

Les éléments principaux depuis la Version 2.0



-
- ☼ **Les Generics**
 - ☼ **Les classes partielles**
 - ☼ **Les méthodes anonymes**
 - ☼ **TAD de listes, piles, files génériques**
 - ☼ **Arbres binaires et classes génériques**
 - ☼ **Principes des bases de données**
 - ☼ **Ado .net et données relationnelles**
 - ☼ **Ado .net et SQL serveur 2005**

Les Generics

C#.net

Plan général: 

1. Les generics

- 1.1 Une liste générale sans les generics
- 1.2 Une liste générale avec les generics
- 1.3 Une méthode avec un paramètre générique
- 1.4 Type générique contraint
- 1.5 Surcharge générique d'une classe

1. Les generics ou types paramétrés

Les generics introduits depuis la version 2.0 du C# permettent de construire des classes des structs, des interfaces, des delegates ou des méthodes qui sont paramétrés par un type qu'ils sont capables de stocker et de manipuler.

Le compilateur C# reconnaît un type paramétré (generic) lorsqu'il trouve un ou plusieurs identificateurs de type entre crochets <...> :

Exemples de syntaxe de classes génériques

```
//les types paramétrés peuvent être appliqués aux classes aux interfaces
```

```
interface IGeneric1<T>
{
}
class ClassGeneric1< UnType, Autre >
{
}
class ClassInt1 : ClassGeneric1< int, int >
{
}
class ClassInt2 <T> : ClassGeneric1< int, T >
{
}
class ClassInt3 <T, U> : ClassGeneric1< int, U >
{
}
```

Exemples de syntaxe de méthodes génériques

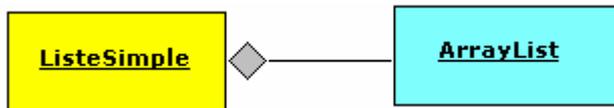
```
//les types paramétrés peuvent être appliqués aux méthodes
```

```
class c1A
{
    public void methode1<T>()
    {
    }
    public T[] methode2<T>()
    {
        return new T[10];
    }
}
```

1.1 Une liste générale sans les generics

On pourrait penser que le type object qui est la classe mère des types références et des types valeurs devrait suffire à passer en paramètre n'importe quel type d'objet. Ceci est en effet possible mais avec des risques de mauvais transtypage dont les tests sont à la charge du développeur.

Considérons une classe `ListeSimple` contenant un objet `ArrayList` :



Nous souhaitons stocker dans cette classe des entiers de type **int** et afficher un élément de la liste de rang fixé. Nous écrivons un programme dans lequel nous rangeons deux entiers dans un objet listeSimpleInt de type **ListeSimple**, puis nous les affichons :

```

public class ListeSimple
{
    public ArrayList liste = new ArrayList();

    public void ajouter(object elt)
    {
        this.liste.Add(elt);
    }
    public int afficherInt(int rang)
    {
        return (int)this.liste[rang];
    }
}

class Program
{
    static void Main(string[] args)
    {
        ListeSimple listeSimpleInt = new ListeSimple();
        listeSimpleInt.ajouter(32);
        listeSimpleInt.ajouter(-58);
        int x;
        for(int i=0; i<=1;i++)
        {
            x = listeSimpleInt. afficherInt (i);
            Console.WriteLine("listeSimpleInt : " + x);
        }
        Console.WriteLine();
    }
}
  
```

Qui affichera lors de l'exécution à la console :

```

listeSimpleInt : 32
listeSimpleInt : -58
  
```

Remarque n°1 :

Dans la méthode afficherInt() l'instruction "**return (int)this.liste[rang];**". Nous sommes obligés de transtyper l'élément de rang "i" car dans l'**ArrayList** nous avons stocké des éléments de type **object**.

Remarque n°2 :

Nous pouvons parfaitement ranger dans le même objet listeSimpleInt de type **ListeSimple** des **string** en construisant la méthode afficherStr() de la même manière que la méthode afficherInt() :

```

public string afficherStr(int rang)
  
```

```

    {
        return (string)this.liste[rang];
    }

```

Soit le code de la classe ainsi construite :

```

public class ListeSimple
{
    public ArrayList liste = new ArrayList();

    public void ajouter(object elt)
    {
        this.liste.Add(elt);
    }
    public int afficherInt(int rang)
    {
        return (int)this.liste[rang];
    }
    public string afficherStr(int rang)
    {
        return (string)this.liste[rang];
    }
}

```

Si nous compilons cette classe, le compilateur n'y verra aucune erreur, car le type **string** hérite du type **object** tout comme le type **int**. Mais alors, si par malheur nous oublions lors de l'écriture du code d'utilisation de la classe, de ne ranger que des éléments du même type (soit uniquement des **int**, soit uniquement des **string**) alors l'un des transtypage produira une erreur.

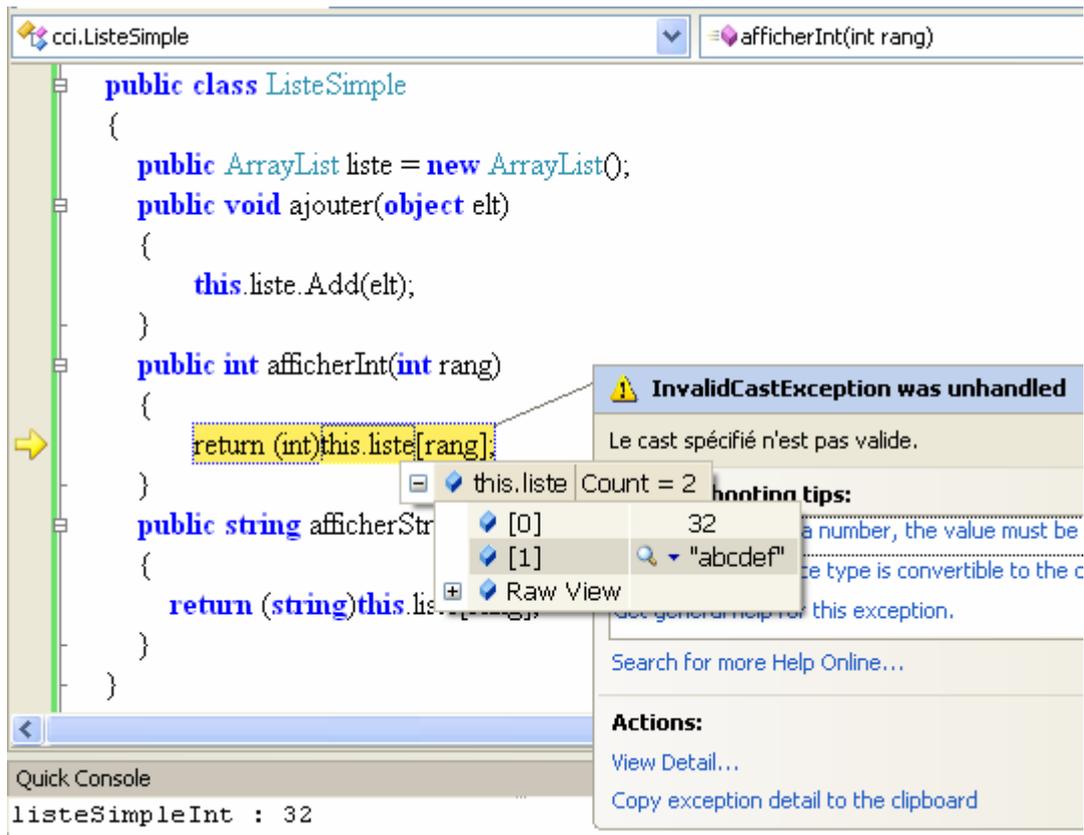
Le programme ci-dessous range dans l'**ArrayList** un premier élément de type **int**, puis un second élément de type **string**, et demande l'affichage de ces deux éléments par la méthode **afficherInt()**. Aucune erreur n'est signalée à la compilation alors qu'il y a incompatibilité entre les types. Ceci est normal puisque le type de l'objet est obtenu dynamiquement :

```

class Program
{
    static void Main(string[] args)
    {
        ListeSimple listeSimpleInt = new ListeSimple();
        listeSimpleInt.ajouter ( 32 );
        listeSimpleInt.ajouter ( "abcdef" );
        int x;
        for(int i=0; i<=1;i++)
        {
            x = listeSimpleInt. afficherInt (i);
            Console.WriteLine("listeSimpleInt : " + x);
        }
        Console.WriteLine();
    }
}

```

Le transtypage du deuxième élément qui est une **string** en un **int**, est une erreur qui est signalée par le CLR lors de l'exécution :



Il revient donc au développeur dans cette classe à prêter une attention de tous les instants sur les types dynamiques des objets lors de l'exécution et éventuellement de gérer les incompatibilités par des gestionnaires d'exception **try...catch**.

Nous allons voir dans le prochain paragraphe comment les generics améliorent la programmation de ce problème.

1.2 Une liste générale avec les generics

Nous construisons une classe de type T générique agrégeant une liste d'éléments de type T générique. Dans ce contexte il n'est plus nécessaire de transtyper, ni de construire autant de méthodes que de types différents à stocker.

```

public class ListeGenerique <T>
{
    List<T> liste = new List<T>();
    public void ajouter ( T elt )
    {
        this.liste.Add(elt);
    }
    public T afficheur(int rang)
    {
        return this.liste[rang];
    }
}

```

C'est dans le code source et donc lors de la compilation que le développeur définit le type <T> de l'élément et c'est le compilateur qui vérifie que chaque élément ajouté est bien de type <T> ainsi que chaque élément affiché est de type <T> :

L'instruction qui suit permet d'instancier à partir de la classe `ListeGenerique` un objet `listeGenricInt` d'éléments de type `int` par exemple :

```
| ListeGenerique<int> listeGenricInt = new ListeGenerique<int>();
```

On peut instancier à partir de la même classe `ListeGenerique` un autre objet `listeGenricStr` d'éléments de type `string` par exemple :

```
| ListeGenerique< string > listeGenricStr = new ListeGenerique< string >();
```

Le compilateur refusera le mélange des types :

```
class Program
{
    static void Main(string[] args)
    {
        ListeGenerique<int> listeGenricInt = new ListeGenerique<int>();
        listeGenricInt.ajouter ( 32 );
        listeGenricInt.ajouter ( "abcdef" ); ← erreur signalée par le compilateur ici !
        int x;
        for(int i=0; i<=1;i++)
        {
            x = listeGenricInt.afficher (i);
            Console.WriteLine("listeSimpleInt : " + x);
        }
        Console.WriteLine();
    }
}
```

Un programme correct serait :

```
class Program
{
    static void Main(string[] args)
    {
        ListeGenerique<int> listeGenricInt = new ListeGenerique<int>();
        listeGenricInt.ajouter ( 32 );
        listeGenricInt.ajouter ( -58 );
        int x;
        for(int i=0; i<=1;i++)
        {
            x = listeGenricInt.afficher (i);
            Console.WriteLine("listeSimpleInt : " + x);
        }
        Console.WriteLine();
    }
}
```

On procéderait d'une manière identique avec un objet "`ListeGenerique< string > listeGenricStr = new ListeGenerique< string >()`" dans lequel le compilateur n'acceptera que l'ajout d'éléments de type `string`.

1.3 Une méthode avec un paramètre générique

Nous reprenons la construction précédente d'une liste générique :

```
public class ListeGenerique <T>
{
    public List<T> liste = new List<T>();
    public void ajouter ( T elt )
    {
        this.liste.Add(elt);
    }
    public T afficher(int rang)
    {
        return this.liste[rang];
    }
}
```

Nous ajoutons dans la classe Program une méthode générique "**static void** afficher<T>(ListeGenerique<T> objet, **int** rang)" qui reçoit en paramètre formel une liste générique nommée objet de type <T>, nous la chargeons d'afficher l'élément de rang k de la liste et le type dynamique de l'objet obtenu par la méthode GetType(). L'appel de la méthode afficher nécessite deux informations :

- a) le type paramétré de définition de l'objet passé en paramètre effectif, soit ici < **int** > ,
- b) le paramètre effectif lui-même soit ici listeGenricInt.

```
class Program
{
    static void afficher<T>(ListeGenerique<T> objet, int k)
    {
        Console.WriteLine(objet.liste[k] + ", de type : " + objet.liste[k].GetType());
    }
    static void Main(string[] args)
    {
        ListeGenerique<int> listeGenricInt = new ListeGenerique<int>();
        listeGenricInt.ajouter ( 32 );
        listeGenricInt.ajouter ( -58 );
        afficher <int> ( listeGenricInt, 0 );
        Console.WriteLine();
    }
}
```

Résultat obtenu lors de l'exécution sur la console :
32, de type : System.Int32

1.4 Type générique contraint

Il est possible de contraindre un type paramétré à hériter d'une ou plusieurs classes génériques ou non et à implémenter une ou plusieurs interfaces classes génériques ou non en utilisant le mot clef where. Cette contrainte apporte deux avantages au développeur :

- 1°) améliorer la sécurité de vérification du typage lors de la compilation,
- 2°) réduire le transtypage.

Les options de contraintes d'un type paramétré sont les suivantes :

Syntaxe de la contrainte	Signification de la contrainte
where T : struct	Le type T doit être un type valeur.
where T : class	Le type T doit être un type référence.
where T : new()	Le type T doit avoir un constructeur sans paramètre explicite.
where T : < classeType >	Le type T doit hériter de la classe classeType
where T : <interfaceType >	Le type T doit implémenter l'interface interfaceType

Le(s) mot(s) clef **where** doit se situer avant le corps de la classe ou de la méthode.

Syntaxe d'utilisation par l'exemple dans une classe :

```
class clA { .... }
interface IGeneric<T> { .... }

class ClassGeneric<UnType, Autre>
  where UnType : clA, new()
  where Autre : class, IGeneric<UnType>
{
  .....
}
```

Dans l'exemple précédent la classe **ClassGeneric** est paramétrée par les deux types **UnType** et **Autre** qui supportent chacun une série de contraintes :

- ❑ Le type **UnType** est contraint d'hériter de la classe **clA** et doit avoir un constructeur sans paramètre explicite.
- ❑ Le type **Autre** est contraint d'être un type référence et doit implémenter l'interface **IGeneric** avec comme type paramétré <UnType>.

Syntaxe d'utilisation par l'exemple dans une méthode :

```
public T meth1 < T, U >()
  where T : new()
  where U : ClassGeneric <T>
{
  return new T();
}
```

Dans l'exemple précédent la méthode **meth1** est paramétrée par les deux types **T** et **U** qui supportent chacun une série de contraintes (elle renvoie un résultat de type **T**) :

- ❑ Le type **T** est contraint d'avoir un constructeur sans paramètre explicite.
- ❑ Le type **U** est contraint d'hériter de la classe **ClassGeneric** avec <T> comme type paramétré.

1.5 Surcharge générique d'une classe

Les types paramétrés font partie intégrante de l'identification d'une classe aussi bien que le nom de la classe elle-même. A l'instar des méthodes qui peuvent être surchargées, il est possible pour une même classe de disposer de plusieurs surcharges : même nom, types paramètres différents.

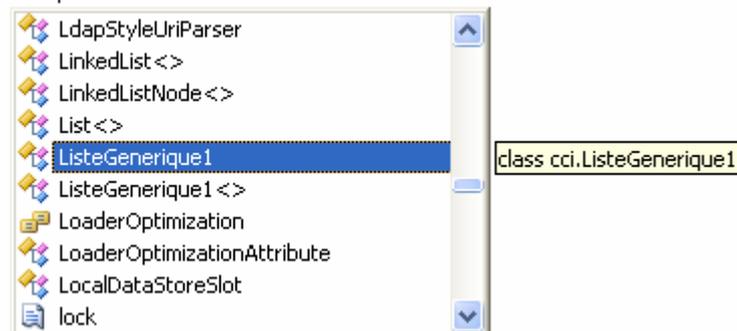
Voici par exemple **trois classes distinctes** pour le compilateur C#, chacune de ces classes n'a aucun rapport avec l'autre si ce n'est qu'elle porte le même nom et qu'elles correspondent chacune à une surcharge différente de la classe `ListeGenerique1` :

```
public class ListeGenerique1<T>
{
}
public class ListeGenerique1
{
}
public class ListeGenerique1<T, U>
{
}
```

Cet aspect comporte pour le développeur non habitué à la surcharge de classe à un défaut apparent de lisibilité largement compensé par l'audit de code dans Visual C#.

Exemple d'instanciation sur la première surcharge générique :

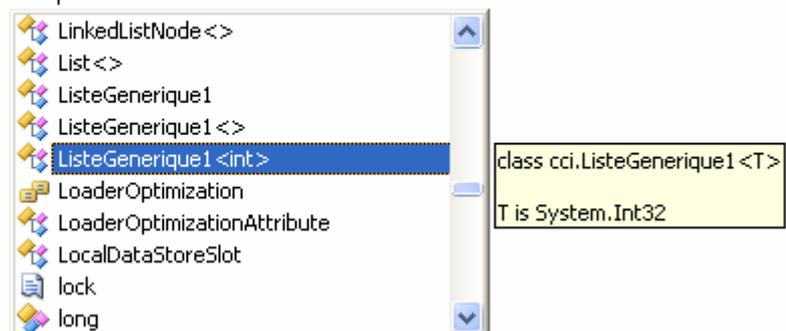
```
ListeGenerique1 obj1 = new |
```



Exemple d'instanciation sur la seconde surcharge générique :

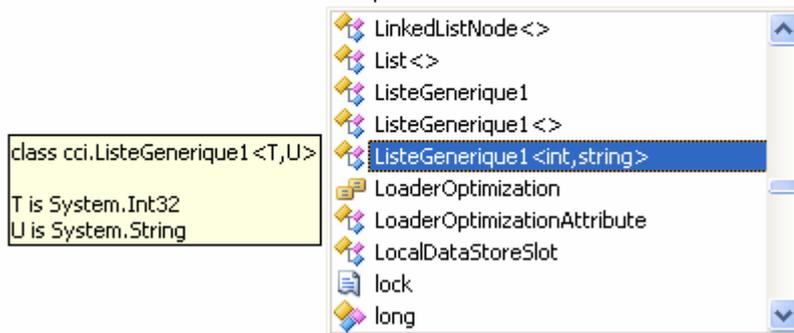
```
ListeGenerique1 obj1 = new ListeGenerique1();
```

```
ListeGenerique1<int> obj2 = new |
```



Exemple d'instanciation sur la troisième surcharge générique :

```
ListeGenerique1 obj1 = new ListeGenerique1();  
ListeGenerique1<int> obj2 = new ListeGenerique1<int>();  
ListeGenerique1<int,string> obj3 = new
```



Les trois instanciations obtenues créent trois objets obj1 , obj2 , obj3 différents et de classe différente :

```
ListeGenerique1 obj1 = new ListeGenerique1();  
ListeGenerique1<int> obj2 = new ListeGenerique1<int>();  
ListeGenerique1<int, string> obj3 = new ListeGenerique1<int, string>();
```

On peut définir une surcharge générique d'une classe à partir d'une **autre surcharge générique de la même classe** :

```
public class ListeGenerique1<T, U>  
{  
}  
public class ListeGenerique1<T> : ListeGenerique1<T, int>  
{  
}
```

Il est bien entendu possible de définir des surcharges génériques d'une classe à partir d'**autres surcharges génériques d'une autre classe** :

```
public class ListeGenerique1<T, U> { ... }  
public class ListeGenerique1<T> : ListeGenerique1<T, int> { ... }  
public class ListeGenerique2<U> : ListeGenerique1<string, U> { ... }  
public class ListeGenerique2 : ListeGenerique1<string, int> { ... }  
public class ListeGenerique2<T,U> : ListeGenerique1<T> { ... }
```

Les types partiels



Plan général: 

1. Les types partiels

- 1.1 Déclaration de classe partielle
- 1.2 classe partielle : héritage, implémentation et imbrication
- 1.3 classe partielle : type générique et contraintes

1. types partiels

Depuis la version 2.0, C# accepte la définition de struct, de classe, d'interface séparées. Ce qui revient à pouvoir définir à plusieurs endroits distincts dans un même fichier, un des trois types précédents ou encore le même type peut voir sa définition répartie sur plusieurs fichiers séparés. Cette notion de classe partielle est largement utilisée depuis 2.0 par Visual C# avec les WinForms.

