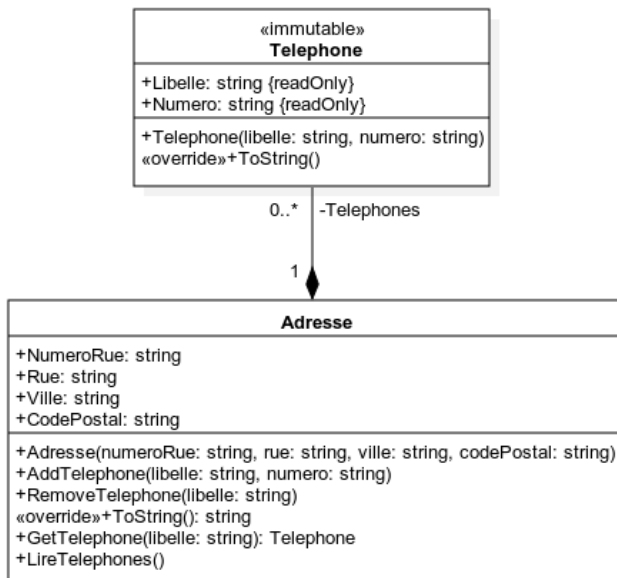
1.2.3 Classe de test

UrBooksLTD/UrBooksLTD/TestMetierTelephone.cs

```
1 |
2 | using System;
3 | using System.Collections.Generic;
4 | using System.Linq;
5 | using System.Text;
6 | using System.Threading.Tasks;
7 |
8 | using Metier;
9 |
10 | namespace UrBooksLTD
11 | {
12 |     class TestMetierTelephone : ITestInterface
13 |     {
14 |         public void Test()
15 |         {
16 |             Console.Write("Merci de saisir le libelle : ");
17 |             string libelle = Console.ReadLine();
18 |             Console.Write("Merci de saisir le numero : ");
19 |             string numero = Console.ReadLine();
20 |
21 |             try
22 |             {
23 |                 Telephone tel = new Telephone(libelle , numero);
24 |                 Console.Write(tel);
25 |             }
26 |             catch (Exception e)
27 |             {
28 |                 Console.WriteLine("Erreur : " + e.Message + "\n" + e.StackTrace)
29 |                 ;
30 |             }
31 |         }
32 |     }
```


1.3 Relation de Composition

1.3.1 Modélisation Statique



Diag 2. Diagramme de Classes avec relation sous forme de Composition



Notons enfin que nous avons défini la propriété *Telephones* de la classe adresse en privé, interdisant ainsi aux autres classes d'accéder à la collection des téléphones. **Cela garantit que la caractéristique de la composition que les téléphones ne sont pas partagés ni modifiables par d'autres instance** que leur instance unique d'adresse. En contrepartie, on n'accède plus aux téléphones que sous forme de `string` pour affichage.

Une autre représentation (pas tout à fait équivalente!) de la relation entre la classe *Adresse* et la classe *Telephone* est la *Composition*. On voit sur ce schéma, par le losange plein représentant une *composition* :

- que la classe *Adresse* va **contenir un ensemble de Telephone**.
- De plus, **les cycles de vie sont liés**. Si une instance de la classe *Adresse* est détruite, toutes les instances de la classe *Telephone* correspondantes seront aussi détruite.
- En particulier, une instance de *Telephone* ne peut pas être partagée par plusieurs instances de la classe *Adresse*.

Notons en outre que nous avons représenté les données de la classe téléphone sous forme de propriétés en lecture seule (typique de C#).

1.3.2 Code source

Voici le code C# de la classe *Adresse* :

UrBooksLTD/Metier/Adresse.cs

```

1 |
2 | using System;
3 | using System.Text;
4 | using System.Text.RegularExpressions;
5 |
6 | namespace Metier
7 | {
8 |     public class Adresse
9 |     {
10 |         public Adresse(string numeroRue, string rue, string codePostal, string
11 |             ville)
  
```

```

12     NumeroRue = numeroRue;
13     Rue = rue;
14     CodePostal = codePostal;
15     Ville = ville;
16     Telephones = new Telephone[MAX_TELEPHONES];
17 }
18
19 private static void checkRegularExp(string pattern, string chaine,
20     string errorMsg)
21 {
22     Regex myRegex = new Regex(pattern);
23     if (!myRegex.IsMatch(chaine))
24     {
25         throw new ArgumentException("Erreur : " + errorMsg);
26     }
27 }
28 private string m_numeroRue;
29 public string NumeroRue
30 {
31     get { return m_numeroRue; }
32     set
33     {
34         checkRegularExp(@"^[a-zA-Z0-9|-\ ]{0,16}$", value, "Numéro de
35             rue invalide");
36         m_numeroRue = value;
37     }
38 }
39 private string m_rue;
40 public string Rue
41 {
42     get { return m_rue; }
43     set
44     {
45         checkRegularExp(@"^[w|s|-\ ]+$", value, "Nom de rue/place
46             invalide");
47         m_rue = value;
48     }
49 }
50 private string m_ville;
51 public string Ville
52 {
53     get { return m_ville; }
54     set
55     {
56         checkRegularExp(@"^[w|s|-\ ]+$", value, "Nom de ville invalide"
57             );
58         m_ville = value;
59     }
60 }
61 private string m_codePostal;
62 public string CodePostal
63 {
64     get { return m_codePostal; }

```

```
64         set
65         {
66             Regex myRegex = new Regex("^[0-9]{5}$");
67             if (!myRegex.IsMatch(value))
68             {
69                 throw new ArgumentException("Code Postal invalide");
70             }
71             m_codePostal = value;
72         }
73     }
74
75     // Nombre Maximum de Numéros de Téléphones
76     private static readonly int MAX_TELEPHONES = 10;
77
78     // propriété en lecture seule
79     private Telephone[] m_telephones;
80     public Telephone[] Telephones
81     {
82         get
83         {
84             return m_telephones;
85         }
86
87         // setter privé !!!
88         private set
89         {
90             m_telephones = value;
91         }
92     }
93
94
95     /// <summary>
96     /// Ajoute un téléphone
97     /// </summary>
98     /// <returns>renvoie true si il reste de la place</returns>
99     public bool AddTelephone(string libelle , string numero)
100    {
101        int indexAjout = Array.IndexOf(Telephones , null);
102        if (indexAjout < MAX_TELEPHONES){
103            Telephones[indexAjout] = new Telephone(libelle , numero);
104            return true;
105        }else{
106            return false;
107        }
108    }
109
110    public bool RemoveTelephone(string libelle)
111    {
112        // On cherche le téléphone avec ce libelle
113        foreach (Telephone tel in Telephones)
114        {
115            if (tel.GetLibelle() == libelle)
116            {
117                // On crée un trou dans le tableau
118                Telephones [Array.IndexOf(Telephones , tel)] = null;
119                return true;

```

```

120         }
121     }
122     return false;
123 }
124
125 public override string ToString()
126 {
127     StringBuilder retour = new StringBuilder();
128     retour.AppendFormat("{0}, {1},\n{2} {3}\nTéléphone(s) :|n",
129         NumeroRue, Rue, CodePostal, Ville);
129     foreach (Telephone tel in Telephones)
130     {
131         if (tel != null)
132             retour.Append(tel);
133     }
134     return retour.ToString();
135 }
136 }
137 }

```

```

1 public class Telephone
2 {
3     private string m_libelle;
4     public string Libelle {
5         get {
6             return m_libelle;
7         }
8
9         private set {
10            Regex myRegex = new Regex("^[a-zA-Z]{4,16}$");
11            if (myRegex.IsMatch(libelle))
12            {
13                m_libelle = libelle;
14            }
15            else
16            {
17                throw new ArgumentException("Le libellé est invalide");
18            }
19        }
20
21        private string m_numero;
22        public string Numero {
23            get {
24                return m_numero;
25            }
26
27            private set {
28                Regex myRegex = new Regex("^[0-9]{10}$");
29                if (myRegex.IsMatch(numero))
30                {
31                    m_numero = numero;
32                }
33                else
34                {
35                    throw new ArgumentException("Le numéro est invalide");

```

```
36     }
37   }
38 }
39 }
```

1.3.3 Classe de tests

Voici la classe de tests correspondant à la classe `Adresse` :

UrBooksLTD/UrBooksLTD/TestMetierAdresse.cs

```
1
2 using System;
3 using System.Text;
4
5 using Metier;
6
7 namespace UrBooksLTD
8 {
9     class TestMetierAdresse : ITestInterface
10    {
11        public void Test()
12        {
13            try
14            {
15                Adresse adresse = new Adresse("12 bis", "Place de la Bourse", "
16                    63000", "Clermont-Ferrand");
17                adresse.AddTelephone("fixe", "0123456789");
18                adresse.AddTelephone("asupprimer", "9876543210");
19                adresse.AddTelephone("mobile", "0623456789");
20                adresse.RemoveTelephone("asupprimer");
21                Console.WriteLine(adresse);
22            }
23            catch (Exception e)
24            {
25                Console.WriteLine("Erreur : " + e.Message + "\n" + e.StackTrace)
26                ;
27            }
28        }
29    }
30 }
```

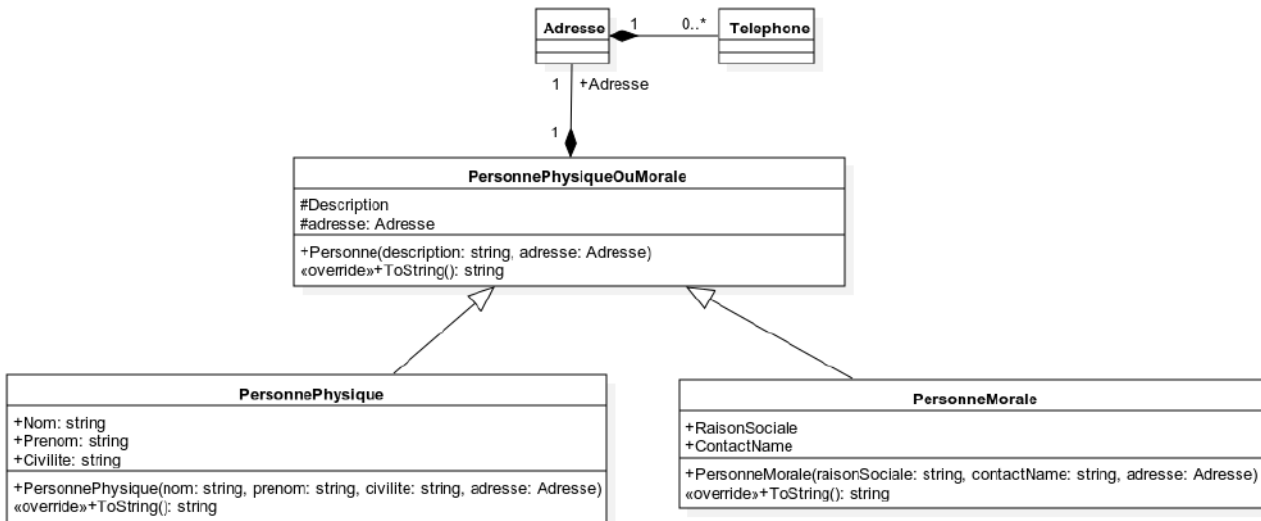
1.4 Pattern *Strategy*

Comme nous l'avons vu dans la partie 1.1.4, nous devons représenter des clients, des fournisseurs et des éditeurs, qui peuvent être, selon les cas, des personnes morales (entreprises) ou des personnes physiques (particuliers). Les informations sur une personne morale (raison sociale, personne à contacter, etc.) ne sont pas forcément les mêmes que les informations sur une personne physique (nom, prénom, civilité, etc.). On souhaite cependant *factoriser* les données et opérations communes aux deux types de personnes (comme les données et opérations liées à l'adresse de la personne/entreprise).

1.4.1 Modélisation statique

On se propose de créer trois classes, une classe `PersonnePhysiqueOuMorale`, générique, qui factorisera les traits communs à tous les types de personnes, une classe `PersonnePhysique`, et une classe `PersonneMorale`.

Ces classes sont liées par des relations d'*héritage*, aussi appelées des relations de *généralisation* (ou de *spécialisation*, selon le point de vue). Ainsi, une `PersonnePhysique` est un cas particulier de `PersonnePhysiqueOuMorale`. Une `PersonneMorale` est aussi un cas particulier de `PersonnePhysiqueOuMorale`.



Diag 3. e pattern *Strategy* pour les types de personnes.

1.4.2 Code source

Voici le code de la classe `PersonnePhysiqueOuMorale`, suivi du code de la classe `PersonneMorale` et enfin de la classe `PersonnePhysique`.

Notez le **chaînage des constructeur** par lequel le constructeur d'une classe dérivée (par exemple le constructeur de `PersonnePhysique` fait appel (avant même le corps du constructeur) au constructeur de la classe de base `PersonnePhysiqueOuMorale` pour initialiser les propriétés de la classe de base.

UrBooksLTD/Metier/PersonnePhysiqueOuMorale.cs

```

1 |
2 | using System;
3 | using System.Text;
4 |
5 | namespace Metier
6 | {
7 |     public class PersonnePhysiqueOuMorale
8 |     {
9 |         protected string Description { get; set; } // Ajouter des tests par
          |         Regex si besoin
10 |
11 |         protected Adresse m_adresse;
12 |
13 |         public PersonnePhysiqueOuMorale(string description, Adresse adresse)
    
```

```

14     {
15         Description = description;
16         m_adresse = adresse;
17     }
18
19     public override string ToString()
20     {
21         return "Personne physique ou personne morale :\n"
22             + Description + ",\n" + m_adresse;
23     }
24 }
25 }

```

UrBooksLTD/Metier/PersonneMorale.cs

```

1
2 using System;
3 using System.Text;
4
5 namespace Metier
6 {
7     public class PersonneMorale : PersonnePhysiqueOuMorale
8     {
9         public string RaisonSociale { get; set; } // Ajouter des tests par Regex
10            si besoin
11
12         public string ContactName { get; set; } // Ajouter des tests par Regex
13            si besoin
14
15         public PersonneMorale(string raisonSociale, string contactName, Adresse
16             adresse)
17             // Appel du constructeur de la classe de base
18             // PersonnePhysiqueOuMorale
19             : base(raisonSociale + ", contacter " + contactName, adresse)
20         {
21             RaisonSociale = raisonSociale;
22             ContactName = contactName;
23         }
24
25         public override string ToString()
26         {
27             return "Personne Morale :\n"
28                 + RaisonSociale + "\nà l'attention de " + ContactName + ",\n"
29                 + m_adresse;
30         }
31     }
32 }

```

UrBooksLTD/Metier/PersonnePhysique.cs

```

1
2 using System;
3 using System.Text;
4
5 namespace Metier
6 {
7     public class PersonnePhysique : PersonnePhysiqueOuMorale

```

```

8      {
9      public enum Civilite {MR, MME, MLE}
10
11     private static string GetCiviliteString(Civilite civilite)
12     {
13         switch(civilite){
14             case Civilite.MLE: return "Mademoiselle";
15             case Civilite.MME: return "Madame";
16             case Civilite.MR: return "Monsieur";
17
18             default : return "";
19         }
20     }
21
22     public string Nom { get; set; } // Ajouter des tests par Regex si besoin
23     public string Prenom { get; set; } // Ajouter des tests par Regex si
        besoin
24     public Civilite m_civilite;
25
26     public PersonnePhysique(string nom, string prenom, Civilite civilite ,
        Adresse adresse)
27         // Appel du constructeur de la classe de base
        PersonnePhysiqueOuMorale
28         : base(GetCiviliteString(civilite) + " " + prenom + " " + nom,
        adresse)
29     {
30
31         Nom = nom;
32         Prenom = prenom;
33         m_civilite = civilite;
34     }
35
36     public string GetCivilite()
37     {
38         return GetCiviliteString(m_civilite);
39     }
40
41     public override string ToString()
42     {
43         return "Personne Physique \n" + base.Description;
44     }
45 }
46 }

```

1.4.3 Classe de Test

Voici la classe de tests correspondant à l'ensemble du pattern *Strategy* pour représenter les personnes :

UrBooksLTD/UrBooksLTD/TestMetierPersonne.cs

```

1
2 using System;
3 using System.Text;
4
5 using Metier;

```



```
6
7 namespace UrBooksLTD
8 {
9     class TestMetierPersonne : ITestInterface
10    {
11        public void Test()
12        {
13            try
14            {
15                Adresse adresse = new Adresse("12 bis", "Place de la Bourse", "
16                    63000", "Clermont-Ferrand");
17
18                // Test de personne morale
19                PersonnePhysiqueOuMorale personne = new PersonneMorale("
20                    Librairie Les Bouts qun", "Christine Kafka",
21                    adresse);
22                Console.WriteLine(personne);
23                PersonneMorale personneMorale = (personne as PersonneMorale);
24                if (personneMorale != null)
25                {
26                    Console.WriteLine("Notre contact chez \" + personneMorale.
27                        RaisonSociale
28                        + \"\" est : \" + personneMorale.
29                        ContactName + \"\n");
30                }
31
32                // Test de personne physique
33                personne = new PersonnePhysique("Kafka", "Christine",
34                    PersonnePhysique.Civilite.MME, adresse);
35                Console.WriteLine(personne);
36
37                PersonnePhysique personnePhysique = (personne as
38                    PersonnePhysique);
39                if (personnePhysique != null)
40                {
41                    Console.WriteLine("\nOn s'adresse à \" + personnePhysique.
42                        Prenom
43                        + \" en disant : \" + personnePhysique.
44                        GetCivilite());
45                }
46            }
47        }
48    }
49 }
```

1.5 Diagrammes de Séquence

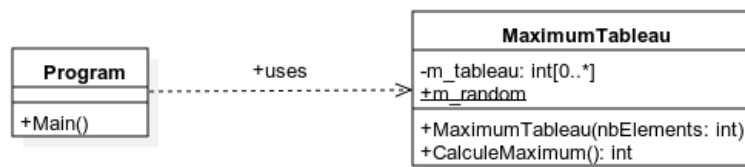
1.5.1 Notion de diagramme de séquence et exemple simple

Les diagrammes de séquences permettent d'avoir une vue dynamique des interactions entre classes et/ou des algorithmes. Généralement, lorsqu'un acteur lance une action (comme un *click* de bouton) dans l'application, un certain nombre de classes interviennent dans la mise en oeuvre de l'action par le programme. Les méthodes de ces classes s'appellent les unes les autres au cours du temps. De plus, à l'intérieur de ces méthodes, des boucles (des *loop* correspondant à des *while* ou *for*), des branchements conditionnels (*opt* pour un *if* ou *alt* pour un *if...else* ou un *switch*) implémentent des algorithmes. En bref, les diagrammes de séquences schématisent l'*algorithmique inter classes* et parfois l'*algorithmique intra classe*.

Nous voyons ici un exemple simple. Une classe `MaximumTableau`, possède un constructeur qui crée un tableau aléatoire d'entiers, dont la référence est mémorisée dans un attribut. Une méthode `CalculeMaximum` retourne la plus grande des valeurs contenues dans le tableau.

Le programme principal crée une instance de `MaximumTableau`, appelle la méthode de `MaximumTableau` qui calcule le maximum, et affiche le résultat.

Voici tout d'abord le diagramme de conception statique (diagramme de classes) :



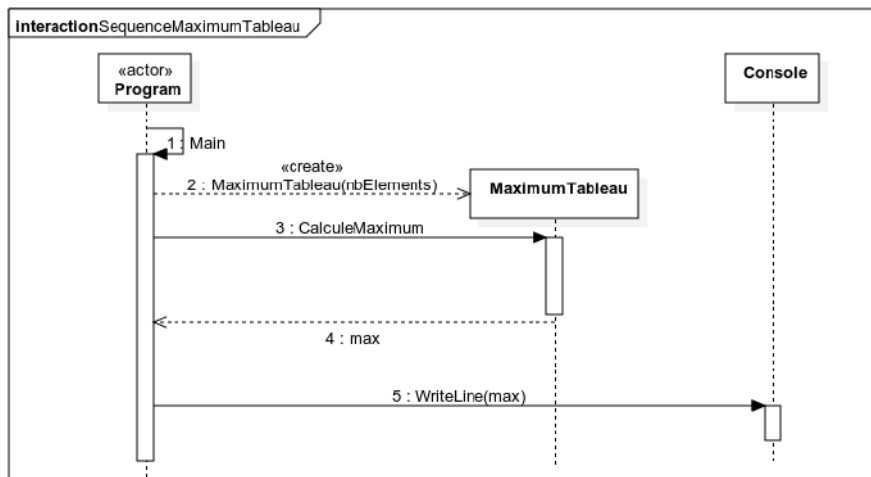
Diag 4. Diagramme de Classes pour le programme créant un tableau aléatoire et calculant son maximum.