Pour permettre ce découpage en plusieurs morceaux d'un même type, il faut obligatoirement utiliser dans la déclaration et juste avant la caractérisation du type (**struct**, **class**, **interface**), le modificateur **partial**.

Syntaxe:

```
partial class Truc1 { ..... }
partial struct Truc2 { ..... }
partial interface Truc3 { ..... }
```

Les autres modificateurs définissant les qualités du type (**static**, **public**, **internal**, **sealed**, **abstract**) doivent se trouver avant le mot clef **partial** :

```
static public partial class Truc1 { ..... }
internal partial struct Truc2 { ..... }
public partial interface Truc3 { ..... }
```

1.1 Déclaration de classe partielle

Nous étudions dans ce paragraphe, les implications de la déclaration d'une classe partielle sur son utilisation dans un programme.

soit un fichier *partie1.cs* contenant la classe **ClassPartielle** possédant un attribut entier public **y** initialisé à 200 dans l'espace de nom **cci** :

```
namespace cci
{
    partial class ClassPartielle
    {
        public int y = 200;
    }
}
```

On peut définir dans un autre fichier *partie2.cs* la "suite" de la classe **ClassPartielle** avec un autre attribut **public** **x** initialisé à 100 dans l'espace de même nom **cci** :

```

namespace cci
{
    partial class ClassPartielle
    {
        public int x = 100;
    }
}

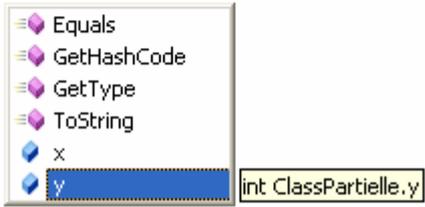
```

Ajoutons dans le code du second fichier *partie2.cs*, une classe **Program** contenant la méthode **Main**, l'audit de code de C#, nous montre bien qu'un objet de type **ClassPartielle**, possède bien les deux attributs x et y :

```

namespace cci
{
    partial class ClassPartielle
    {
        public int x = 100;
    }
    class Program
    {
        static void Main(string[] args)
        {
            ClassPartielle Obj = new ClassPartielle();
            Obj.
        }
    }
}

```



3 déclarations partielles de la même classe	Implications sur la classe
<pre> partial class ClassPartielle { public int x = 100; } sealed partial class ClassPartielle { public int y = 200; } internal partial class ClassPartielle { public int z = 300; } </pre>	<p>La classe est considérée par le compilateur comme possédant la réunion des deux qualificatifs sealed et internal :</p> <pre> sealed internal partial class ClassPartielle { } </pre>

Attention le compilateur C# vérifie la cohérence entre tous les déclarations différentes des qualificateurs d'une même classe partielle. Il déclençera par exemple une erreur dans les

déclarations suivantes :

```
public partial class ClassPartielle {  
    public int x = 100;  
}  
....  
internal partial class ClassPartielle {  
    public int y = 200;  
}
```

Dans l'exemple ci-dessus, nous aurons le message d'erreur du compilateur C# suivant :
Error : Les déclarations partielles de 'ClassPartielle' ont des modificateurs d'accessibilité en conflit.

Conseil : pour maintenir une bonne lisibilité du programme mettez tous les qualificatifs devant chacune des déclarations de classe partielle.

```
sealed internal partial class ClassPartielle {  
    public int x = 100;  
}  
....  
sealed internal partial class ClassPartielle {  
    public int y =  
}  
....etc
```

Conseil : ne pas abuser du concept partial, car bien que pratique, la dissémination importante des membres d'une classe dans plusieurs fichiers peut nuire à la lisibilité du programme !

1.2 classe partielle : héritage, implémentation et imbrication

Héritage de classe

Si une classe partielle `ClassPartielle` hérite dans l'une de ses définitions d'une autre classe `ClasseA` partielle ou non, cet héritage s'étend implicitement à toutes les autres définitions de `ClassPartielle` :

```
public class ClasseA {  
    ....  
}  
public partial class ClassPartielle : ClasseA {  
    public int x = 100;  
}....  
public partial class ClassPartielle {  
    public int y = 200;  
}....  
public partial class ClassPartielle {  
    public int z = 300;  
}....
```

Conseil : pour maintenir une bonne lisibilité du programme mettez la qualification d'héritage dans chacune des déclarations de la classe partielle.

```

public class ClasseA {
....
}
public partial class ClassPartielle : ClasseA {
    public int x = 100;
}
public partial class ClassPartielle : ClasseA {
    public int y = 200;
}
public partial class ClassPartielle : ClasseA {
    public int z = 300;
}

```

Implémentation d'interface

Pour une classe partielle implémentant une ou plusieurs interfaces le comportement est identique à celui d'une classe partielle héritant d'une autre classe. Vous pouvez écrire :

```

public interface InterfA { ... }
public interface InterfB { ... }

public partial class ClassPartielle : InterfA {
    public int x = 100;
}....
public partial class ClassPartielle {
    public int y = 200;
}....
public partial class ClassPartielle : InterfB {
    public int z = 300;
}....

```

Ecriture plus lisible conseillée :

```

public interface InterfA { ... }
public interface InterfB { ... }

public partial class ClassPartielle : InterfA , InterfB {
    public int x = 100;
}....
public partial class ClassPartielle : InterfA , InterfB {
    public int y = 200;
}....
public partial class ClassPartielle : InterfA , InterfB {
    public int z = 300;
}....

```

Classe imbriquée

Les classes imbriquées de C# peuvent être définies sous forme de classes partielles.

Exemple d'imbrication dans une classe non partielle :

```

class classeMere
{
    partial class classeA

```

```

{
}
.....
partial class classeA
{
}
}

```

imbrication dans une classe partielle :

```

partial class classeMere
{
    partial class classeA
    {
    }
    .....
    partial class classeA
    {
    }
}
.....
partial class classeMere
{
    partial class classeA
    {
    }
    .....
    partial class classeA
    {
    }
}
}

```

1.3 classe partielle : type générique et contraintes

Une classe générique peut être partielle

soit la classe de liste générique définie plus haut mise sous forme de 3 définitions partielles :

Première définition partielle :

```

public partial class ListeGenerique <T>
{
    List<T> liste = new List<T>();
}

```

Seconde définition partielle :

```

public partial class ListeGenerique <T>
{
    public void ajouter ( T elt )
    {
        this.liste.Add(elt);
    }
}

```

Troisième définition partielle :

```
public partial class ListeGenerique <T>
{
    public T afficher(int rang)
    {
        return this.liste[rang];
    }
}
```

Notons que le ou les types génériques paramétrant la classe partielle doivent obligatoirement être présents, sauf à respecter les conventions de la surcharge générique de classe vue plus haut.

Une classe partielle peut avoir des surcharges génériques partielles

Dans ce cas aussi nous ne saurions que trop conseiller au lecteur d'écrire explicitement dans toutes les définitions partielles d'une même surcharge, les mêmes listes de types paramétrés de la définition.

Notre exemple ci-dessous montre trois surcharges génériques d'une classe `ListeGenerique1` : soit `ListeGenerique1`, `ListeGenerique1<T>` et `ListeGenerique1<T, U>`, chaque surcharge générique est déclarée sous forme de deux définitions de classes partielles.

```
public partial class ListeGenerique1<T>
{
}
public partial class ListeGenerique1<T>
{
}
public partial class ListeGenerique1
{
}
public partial class ListeGenerique1
{
}
public partial class ListeGenerique1<T, U>
{
}
public partial class ListeGenerique1<T, U>
{
}
```

Types génériques contraints dans les classes partielles

Comme il est illisible de jongler avec les exclusions possibles ou non sur la position des différents paramètres de la clause **where** dans plusieurs déclarations d'une même classe partielle, le langage C# **exige de déclarer explicitement** la (les) même(s) clause(s) **where** dans chaque déclaration de la classe partielle :

Première déclaration partielle :

```
public class ClasseA { ... }
public interface InterfB { ... }
```

```
public partial class ListeGenerique <T>  
where T : ClasseA , InterfB , new()  
{  
    List<T> liste = new List<T>();  
}
```

Seconde déclaration partielle :

```
public partial class ListeGenerique <T>  
where T : ClasseA , InterfB , new()  
{  
    public void ajouter ( T elt )  
    {  
        this.liste.Add(elt);  
    }  
}
```

Troisième déclaration partielle :

```
public partial class ListeGenerique <T>  
where T : ClasseA , InterfB , new()  
{  
    public T afficher(int rang)  
    {  
        return this.liste[rang];  
    }  
}
```

Les méthodes anonymes



Plan général: 

1. Les méthodes anonymes

- 1.1 délégué et méthode anonyme
- 1.2 Création pas à pas d'une méthode anonyme
- 1.3 Gestionnaire anonyme d'événement classique sans information
- 1.4 Gestionnaire anonyme d'événement personnalisé avec information
- 1.5 Les variables capturées par une méthode anonyme
- 1.6 Les méthodes anonymes sont implicitement de classe ou d'instance
- 1.7 Comment les méthodes anonymes communiquent entre elles

1. Les méthodes anonymes

Le concept de méthode anonyme dans C# est semblable au concept de classe anonyme en Java très utilisé pour les écouteurs.

L'objectif est aussi dans C# de réduire les lignes de code superflues tout en restant lisible. Les méthodes anonymes en C# sont intimement liées aux délégués.

1.1 Délégué et méthode anonyme

Là où le développeur peut mettre un objet délégué, il peut aussi bien mettre une méthode anonyme. Lorsque dans le programme, il n'est pas nécessaire de connaître le nom de la méthode vers laquelle pointe un délégué, les méthodes anonymes sont alors un bon outil de simplification du code.

Prenons un exemple d'utilisation classique d'un délégué censé permettre fictivement de pointer vers des méthodes de recherche d'un nom dans une liste. Rappelons la démarche générale pour la manipulation de ce concept :

```
// 1°) la classe de délégué
delegate string DelegListe(int rang);

class ClassMethodeAnonyme
{
    // 2°) la méthode vers laquelle va pointer le délégué
    private string searchNom1(int x)
    {
        return "found n° "+Convert.ToString(x);
    }
    public void utilise()
    {
        // 3°) création d'un objet délégué pointant vers la méthode
        DelegListe search1 = new DelegListe(searchNom1);
        Console.WriteLine(search1(1));
    }
}
```

Reprenons les mêmes éléments mais avec une méthode anonyme :

```
// 1°) la classe de délégué
delegate string DelegListe(int rang);

class ClassMethodeAnonyme
{
    // 2°) la méthode anonyme vers laquelle pointe le délégué
    private DelegListe searchNom2 = delegate ( int x )
    {
        return "found n° "+Convert.ToString(x);
    };
}
```

```

public void utilise()
{
    Console.WriteLine(searchNom2 (2));
}
}

```

1.2 Création pas à pas d'une méthode anonyme

Explicitons la démarche ayant conduit à l'écriture du remplacement dans le premier paragraphe précédent des étapes 2° et 3° par la seule étape 2° avec une méthode anonyme dans le second paragraphe.

1°) Une méthode anonyme est déclarée par le mot clef général **delegate** avec son corps de méthode complet :

```

... = delegate ( int x )
{
    return "found n° "+Convert.ToString(x);
};

```

2°) On utilise une référence de **delegate** , ici searchNom2 de type **DelegListe** :

```

private DelegListe searchNom2;

```

3°) Une méthode anonyme possède la signature de la classe **delegate** avec laquelle on souhaite l'utiliser (ici **DelegListe**) directement par l'affectation à une référence de **delegate** appropriée :

```

private DelegListe searchNom2 = delegate ( int x )
{
    return "found n° "+Convert.ToString(x);
};

```

Si le lecteur souhaite exécuter l'exemple complet, voici le code de la méthode Main de la classe principale affichant les résultats fictifs :

```

class Program
{
    static void Main(string[] args)
    {
        ClassMethodeAnonyme obj = new ClassMethodeAnonyme();
        obj.utilise();
        Console.ReadLine();
    }
}

```

1.3 Gestionnaire anonyme d'événement classique sans information

Reprenons un exemple de construction et d'utilisation d'un **événement classique sans information** comme le click sur le bouton dans la fenêtre ci-dessous, et utilisons un gestionnaire d'événements anonyme.



La démarche classique (prise en charge automatiquement dans les environnements Visual Studio, Borland Studio, sharpDevelop) est la suivante :

- ❑ Créer et faire pointer l'événement Click (qui est un **delegate**) vers un nom de gestionnaire déjà défini (ici `button1_Click`) :

```
this.button1.Click += new System.EventHandler(this.button1_Click);
```

- ❑ Définir le gestionnaire `button1_Click` selon la signature du **delegate** :

```
private void button1_Click(object sender, EventArgs e)
{
    textBox1.Text = "click sur bouton-1";
}
```

L'utilisation d'un **gestionnaire anonyme** réduit ces deux instructions à une seule et au même endroit :

```
this.button1.Click += delegate (object sender, System.EventArgs e)
{
    textBox1.Text = "click sur bouton-1";
};
```

1.4 Gestionnaire anonyme d'événement personnalisé avec information

Soit un exemple d'utilisation de 4 gestionnaires anonymes d'un même événement personnalisé avec information appelé **Enlever**. Pour la démarche détaillée de création d'un tel événement que nous appliquons ici, nous renvoyons le lecteur au chapitre "Evénements" de cet ouvrage.

Nous déclarons une classe d'informations sur un événement personnalisé :

```
| public class EnleverEventArgs : EventArgs { ... }
```

Nous déclarons la classe delegate de l'événement personnalisé :

```
public delegate void DelegateEnleverEventHandler (object sender,  
EnleverEventArgs e);
```

Nous donnons le reste sans explication supplémentaire autre que les commentaires inclus dans le code source,

```
using System;  
using System.Collections;  
  
namespace cci {  
  
    //--> 1°) classe d'informations personnalisées sur l'événement  
    public class EnleverEventArgs : EventArgs  
    {  
        public string info;  
        public EnleverEventArgs(string s)  
        {  
            info = s;  
        }  
    }  
  
    //--> 2°) déclaration du type délégation normalisé  
    public delegate void DelegateEnleverEventHandler(object sender, EnleverEventArgs e);  
  
    public class ClassA  
    {  
        //--> 3°) déclaration d'une référence event de type délégué :  
        public event DelegateEnleverEventHandler Enlever;  
        //--> 4.1°) méthode protégée qui déclenche l'événement :  
        protected virtual void OnEnlever(object sender, EnleverEventArgs e)  
        {  
            //....  
            if (Enlever != null) Enlever(sender, e);  
            //....  
        }  
        //--> 4.2°) méthode publique qui lance l'événement :  
        public void LancerEnlever()  
        {  
            //....  
            EnleverEventArgs evt = new EnleverEventArgs("événement déclenché");  
            OnEnlever(this, evt);  
            //....  
        }  
    }  
  
    public class ClasseUse  
    {  
        //--> 5°) la méthode permettant l'utilisation des gestionnaires anonymes  
        static public void methodUse()  
        {  
            ClassA ObjA = new ClassA();  
            ClasseUse ObjUse = new ClasseUse();  
  
            //--> 6°) abonnement et définition des 4 gestionnaires anonymes :  
            ObjA.Enlever += delegate(object sender, EnleverEventArgs e)  
            {  
                //...gestionnaire d'événement Enlever: méthode d'instance.  
                System.Console.WriteLine("information utilisateur 100 : " + e.info);  
            };  
        }  
    }  
}
```

```

ObjA.Enlever += delegate(object sender, EnleverEventArgs e)
{
    //...gestionnaire d'événement Enlever: méthode d'instance.
    System.Console.WriteLine("information utilisateur 101 : " + e.info);
};
ObjA.Enlever += delegate(object sender, EnleverEventArgs e)
{
    //...gestionnaire d'événement Enlever: méthode d'instance.
    System.Console.WriteLine("information utilisateur 102 : " + e.info);
};
ObjA.Enlever += delegate(object sender, EnleverEventArgs e)
{
    //...gestionnaire d'événement Enlever: méthode de classe.
    System.Console.WriteLine("information utilisateur 103 : " + e.info);
};

//--> 7°) consommation de l'événement:
ObjA.LancerEnlever(); //...l'appel à cette méthode permet d'invoquer l'événement Enlever
}
static void Main(string[] args)
{
    ClasseUse.methodUse();
    Console.ReadLine();
}
}
}

```

Résultats d'exécution du programme précédent avec gestionnaires anonymes :

```

C:\D:\CsBuilder\EventPerso\bin\Debug\ProjEventPerso.exe
information utilisateur 100 : événement déclenché
information utilisateur 101 : événement déclenché
information utilisateur 102 : événement déclenché
information utilisateur 103 : événement déclenché

```

1.5 Les variables capturées par une méthode anonyme

Une méthode anonyme accède aux données locales du bloc englobant dans lequel elle est définie.

- ❑ Les variables locales utilisables par une méthode anonyme sont appelées **variables externes**.
- ❑ Les variables utilisées par une méthode anonyme sont appelées les **variables externes capturées**.

Exemple :

```

int y = 100; // variable externe au bloc anonyme

.... = delegate (int x)
{
    y = x - 3; // variable externe x capturée par le bloc anonyme
};

....

```

Tous type de variables (valeur ou référence) peut être capturé par un bloc anonyme.