Voici maintenant le diagramme de séquence de l'algorithme grossier (inter classes) pour le même programme, qui schématise les appels de méthodes entre la classe `Program` et la classe `MaximumTableau`.

Dans ce diagramme, l'axe vertical représente le temps. Les lignes verticales représentent des cycles de vies de classes ou d'instances de classes.

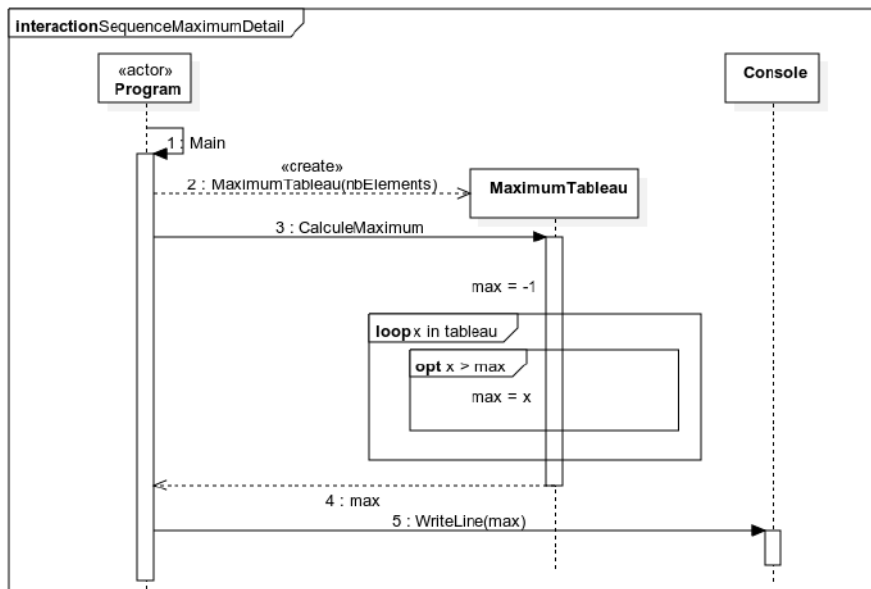
La flèche avec le stéréotype *create* est un appel du constructeur de `MaximumTableau`, lequel génère le tableau aléatoire. La flèche continue est un appel de la méthode `CalculeMaximum` dans la fonction `Program.Main`.

Les rectangles verticaux représentent des méthodes (appelés *messages*) et les flèches en pointillés qui en sont issues des valeurs retournées par ces méthodes (appelés *messages de retour*).



Diag 5. Diagramme de S quences “vu de loin” (algorithmique inter classe) pour le programme cr ant un tableau al atoire et calculant son maximum.

Voici enfin l’algorithme d taill  pour le m me programme, qui inclut l’algorithme de calcul du maximum   l’int rieur de la m thode `MaximumTableau.CalculeMaximum`



Diag 6. Diagramme de S quences “vu de pr s” (incluant l’algorithme intra classe) pour le programme cr ant un tableau al atoire et calculant son maximum.

Voici maintenant le code source C# de la classe `MaximumTableau`, puis de la classe `Program`.

ExemplesDiagSeq/ExemplesDiagSeq/Program.cs

```

1 |
2 | using System;
3 | using System.Text;
4 |
5 | namespace ExemplesDiagSeq
  
```

```

6 {
7   class Program
8   {
9       static void Main(string[] args)
10      {
11          MaximumTableau myTableau = new MaximumTableau(50);
12          int max = myTableau.CalculeMaximum();
13          Console.WriteLine("Maximum du Tableau : {0}", max);
14          Console.ReadKey();
15      }
16  }
17 }

```

ExemplesDiagSeq/ExemplesDiagSeq/MaximumTableau.cs

```

1
2 using System;
3 using System.Text;
4
5 namespace ExemplesDiagSeq
6 {
7     class MaximumTableau
8     {
9         // Générateur aléatoire, OBLIGATOIREMENT statique,
10        // Initialisé au début du programme par le constructeur
11        private static Random random = new Random();
12
13        private int[] m_tableau;
14
15        public MaximumTableau(int nbElements)
16        {
17            m_tableau = new int[nbElements];
18            for (int i = 0; i < nbElements; i++)
19            {
20                m_tableau[i] = random.Next(0, 100);
21            }
22        }
23
24        public int CalculeMaximum()
25        {
26            int max = -1;
27            foreach (int x in m_tableau)
28            {
29                if (x > max)
30                {
31                    max = x;
32                }
33            }
34            return max;
35        }
36    }
37 }
38 }

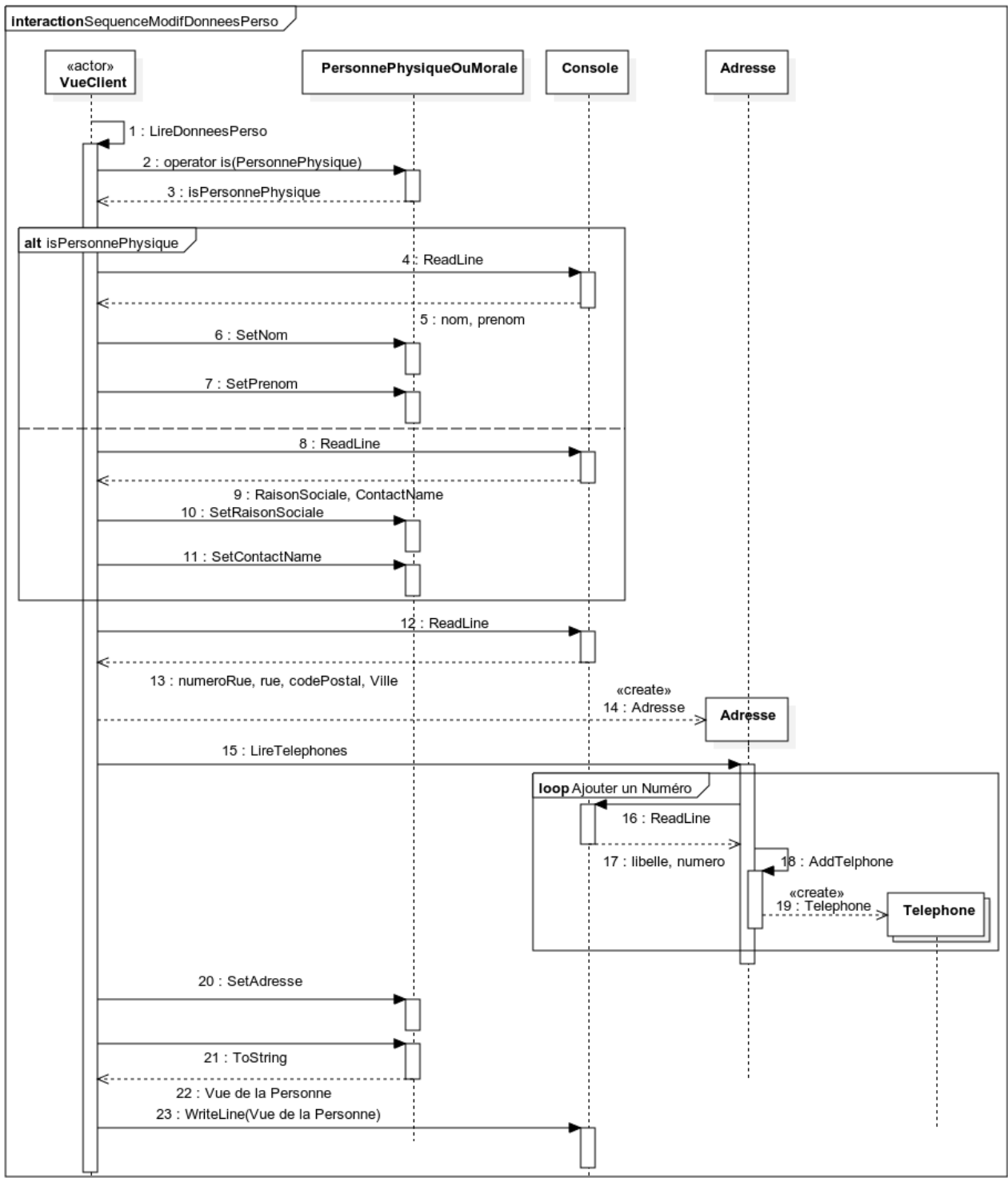
```

1.5.2 Exemple de diagramme de séquence pour *UrBookLTD*

Un diagramme de séquence correspond à une action (avec éventuellement des hypothèses sur les branchements). Recenser les acteurs et les actions fait partie de l'*analyse fonctionnelle de l'application*.

Nous avons recensé dans la partie 1.1.3 les principales actions pour l'acteur *Client* et pour l'acteur *Manager*. Voyons un exemple de diagramme de séquence pour l'action *modifier ses données personnelles* d'un *Client*.

Deux cas sont possibles : le client est une personne physique ou une personne morale. On fait saisir les données au client, y compris concernant son adresse et son numéro de téléphone. On affiche enfin le résultat.



Diag 7. Diagramme de Séquences de la saisie des données personnelles par une personne (physique ou morale).

Chapitre 2

Delegates et Expressions Lambda

2.1 Fonctions de comparaison basiques

Considérons trois types qui admettent la comparaison (un ordre) entre les éléments :

- Le type `int` avec la relations d'ordre usuelle sur les nombres entiers ;
- Le type `string` avec l'ordre alphabétique ;
- Le type `MyDate` défini ci-dessous avec l'ordre chronologique sur les dates.

Voici la définition de la classe `MyDate` :

CoursLinq/ExemplesDelegate/ex02_MyDate.cs

```
1
2 using System;
3 using System.Collections.Generic;
4 using System.Linq;
5 using System.Text;
6 using System.Threading.Tasks;
7 using System.Text.RegularExpressions;
8
9 namespace ExemplesDelegate
10 {
11     public partial class MyDate
12     {
13         public int Annee {get; private set;}
14         public int Mois {get; private set;}
15         public int Jour { get; private set; }
16
17         public MyDate(string dateJJMMAAA)
18         {
19             string[] elementsDate = dateJJMMAAA.Split(new[] { '/' });
20             Annee = int.Parse(elementsDate[2]);
21             Mois = int.Parse(elementsDate[1]);
22             Jour = int.Parse(elementsDate[0]);
23         }
24
25         public override string ToString()
26         {
27             return Jour.ToString().PadLeft(2, '0')
```

```

28         + "/" + Mois.ToString().PadLeft(2, '0') + "/"
29         + Annee.ToString().PadLeft(2, '0');
30     }
31
32     public string ToStringReverse()
33     {
34         return Annee.ToString().PadLeft(2, '0')
35             + "/" + Mois.ToString().PadLeft(2, '0') + "/"
36             + Jour.ToString().PadLeft(2, '0');
37     }
38 }
39 }

```

Nous pouvons définir trois fonctions différentes pour coparer respectivement ces trois types de données. Le fonctionnement est à peu près le même, chaque méthode renvoie une valeur négative, nulle ou positive suivant les cas sur l'ordre ou l'égalité des deux paramètres (comme la fonction `strcmp` pour le langage *C*).

CoursLinq/ExemplesDelgate/ex01_ExCompareBasic.cs

```

1
2 using System;
3 using System.Collections.Generic;
4 using System.Linq;
5 using System.Text;
6 using System.Threading.Tasks;
7
8 namespace ExemplesDelegate
9 {
10     public class ExCompareBasic
11     {
12         // Fonction de comparaison de deux entiers
13         // Retourne 0 si les a==b, positif si a<b, négatif si b<a
14         public static int? CompareInt(int? a, int? b)
15         {
16             return a - b;
17         }
18
19         // Fonction de comparaison de deux chaîne par ordre alphabétique
20         // Retourne 0 si les a==b, positif si a<b, négatif si b<a
21         public static int CompareString(string a, string b)
22         {
23             return String.Compare(a, b);
24         }
25
26         // Fonction de comparaison de deux dates par ordre alphabétique
27         // Retourne 0 si les a==b, positif si a<b, négatif si b<a
28         public static int CompareDate(MyDate a, MyDate b)
29         {
30             return String.Compare(a.ToStringReverse(), b.ToStringReverse());
31         }
32     }
33 }

```

Ces méthodes s'utilisent comme dans les méthodes de test suivantes :

CoursLinq/ExemplesDelgate/ex03_TestCompareBasic.cs


```
1
2 using System;
3 using System.Collections.Generic;
4 using System.Linq;
5 using System.Text;
6 using System.Threading.Tasks;
7
8 namespace ExemplesDelegate
9 {
10     public class TestCompareBasic
11     {
12         // Test de la comparaison de deux entiers
13         public static void TestCompareInt()
14         {
15             Console.Write("Merci d'entrer deux entiers s par s pas un / : ");
16             string[] chainesAB = Console.ReadLine().Split(new[] { '/' });
17             int a = int.Parse(chainesAB[0]);
18             int b = int.Parse(chainesAB[1]);
19             if (ExCompareBasic.CompareInt(a, b) == 0)
20                 Console.WriteLine("{0}  gale {1}", a, b);
21             else
22                 if (ExCompareBasic.CompareInt(a, b) < 0)
23                     Console.WriteLine("{0} inf rieur   {1}", a, b);
24                 else
25                     Console.WriteLine("{0} sup rieur   {1}", a, b);
26         }
27
28         // Test de la comparaison alphab tique de deux cha nes
29         public static void TestCompareString()
30         {
31             Console.Write("Merci d'entrer deux cha nes alphab tiques s par es
32                 pas un / : ");
33             string[] chainesAB = Console.ReadLine().Split(new[] { '/' });
34             string a = chainesAB[0];
35             string b = chainesAB[1];
36             if (ExCompareBasic.CompareString(a, b) == 0)
37                 Console.WriteLine("{0}  gale {1}", a, b);
38             else
39                 if (ExCompareBasic.CompareString(a, b) < 0)
40                     Console.WriteLine("{0} inf rieur   {1}", a, b);
41                 else
42                     Console.WriteLine("{0} sup rieur   {1}", a, b);
43         }
44
45         // Test de la comparaison chronologique de deux dates
46         public static void TestCompareDate()
47         {
48             Console.Write("Merci d'entrer une date au format jj/mm/aaaa : ");
49             MyDate a = new MyDate(Console.ReadLine());
50             Console.Write("Merci d'entrer une autre date au format jj/mm/aaaa :
51                 ");
52             MyDate b = new MyDate(Console.ReadLine());
53
54             if (ExCompareBasic.CompareDate(a, b) == 0)
55                 Console.WriteLine("{0}  gale {1}", a, b);
56             else
```

```

55         if (ExCompareBasic.CompareDate(a, b) < 0)
56             Console.WriteLine("{0} inférieur à {1}", a, b);
57         else
58             Console.WriteLine("{0} supérieur à {1}", a, b);
59     }
60
61     // Méthode qui exécute les tests des trois méthodes de comparaison
62     public static void TestAllCompareMethods()
63     {
64         TestCompareInt();
65         TestCompareString();
66         TestCompareDate();
67     }
68 }
69 }

```

2.2 Uniformisation de la signature des fonctions

Pour pouvoir utiliser la comparaison sur des types génériques (par exemple, comme nous le verrons ci-dessous, pour écrire des algorithmes de tris qui fonctionneront indépendamment du type), nous allons créer des méthodes de comparaison dont la signature est générique. Autrement dit, **la signature des trois méthodes de comparaison est la même, ainsi que la signature des trois méthodes de lecture dans la console.**

On s'appuie pour cela sur la classe `Objet`, qui est ancêtre de toutes les classes en `C#`. Dans la définition des méthodes, on réalise un *downcasting* pour utiliser la comparaison sur des types plus spécialisés que la classe `Objet` (ici les types `int`, `string` et `MyDate`).

CoursLinq/ExemplesDelgate/ex04_ExUniformSignature.cs

```

1
2 using System;
3 using System.Collections.Generic;
4 using System.Linq;
5 using System.Text;
6 using System.Threading.Tasks;
7
8 namespace ExemplesDelegate
9 {
10     public class ExUniformSignature
11     {
12         public static Object ReadInt()
13         {
14             Console.Write("Merci d'entrer un nombre entier : ");
15             int a = int.Parse(Console.ReadLine());
16             return a;
17         }
18
19         public static Object ReadString()
20         {
21             Console.Write("Merci d'entrer une chaîne alphabétique : ");
22             string a = Console.ReadLine();
23             return a;
24         }
25     }

```

```

26     public static Object ReadDate()
27     {
28         Console.WriteLine("Merci d'entrer une date au format jj/mm/aaaa : ");
29         MyDate a = new MyDate(Console.ReadLine());
30         return a;
31     }
32
33     // Fonction de comparaison de deux entiers
34     // Retourne 0 si les a==b, positif si a<b, n gatif si b<a
35     public static int CompareInt(Object a, Object b)
36     {
37         int? aa = a as int?;
38         int? bb = b as int?;
39         return aa == bb ? 0 : (aa < bb ? -1 : 1);
40     }
41
42     // Fonction de comparaison de deux cha ne par ordre alphab tique
43     // Retourne 0 si les a==b, positif si a<b, n gatif si b<a
44     public static int CompareString(Object a, Object b)
45     {
46         string aa = a as string;
47         string bb = b as string;
48         return String.Compare(aa, bb);
49     }
50
51     // Fonction de comparaison de deux dates par ordre chronologique
52     // Retourne 0 si les a==b, positif si a<b, n gatif si b<a
53     public static int CompareDate(Object a, Object b)
54     {
55         MyDate aa = a as MyDate;
56         MyDate bb = b as MyDate;
57         return String.Compare(aa.ToStringReverse(), bb.ToStringReverse());
58     }
59 }
60 }

```

Dans les trois fonctions de test suivantes, on constate que le code est exactement le m me, sauf le nom des m thodes et les downcasts. Comme nous allons le voir dans la partie suivante, on peut en fait **factoriser ce code** en passant les fonctions de lecture dans la console et de comparaison en param tre.

CoursLinq/ExemplesDelegate/ex05_TestUniformSignature.cs

```

1
2 using System;
3 using System.Collections.Generic;
4 using System.Linq;
5 using System.Text;
6 using System.Threading.Tasks;
7
8 namespace ExemplesDelegate
9 {
10     public class TestUniformSignature
11     {
12         public static void TestCompareInt()
13         {
14             int? a = ExUniformSignature.ReadInt() as int?;

```

```

15     int? b = ExUniformSignature.ReadInt() as int?;
16
17     if (ExUniformSignature.CompareInt(a, b) == 0)
18         Console.WriteLine("{0} égale {1}", a, b);
19     else
20         if (ExUniformSignature.CompareInt(a, b) < 0)
21             Console.WriteLine("{0} inférieur à {1}", a, b);
22         else
23             Console.WriteLine("{0} supérieur à {1}", a, b);
24     }
25
26     public static void TestCompareString()
27     {
28         string a = ExUniformSignature.ReadString() as string;
29         string b = ExUniformSignature.ReadString() as string;
30
31         if (ExUniformSignature.CompareString(a, b) == 0)
32             Console.WriteLine("{0} égale {1}", a, b);
33         else
34             if (ExUniformSignature.CompareString(a, b) < 0)
35                 Console.WriteLine("{0} inférieur à {1}", a, b);
36             else
37                 Console.WriteLine("{0} supérieur à {1}", a, b);
38     }
39
40     public static void TestCompareDate()
41     {
42
43         Object a = ExUniformSignature.ReadDate();
44         Object b = ExUniformSignature.ReadDate();
45
46         if (ExUniformSignature.CompareDate(a, b) == 0)
47             Console.WriteLine("{0} égale {1}", a, b);
48         else
49             if (ExUniformSignature.CompareDate(a, b) < 0)
50                 Console.WriteLine("{0} inférieur à {1}", a, b);
51             else
52                 Console.WriteLine("{0} supérieur à {1}", a, b);
53     }
54
55     // Méthode qui exécute les tests des trois méthodes de comparaison
56     public static void TestAllCompareMethods()
57     {
58         TestCompareInt();
59         TestCompareString();
60         TestCompareDate();
61     }
62 }
63 }

```

2.3 Types delegate

Définition. Un type délégué est un type de donnée pouvant contenir des méthodes C#. Pour un type délégué donné, la signature des méthodes est fixée, dans la définition du type délégué.

Les types délégués sont analogues aux types pointeurs de fonctions en langage C : ils peuvent contenir des fonctions.

Exemple. Voici la déclaration d'un type délégué qui peut être utilisé pour des méthodes de comparaison.

```

1 // Type délégué pour les fonctions de comparaison
2 // signature : int fonction(Object, Object)
3 public delegate int MonTypeFonctionComparaison(Object a, Object b);
4
5 // Ajout d'une éthode dans une instance de délégué (abonnement).
6 // La méthode doit avoir la même signature que le type délégué
7 MonTypeFonctionLecture readFunction = ExUniformSignature.ReadInt;
```

En fait, nous verrons plus loin que les instances de types délégués en C# peuvent contenir plusieurs méthodes de même signature. On parle alors de *délégués multicast*.