Le bloc englobant peut être une méthode :

```
delegate string DelegateListe(int rang);
.....
public void utilise ()
{
    int entier = 100; //...variable locale type valeur
    object Obj = null; //... variable locale type référence

    DelegateListe searchNum = delegate(int x)
    {
        entier = 99; //...variable capturée
        Obj = new object (); //...variable capturée
        return "found n° " + Convert.ToString(x);
    };
    Console.WriteLine("static : " + searchNum(2));
}
```

Le bloc englobant peut être une classe :

```
delegate string DelegateListe(int rang);

class ClasseA
{
    static int entierStatic = 100; //...champ de classe type valeur
    int entierInstance = 200; //...champ d'instance type valeur
    object Obj = null; //... champ d'instance type référence

    public void utiliseMembre ()
    {
        DelegateListe searchNum = delegate(int x)
        {
            entierStatic++; //...variable capturée
            entierInstance++; //...variable capturée
            Obj = new object (); //...variable capturée
            return "found n° " + Convert.ToString(x);
        };
        Console.WriteLine( searchNum(1) );
    }
}
```

Il est bien sûr possible de combiner les deux genres de variables capturées, soit local, soit membre.

1.6 Les méthode anonymes sont implicitement de classe ou d'instance

Une méthode anonyme peut être implicitement de classe ou bien implicitement d'instance **uniquement**, selon que le délégué qui pointe vers la méthode est lui-même dans une méthode de classe ou une méthode d'instance.

Le fait que le délégué qui pointe vers une méthode anonyme soit explicitement **static**, n'induit pas que la méthode anonyme soit de classe; c'est la méthode englobant la méthode anonyme qui impose son modèle **static** ou d'instance.

a) Exemple avec **méthode englobante d'instance** et **membre délégué d'instance** explicite :

```
delegate string DelegateListe(int rang);

class ClasseA
{
    static int entierStatic = 100; //... membre de classe type valeur
    int entierInstance = 200; //... membre d'instance type valeur
    object Obj = null; //... membre d'instance type référence
    DelegateListe searchNum; //...membre délégué d'instance

    public void utiliseMembre ()
    {
        searchNum = delegate(int x)
        {
            entierStatic++; //...variable de classe capturée
            entierInstance++; //...variable d'instance capturée
            Obj = new object (); //...variable d'instance capturée
            return "found n° " + Convert.ToString(x);
        };
        Console.WriteLine( searchNum(1) );
    }
}
```

b) Exemple avec **méthode englobante d'instance** et **membre délégué de classe** explicite :

```
delegate string DelegateListe(int rang);

class ClasseA
{
    static int entierStatic = 100; //... membre de classe type valeur
    int entierInstance = 200; //... membre d'instance type valeur
    object Obj = null; //... membre d'instance type référence
    static DelegateListe searchNum; //...membre délégué de classe

    public void utiliseMembre ()
    {
        searchNum = delegate(int x)
        {
            entierStatic++; //...variable de classe capturée
            entierInstance++; //...variable d'instance capturée
            Obj = new object (); //...variable d'instance capturée
            return "found n° " + Convert.ToString(x);
        };
        Console.WriteLine( searchNum(2) );
    }
}
```

c) Exemple avec **méthode englobante de classe** et **membre délégué de classe** explicite :

```
delegate string DelegateListe(int rang);
```

```

class ClasseA
{
    static int entierStatic = 100; //... membre de classe type valeur
    int entierInstance = 200; //... membre d'instance type valeur
    object Obj = null; //... membre d'instance type référence
    static DelegateListe searchNum; //...membre délégué de classe

    public static void utiliseMembre ( )
    {
        searchNum = delegate(int x)
        {
            entierStatic++; //...variable static capturée
            entierInstance++; ← erreur de compilation membre d'instance non autorisé
            Obj = new object ( ); ← erreur de compilation membre d'instance non autorisé
            return "found n° " + Convert.ToString(x);
        };
        Console.WriteLine( searchNum(3) );
    }
}

```

d) Exemple avec **méthode englobante de classe** et **membre délégué d'instance** explicite :

```

delegate string DelegateListe(int rang);

class ClasseA
{
    static int entierStatic = 100; //... membre de classe type valeur
    int entierInstance = 200; //... membre d'instance type valeur
    object Obj = null; //... membre d'instance type référence
    DelegateListe searchNum; //...membre délégué d'instance

    public static void utiliseMembre ( )
    {
        searchNum = delegate(int x) ← erreur de compilation le membre doit être static
        {
            entierStatic++; //...variable static capturée
            entierInstance++; ← erreur de compilation membre d'instance non autorisé
            Obj = new object ( ); ← erreur de compilation membre d'instance non autorisé
            return "found n° " + Convert.ToString(x);
        };
        Console.WriteLine( searchNum(4) );
    }
}

```

Nous remarquons dans les exemples précédents, que le délégué **searchNum** qui pointe vers la méthode anonyme **delegate(int x) { ... }** possède bien les mêmes caractéristiques que la méthode englobante **utiliseMembre ()**. C'est pourquoi dans les exemples (c) et (d) **searchNum** est implicitement **static** comme **utiliseMembre ()** et donc les erreurs signalées par le compilateur dans la méthode anonyme, sont des erreurs classiques commises sur une méthode qui est **static** et qui ne peut accéder qu'à des entités elles-mêmes **static**.

1.7 Comment les méthode anonymes communiquent entre elles

La notion de variable externe capturée par une méthode anonyme permet à plusieurs méthodes anonymes déclarées dans le même bloc englobant de partager et donc capturer les mêmes variables externes.

Ces variables sont donc comme des variables communes à toutes les méthodes anonymes qui peuvent ainsi communiquer entre elles comme dans l'exemple ci-dessous où deux méthodes anonymes pointées l'une par le délégué **searchNum**, l'autre par le délégué **combienDeSearch** partagent les mêmes variables **numero** (variable locale au bloc englobant) et **nbrDeRecherche** membre d'instance de la classe :

```
class ClasseA
{
    int nbrDeRecherche = 0; //... membre d'instance type valeur
    DelegateListe searchNum; //...membre délégué d'instance
    DelegateConsulter combienDeSearch; //...membre délégué d'instance

    public void utiliseMembre()
    {
        int numero = 12345;
        searchNum = delegate(int x)
        {
            nbrDeRecherche++; //...variable d'instance capturée
            numero++; //...variable locale capturée
            return "found n° " + Convert.ToString(x);
        };
        combienDeSearch = delegate()
        {
            return "nombre de recherches effectuées : " + Convert.ToString(nbrDeRecherche)
                + " , numéro : " + Convert.ToString(numero);
        };
        Console.WriteLine( searchNum(50) );
        Console.WriteLine( combienDeSearch( ) );
    }
}
```

Résultats obtenu lors de l'exécution sur la console :

```
found n° 50
nombre de recherches effectuées : 1 , numéro : 12346
```

Attention : une méthode anonyme ne s'exécute que lorsque le délégué qui pointe vers elle est lui-même invoqué, donc les actions effectuées sur des variables capturées ne sont effectives que lors de l'invocation du délégué.

Si nous reprenons la classe précédente et que nous reportons l'initialisation "**numero = 12345;**" de la variable locale **numero** dans la première méthode anonyme, nous aurons un message d'erreur du compilateur :

```
class ClasseA
{
    int nbrDeRecherche = 0; //... membre d'instance type valeur
    DelegateListe searchNum; //...membre délégué d'instance
    DelegateConsulter combienDeSearch; //...membre délégué d'instance
```

```

public void utiliseMembre()
{
    int numero; //...variable locale non initialisée
    searchNum = delegate(int x)
    {
        nbrDeRecherche++; //...variable d'instance capturée
        numero= 12345; //...variable locale capturée et initialisée
        return "found n° " + Convert.ToString(x);
    };
    combienDeSearch = delegate()
    {
        return "nombre de recherches effectuées : " + Convert.ToString(nbrDeRecherche)
            + " , numéro : "
            + Convert.ToString(numero); ← erreur de compilation variable numero non initialisée !
    };
    Console.WriteLine( searchNum(50) );
    Console.WriteLine( combienDeSearch( ) );
}
}

```

Types abstraits de données , implantation avec classes génériques



Plan du chapitre: 

1. Types abstraits de données et classes génériques en C#

- 1.1 Traduction générale TAD → classes
- 1.2 Exemples de Traduction TAD → classes
- 1.3 Variations sur les spécifications d'implantation
- 1.4 Exemples d'implantation de la liste linéaire
- 1.5 Exemples d'implantation de la pile LIFO
- 1.6 Exemples d'implantation de la file FIFO

1. Types abstraits de données et classes en C#

Dans cette partie nous adoptons un point de vue pratique dirigé par l'implémentation sous une forme accessible à un débutant des notions de type abstrait de donnée.

Nous allons proposer une écriture des TAD avec des classes C# :

- La notion classique de **classe**, contenue dans tout langage orienté objet, se situe au niveau 2 de cette hiérarchie **constitue une meilleure approche de la notion de TAD**.

En fait un TAD sera bien décrit par les membres **public** d'une **classe** et se traduit presque immédiatement ; le travail de traduction des préconditions est à la charge du développeur et se trouvera dans le corps des méthodes.

1.1 Traduction générale TAD → classe C#

Nous proposons un tableau de correspondance pratique entre la signature d'un TAD et les membres d'une classe :

<i>syntaxe du TAD</i>	<i>squelette de la classe associée</i>
<u>TAD</u> Truc	class Truc {
<u>utilise</u> TAD ₁ , TAD ₂ ,...,TAD _n	Les classes TAD ₁ ,...,TAD _n sont déclarées dans le même espace de nom que truc (sinon le global par défaut)
<u>champs</u>	attributs
<u>opérations</u> Op1 : E x F → G Op2 : E x F x G → H x S	public G Op1(E x, F y) {} public void Op2(E x, F y, G, ref H t, ref S u) { }
<u>FinTAD</u> -Truc	}

Reste à la charge du programmeur l'écriture du code dans le corps des méthodes Op1 et Op2

1.2 Exemples de Traduction TAD → classe C#

Le TAD Booléens implanté sous deux spécifications concrètes en C# avec deux types scalaires différents.

Spécification opérationnelle concrète n°1

Les constantes du type Vrai, Faux sont représentées par deux attributs de type entier dans un type structure nommé « logique » :

```
public struct logique
{
    static public int Faux = 0;
    static public int Vrai = 1;
}
```

Voici l'interface de la unit traduite et le TAD :

<p>TAD : Booléens Champs : Opérations : Vrai : → Booléens Faux : → Booléens Et : Booléens x Booléens → Booléens Ou : Booléens x Booléens → Booléens Non : Booléens → Booléens</p> <p>FINTAD-Booléens</p>	<pre>class Booleens { public logique val; public Booleens Et (Booleens x, Booleens y) { } public Booleens Ou (Booleens x, Booleens y) { } public Booleens Non (Booleens x) { } }</pre>
---	---

Spécification opérationnelle concrète n°2

Les constantes du type Vrai, Faux sont représentées par deux identificateurs C# dans un type énuméré nommé « logique » :

```
public enum logique { Faux, Vrai };
```

Voici l'interface de la unit traduite et le TAD :

<p>TAD : Booléens Champs : Opérations : Vrai : → Booléens Faux : → Booléens Et : Booléens x Booléens → Booléens Ou : Booléens x Booléens → Booléens Non : Booléens → Booléens</p> <p>FINTAD-Booléens</p>	<pre>class Booleens { public logique val ; public Booleens Et (Booleens x, Booleens y) { } public Booleens Ou (Booleens x, Booleens y) { } public Booleens Non (Booleens x) { } }</pre>
---	--

Nous remarquons la forte similarité des deux spécifications concrètes :

Implantation avec des entiers	Implantation avec des énumérés
<pre>public struct logique { static public int Faux = 0; static public int Vrai = 1; }</pre>	<pre>public enum logique = (faux , vrai) ;</pre>

<pre> class Booleens { public int val ; public Booleens (int init) { val = init; } public Booleens Et (Booleens x, Booleens y) { } public Booleens Ou (Booleens x, Booleens y) { } public Booleens Non (Booleens x) { } } </pre>	<pre> class Booleens { public logique val ; public Booleens (logique init) { val = init; } public Booleens Et (Booleens x, Booleens y) { } public Booleens Ou (Booleens x, Booleens y) { } public Booleens Non (Booleens x) { } } </pre>
---	---

1.3 Variations sur les spécifications d'implantation

Cet exercice ayant été proposé à un groupe d'étudiants, nous avons eu plusieurs genres d'implantation des opérations : « et », « ou », « non ». Nous exposons au lecteur ceux qui nous ont paru être les plus significatifs :

Implantation d'après spécification concrète n°1

Fonction Et	Fonction Et
<pre> public Booleens Et (Booleens x, Booleens y) { return x.val * y.val ; } </pre>	<pre> public Booleens Et (Booleens x, Booleens y) { if (x.val == logique.Faux) return new Booleens (logique.Faux); else return new Booleens (y.val); } </pre>

Fonction Ou	Fonction Ou
<pre> public Booleens Ou (Booleens x, Booleens y) { return x.val +y.val - x.val *y.val ; } </pre>	<pre> public Booleens Ou (Booleens x, Booleens y) { if (x.val == logique.Vrai) return new Booleens (logique.Vrai); else return new Booleens (y.val); } </pre>

Fonction Non	Fonction Non
<pre> public Booleens Non (Booleens x) { return 1-x.val ; } </pre>	<pre> public Booleens Non (Booleens x) { if (x.val == logique.Vrai) return new Booleens (logique.Vrai); else return new Booleens (logique.Faux); } </pre>

Dans la colonne de gauche de chaque tableau, l'analyse des étudiants a été dirigée par le choix de la spécification concrète sur les entiers et sur un modèle semblable aux fonctions indicatrices des ensembles. Ils ont alors cherché des combinaisons simples d'opérateurs sur les entiers fournissant les valeurs adéquates.

Dans la colonne de droite de chaque tableau, l'analyse des étudiants a été dirigée dans ce cas par des considérations axiomatiques sur une algèbre de Boole. Ils se sont servis des propriétés d'absorption des éléments neutres de la loi " ou " et de la loi " et ". Il s'agit là d'une structure algébrique abstraite.

Influence de l'abstraction sur la réutilisation

A cette étape du travail nous avons demandé aux étudiants quel était, s'il y en avait un, le meilleur choix d'implantation quant à sa réutilisation pour l'implantation d'après la spécification concrète n°2.

Les étudiants ont compris que la version dirigée par les axiomes l'emportait sur la précédente, car sa qualité d'abstraction due à l'utilisation de l'axiomatique a permis de la réutiliser sans aucun changement dans la partie **implementation** de la unit associée à spécification concrète n°2 (en fait toute utilisation des axiomes d'algèbre de Boole produit la même efficacité).

Conclusion :

l'abstraction a permis ici une réutilisation totale et donc un gain de temps de programmation dans le cas où l'on souhaite changer quelle qu'en soit la raison, la spécification concrète.

1.4 Exemples d'implantation de liste linéaire générique

Rappel de l'écriture du TAD de liste linéaire

TAD Liste

utilise : \mathbf{N} , T_0 , Place

Champs : (a_1, \dots, a_n) suite finie dans T_0

opérations :

liste_vide : \rightarrow Liste

acces : Liste \times $\mathbf{N} \rightarrow$ Place

contenu : Place $\rightarrow T_0$

kème : Liste \times $\mathbf{N} \rightarrow T_0$

long : Liste $\rightarrow \mathbf{N}$

supprimer : Liste \times $\mathbf{N} \rightarrow$ Liste

insérer : Liste \times $\mathbf{N} \times T_0 \rightarrow$ Liste

ajouter : Liste $\times T_0 \rightarrow$ Liste

succ : Place \rightarrow Place

préconditions :

acces(L,k) **def ssi** $1 \leq k \leq \text{long}(L)$

supprimer(L,k) **def ssi** $1 \leq k \leq \text{long}(L)$

insérer(L,k,e) **def ssi** $1 \leq k \leq \text{long}(L) + 1$

kème(L,k) **def ssi** $1 \leq k \leq \text{long}(L)$

Fin-Liste

Dans les exemples qui suivent, la notation \cong , indique la traduction en langage C#.

spécification proposée en C# :

Liste \cong	<pre> interface IListe<T0> { int longueur { get; } T0 this[int index] { get; set; } bool est_Vide(); void ajouter(T0 Elt); void inserer(int k, T0 x); void supprimer(int k); bool estPresent(T0 x); int rechercher(T0 x); } class Liste<T0> : IListe<T0> { public static readonly int max_elt = 100; private T0[] t; private int longLoc; public Liste() { t = new T0[max_elt]; longLoc = 0; } // Le reste implante l'interface IListe<T0> } </pre>
liste_vide \cong	Propriété : longueur = 0
acces \cong	Indexeur : this[index]
contenu \cong	Indexeur : this[index]
kème \cong	Indexeur : this[index]
long \cong	Propriété : longueur
succ \cong	Indexeur : this[index]
supprimer \cong	<code>public void supprimer (int k) { }</code>
inserer \cong	<code>public void inserer (int k , T0 Elt) { }</code>

La précondition de l'opérateur **supprimer** peut être ici implantée par le test :

```
if (0 <= k && k <= longueur - 1) .....
```

La précondition de l'opérateur **insérer** peut être ici implantée par le test :

```
if (longueur < max_elt && (0 <= k && k <= longueur) ) .....
```

Les deux objets **accès** et **contenu** ne seront pas utilisés en pratique, car l'indexeur de la classe les implante automatiquement d'une manière transparente pour le programmeur.

Le reste du programme est laissé au soin du lecteur qui pourra ainsi se construire à titre didactique, sur sa machine, une base de types en C# de base, il en trouvera une correction à la fin de ce chapitre.