Voici maintenant des exemples d'utilisation des délégués pour créer des méthodes qui peuvent s'appliquer à la comparaison d'objets de différents types.

La première méthode, plus simple, montre comment on affecte une méthode (ici `ExUniformSignature.ReadInt` ou `ExUniformSignature.CompareInt`). Dans la deuxième méthodes, les instances de types délégués sont en paramètre de la fonction, ce qui permet, lors de l'appel de la fonction, de passer la méthode correspondant au type (`int`, `string` et `MyDate`) utilisé (voir troisième méthode ci-dessous).

CoursLinq/ExemplesDelgate/ex06_BasicDelegateExemples.cs

```

1
2 using System;
3 using System.Collections.Generic;
4 using System.Linq;
5 using System.Text;
6 using System.Threading.Tasks;
7
8 namespace ExemplesDelegate
9 {
10     public class BasicDelegateExemples
11     {
12         ////////////////////////////////////////////////////
13         // Déclaration de types délégués
14
15         // Type délégué pour les fonctions de lecture
16         // signature : Object fonction(void)
17         public delegate Object MonTypeFonctionLecture();
18
19         // Type délégué pour les fonctions de comparaison
20         // signature : int fonction(Object, Object)
21         public delegate int MonTypeFonctionComparaison(Object a, Object b);
22
23         ////////////////////////////////////////////////////
24         // Exemples d'utilisation de délégués
25
26         // Exemple de méthode utilisant des variables locales de type délégué
27         public static void ExempleDelegatesUnicastDeBase()
28         {
29             // Ajout d'une éthode dans une instance de délégué (abonnement).
```

```

30 // La méthode doit avoir la même signature que le type délégué
31 MonTypeFonctionLecture readFunction = ExUniformSignature.ReadInt;
32
33 // Ajout d'une éthode dans une instance de délégué (abonnement).
34 // La méthode doit avoir la même signature que le type délégué
35 MonTypeFonctionComparaison compareFunction = ExUniformSignature.
    CompareInt;
36
37 // Exécution de la méthode contenue dans l'instance de délégué
    readFunction
38 // Syntaxiquement, ça fonctionne comme un appel de fonction
39 // Le type
40 Object a = readFunction();
41 Object b = readFunction();
42
43 // Exécution de la méthode contenue dans l'instance de délégué
    compareFunction
44 if (compareFunction(a, b) == 0)
45     Console.WriteLine("{0} égale {1}", a, b);
46 else
47     if (compareFunction(a, b) < 0)
48         Console.WriteLine("{0} inférieur à {1}", a, b);
49     else
50         Console.WriteLine("{0} supérieur à {1}", a, b);
51 }
52
53 // Passage de paramètres de type délégué
54 // La méthode ici définie marche pour nos trois types int, string et
    MyDate
55 // Les méthodes de lecture et de comparaison sont passées en paramètre
    !!!!
56 public static void TestCompareGeneric(MonTypeFonctionLecture
    readFunction,
57                                     MonTypeFonctionComparaison
    compareFunction)
58 {
59     Object a = readFunction();
60     Object b = readFunction();
61
62     if (compareFunction(a, b) == 0)
63         Console.WriteLine("{0} égale {1}", a, b);
64     else
65         if (compareFunction(a, b) < 0)
66             Console.WriteLine("{0} inférieur à {1}", a, b);
67         else
68             Console.WriteLine("{0} supérieur à {1}", a, b);
69
70 }
71
72 // Utilisation d'une méthode avec paramètres de type délégué
73 public static void testPassageParametreDelegate()
74 {
75     TestCompareGeneric(ExUniformSignature.ReadInt, ExUniformSignature.
        CompareInt);
76     TestCompareGeneric(ExUniformSignature.ReadString, ExUniformSignature.
        CompareString);

```

```
77         TestCompareGeneric(ExUniformSignature.ReadDate, ExUniformSignature.  
78             CompareDate);  
79     }  
80 }
```

2.4 Exemple d'utilisation : m thode de tri g n rique

Voici une m thode de tri, impl mentation de l'algorithme du tri par s lection, qui peut fonctionner pour nos trois types de donn es : `int`, `string` et `MyDate`. La liste des objets   trier, ainsi que la m thode de comparaison   utiliser pour comparer les  l ments de la liste, sont pass es en param tre.

CoursLinq/ExemplesDelegate/ex07_ExTriGenerique.cs

```
1  
2 using System;  
3 using System.Collections.Generic;  
4 using System.Linq;  
5 using System.Text;  
6 using System.Threading.Tasks;  
7  
8 namespace ExemplesDelegate  
9 {  
10     public class ExTriGenerique  
11     {  
12         // Type d l gu  pour les fonctions de comparaison  
13         // signature : int fonction(Object, Object)  
14         public delegate int MonTypeFonctionComparaison(Object a, Object b);  
15  
16         // M thode de tri qui fonctionne pour tous nos types de donn es : int;  
17         // string ou MyDate  
18         // La fonction permettant de comparer les  l ments   trier est pass e en  
19         // param tre  
20         // La liste est pass e par r f rence et la r f rence est modifi e   la  
21         // fin de la fonction  
22         //  
23         // L'algorithme utilis  pour ce tri "  la main" est le tri par s lection  
24         // :  
25         // tant que la liste est non vide  
26         //     chercher le plus petit  l ment min de la liste  
27         //     ajouter min en queue de liste au r sultat  
28         //     supprimer min de la liste  
29         public static void TriSelection(ref List<Object> liste ,  
30             MonTypeFonctionComparaison fonctionCompare)  
31         {  
32             List<Object> listeTrie = new List<Object>();  
33             // tant que la liste est non vide  
34             while (liste.Count() != 0)  
35             {  
36                 // Recherche du minimum de la liste restante :  
37                 Object min = liste.First();  
38                 foreach (Object element in liste)  
39                 {  
40                     if (fonctionCompare(element, min) < 0)41                     {  
42                         min = element;  
43                     }  
44                 }  
45                 listeTrie.Add(min);  
46                 liste.Remove(min);  
47             }  
48             return listeTrie;  
49         }  
50     }  
51 }
```

```

36         {
37             min = element;
38         }
39     }
40     listeTriee.Add(min); // ajouter min en queue de liste au
        resultat
41     liste.Remove(min); // supprimer min de la liste
42 }
43 // Retour de la liste triée par référence
44 liste = listeTrie;
45 }
46
47 // Génération d'une liste (non ordonnée) de MyDate
48 public static List<Object> generateInitialList()
49 {
50     List<Object> listeDates = new List<Object>();
51     listeDates.Add(new MyDate("20/09/2014"));
52     listeDates.Add(new MyDate("20/09/2012"));
53     listeDates.Add(new MyDate("20/09/2015"));
54     listeDates.Add(new MyDate("20/10/2014"));
55     listeDates.Add(new MyDate("10/09/2014"));
56     listeDates.Add(new MyDate("20/09/2013"));
57     listeDates.Add(new MyDate("20/08/2015"));
58     listeDates.Add(new MyDate("20/09/2014"));
59     listeDates.Add(new MyDate("12/09/2013"));
60     return listeDates;
61 }
62
63 // Méthode de test de la méthode de tri générique, appliquée à une liste
        de dates.
64 public static void TestTriBulleDelegate()
65 {
66     // Génération de la liste de MyDate
67     List<Object> listeDates = generateInitialList();
68
69     // Appel de la méthode de tri avec paramètre délégué pour
70     // la méthode de comparaison
71     TriSelection(ref listeDates, ExUniformSignature.CompareDate);
72
73     // Affichage de la liste triée
74     foreach (MyDate date in listeDates)
75         Console.WriteLine(date + ", ");
76     Console.WriteLine();
77 }
78 }
79 }

```

2.5 Expressions lambda

Les *Expressions lambda* sont un moyen de définir des méthodes anonymes en C#. Voici un exemple de méthode de comparaison défini par une expression lambda, que nous passerons en paramètre à la méthode `ExTriGenerique.TriSelection`.

CoursLinq/ExemplesDelgate/ex08_ExempleExpressionLambda.cs


```
1
2 using System;
3 using System.Collections.Generic;
4 using System.Linq;
5 using System.Text;
6 using System.Threading.Tasks;
7
8 namespace ExemplesDelegate
9 {
10     public class ExempleExpressionLambda
11     {
12         public static void TestTriBulleExprssionLambda()
13         {
14
15             List<Object> listeDates = ExTriGenerique.generateInitialList();
16             // Utilisation d'une expression lambda (m thode anonyme)
17             // Param tres de la m thode anonyme : a et b de type Object
18             // Retour de la m thode anonyme : int (type d termin  dynamiquement
19             // par le return)
20             ExTriGenerique.TriSelection(ref listeDates, (Object a, Object b) =>
21             {
22                 MyDate aa = a as MyDate;
23                 MyDate bb = b as MyDate;
24                 return String.Compare(aa.ToStringReverse(), bb.ToStringReverse()
25                 );
26             });
27
28             // Affichage de la liste tri e
29             foreach (MyDate date in listeDates)
30                 Console.WriteLine(date + ", ");
31         }
32     }
```

Chapitre 3

Collections

Les *Collections* sont des classes qui gèrent des structures de données représentant des ensembles, listes, tableaux, etc. d'objets. Les opérations de base des collections sont typiquement l'ajout, la suppression d'éléments, le tri suivant un certain ordre (utilisant une fonction de comparaison), etc.

Certaines collections, comme les files (gestion *First In First Out* ou encore *FIFO*) et les piles (gestion *Last In First Out LIFO*), ont un ensemble de primitives spécifiques, qui permettent une gestion particulière des éléments qui ont de bonnes propriétés algorithmiques.

Les *dictionnaires* permettent, eux, de représenter des applications (au sens mathématique, associant à un élément d'un ensemble un élément d'un autre ensemble). Un dictionnaire associe à différentes clefs des valeurs. Chaque clef est unique (pas de doublon) et sa valeur est donnée par le dictionnaire.

Nous abordons ici les collections appelées *génériques*, parce qu'elles dépendent d'un *template*, à savoir, le type de données qui constitue les éléments de la collection (ou encore les clefs et les valeurs dans le cas d'un dictionnaire). Elles se trouvent dans le namespace suivant :

`System.Collections.Generic`

Dont la documentation peut être consultée via la *MSDN* en ligne ou dans la visionneuse d'aide de *Visual Studio*.

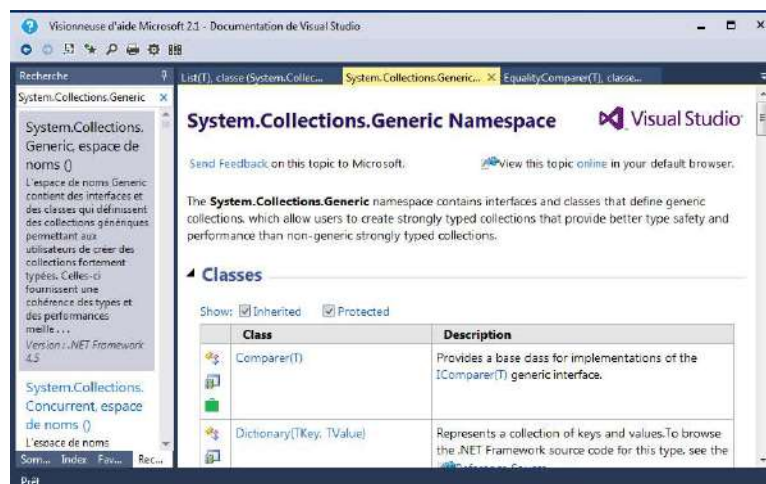


FIGURE 3.1 : Capture de la Visionneuse d'Aide de *Visual Studio*

3.1 Listes

CoursLinq/ExemplesCollections/ex01_ExempleList.cs

```
1
2 using System;
3 using System.Collections.Generic;
4 using System.Linq;
5 using System.Text;
6 using System.Threading.Tasks;
7
8 namespace ExemplesCollections
9 {
10     public class ExempleList
11     {
12         public static void ExempleListe()
13         {
14             // D claration d'une r f rence sur des List<int>
15             List<int> liste;
16             // Allocation OBLIGATOIRE : cr ation d'une liste vide
17             // (par exemple dans un constructeur, pour initialiser un attribut
18             // liste)
19             liste = new List<int>();
20
21             // La m thode Add ajoute un  l ment en queue de liste
22             liste.Add(4); //  l ments : 4
23             liste.Add(7); //  l ments : 4, 7
24             liste.Add(9); //  l ments : 4, 7, 9
25             liste.Add(7); //  l ments : 4, 7, 9, 7
26             // Insertion de 34 en t te de liste
27             liste.Insert(0, 34); //  l ments : 34, 4, 7, 9, 7
28             liste.Insert(2, 12); //  l ments : 34, 4, 12, 7, 9, 7
29             // Suppression de la premi re occurrence d'un  l ment :
30             liste.Remove(7); //  l ments : 34, 4, 12, 9, 7
31
32             // Nombre d' l ments de la liste :
33             Console.WriteLine("Nombre d' l ments : {0}", liste.Count);
34             // Parcours de la liste par foreach : affiche "34, 4, 12, 9, 7,"
35             foreach (int element in liste)
36             {
37                 Console.Write(element + ", ");
38             }
39             liste.Sort(); // Tri de la liste
40             Console.WriteLine("\nListe Tri e :");
41             foreach (int element in liste)
42             {
43                 Console.Write(element + ", ");
44             }
45         }
46     }
47 }
```

3.2 Files

Une *file* est une structures de données dans laquelle on peut ajouter et supprimer des éléments suivant la règle du *premier arrivé premier sorti*, ou encore *FIFO* (*First In First Out*).

Le nom de file vient d'une analogie avec une file d'attente à un guichet, dans laquelle le premier arrivé sera la premier servi. Les usagers arrivent en queue de file, et sortent de la file à sa tête.

Les files correspondent à la classe `C#Queue<T>` Les principales primitives de gestion des files sont les suivantes :

- **Constructeur** : cette fonction crée une file vide.
- **EstVide** : renvoie 1 si la file est vide, 0 sinon.
- **EstPleine** : renvoie 1 si la file est pleine, 0 sinon.
- **AccederTete** : cette fonction permet l'accès à l'information contenue dans la tête de file.
- **Enfiler** : cette fonction permet d'ajouter un élément en queue de file. La fonction renvoie un code d'erreur si besoin en cas de manque de mémoire.
- **Defiler** : cette fonction supprime la tête de file. L'élément supprimé est retourné par la fonction `Defiler` pour pouvoir être utilisé.
- **Vider** : cette fonction vide la file.
- **Detruire** : cette fonction permet de détruire la file.

CoursLinq/ExemplesCollections/ex02_ExempleQueue.cs

```

1
2 using System;
3 using System.Collections.Generic;
4 using System.Linq;
5 using System.Text;
6 using System.Threading.Tasks;
7
8 namespace ExemplesCollections
9 {
10     public class ExempleQueue
11     {
12         public static void ExempleFile()
13         {
14             // déclaration d'une variable référence vers des Queues<int>
15             Queue<int> file;
16             // Allocation OBLIGATOIRE : création d'une file vide
17             // (par exemple dans un constructeur, pour initialiser un attribut
18             // file)
19             file = new Queue<int>();
20             // La méthode Enqueue ajoute un élément en queue de file (FIFO)
21             file.Enqueue(4); // éléments : 4
22             file.Enqueue(7); // éléments : 4, 7
23             file.Enqueue(9); // éléments : 4, 7, 9
24             // Suppression de la tête de file (l'élément supprimé est retourné)

```

```

24         int tete = file.Dequeue(); //  l ment supprim  : 4 (gestion FIFO)
25         Console.WriteLine("La t te de file {0} a  t  supprim e", tete);
26         Console.WriteLine("La t te de file est maintenant {0}", file.Peek())
           ;
27
28         // Parcours de la liste par foreach : affiche "7, 9"
29         foreach (int element in file)
30         {
31             Console.Write(element + ", ");
32         }
33     }
34
35 }
36
37 }

```

3.3 Dictionnaires

Comme nous l'avons dit, un dictionnaire permet d'associer   des clefs d'un certain type des valeurs d'un autre type. Voici un exemple dans lequel notre dictionnaire associe   une cha ne de caract re (type `string`) repr sentant le nom d'un aliment, un nombre repr sentant le poids de cet aliment en grammes.

CoursLinq/ExemplesCollections/ex03_ExempleDictionnaire.cs

```

1
2 using System;
3 using System.Collections.Generic;
4 using System.Linq;
5 using System.Text;
6 using System.Threading.Tasks;
7
8 namespace ExemplesCollections
9 {
10     public class ExempleDictionnaire
11     {
12         public static void ExempleDico()
13         {
14             // d claration d'une variable r f rence vers des Dictionary<string,
              int>
15             Dictionary<string, int> dicoPoids;
16             // Allocation OBLIGATOIRE : cr ation d'une dictionnaire vide
17             // (par exemple dans un constructeur, pour initialiser un attribut
              dicoPoids)
18             dicoPoids = new Dictionary<string, int>(); // poids en gramme d'un
              aliment
19
20             try
21             {
22                 dicoPoids.Add("Barquette de frites", 100);
23                 dicoPoids.Add("Yoghurt nature", 125);
24                 dicoPoids.Add("Yoghurt aux fruit", 140);
25                 dicoPoids.Add("Litre de lait", 1000);
26             }
27             catch (Exception e)

```

```
28     {
29         Console.WriteLine("Erreur : " + e.StackTrace);
30     }
31
32     if (dicoPoids.Remove("Yoghurt aux fruit"))
33     {
34         Console.WriteLine("Suppression de Yoghurt aux fruit...");
35     } else
36     {
37         Console.WriteLine("Yoghurt aux fruit : clef introuvable");
38     }
39
40     // Parcours de l'ensemble des couples (clefs, valeurs)
41     // 1) Récupération de la collection des clefs :
42     Dictionary<string, int>.KeyCollection lesClefs = dicoPoids.Keys;
43     // 2) Parcours de la collection des clefs et affichage des valeurs
44     :
45     foreach (string clef in lesClefs)
46     {
47         Console.WriteLine("Le poids de {0} est de {1} grammes",
48             clef, dicoPoids[clef]);
49     }
50
51     // Parcours de l'ensemble des clefs valeurs
52     Dictionary<string, int>.ValueCollection lesValeurs = dicoPoids.
53     Values;
54     Console.WriteLine("Les poids possibles pour les aliments présents
55     sont :");
56     foreach (int val in lesValeurs)
57     {
58         Console.WriteLine("{0} grammes, ", val);
59     }
60
61     try
62     {
63         Console.WriteLine("Voici le poids de la boîte d'oeufs : ",
64             dicoPoids["Boîte de 6 oeufs"]);
65     }
66     catch (KeyNotFoundException)
67     {
68         Console.WriteLine("Euh... En fait, nous n'avons pas d'oeufs...")
69         ;
70     }
71
72     try
73     {
74         // Les doublons ne sont pas permis : chaque clef est unique
75         // et est associée à une seule valeur
76         dicoPoids.Add("Yoghurt nature", 150);
77     }
78     catch (Exception e)
79     {
80         Console.WriteLine("Erreur : chaque clef est unique!!!");
81         Console.WriteLine(e.StackTrace);
82     }
83 }
```

```
80         try
81         {
82             // Recherche du premier élément satisfaisant un prédicat (poids
83             // >= 150 grammes)
84             // Le prédicat est une expression lambda prenant une paire clef
85             // valeur
86             // et retournant un booléen.
87             KeyValuePair<string, int> clefValeurPoidsSuperieurA150 =
88                 dicoPoids.First(paireClefValeur => paireClefValeur.Value >=
89                 150);
90             Console.WriteLine("Le premier poids supérieur à 150 grammes dans
91             le dictionnaire est ");
92             Console.WriteLine("le poids de {0}, soit {1} grammes",
93                 clefValeurPoidsSuperieurA150.Key,
94                 clefValeurPoidsSuperieurA150.Value);
95         }
96     }
97 }
98 }
```

3.4 Protocole de comparaison

Le protocole de comparaison permet de définir l'ordre sur les éléments d'une collection pour certaines opérations comme, typiquement, le tri des éléments.

La définition d'une classe fille de *Comparer* permet la comparaison d'instances, par exemple pour trier les objets. Cette classe *Comparer* contient une méthode abstraite : la méthode de comparaison, qui doit être implémentée. La classe *Comparer* est générique (elle s'applique à un template).