Nous pouvons "**enrichir**" le TAD Liste en lui adjoignant deux opérateurs test et rechercher (rechercher un élément dans une liste). Ces adjonctions ne posent aucun problème. Il suffit pour cela de rajouter au TAD les lignes correspondantes :

opérations

estPresent : Liste x $T_0 \rightarrow$ Booléen

rechercher : Liste x $T_0 \rightarrow$ Place

précondition

rechercher(L,e) **def ssi** Test(L,e) = V

Le lecteur construira à titre d'exercice l'implantation C# de ces deux nouveaux opérateurs en étendant le programme déjà construit. Il pourra par exemple se baser sur le schéma de représentation C# suivant :

```
public bool estPresent(T0 Elt)
{
    return !(rechercher(Elt) == -1);
}

public int rechercher(T0 Elt)
{
    // il s'agit de fournir le rang de x dans la liste
    // utiliser par exemple un algo de recherche séquentielle
}
```

1.5 Exemples d'implantation de pile LIFO générique

Rappel de l'écriture du TAD Pile LIFO

TAD PILIFO

<p>utilise : T₀, Booléens</p> <p>Champs : (a₁,...,a_n) suite finie dans T₀</p> <p>opérations :</p> <p>sommet : → PILIFO</p> <p>Est_vide : PILIFO → Booléens</p> <p>empiler : PILIFO x T₀ x sommet → PILIFO x sommet</p> <p>dépiler : PILIFO x sommet → PILIFO x sommet x T₀</p> <p>premier : PILIFO → T₀</p> <p>préconditions :</p> <p>dépiler(P) def ssi est_vide(P) = Faux</p> <p>premier(P) def ssi est_vide(P) = Faux</p> <p>FinTAD-PILIFO</p>

Nous allons utiliser un **tableau** avec une case supplémentaire permettant d’indiquer que le fond de pile est atteint (la case 0 par exemple, qui ne contiendra jamais d’élément).

spécification proposée en C# :

Pilifo ≅	<pre>interface ILifo<T0> { int nbrElt { get; } bool estVide(); void empiler(T0 Elt); T0 depiler(); T0 premier(); } class Lifo<T0> : ILifo<T0> { public static readonly int max_elt = 100; private T0[] t; private int sommet, fond; public Lifo() { t = new T0[max_elt]; fond = -1; sommet = fond; // Le reste implante l'interface ILifo<T0> } }</pre>
depiler ≅	public T0 depiler ()
empiler ≅	public T0 empiler (T ₀ Elt)
premier ≅	public T0 premier ()
Est_vide ≅	public bool estVide ()

Le contenu des méthodes est conseillé au lecteur à titre d’exercice, il en trouvera une correction à la fin de ce chapitre..

Remarque :

Il est aussi possible de construire une spécification opérationnelle à l'aide du TAD Liste en remplaçant dans l'étude précédente le mot " tableau " par le mot " liste ". Il est vivement conseillé au lecteur d'écrire cet exercice en C# pour bien se convaincre de la différence entre les niveaux d'abstractions.

1.6 Exemples d'implantation de file FIFO générique

Rappel de l'écriture du TAD file FIFO

TAD FIFO

utilise : T_0 , Booléens

Champs : (a_1, \dots, a_n) suite finie dans T_0

opérations :

tête : \rightarrow FIFO

fin : \rightarrow FIFO

Est_vide : FIFO \rightarrow Booléens

ajouter : FIFO \times T_0 \times fin \rightarrow PILIFO \times fin

retirer : FIFO \times tête \rightarrow FIFO \times tête \times T_0

premier : FIFO \rightarrow T_0

préconditions :

retirer(F) **def ssi** est_vide(F) = **Faux**

premier(F) **def ssi** est_vide(F) = **Faux**

FinTAD-FIFO

Nous allons utiliser ici aussi un **tableau** avec une case supplémentaire permettant d'indiquer que la file est vide (la case 0 par exemple, qui ne contiendra jamais d'élément).

spécification proposée en C# :

Fifo \cong	<pre> interface IFifo<T0> { int nbrElt { get; } bool estVide(); void ajouter(T0 Elt); T0 retirer(); T0 premier(); } class Fifo<T0> : IFifo<T0> { public static readonly int max_elt = 100; private T0[] t; private int tete, fin; public Fifo() { t = new T0[max_elt]; tete = -1; fin = 0; } </pre>
--------------	--

	<pre> } } </pre>
retirer ≅	<code>public T0 retirer ()</code>
ajouter ≅	<code>public void ajouter (T0 Elt)</code>
premier ≅	<code>public T0 premier ()</code>
Est_vide ≅	<code>public bool estVide ()</code>

Le contenu des méthodes est conseillé au lecteur à titre d'exercice, il en trouvera une correction à la fin de ce chapitre..

Remarque :

Comme pour le TAD Pilifo, il est aussi possible de construire une spécification opérationnelle du TAD FIFO à l'aide du TAD Liste en remplaçant dans l'étude précédente l'objet de tableau « `private T0[] t` » par l'objet de liste « `private Liste<T0> t` » .

Une solution d'implantation de liste linéaire générique en C

```
using System;
using System.Collections.Generic;
using System.Text;
```

```
interface IListe<T0>
{
    int longueur
    {
        get;
    }
    T0 this[int index]
    {
        get;
        set;
    }
    bool est_Vide();
    void ajouter(T0 Elt);
    void inserer(int k, T0 x);
    void supprimer(int k);
    bool estPresent(T0 x);
    int rechercher(T0 x);
}
class Liste<T0> : IListe<T0>
{
    public static readonly int max_elt = 100;
    private T0[] t;
    private int longLoc;
    public Liste()
    {
        t = new T0[max_elt];
        longLoc = 0;
    }

    public void ajouter(T0 Elt)
    {
        if (longueur < max_elt)
        {
            t[longueur] = Elt;
            longueur++;
        }
        else
            System.Console.WriteLine("ajout impossible : capacité maximale atteinte !");
    }

    public bool est_Vide()
    {
        return longueur == 0;
    }

    public T0 this[int index]
    {
        get
        {
            if (!est_Vide())
            {
```

```

        if (0 <= index && index <= longueur)
        {
            return t[index];
        }
        else
        {
            System.Console.WriteLine("indexage, lecture incorrecte : indice (" + index + ") hors de la liste.");
            return default(T0);
        }
    }
    else
    {
        System.Console.WriteLine("indexage, lecture incorrecte : Désolé la liste est vide.");
        return default(T0);
    }
}
set {
    if (0 <= index && index <= longueur - 1)
    {
        t[index] = value;
    }
    else
        System.Console.WriteLine("indexage, écriture impossible : indice (" + index + ") hors de la liste.");
}
}

public int longueur
{
    get { return longLoc; }
    protected set { longLoc = value; }
}

public void supprimer(int k)
{
    if (!lest_Vide())
    {
        if (0 <= k && k <= longueur - 1)
        {
            for (int i = k; i < longueur - 1; i++)
                t[i] = t[i + 1];
            longueur--;
        }
        else
            System.Console.WriteLine("suppression impossible : indice (" + k + ") hors de la liste.");
    }
}

public void inserer(int k, T0 Elt)
{
    if (!lest_Vide())
    {
        if (longueur < max_elt && (0 <= k && k <= longueur))
        {
            for (int i = longueur - 1; i >= k; i--)
                t[i + 1] = t[i];
            t[k] = Elt;
            longueur++;
        }
        else
            if (longueur >= max_elt)

```

```

        System.Console.WriteLine("insertion impossible : capacité maximale atteinte !");
    else
        System.Console.WriteLine("insertion impossible : indice (" + k + ") hors de la liste.");
    }
    else
        ajouter(Elt);
}

public bool estPresent(T0 Elt)
{
    return !(rechercher(Elt) == -1);
}

public int rechercher(T0 Elt)
{
    int k;
    for (k = 0; k < longueur; k++)
        if (Elt.Equals(t[k])) break;
    if (k == longueur)
        return -1;
    else
        return k;
}

public void afficher()
{
    for (int k = 0; k < longueur; k++)
        System.Console.Write(t[k] + ", ");
    System.Console.WriteLine(": long = " + longueur);
}
}

```

Exemple de classe principale testant la classe « Liste » sur $T_0 = \mathbf{int}$:

```

class ProgramListe
{
    static void Main(string[] args)
    {
        Liste<int> liste = new Liste<int>();
        System.Console.WriteLine(liste[0]);
        for (int i = 0; i < Liste<int>.max_elt / 10; i++)
            liste.inserer(i, i);
        liste.afficher();
        liste.inserer(20, 97);
        liste.inserer(liste.longueur, 98);
        liste.supprimer(liste.longueur - 1);
        liste.afficher();
        .....
    }
}

```

Une solution d'implantation de pile Lifo générique en C#

```
interface ILifo<T0>
{
    int nbrElt
    {
        get;
    }
    bool estVide();
    void empiler(T0 Elt);
    T0 depiler();
    T0 premier();
}

class Lifo<T0> : ILifo<T0>
{
    public static readonly int max_elt = 100;
    private T0[] t;
    private int sommet, fond;
    public Lifo()
    {
        t = new T0[max_elt];
        fond = -1;
        sommet = fond;
    }
    public int nbrElt
    {
        get { return sommet + 1; }
    }

    public bool estVide()
    {
        return sommet == fond;
    }

    public void empiler(T0 Elt)
    {
        if (sommet < max_elt)
        {
            sommet++;
            t[sommet] = Elt;
        }
        else
            System.Console.WriteLine("empilement impossible : capacité maximale atteinte !");
    }

    public T0 depiler()
    {
        if (!estVide())
        {
            T0 Elt = t[sommet];
            sommet--;
            return Elt;
        }
        else
        {

```

```

        System.Console.WriteLine("dépilement impossible : pile vide !");
        return default(T0);
    }
}

public T0 premier()
{
    if (!estVide())
    {
        return t[sommet];
    }
    else
    {
        System.Console.WriteLine("premier impossible : pile vide !");
        return default(T0);
    }
}

public void afficher()
{
    for (int k = 0; k <= sommet; k++)
        System.Console.Write(t[k] + ", ");
    System.Console.WriteLine(": nbr elmt = " + this.nbrElt);
}
}

```

Exemple de classe principale testant la classe « Lifo » sur $T_0 = \mathbf{int}$:

```

class ProgramListe
{
    static void Main(string[] args)
    {
        Lifo<int> pile = new Lifo<int>();
        pile.afficher();
        pile.depiler();
        System.Console.WriteLine("premier=" + pile.premier());
        for (int i = 0; i < Lifo<int>.max_elt / 10; i++)
            pile.empiler(i);
        pile.afficher();
        System.Console.WriteLine("on dépile : " + pile.depiler());
        System.Console.WriteLine("on dépile : " + pile.depiler());
        pile.afficher();
    }
}

```

Une solution d'implantation de file Fifo générique en C#

```
using System;
using System.Collections.Generic;
using System.Text;

interface IFifo<T0>
{
    int nbrElt
    {
        get;
    }
    bool estVide();
    void ajouter(T0 Elt);
    T0 retirer();
    T0 premier();
}

class Fifo<T0> : IFifo<T0>
{
    public static readonly int max_elt = 100;
    private T0[] t;
    private int tete, fin;

    private void decaleUn()
    {
        if (tete < max_elt)
        {
            tete++;
            for (int k = tete; k >= 0; k--)
                t[k + 1] = t[k];
        }
    }

    public Fifo()
    {
        t = new T0[max_elt];
        tete = -1;
        fin = 0;
    }

    public int nbrElt
    {
        get { return tete + 1; }
    }

    public bool estVide()
    {
        return tete == -1;
    }

    public void ajouter(T0 Elt)
    {
        if (tete < max_elt)
        {
            decaleUn();
            t[fin] = Elt;
        }
    }
}
```

```

        else
            System.Console.WriteLine("ajout impossible : capacité maximale atteinte !");
    }

    public T0 retirer()
    {
        if (!estVide())
        {
            T0 Elt = t[tete];
            tete--;
            return Elt;
        }
        else
        {
            System.Console.WriteLine("ajout impossible : file vide !");
            return default(T0);
        }
    }

    public T0 premier()
    {
        if (!estVide())
        {
            return t[tete];
        }
        else
        {
            System.Console.WriteLine("premier impossible : file vide !");
            return default(T0);
        }
    }

    public void afficher()
    {
        for (int k = fin; k <= tete; k++)
            System.Console.Write(t[k] + ", ");
        System.Console.WriteLine(" : nbr elmt = " + this.nbrElt);
    }
}

class ProgramFifo
{
    static void Main(string[] args)
    {
        Fifo<int> file = new Fifo<int>();
        file.afficher();
        file.retirer();
        System.Console.WriteLine("premier = " + file.premier());
        for (int i = 0; i < Fifo<int>.max_elt / 10; i++)
            file.ajouter(i);
        file.afficher();
        System.Console.WriteLine("on retire : " + file.retirer());
        System.Console.WriteLine("on retire : " + file.retirer());
        file.afficher();
    }
}

```

Exemple de classe principale testant la classe « Fifo » sur $T_0 = \text{int}$:

```

class ProgramFifo
{
    static void Main(string[] args)
    {
        Fifo<int> file = new Fifo<int>();
        file.afficher();
        file.retirer();
        System.Console.WriteLine("premier = " + file.premier());
        for (int i = 0; i < Fifo<int>.max_elt / 10; i++)
            file.ajouter(i);
        file.afficher();
        System.Console.WriteLine("on retire : " + file.retirer());
        System.Console.WriteLine("on retire : " + file.retirer());
        file.afficher();
    }
}

```

Résultats d'exécution sur la console :

```

: nbr elmt = 0
ajout impossible : file vide !
premier impossible : file vide !
premier = 0
9, 8, 7, 6, 5, 4, 3, 2, 1, 0, : nbr elmt = 10
on retire : 0
on retire : 1
9, 8, 7, 6, 5, 4, 3, 2, : nbr elmt = 8

```

Structures d'arbres binaires et classes génériques



Plan du chapitre:

1. Notions générales sur les arbres

1.1 Vocabulaire employé sur les arbres :

- Etiquette Racine, noeud, branche, feuille
- Hauteur, profondeur ou niveau d'un noeud
- Chemin d'un noeud, Noeuds frères, parents, enfants, ancêtres
- Degré d'un noeud
- Hauteur ou profondeur d'un arbre
- Degré d'un arbre
- Taille d'un arbre

1.2 Exemples et implémentation d'arbre

- Arbre de dérivation
- Arbre abstrait
- Arbre lexicographique
- Arbre d'héritage
- Arbre de recherche

2. Arbres binaires

2.1 TAD d'arbre binaire

2.2 Exemples et implémentation d'arbre

- *tableau statique*
- *variable dynamique*
- *classe*

2.3 Arbres binaires de recherche

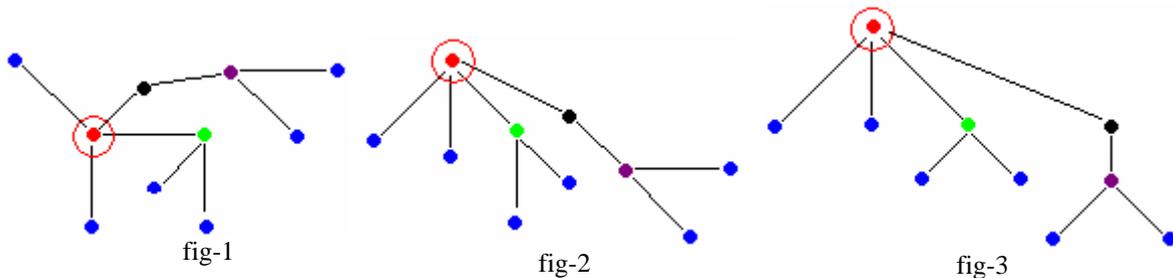
2.4 Arbres binaires partiellement ordonnés (*tas*)

2.5 Parcours en largeur et profondeur d'un arbre binaire

- Parcours d'un arbre
- Parcours en largeur
- Parcours préfixé
- Parcours postfixé
- Parcours infixé
- Illustration d'un parcours en profondeur complet
- Exercice

1. Notions générales sur les structures d'arbres

La structure d'arbre est très utilisée en informatique. Sur le fond on peut considérer un arbre comme une généralisation d'une liste car les listes peuvent être représentées par des arbres. La complexité des algorithmes d'insertion de suppression ou de recherche est généralement plus faible que dans le cas des listes (cas particulier des arbres équilibrés). Les mathématiciens voient les arbres eux-même comme des cas particuliers de graphes non orientés connexes et acycliques, donc contenant des sommets et des arcs :



Ci dessus 3 représentations graphiques de la même structure d'arbre : dans la figure fig-1 tous les sommets ont une disposition équivalente, dans la figure fig-2 et dans la figure fig-3 le sommet "**cerclé**" se distingue des autres.

Lorsqu'un sommet est distingué par rapport aux autres, on le dénomme **racine** et la même structure d'arbre s'appelle une **arborescence**, par abus de langage dans tout le reste du document nous utiliserons le vocable **arbre** pour une **arborescence**.

Enfin certains arbres particuliers nommés arbres binaires sont les plus utilisés en informatique et les plus simples à étudier. En outre il est toujours possible de "**binariser**" un arbre non binaire, ce qui nous permettra dans ce chapitre de n'étudier que les structures d'arbres binaires.

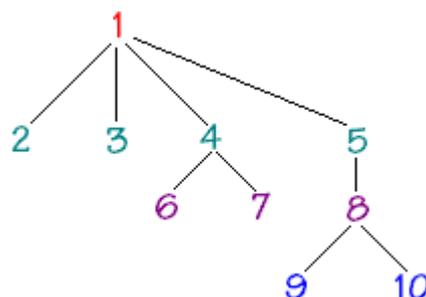
1.1 Vocabulaire employé sur les arbres



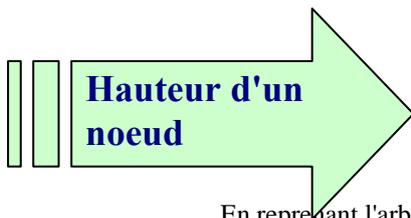
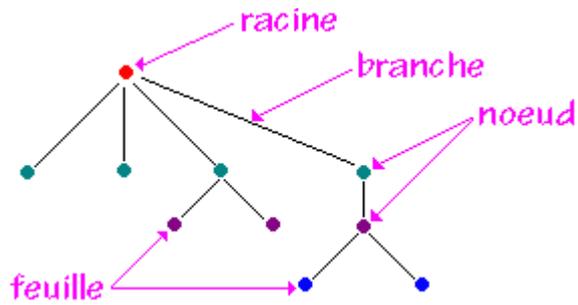
Un arbre dont tous les noeuds sont nommés est dit **étiqueté**. L'étiquette (ou nom du sommet) représente la "valeur" du noeud ou bien l'information associée au noeud.

Ci-dessous un arbre étiqueté dans les entiers entre 1 et 10 :

Ci-dessous un arbre étiqueté dans les entiers entre 1 et 10 :

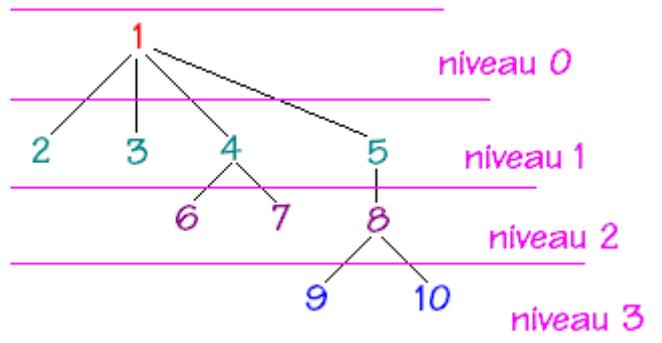


Nous rappelons la terminologie de base sur les arbres:



Nous conviendrons de définir la **hauteur** (ou **profondeur** ou **niveau d'un noeud**) d'un noeud X comme égale **au nombre de noeuds à partir de la racine** pour aller jusqu'au noeud X.