Dans l'exemple suivant, on compare des dates (de type *MyDate*) pour les trier dans l'ordre chronologique.

CoursLinq/ExemplesDelegate/ex09_MyDateComparer.cs

```
1
2 using System;
3 using System.Collections;
4 using System.Collections.Generic;
5 using System.Linq;
6 using System.Text;
7 using System.Threading.Tasks;
8
9 namespace ExemplesDelegate
10 {
11     // L'implémentation de l'interface IComparer permet
12     // la comparaison d'instances, par exemple pour trier les objets
13     public class MyDateComparer : Comparer<MyDate>
14     {
15         public override int Compare(MyDate a, MyDate b)
16         {
```

```

17         return String.Compare(a.ToStringReverse(), b.ToStringReverse());
18     }
19 }
20 }

```

Voici un exemple d'utilisation pour trier une liste de dates avec la méthode `List<T>.Sort`.

CoursLinq/ExemplesDelegate/ex10_TestComparerClass.cs

```

1
2 using System;
3 using System.Collections.Generic;
4 using System.Linq;
5 using System.Text;
6 using System.Threading.Tasks;
7
8 namespace ExemplesDelegate
9 {
10     public class TestComparerClass
11     {
12         public static void TestSortDates()
13         {
14             List<MyDate> listeDates = new List<MyDate>();
15             listeDates.Add(new MyDate("20/09/2014"));
16             listeDates.Add(new MyDate("20/09/2012"));
17             listeDates.Add(new MyDate("20/09/2015"));
18             listeDates.Add(new MyDate("20/10/2014"));
19             listeDates.Add(new MyDate("10/09/2014"));
20             listeDates.Add(new MyDate("20/09/2013"));
21             listeDates.Add(new MyDate("20/08/2015"));
22             listeDates.Add(new MyDate("20/09/2014"));
23             listeDates.Add(new MyDate("12/09/2013"));
24
25             listeDates.Sort(new MyDateComparer());
26
27             foreach (MyDate date in listeDates)
28                 Console.Write(date + ", ");
29             Console.WriteLine();
30         }
31     }
32 }

```

3.5 Protocole d'égalité

Le protocole d'égalité permet de définir les critères sur les éléments d'une collection pour les considérer comme *égaux*, ou plus exactement, comme *équivalents*. En effet, par défaut, les *références* des instances de classes sont utilisées pour tester l'égalité. Ça n'est pas toujours pertinent.

Si l'on reprend l'exemple de notre classe `MyDate` de la partie 2.1, deux instances différentes (donc avec des références différentes) de la classe `MyDate` pourraient représenter la même date si les ont même année, même mois, et même jour. Le programmeur doit ainsi spécifier les critères à prendre en compte pour l'égalité, ce qui se définit par une fonction booléenne.

L'implémentation d'une classe dérivée de la classe `EqualityComparer` permet la comparaison d'instances pour tester l'égalité (ou l'équivalence) des objets. Cela nécessite l'implémentation

de deux méthodes (qui sont virtuelles au niveau de la classe *EqualityComparer*) : la méthode booléenne d'égalité `Equals`, et la méthode `GetHashCode` qui donne le code de hachage. La classe est générique (elle s'applique à un template).

Dans l'exemple suivant, on compare des dates (de type `MyDate`) pour définir s'il s'agit "d'une même date".

CoursLinq/ExemplesDelegate/ex11_MyDateEqualityComparer.cs

```

1
2 using System;
3 using System.Collections.Generic;
4 using System.Linq;
5 using System.Text;
6 using System.Threading.Tasks;
7
8 namespace ExemplesDelegate
9 {
10     class MyDateEqualityComparer : EqualityComparer<MyDate>
11     {
12         public override bool Equals(MyDate d1, MyDate d2)
13         {
14             return d1.Annee == d2.Annee && d1.Mois == d2.Mois && d1.Jour == d2.
15                 Jour;
16         }
17         public override int GetHashCode(MyDate d)
18         {
19             // Chaque date correspondra à un entier unique (c'est le mieux)
20             return int.Parse(d.Annee.ToString().PadLeft(4, '0')*10000 + d.Mois.
21                 ToString().PadLeft(2, '0')*100 + d.Jour.ToString().PadLeft(2));
22         }
23     }

```

Dans l'exemple suivant, on voit comment utiliser ce protocole d'égalité pour gérer l'interdiction des doublons dans la collection des clefs d'un dictionnaire (chaque clef est unique, mais on ne parle pas forcément de l'égalité des références).

Dans notre cas des dates, si on veut associer à chaque date une liste d'événements (ou autre), il s'agit que les événements d'une même date soient regroupés dans le dictionnaire et accessible avec la clef correspondant à cette date.

CoursLinq/ExemplesDelegate/ex12_TestEqualityComparer.cs

```

1
2 using System;
3 using System.Collections.Generic;
4 using System.Linq;
5 using System.Text;
6 using System.Threading.Tasks;
7
8 namespace ExemplesDelegate
9 {
10     public class TestEqualityComparerClass
11     {
12         static void TryAdd(Dictionary<MyDate, string> dico, MyDate date, string
13             chaine)

```

```
14         try
15         {
16             dico.Add(date, chaine);
17         } catch
18         {
19             if (chaine != null)
20                 Console.Error.WriteLine("Tentative d'ajout d'une clé
21                                     existante dans le dico.");
22         }
23     }
24     public static void TestInsertDates()
25     {
26         Dictionary<MyDate, string> dico = new Dictionary<MyDate, string>(new
27             MyDateEqualityComparer());
28         TryAdd(dico, new MyDate("20/09/2014"), "1ère date 20/09/2014");
29         TryAdd(dico, new MyDate("20/09/2012"), "2ème date 20/09/2012");
30         TryAdd(dico, new MyDate("20/09/2015"), "3ème date 20/09/2015");
31         TryAdd(dico, new MyDate("20/10/2014"), "4ème date 22/10/2014");
32         TryAdd(dico, new MyDate("10/09/2014"), "5ème date 10/09/2014");
33         TryAdd(dico, new MyDate("20/09/2015"), "6ème date 20/09/2013");
34         TryAdd(dico, new MyDate("20/08/2015"), "7ème date 20/08/2015");
35         TryAdd(dico, new MyDate("20/09/2014"), "8ème date 20/09/2014");
36         TryAdd(dico, new MyDate("12/09/2013"), "9ème date 12/09/2013");
37
38         foreach (string valeur in dico.Values)
39             Console.Write(valeur + ", ");
40         Console.WriteLine();
41     }
42 }
```

Chapitre 4

Requêtes *LINQ*

4.1 Types délégués anonymes

Comme nous l'avons vu dans le chapitre 2, les types délégués sont des types dont les instances peuvent contenir des méthodes. Par exemple, si l'on considère un type délégué pour un prédicat qui, pour une personne, renvoie un booléen. On pourra par exemple affecter dans une instance de ce type délégué une méthode qui renvoie vrai si l'âge de la personne est supérieur ou égal à 18 ans. Le code correspondant serait, en supposant que l'on dispose d'une méthode `Personne.getAge()` qui renvoie un nombre d'années révolues depuis la date de naissance de la personne (instance d'une classe `Personne`).

```
1 class DemoDelegate{
2     // Type délégué pour des prédicats sur des personnes :
3     public delegate bool MonTypePredicatSurPersonne(Personne personne);
4
5     // Méthode qui teste une prédicat sur une Personne
6     public static void TestPredicatPersonne(MonTypePredicatSurPersonne predicat)
7     {
8         // Création d'une instance de Personne (appel de fabrique)
9         Personne personne = Personne.fabriqueDePersonne();
10        if (predicat(personne))
11        {
12            Console.WriteLine("La personne vérifie le prédicat.");
13        }else
14        {
15            Console.WriteLine("La personne vérifie le prédicat.");
16        }
17    }
18
19    public static AppelleTestPredicatPersonne()
20    {
21        TestPredicatPersonne((Personne personne) => {
22            return personne.GetAge() >= 18;
23        });
24    }
25 }
```

Avec cette syntaxe pour définir des types délégués, nous devons obligatoirement définir notre type délégué et lui donner un nom avant de l'utiliser pour définir des fonctions. En fait,

il existe en C# une syntaxe pour définir des types délégués à la volée.

Reprenons notre exemple, ci-dessus, du prédicat qui, pour une personne, renvoie un booléen. Nous aurions pu définir notre méthode `TestPredicatPersonne` directement sans définir auparavant un type `MonTypePredicatSurPersonne` :

```

1 class DemoDelegate{
2     // Méthode qui teste une prédicat sur une Personne
3     public static void TestPredicatPersonne(Fonc<Personne, bool> predicat)
4     {
5         // Création d'une instance de Personne (appel de fabrique)
6         Personne personne = Personne.fabriqueDePersonne();
7         if (predicat(personne))
8         {
9             Console.WriteLine("La personne vérifie le prédicat.");
10        }else
11        {
12            Console.WriteLine("La personne vérifie le prédicat.");
13        }
14    }
15
16    public static AppelleTestPredicatPersonne()
17    {
18        TestPredicatPersonne((Personne personne) => {
19            return personne.GetAge() >= 18;
20        });
21    }
22 }

```

Ainsi, `Fonc<Personne, bool>` désigne un type délégué qui à une `Personne` associe un booléen (prédicat sur une `Personne`). De même, `Fonc<Personne, int>` désigne un type délégué qui à une `Personne` associe un entier.

4.2 Méthodes d'extension

Les méthodes d'extension permettent d'ajouter des méthodes à une classe existante sans retoucher le code de la classe proprement dite. Les méthodes de LINQ sont définies comme des méthodes d'extension sur les différentes classes de collections. Ça n'est pas difficile à utiliser, mais pour comprendre les prototypes des méthodes de LINQ, il faut savoir lire le prototype d'une méthode d'extension.