En reprenant l'arbre précédent et en notant **h** la fonction hauteur d'un noeud :



Pour atteindre le noeud étiqueté 9, il faut parcourir le lien 1--5, puis 5--8, puis enfin 8--9 soient 4 noeuds donc 9 est de profondeur ou de hauteur égale à 4, soit $h(9) = 4$.

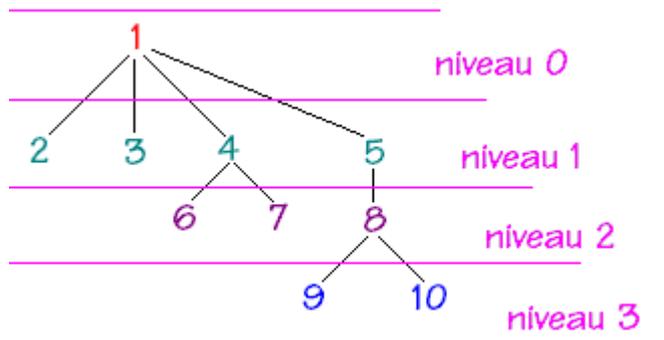
Pour atteindre le noeud étiqueté 7, il faut parcourir le lien 1--4, et enfin 4--7, donc 7 est de profondeur ou de hauteur égale à 3, soit $h(7) = 3$.

Par définition la hauteur de la racine est égal à 1.
 $h(\text{racine}) = 1$ (pour tout arbre non vide)

(Certains auteurs adoptent une autre convention pour calculer la hauteur d'un noeud: la racine a pour hauteur 0 et donc n'est pas comptée dans le nombre de noeuds, ce qui donne une hauteur inférieure d'une unité à notre définition).



On appelle chemin du noeud X la **suite des noeuds** par lesquels il faut passer pour aller de la racine vers le noeud X.



Chemin du noeud 10 = (1,5,8,10)

Chemin du noeud 9 = (1,5,8,9)

.....

Chemin du noeud 7 = (1,4,7)

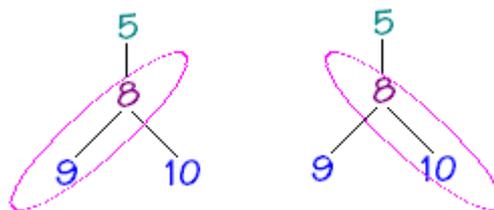
Chemin du noeud 5 = (1,5)

Chemin du noeud 1 = (1)

Remarquons que la hauteur h d'un noeud X est égale au nombre de noeuds dans le chemin :

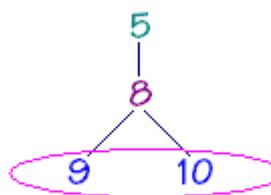
$$h(X) = \text{NbrNoeud}(\text{Chemin}(X)).$$

Le vocabulaire de lien entre noeuds de niveau différents et reliés entre eux est emprunté à la généalogie :



9 est l'enfant de 8
8 est le parent de 9

10 est l'enfant de 8
10 est le parent de 8



noeuds frères

- 9 et 10 sont des frères
- 5 est le parent de 8 et l'ancêtre de 9 et 10.

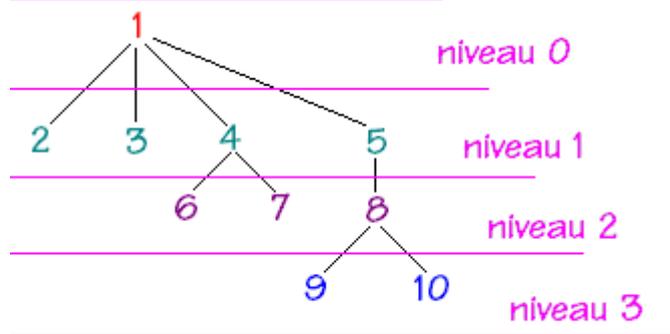
On parle aussi d'ascendant, de descendant ou de fils pour évoquer des relations entre les noeuds d'un même arbre reliés entre eux.

Nous pouvons définir récursivement la hauteur h d'un noeud X à partir de celle de son parent :



$h(\text{racine}) = 1;$
 $h(X) = 1 + h(\text{parent}(X))$

Reprenons l'arbre précédent en exemple :



Calculons récursivement la hauteur du noeud 9, notée $h(9)$:

$$h(9) = 1 + h(8)$$

$$h(8) = 1 + h(5)$$

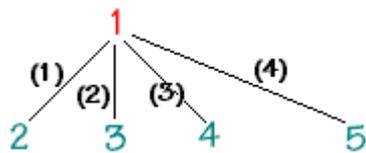
$$h(5) = 1 + h(1)$$

$$h(1) = 1 = h(5) = 2 = h(8) = 3 = h(9) = 4$$

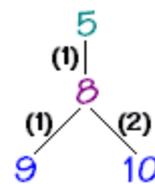


Par définition le **degré** d'un noeud est égal au **nombre de ses descendants** (enfants).

Soient les deux exemples ci-dessous extraits de l'arbre précédent :



Le noeud 1 est de degré 4, car il a 4 enfants



*Le noeud 5 n'ayant qu'un enfant son degré est 1.
 Le noeud 8 est de degré 2 car il a 2 enfants.*

Remarquons

que lorsqu'un arbre a **tous ses noeuds de degré 1**, on le nomme **arbre dégénéré** et que c'est en fait une **liste**.



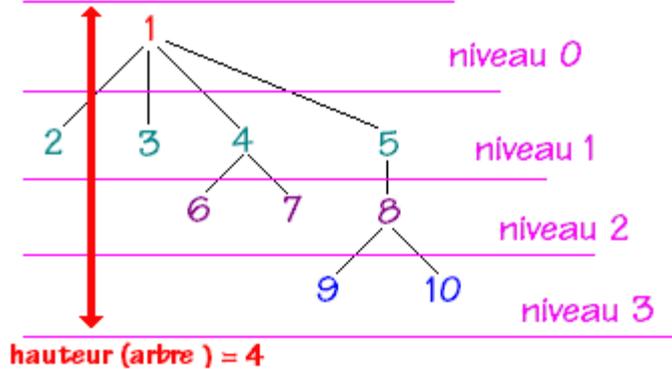
Par définition c'est le **nombre de noeuds du chemin le plus long** dans l'arbre.; on dit aussi profondeur de l'arbre.

La hauteur h d'un arbre correspond donc au nombre maximum de niveaux :

Hauteur d'un arbre

$h(\text{Arbre}) = \max \{ h(X) / \forall X, X \text{ noeud de Arbre} \}$
 si $\text{Arbre} = \emptyset$ alors $h(\text{Arbre}) = 0$

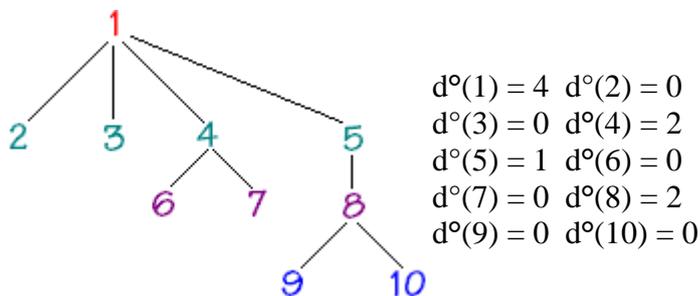
La hauteur de l'arbre ci-dessous :



Degré d'un arbre

Le degré d'un arbre est égal au **plus grand des degrés de ses nœuds** :
 $d^\circ(\text{Arbre}) = \max \{ d^\circ(X) / \forall X, X \text{ noeud de Arbre} \}$

Soit à répertorier dans l'arbre ci-dessous le degré de chacun des noeuds :

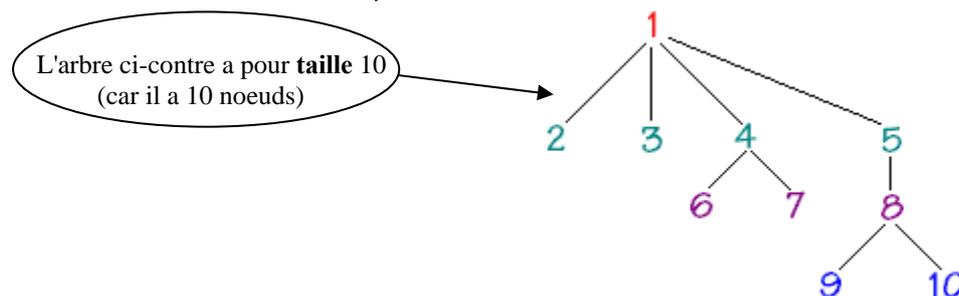


La valeur maximale est 4, donc cet arbre est de degré 4.

Taille d'un arbre

On appelle **taille** d'un arbre le nombre total de noeuds de cet arbre :

$\text{taille}(\langle r, fg, fd \rangle) = 1 + \text{taille}(fg) + \text{taille}(fd)$



1.2 Exemples et implémentation d'arbre

Les structures de données arborescentes permettent de représenter de nombreux problèmes, nous proposons ci-après quelques exemples d'utilisations d'arbres dans des contextes différents.



Soit la grammaire $G_2 : V_N = \{S\}$

$V_T = \{ (,) \}$

Axiome : S

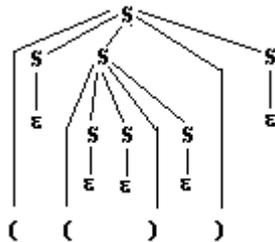
Règles

1 : $S \rightarrow (SS)S$

2 : $S \rightarrow \varepsilon$

Le langage $L(G_2)$ se dénomme langage des parenthèses bien formées.

Soit le mot $(())$ de G_2 , voici un arbre de dérivation de $(())$ dans G_2 :



Exemple - 2
arbre abstrait

Soit la grammaire G_{exp} :

$G_{exp} = (V_N, V_T, \text{Axiome}, \text{Règles})$

$V_T = \{ 0, \dots, 9, +, -, /, *,), (\}$

$V_N = \{ \langle \text{Expr} \rangle, \langle \text{Nbr} \rangle, \langle \text{Cte} \rangle, \langle \text{Oper} \rangle \}$

Axiome : $\langle \text{Expr} \rangle$

Règles :

1 : $\langle \text{Expr} \rangle \rightarrow \langle \text{Nbr} \rangle \mid (\langle \text{Expr} \rangle) \mid \langle \text{Expr} \rangle \langle \text{Oper} \rangle \langle \text{Expr} \rangle$

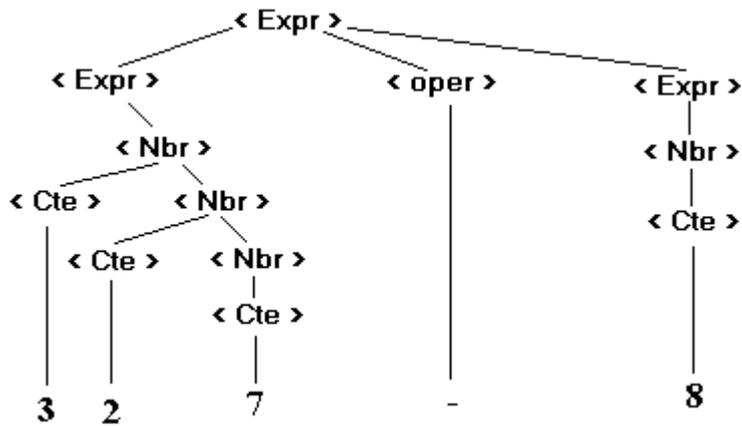
2 : $\langle \text{Nbr} \rangle \rightarrow \langle \text{Cte} \rangle \mid \langle \text{Cte} \rangle \langle \text{Nbr} \rangle$

3 : $\langle \text{Cte} \rangle \rightarrow 0 \mid 1 \mid \dots \mid 9$

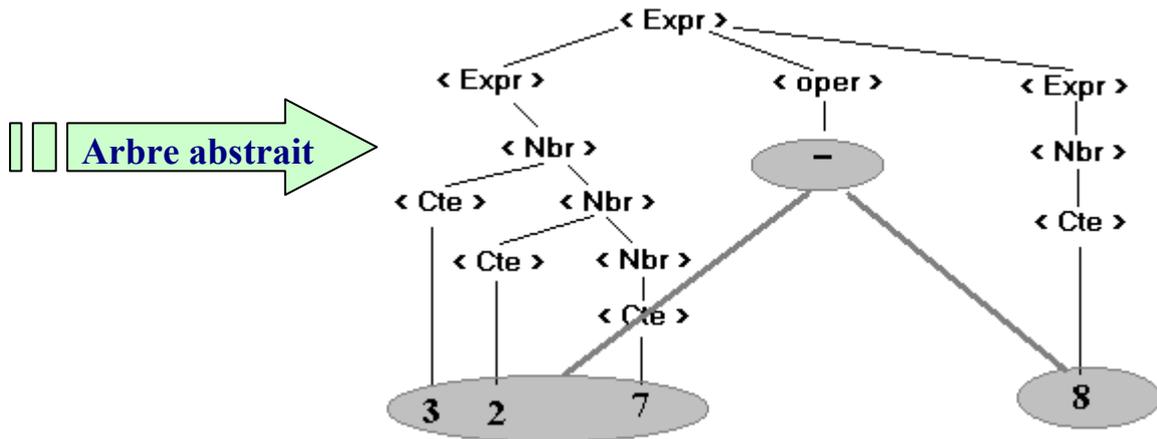
4 : $\langle \text{Oper} \rangle \rightarrow + \mid - \mid * \mid /$

soit : **327 - 8** un mot de $L(G_{exp})$

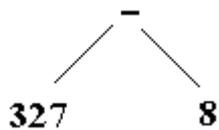
Soit son arbre de dérivation dans G_{exp} :



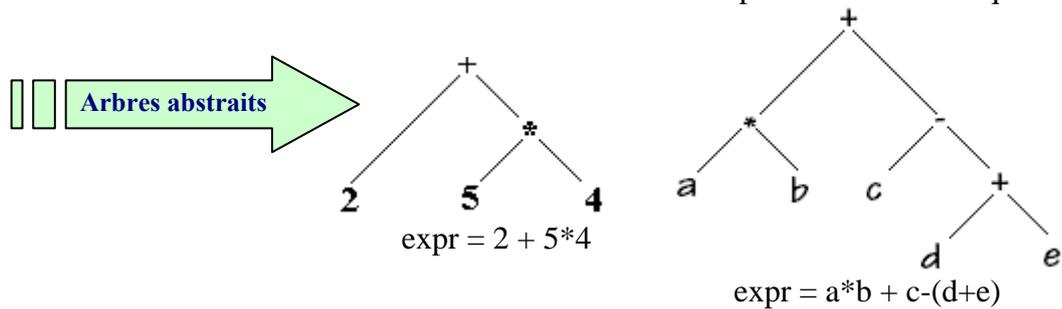
L'arbre obtenu ci-dessous en grisé à partir de l'arbre de dérivation s'appelle l'arbre abstrait du mot " 327-8 " :



On note ainsi cet arbre abstrait :



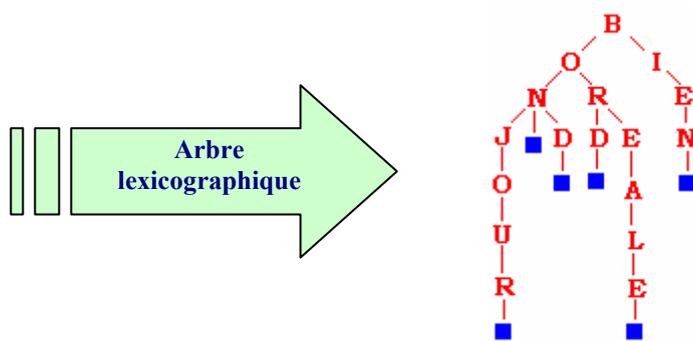
Voici d'autres arbres abstraits d'expressions arithmétiques :



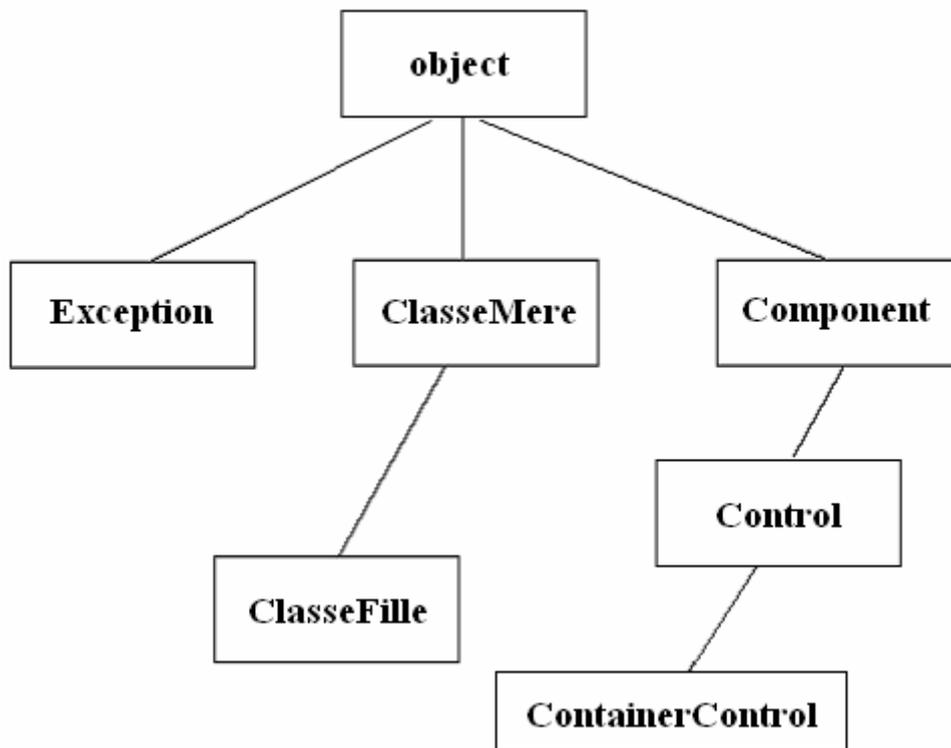
Arbre lexicographique

Rangement de mots par ordre lexical (alphabétique)

Soient les mots BON, BONJOUR, BORD, BOND, BOREALE, BIEN, il est possible de les ranger ainsi dans une structure d'arbre :



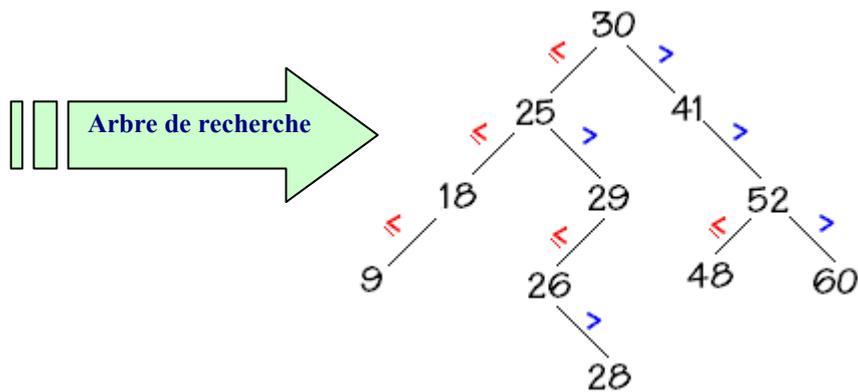
Arbre d'héritage en C#



Arbre de recherche

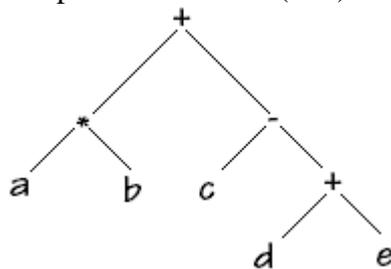
Voici à titre d'exemple que nous étudierons plus loin en détail, un arbre dont les noeuds sont de degré 2 au plus et qui est tel que pour chaque noeud la valeur de son enfant de gauche lui est inférieure ou égale, la valeur de son enfant de droite lui est strictement supérieure.

Ci-après un tel arbre ayant comme racine 30 et stockant des entiers selon cette répartition :



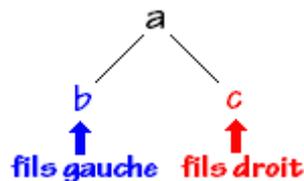
2 Les arbres binaires

Un arbre **binaire** est un arbre de degré 2 (dont les noeuds sont de degré 2 au plus). L'arbre abstrait de l'expression $a*b + c-(d+e)$ est un arbre binaire :



Vocabulaire :

Les descendants (enfants) d'un noeud sont lus de gauche à droite et sont appelés respectivement **fil gauche** (descendant gauche) et **fil droit** (descendant droit) de ce noeud.



Les arbres binaires sont utilisés dans de très nombreuses activités informatiques et comme nous l'avons déjà signalé il est toujours possible de représenter un arbre général (de degré 2) par un arbre binaire en opérant une "binarisation".

Nous allons donc étudier dans la suite, le comportement de cette structure de donnée récursive.

2.1 TAD d'arbre binaire

Afin d'assurer une cohérence avec les autres structures de données déjà vues (**liste**, **pile**, **file**) nous proposons de décrire une abstraction d'un arbre binaire avec un TAD. Soit la signature du TAD d'arbre binaire :

```

TAD ArbreBin
utilise : T0, Noeud, Booleens
opérations :
  ∅ : →ArbreBin
  Racine : ArbreBin →Noeud
  filsG : ArbreBin →ArbreBin
  filsD : ArbreBin →ArbreBin
  Constr : Noeud x ArbreBin x ArbreBin →ArbreBin
  Est_Vide : ArbreBin → Booleens
  Info : Noeud →T0

préconditions :

  Racine(Arb) def_ssi Arb ≠ ∅
  filsG(Arb) def_ssi Arb ≠ ∅
  filsD(Arb) def_ssi Arb ≠ ∅
axiomes :

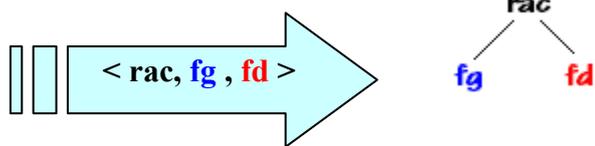
  ∀rac ∈ Noeud , ∀fg ∈ ArbreBin , ∀fd ∈ ArbreBin
  Racine(Constr(rac,fg,fd)) = rac
  filsG(Constr(rac,fg,fd)) = fg
  filsD(Constr(rac,fg,fd)) = fd
  Info(rac) ∈T0

FinTAD- ArbreBin

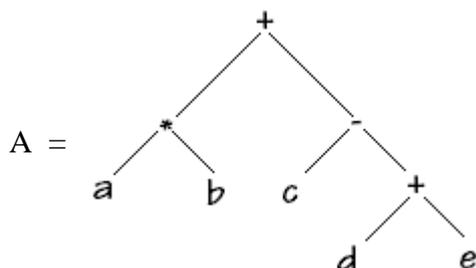
```

- T₀ est le type des données rangées dans l'arbre.
- L'opérateur **filsG()** renvoie le sous-arbre gauche de l'arbre binaire, l'opérateur **filsD()** renvoie le sous-arbre droit de l'arbre binaire, l'opérateur Info() permet de stocker des informations de type T₀ dans chaque noeud de l'arbre binaire.

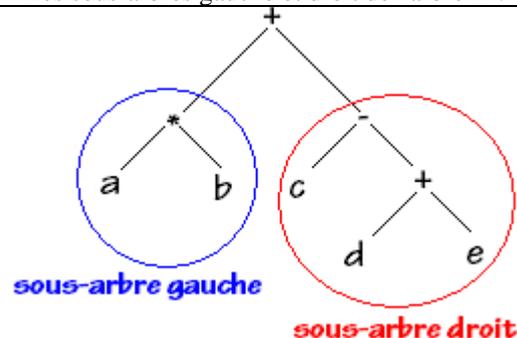
Nous noterons **< rac, fg, fd >** avec conventions implicites un arbre binaire dessiné ci-dessous :



Exemple, soit l'arbre binaire A :



Les sous-arbres gauche et droit de l'arbre A :



filsG(A) = < *, a, b >
 filsD(A) = < -, c, <+, d, e >>

2.2 Exemples et implémentation d'arbre binaire étiqueté

Nous proposons de représenter un **arbre binaire étiqueté** selon deux spécifications différentes classiques :

1°) Une implantation fondée sur une structure de tableau en **allocation de mémoire statique**, nécessitant de connaître au préalable le nombre maximal de noeuds de l'arbre (ou encore sa taille).

2°) Une implantation fondée sur une structure d'**allocation de mémoire dynamique** implémentée soit par des pointeurs (variables dynamiques) soit par des références (objets) .

Implantation dans un tableau statique

Spécification concrète

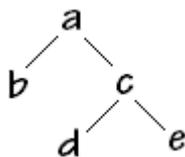
Un noeud est une structure statique contenant 3 éléments :

- l'information du noeud
- le fils gauche
- le fils droit

Pour un arbre binaire de taille = n, **chaque noeud de l'arbre binaire est stocké dans une cellule d'un tableau** de dimension 1 à n cellules. Donc chaque noeud est repéré dans le tableau par un indice (celui de la cellule le contenant).

Le champ fils gauche du noeud sera l'**indice de la cellule contenant le descendant gauche**, et le champ fils droit vaudra l'**indice de la cellule contenant le descendant droit**.

Exemple



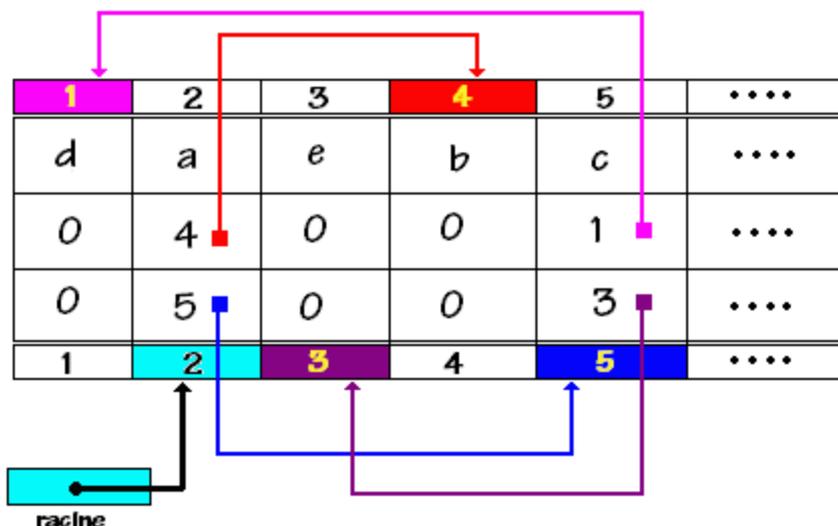
Soit l'arbre binaire ci-contre :

Selon l'implantation choisie, par hypothèse de départ, la racine <a, vers b, vers c >est contenue dans la cellule d'indice 2 du tableau, les autres noeuds sont supposés être rangés dans les cellules 1, 3,4,5 :

Nous avons :

```

racine = table[2]
table[1] = < d , 0 , 0 >
table[2] = < a , 4 , 5 >
table[3] = < e , 0 , 0 >
table[4] = < b , 0 , 0 >
table[5] = < c , 1 , 3 >
  
```



Explications :

table[2] = < a , 4 , 5 > signifie que le fils gauche de ce noeud est dans table[4] et son fils droit dans table[5]
table[5] = < c , 1 , 3 > signifie que le fils gauche de ce noeud est dans table[1] et son fils droit dans table[3]
table[1] = < d , 0 , 0 > signifie que ce noeud est une feuille
...etc

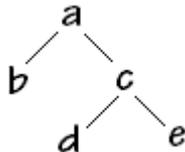
Implantation dynamique avec une classe

Spécification concrète

Le noeud est un objet contenant 3 éléments dont 2 sont eux-mêmes des objets de noeud :

- l'information du noeud
- une référence vers le fils gauche
- une référence vers le fils droit

Exemple

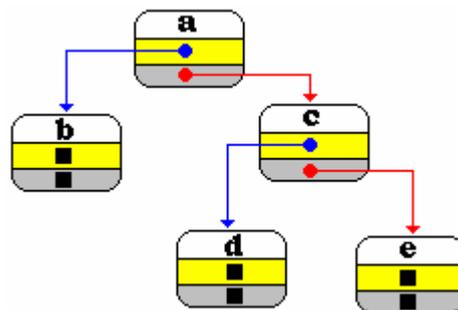


Soit l'arbre binaire ci-contre :

Selon l'implantation choisie, par hypothèse de départ, le premier objet appelé racine de l'arbre est < a, ref vers b, ref vers c >

Nous avons :

racine → < a, ref vers b, ref vers c >
ref vers b → < b, null, null >
ref vers c → < a, ref vers d, ref vers e >
ref vers d → < d, null, null >
ref vers e → < e, null, null >



Spécification d'implantation en



Nous livrons ci-dessous une écriture de la signature et l'implémentation minimale d'une classe d'arbre binaire nommée `ArbreBin` en C# dans laquelle les informations portées par un nœud sont de type `T0` :

```
interface IArbreBin<T0>
{
    T0 Info { get; set; }
}
class ArbreBin<T0> : IArbreBin<T0>
{
    private T0 InfoLoc;
```

```

private ArbreBin<T0> fg;
private ArbreBin<T0> fd;

public ArbreBin(T0 s) : this () {
    InfoLoc = s;
}
public ArbreBin ()
{
    InfoLoc = default(T0);
    fg = default(ArbreBin<T0>);
    fd = default(ArbreBin<T0>);
}
public T0 Info
{
    get { return InfoLoc; }
    set { InfoLoc = value; }
}

public ArbreBin<T0> filsG
{
    get { return this.fg; }
    set { fg = new ArbreBin<T0>(value.Info); }
}
public ArbreBin<T0> filsD
{
    get { return this.fd; }
    set { fd = new ArbreBin<T0>(value.Info); }
}
}

```

2.3 Arbres binaires de recherche

- Nous avons étudié précédemment des algorithmes de recherche en table, en particulier la recherche dichotomique dans une table triée dont la recherche s'effectue en $O(\log(n))$ comparaisons.
- Toutefois lorsque le nombre des éléments varie (ajout ou suppression) ces ajouts ou suppressions peuvent nécessiter des temps en $O(n)$.
- En utilisant une liste chaînée qui approche bien la structure dynamique (plus gourmande en mémoire qu'un tableau) on aura en moyenne des temps de suppression ou de recherche au pire de l'ordre de $O(n)$. L'ajout en fin de liste ou en début de liste demandant un temps constant noté $O(1)$.

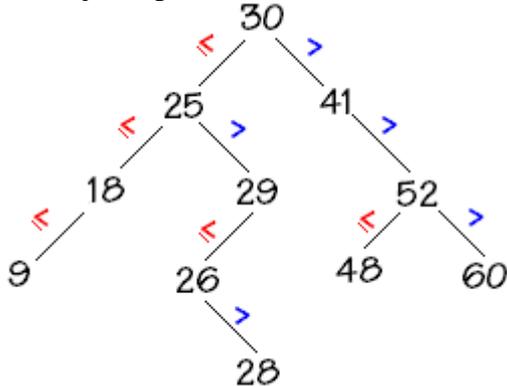
Les arbres binaires de recherche sont un bon compromis pour un temps **équilibré entre ajout, suppression et recherche**.

Un arbre binaire de recherche satisfait aux critères suivants :

- L'ensemble des étiquettes est **totalelement ordonné**.
- Une étiquette est dénommée **clef**.
- Les **clefs** de tous les noeuds du sous-arbre **gauche** d'un noeud X, sont **inférieures ou égales** à la clef de X.

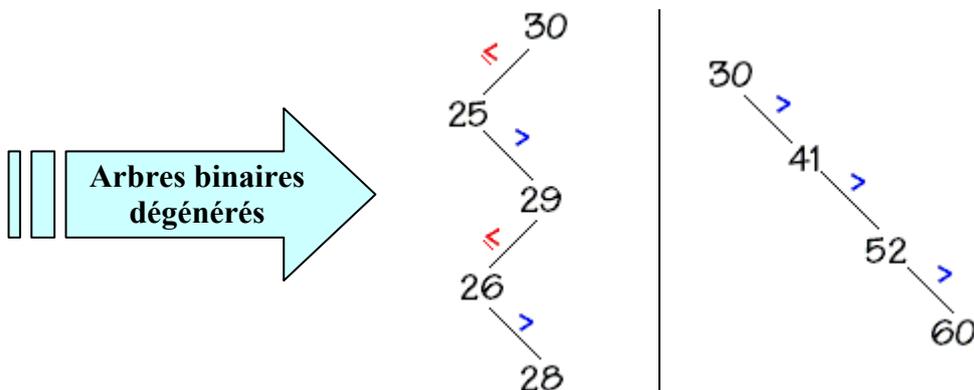
- Les **clefs** de tous les noeuds du sous-arbre **droit** d'un noeud X, sont **supérieures** à la clef de X.

Nous avons déjà vu plus haut un arbre binaire de recherche :



Prenons par exemple le noeud (25) son sous-arbre droit est bien composé de noeuds dont les clefs sont supérieures à 25 : (29,26,28). Le sous-arbre gauche du noeud (25) est bien composé de noeuds dont les clefs sont inférieures à 25 : (18,9).

On appelle arbre binaire dégénéré un arbre binaire dont le degré = 1, ci-dessous 2 arbres binaires de recherche dégénérés :

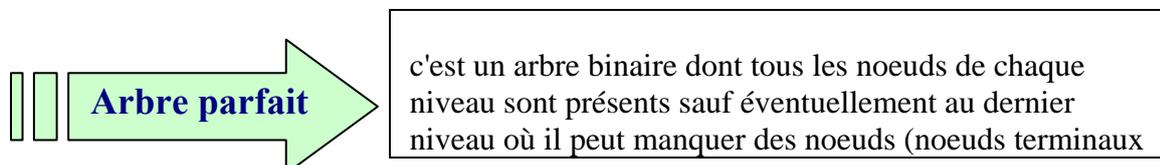


Nous remarquons dans les deux cas que nous avons affaire à une liste chaînée donc le nombre d'opération pour la suppression ou la recherche est au pire de l'ordre de $O(n)$.

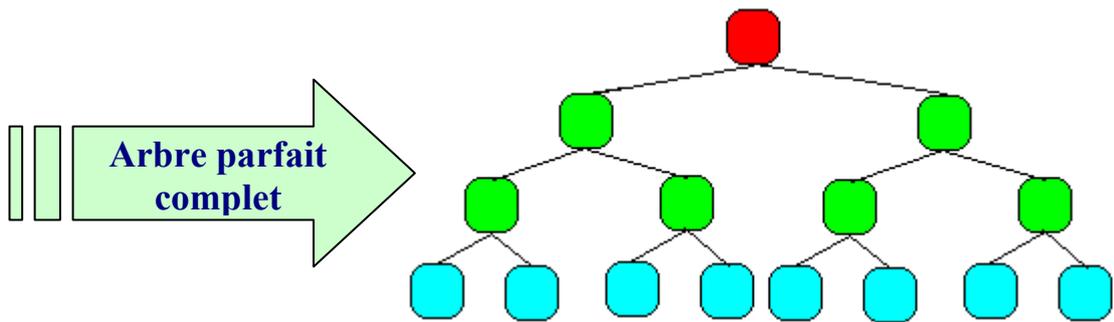
Il faudra donc utiliser une catégorie spéciale d'arbres binaires qui restent équilibrés (leurs feuilles sont sur 2 niveaux au plus) pour assurer une recherche au pire en $O(\log(n))$.

2.4 Arbres binaires partiellement ordonnés (tas)

Nous avons déjà évoqué la notion d'arbre parfait lors de l'étude du tri par tas, nous récapitulons ici les éléments essentiels le lecteur

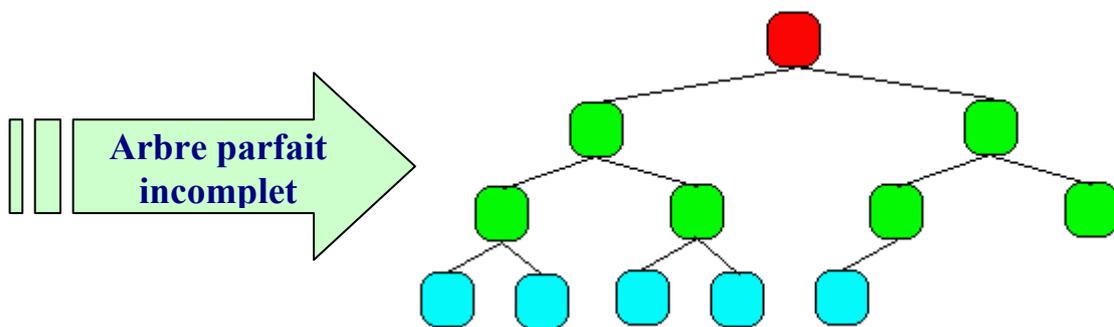


= feuilles), dans ce cas l'arbre parfait est un arbre binaire incomplet et les feuilles du dernier niveau **doivent être regroupées à partir de la gauche** de l'arbre.