Par exemple, supposons que nous souhaitions ajouter à la classe `string` une méthode qui calcule le nombre d'occurrences d'un caractère passé en paramètre. Voici comme on peut faire :

```

1 using System;
2 using System.Linq;
3 using System.Text;
4
5 namespace DemoExtension
6 {
7     // Les méthodes d'extension doivent être définies dans une classe statique
8     public static class StringExtension

```

```

9      {
10     // Ceci est la m thode d'extension.
11     // Le premier param tre est d clar  avec le modificateur this
12     // La m thode s'appliquera   des instances de la classe
13     // correspondant au premier param tre (ici String).
14     public static int CompteOccurrences(this String chaine, Char caractere)
15     {
16         // La m thode Where s lectionne les caract res suivant un predicat
17         // (voir d tails de la m thode ci-dessous
18         return chaine.Where((Char c) => {return c==caractere;})
19             .ToList().Count();
20     }
21 }
22
23 class Program
24 {
25     static void Main(string[] args)
26     {
27         string s = "C# peut repr senter un paradigme du langage objet";
28         // Appel de la m thode d'extension qui compte les
29         // occurrences de la lettre 'a' dans la chaine s
30
31         int nOccurrences = s.CompteOccurrences('a');
32         Console.WriteLine("Il y a {0} caract res {1} dans la chaine {2}",
33             nOccurrences, 'a', s);
34     }
35 }
36 }

```



Notez que le premier param tre dans la d finition de la m thode d'extention, qui est modifi  par le mot cl  **this**, ne fait pas vraiment partie des param tres de la m thode, mais il repr sente l'instance de classe sur laquelle s'appliquera la m thode.

4.3 Interface IEnumerable<T>

L'interface `IEnumerable<T>` est l'interface de base de toutes les classes de collections g n riques. Elle ne contient qu'une seule m thode, qui renvoie un * num rateur* (aussi appel  *it rateur*).

```
1 | IEnumerable<T> GetEnumerator()
```

L' num rateur poss de les m thodes qui permettent de parcourir une collection (se positionner au d but, obtenir l' l ment suivant, etc.). L'interface `IEnumerable<T>` esst ainsi le strict n cesssaire pour parcourir une collection.

La plupart des m thodes de *LINQ*, qui s'appliquent   une collection renvoient une collection en sortie, sont des m thodes d'extension de l'interface `IEnumerable<T>`. Elles auront donc un premier param tre **this IEnumerable<T>** :

```
1 | IEnumerable<T> MaMethodeDeLINQ(this IEnumerable<T>, etc ...)
```

4.4 Quelques méthodes de *LINQ*

4.4.1 Filtre *Where*

La méthode d'extension *Where*, qui est définie sur les principaux types de collections en *C#*.

```
1 public static IEnumerable<TSource> Where<TSource>(
2     this IEnumerable<TSource> source ,
3     Func<TSource, bool> predicate
4 )
```

Paramètres :

- **source** : une collection générique qui implémente *IEnumerable<TSource>*, dont les éléments sont d'un type *TSource*.
- **predicate** : Prédicat sur le type *TSource* des éléments de la collection source (qui à une instance de *TSource* associe un booléen).

Valeur retournée : La sous collection (du type *IEnumerable<TSource>*) de la collection source formée des éléments qui satisfont le prédicat (le prédicat renvoie vrai sur ces éléments).

4.4.2 Méthode de tri *OrderBy*

La méthode d'extension *OrderBy*, qui est définie sur les principaux types de collections en *C#*, permet de trier les éléments suivant un critère de tri (implémentation de *IComparer* (voir partie 3.4)).

```
1 public static IOrderedEnumerable<TSource> OrderBy<TSource, TKey>(
2     this IEnumerable<TSource> source ,
3     Func<TSource, TKey> keySelector ,
4     IComparer<TKey> comparer
5 )
```

Paramètres :

- **source** : une collection générique qui implémente *IEnumerable<TSource>*, dont les éléments sont d'un type *TSource*.
- **keySelector** : Sélection de clef sur le type *TSource* des éléments de la collection source (qui à une instance de *TSource* associe une clef de type *TKey*).
- **comparer** : instance d'une classe de comparaison des clefs (qui implémente l'interface *IComparer*). Si le paramètre **comparer** est omis, le comparateur par défaut sur les clefs, s'il existe, est utilisé.

Valeur retournée : La collection (du type *IEnumerable<TSource>*) qui comprend les mêmes éléments que la collection source, mais triés suivant l'ordre défini par la méthode *Compare* du *comparer*.

4.4.3 Filtre `Where` sur un dictionnaire

La m thode d'extension `Where`, qui est d finie sur les principaux types de collections en C#.

```
1 public static IEnumerable<KeyValuePair<TKey, TValue>> Where<KeyValuePair<TKey,
2     TValue>>(
3     this IEnumerable<KeyValuePair<TKey, TValue>> source,
4     Func<KeyValuePair<TKey, TValue>, Boolean> predicate
5 )
```

Param tres :

- `source` : un dictionnaire qui impl mente `IDictionary<TKey, TValue>`, dont les clef sont d'un type `TKey` et les valeurs d'un type `TValue`,
- `predicate` : Pr dicat sur le type `KeyValuePair<TKey, TValue>` des  l ments de la collection `source` (qui   une instance de `KeyValuePair<TKey, TValue>` associe un bool en.

Valeur retourn e : La sous collection (du type `KeyValuePair<TKey, TValue>`) des couple clef/valeur du dictionnaire `source` form e des couples qui satisfont le pr dicat (le pr dicat renvoie vrai sur ces  l ments).

4.4.4 M thode de regroupement `GroupBy`

La m thode d'extension `GroupBy`, qui est d finie sur les principaux types de collections en C#, permet de regroupe les  l ments suivant un crit re de projection et de comparaison (impl mentation de `IEqualityComparer` (voir partie 3.5)).

```
1 public static IEnumerable<IGrouping<TKey, TSource>> GroupBy<TSource, TKey>(
2     this IEnumerable<TSource> source,
3     Func<TSource, TKey> keySelector,
4     IEqualityComparer<TKey> comparer
5 )
6
7 public interface IGrouping<out TKey, out TElement> : IEnumerable<TElement>,
8     IEnumerable
```

Param tres :

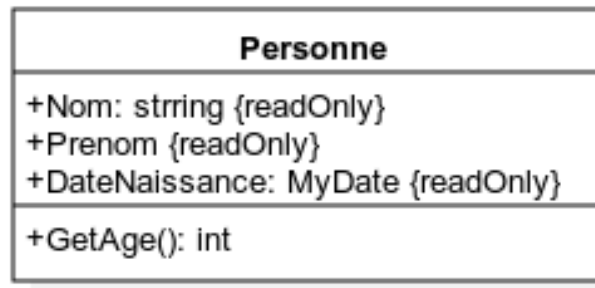
- `source` : une collection g n rique qui impl mente `IEnumerable<TSource>`, dont les  l ments sont d'un type `TSource`.
- `keySelector` : S lection de clef sur le type `TSource` des  l ments de la collection `source` (qui   une instance de `TSource` associe une clef de type `TKey`).
- `comparer` : instance d'une classe de comparaison des clefs (qui impl mente l'interface `IEqualityComparer`) (voir partie 3.5). Si le param tre `comparer` est omis, le comparateur d' galit  par d faut sur les clefs, s'il existe, est utilis .

Valeur retourn e : L'interface `IGrouping<out TKey, out TElement>`, similaire   un dictionnaire, repr sente une association qui   chaque clef unique (suivant le protocole d' galit  d fini par `comparer`) associe la collection (du type `IEnumerable<TSource>`) qui comprend les  l ments que la collection `source` poss dent cette clef.

On peut parcourir la collection regroup e par une double boucle `foreach` imbriqu e.

4.4.5 Exemples

Voici le diagramme d'une classe personne, avec un nom, un prénom et une date de naissance. Une méthode `GetAge` permet d'obtenir l'âge de la personne en années révolues (au format `int`).



Diag 8. Diagramme de Classes : la classe Personne.

Voici maintenant des exemples sélectionnant, dans une liste de personnes, les personnes majeures, triées et/ou regroupées suivant différents critères.

```

1 using System;
2 using System.Collections.Generic;
3 using System.Linq;
4
5 namespace ExemplesLINQ
6 {
7     public class ExemplesPersonnesLINQ
8     {
9         // L'implémentation de l'interface IComparer permet
10        // la comparaison d'instances, par exemple pour trier les objets
11        public class MyDateComparer : Comparer<MyDate>
12        {
13            public override int Compare(MyDate a, MyDate b)
14            {
15                return String.Compare(a.ToStringReverse(), b.ToStringReverse());
16            }
17        }
18
19        // L'héritage de la classe EqualityComparer<MyDate> permet
20        // la comparaison d'instances pour tester l'égalité
21        // Ou plutôt, l'équivalence, suivant un certain critère, ici l'année.
22        class MyDateYearEquality : EqualityComparer<MyDate>
23        {
24            public override bool Equals(MyDate d1, MyDate d2)
25            {
26                return d1.Annee == d2.Annee;
27            }
28
29            public override int GetHashCode(MyDate d)
30            {
31                return int.Parse(d.Annee);
32            }
33        }
  
```



```
34
35 // M thode de d monstration d'exemples LINQ sur les Personne
36 public static List<Personne> DemoLINQ_Personne1(List<Personne> liste)
37 {
38     // Exemple avec comparateurs de string par d faut
39     // (ordre alphab tique)
40     return liste.Where((Personne p) => {return (p.GetAge() >= 18;)})
41         .OrderBy((Personne p) => {return p.Nom;})
42         .ThenBy((Personne p) => {return p.Prenom;}).ToList();
43 }
44
45 // M thode de d monstration d'exemples LINQ sur les Personne
46 public static List<Personne> DemoLINQ_Personne2(List<Personne> liste)
47 {
48     // M me exemple avec syntaxe d'expression lambda plus l g re
49     // Types des param tres implicites, corps de fonction r duit
50     return liste.Where(p => (p.GetAge() >= 18))
51         .OrderBy(p => p.Nom)
52         .ThenBy(p => p.Prenom).ToList();
53 }
54
55 // M thode de d monstration d'exemples LINQ sur les Personne
56 public static List<Personne> DemoLINQ_Personne3(List<Personne> liste)
57 {
58     // Exemple avec comparateurs de MyDate par ordre
59     // chronologique
60     // Voir partie 3.4 du poly (protocole de comparaison)
61     return liste.Where(p => {return (p.GetAge() >= 18;)})
62         .OrderBy(p => p.DateNaissance, new MyDateComparer());
63 }
64
65 // M thode de d monstration d'exemples LINQ sur les Personne
66 public static List<Personne> DemoLINQ_Personne4(List<Personne> liste)
67 {
68     // Exemple avec comparateurs de MyDate par ordre
69     // chronologique
70     // Voir partie 3.4 du poly (protocole de comparaison)
71     return liste.Where(p => {return (p.GetAge() >= 18;)})
72         .OrderBy(p => p.DateNaissance, new MyDateComparer());
73         .GroupBy(p => p.DateNaissance, new MyDateYearEquality())
74         ;
75 }
76 }
77 }
```