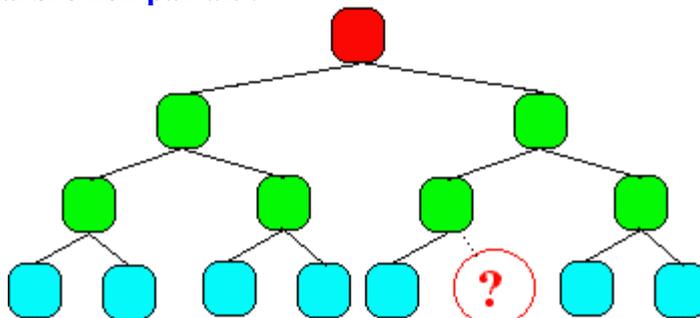
parfait complet : le dernier niveau est complet car il contient tous les enfants

un arbre **parfait** peut être incomplet lorsque le dernier niveau de l'arbre est incomplet (dans le cas où manquent des feuilles à la droite du dernier niveau, les feuilles sont regroupées à gauche)



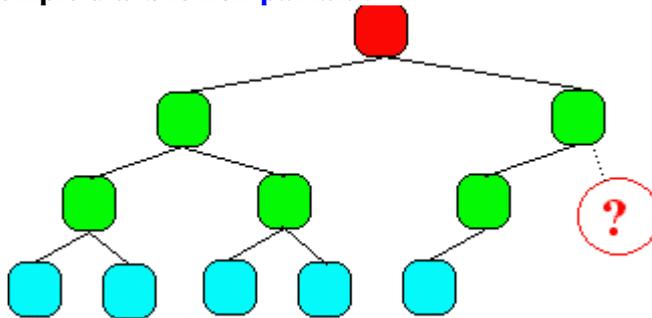
parfait incomplet: le dernier niveau est incomplet car il manque 3 enfants à la droite

Exemple d'arbre **non parfait** :



(non parfait : les feuilles ne sont pas regroupées à gauche)

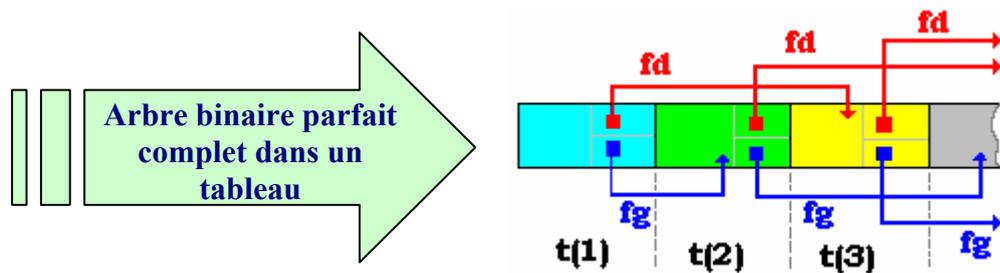
Autre exemple d'arbre non parfait :



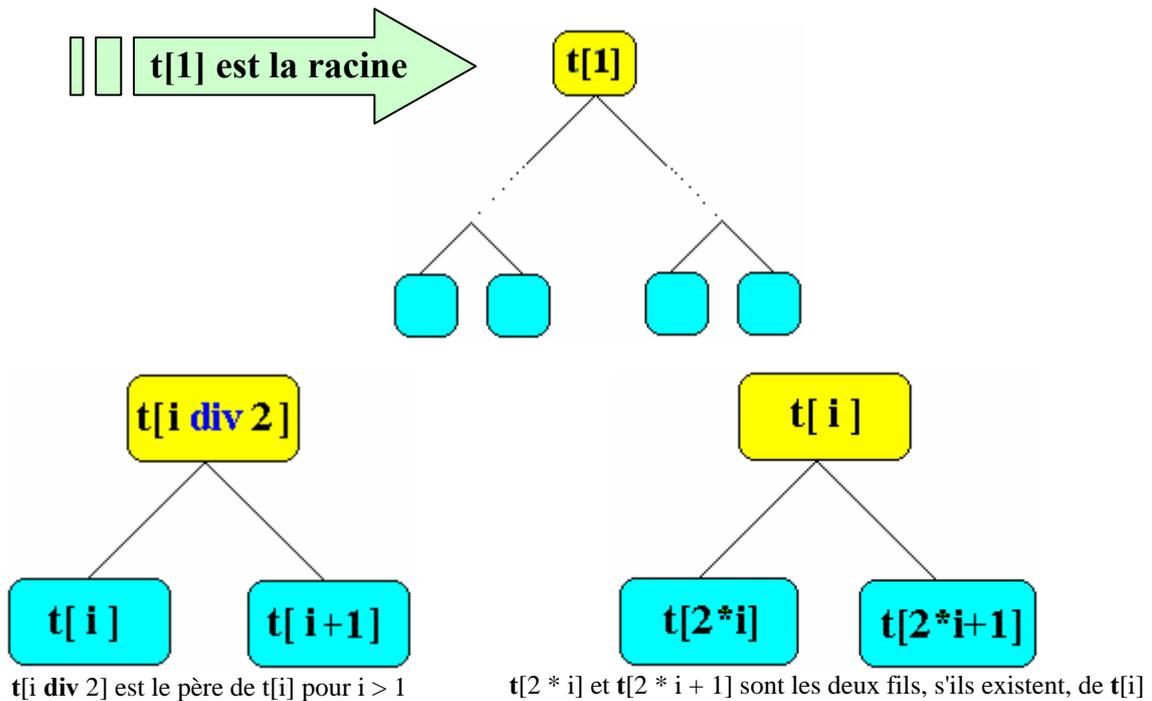
(non parfait : les feuilles sont bien regroupées à gauche, mais il manque 1 enfant à l'avant dernier niveau)

Un arbre binaire parfait se représente classiquement dans un tableau :

Les noeuds de l'arbre sont dans les cellules du tableau, il n'y a pas d'autre information dans une cellule du tableau, l'accès à la topologie arborescente est simulée à travers un calcul d'indice permettant de parcourir les cellules du tableau selon un certain 'ordre' de numérotation correspondant en fait à un **parcours hiérarchique** de l'arbre. En effet ce sont les numéros de ce parcours qui servent d'indice aux cellules du tableau nommé **t** ci-dessous :



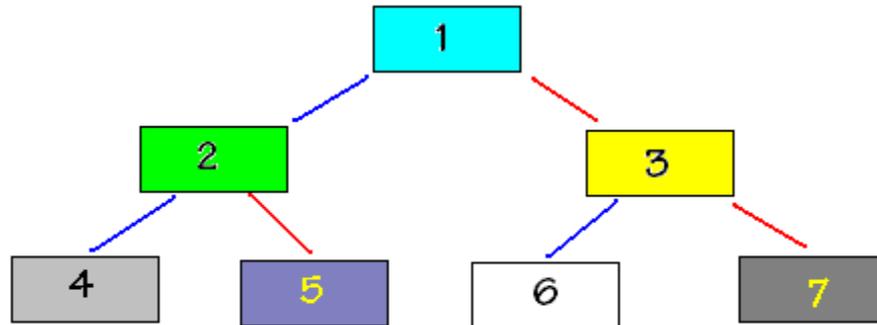
Si **t** est ce tableau, nous avons les règles suivantes :



si p est le nombre de noeuds de l'arbre et si $2 * i = p$, $t[i]$ n'a qu'un fils, $t[p]$.
 si i est supérieur à $p \text{ div } 2$, $t[i]$ est une feuille.

Exemple de rangement d'un tel arbre dans un tableau

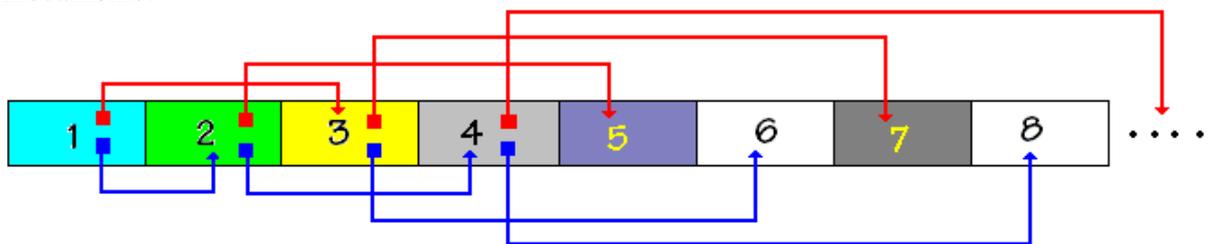
(on a figuré l'indice de numérotation hiérarchique de chaque noeud dans le rectangle associé au noeud)



Cet arbre sera stocké dans un tableau en disposant séquentiellement et de façon contiguë les noeuds selon la numérotation hiérarchique (l'index de la cellule = le numéro hiérarchique du noeud).

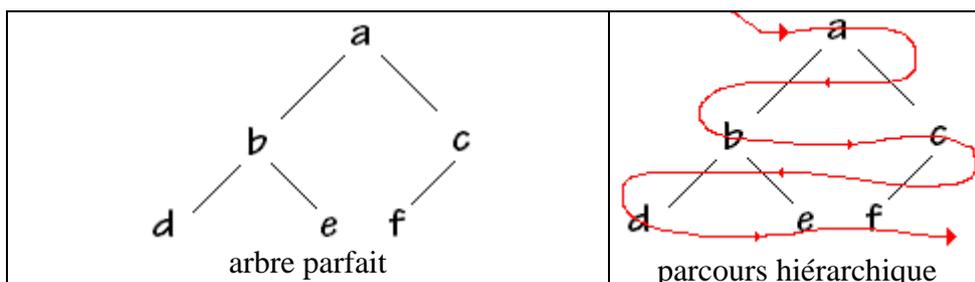
Dans cette disposition le passage d'un noeud de numéro k (indice dans le tableau) vers son fils gauche s'effectue par calcul d'indice, le fils gauche se trouvera dans la cellule d'index $2 * k$ du tableau, son fils droit se trouvant dans la cellule d'index $2 * k + 1$ du tableau. Ci-dessous l'arbre précédent est stocké dans un tableau : le noeud d'indice hiérarchique 1 (la racine) dans la cellule d'index 1, le noeud d'indice hiérarchique 2 dans la cellule d'index 2, etc...

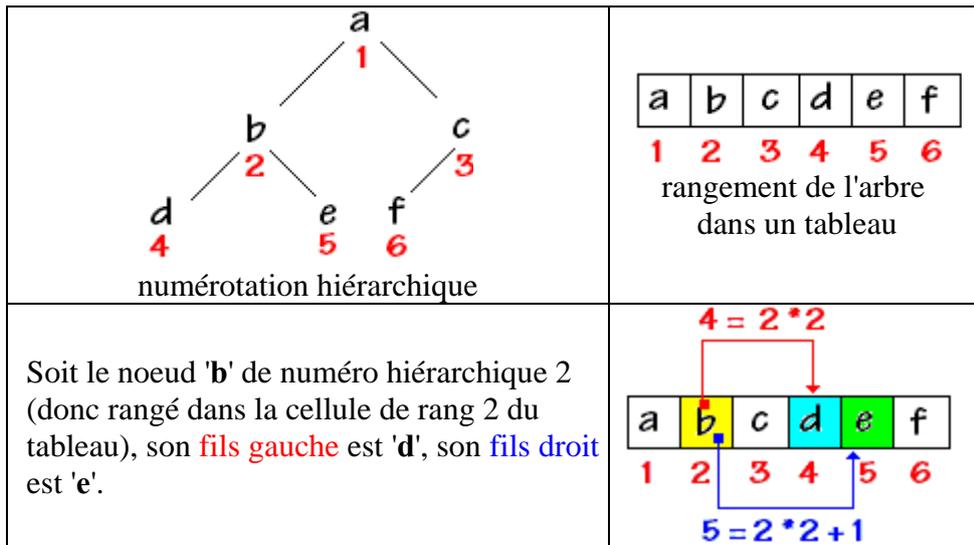
Le nombre qui figure dans la cellule (nombre qui vaut l'index de la cellule = le numéro hiérarchique du noeud) n'est mis là qu'à titre pédagogique afin de bien comprendre le mécanisme.



On voit par exemple, que par calcul on a bien le fils gauche du noeud d'indice 2 est dans la cellule d'index $2 * 2 = 4$ et son fils droit se trouve dans la cellule d'index $2 * 2 + 1 = 5$...

Exemple d'un arbre parfait étiqueté avec des caractères :



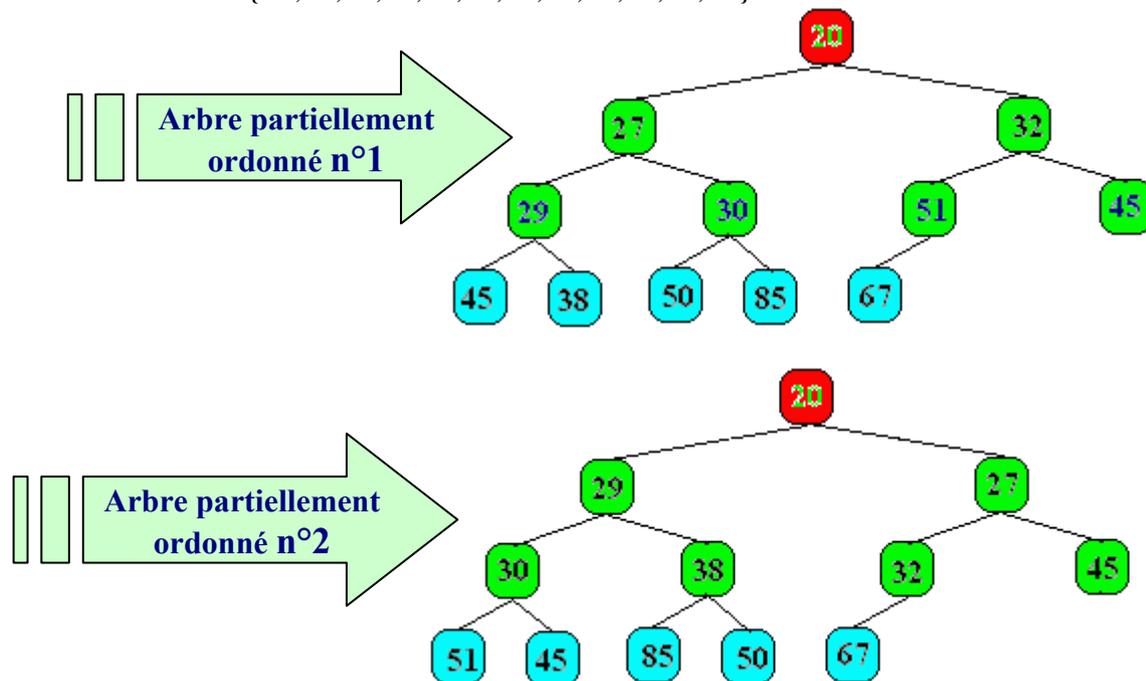


Arbre partiellement ordonné

C'est un arbre étiqueté dont les valeurs des noeuds appartiennent à un ensemble muni d'une **relation d'ordre total** (les nombres entiers, réels etc... en sont des exemples) tel que pour un noeud donné tous ses **fil**s ont une valeur supérieure ou égale à celle de leur père.

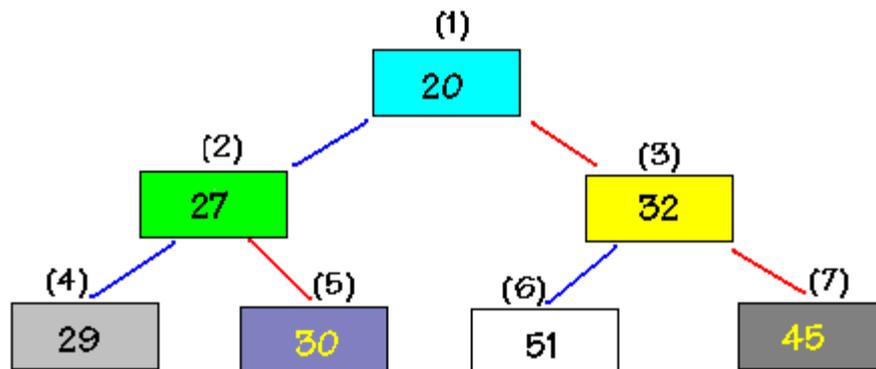
Exemple de **deux** arbres partiellement ordonnés

sur l'ensemble {20,27,29,30,32,38,45,45,50,51,67,85} d'entiers naturels :

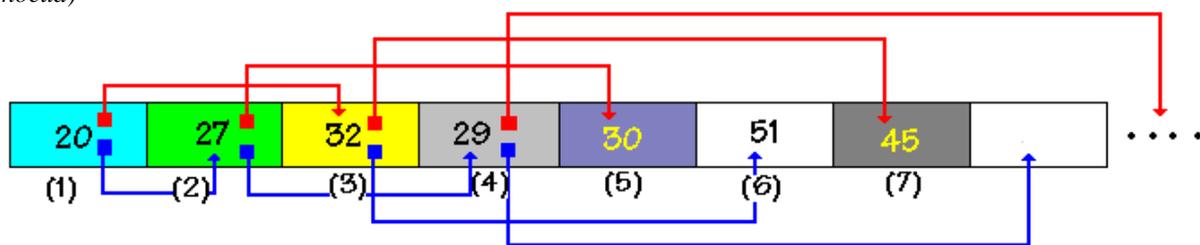


Nous remarquons que **la racine d'un tel arbre est toujours l'élément de l'ensemble possédant la valeur minimum** (le plus petit élément de l'ensemble), car la valeur de ce noeud par construction est inférieure à celle de ses fils et par transitivité de la relation d'ordre à celles de ses

descendants c'est le minimum. Si donc nous arrivons à ranger une liste d'éléments dans un tel arbre le minimum de cette liste est atteignable immédiatement comme racine de l'arbre. En reprenant l'exemple précédent sur 3 niveaux : (entre parenthèses le numéro hiérarchique du noeud)



Voici réellement ce qui est stocké dans le tableau : (entre parenthèses l'index de la cellule contenant le noeud)



On appelle **tas** un tableau représentant un **arbre parfait partiellement ordonné**.

L'intérêt d'utiliser un arbre parfait complet ou incomplet réside dans le fait que le tableau est toujours **compacté**, les cellules vides s'il y en a se situent à la fin du tableau. Le fait d'être partiellement ordonné sur les valeurs permet d'avoir immédiatement un **extremum** à la racine.

2.5 Parcours d'un arbre binaire

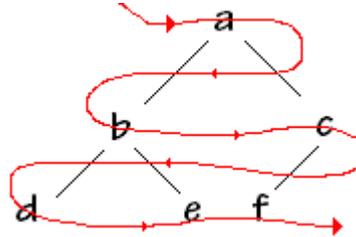
Objectif : les arbres sont des structures de données. Les informations sont contenues dans les noeuds de l'arbre, afin de construire des algorithmes effectuant des opérations sur ces informations (ajout, suppression, modification,...) il nous faut pouvoir examiner tous les noeuds d'un arbre. Examinons les différents moyens de parcourir ou de traverser chaque noeud de l'arbre et d'appliquer un traitement à la donnée rattachée à chaque noeud.

Parcours d'un arbre

L'opération qui consiste à **retrouver** systématiquement tous les noeuds d'un arbre et d'y appliquer un **même traitement** se dénomme **parcours** de l'arbre.

Parcours en largeur ou hiérarchique

Un algorithme classique consiste à **explorer** chaque noeud d'un niveau donné de **gauche à droite**, puis de passer au niveau suivant. On dénomme cette stratégie le parcours en largeur de l'arbre.



Parcours en profondeur

La stratégie consiste à **descendre** le plus profondément soit **jusqu'aux feuilles** d'un noeud de l'arbre, puis lorsque toutes les feuilles du noeud ont été visitées, l'algorithme "**remonte**" au noeud plus haut dont les feuilles n'ont pas encore été visitées.

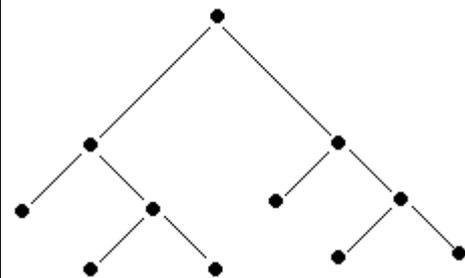
Notons que ce parcours peut s'effectuer systématiquement en commençant par le fils gauche, puis en examinant le fils droit ou bien l'inverse.

Parcours en profondeur par la gauche

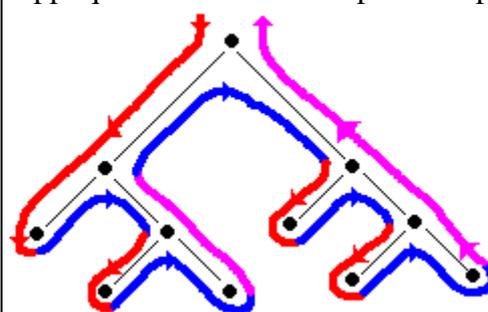
Traditionnellement c'est l'exploration **fils gauche, puis ensuite fils droit** qui est retenue on dit alors que l'on traverse l'arbre en "**profondeur par la gauche**".

Schémas montrant le principe du parcours exhaustif en "**profondeur par la gauche**" :

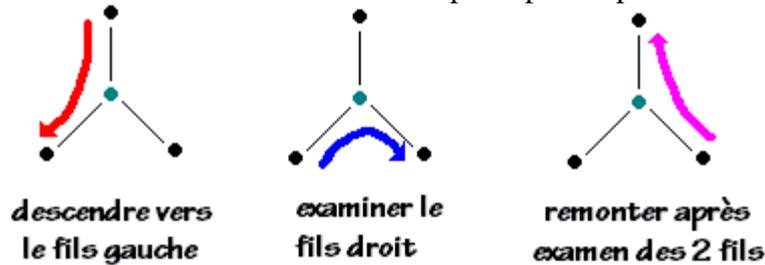
Soit l'arbre binaire suivant:



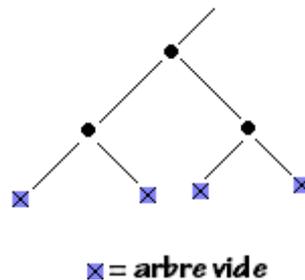
Appliquons la méthode de parcours proposée :



Chaque noeud a bien été examiné selon les principes du parcours en profondeur :



En fait pour ne pas surcharger les schémas arborescents, nous omettons de dessiner à la fin de chaque noeud de type feuille les deux **noeuds enfants vides** qui permettent de reconnaître que le parent est une feuille :



Lorsque la compréhension nécessitera leur dessin nous conseillons au lecteur de faire figurer explicitement dans son schéma arborescent les noeuds vides au bout de chaque feuille.

Nous proposons maintenant, de donner une description en langage algorithmique LDFA du parcours en profondeur d'un arbre binaire sous forme récursive.

Algorithme général récursif de parcours en profondeur par la gauche

```

parcourir ( Arbre )
si Arbre ≠ ∅ alors
    Traiter-1 (info(Arbre.Racine)) ;
    parcourir ( Arbre.filsG ) ;
    Traiter-2 (info(Arbre.Racine)) ;
    parcourir ( Arbre.filsD ) ;
    Traiter-3 (info(Arbre.Racine)) ;
Fsi
    
```

Les différents traitements **Traiter-1** ,**Traiter-2** et **Traiter-3** consistent à traiter l'information située dans le noeud actuellement traversé soit lorsque l'on descend vers le fils gauche (**Traiter-1**), soit en allant examiner le fils droit (**Traiter-2**), soit lors de la remonté après examen des 2 fils (**Traiter-3**).

En fait on n'utilise en pratique que trois variantes de cet algorithme, celles qui constituent des parcours ordonnés de l'arbre en fonction de l'application du traitement de l'information située aux noeuds. Chacun de ces 3 parcours définissent un ordre implicite (préfixé, infixé, postfixé) sur l'affichage et le traitement des données contenues dans l'arbre.

Algorithme de parcours en pré-ordre :

```
parcourir ( Arbre )  
si Arbre ≠ ∅ alors  
    Traiter-1 (info(Arbre.Racine)) ;  
    parcourir ( Arbre.filsG ) ;  
    parcourir ( Arbre.filsD ) ;  
Fsi
```

Ordre préfixé

Algorithme de parcours en post-ordre :

```
parcourir ( Arbre )  
si Arbre ≠ ∅ alors  
    parcourir ( Arbre.filsG ) ;  
    parcourir ( Arbre.filsD ) ;  
    Traiter-3 (info(Arbre.Racine)) ;  
Fsi
```

Ordre postfixé

Algorithme de parcours en ordre symétrique :

```
parcourir ( Arbre )  
si Arbre ≠ ∅ alors  
    parcourir ( Arbre.filsG ) ;  
    Traiter-2 (info(Arbre.Racine)) ;  
    parcourir ( Arbre.filsD ) ;  
Fsi
```

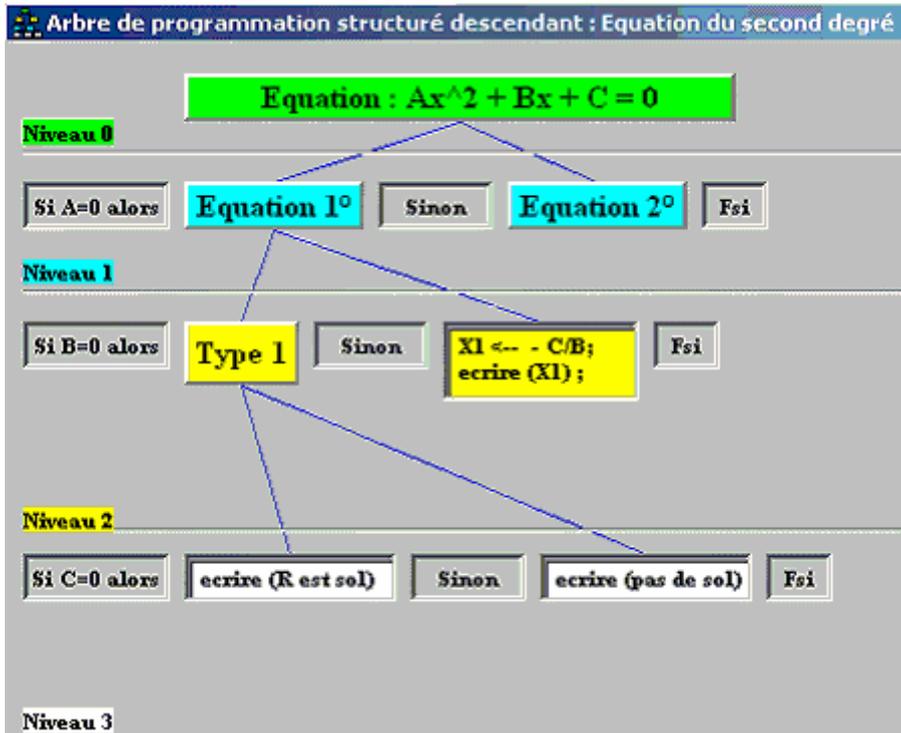
Ordre infixé

Illustration pratique d'un parcours général en profondeur

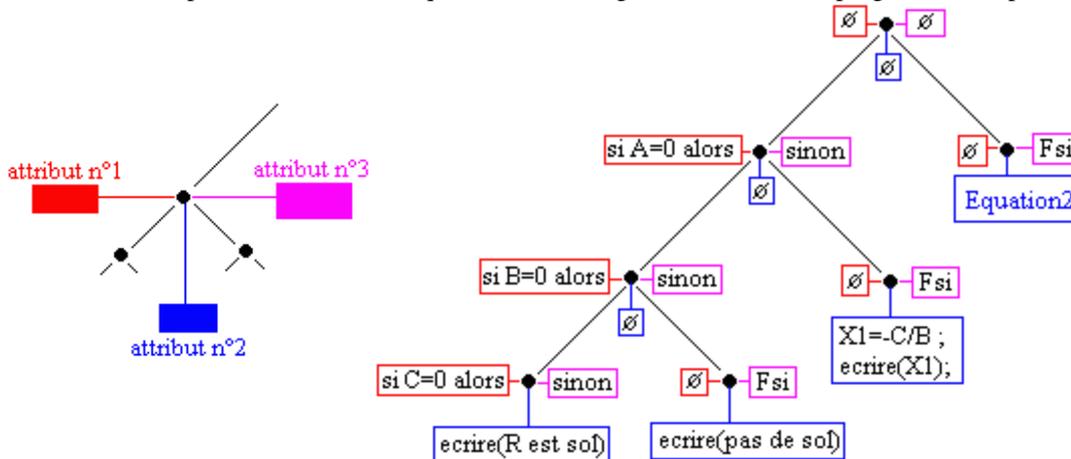
Le lecteur trouvera plus loin des exemples de parcours selon l'un des 3 ordres infixé, préfixé, postfixé, nous proposons ici un exemple didactique de parcours général avec les 3 traitements.

Nous allons voir comment utiliser une telle structure arborescente afin de restituer du texte algorithmique linéaire en effectuant un parcours en profondeur.

Voici ce que nous donne une analyse descendante du problème de résolution de l'équation du second degré (nous avons fait figurer uniquement la branche gauche de l'arbre de programmation) :

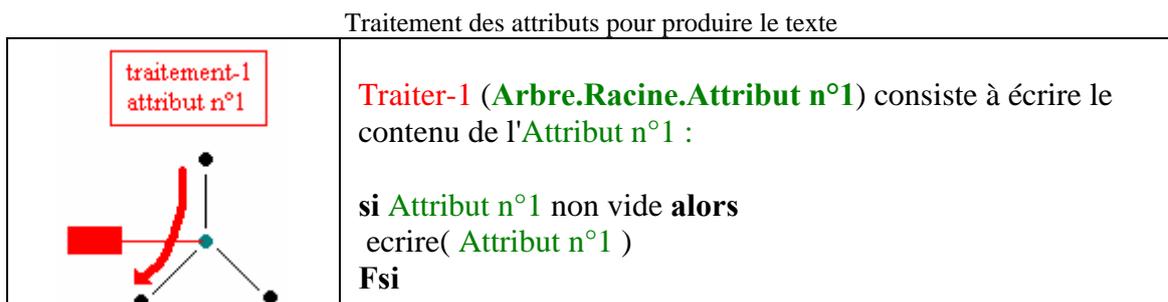


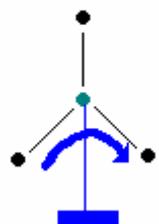
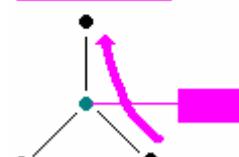
Ci-dessous une représentation schématique de la branche gauche de l'arbre de programmation précédent :



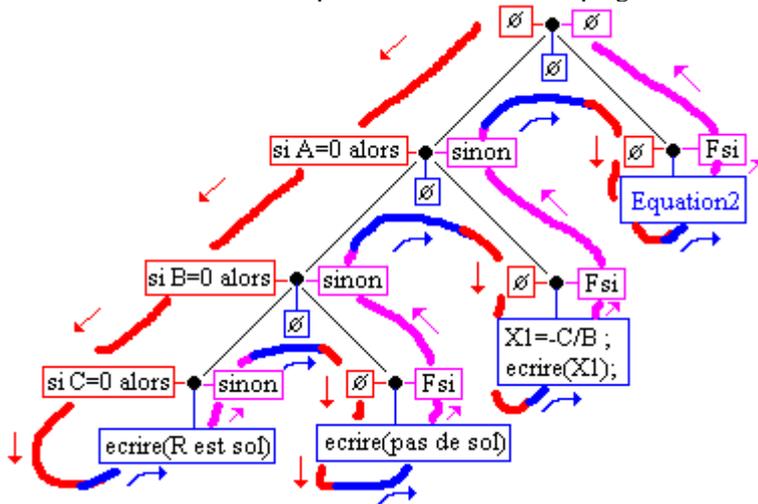
Nous avons établi un modèle d'arbre (binaire ici) où les informations au noeud sont au nombre de 3 (nous les nommerons **attribut n°1**, **attribut n°2** et **attribut n°3**). Chaque attribut est une **chaîne de caractères**, vide s'il y a lieu.

Nous noterons ainsi un attribut contenant une chaîne vide : \emptyset



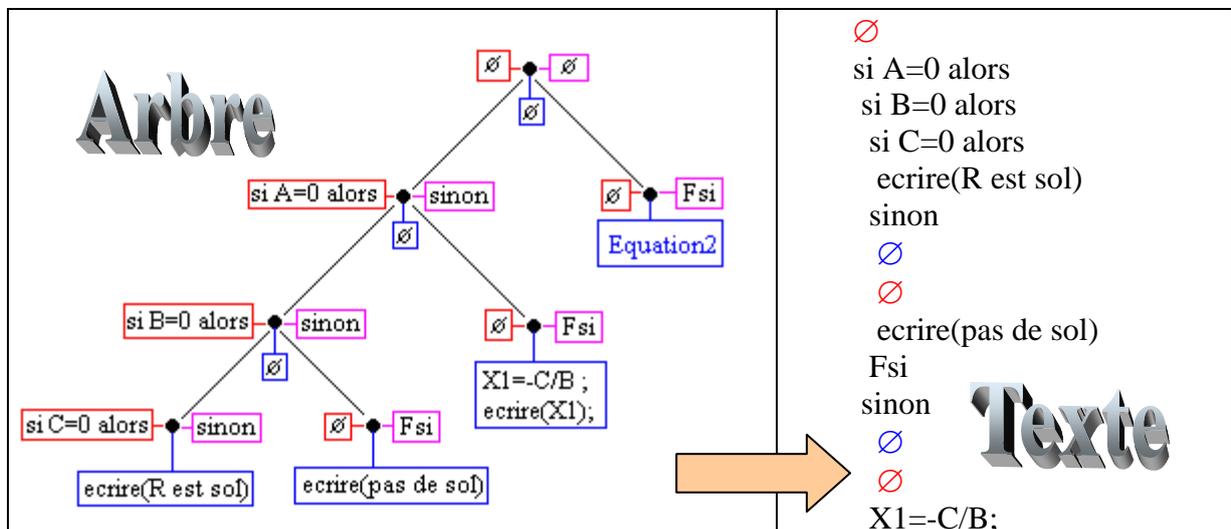
<p>traitement-2 attribut n°2</p> 	<p>Traiter-2 (Arbre.Racine.Attribut n°2) consiste à écrire le contenu de l'Attribut n°2 :</p> <p>si Attribut n°2 non vide alors écrire(Attribut n°2) Fsi</p>
<p>traitement-3 attribut n°3</p> 	<p>Traiter-3 (Arbre.Racine.Attribut n°3) consiste à écrire le contenu de l'Attribut n°3 :</p> <p>si Attribut n°3 non vide alors écrire(Attribut n°3) Fsi</p>

Parcours en profondeur de l'arbre de programmation de l'équation du second degré :



parcourir (**Arbre**)

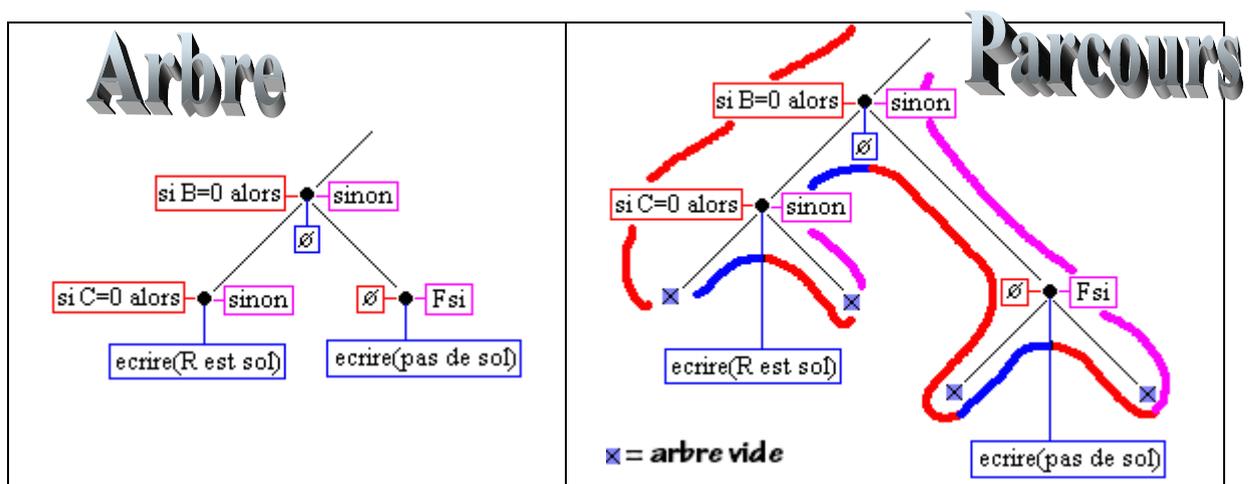
si **Arbre** ≠ ∅ **alors**
Traiter-1 (**Attribut n°1**) ;
parcourir (**Arbre.filsG**) ;
Traiter-2 (**Attribut n°2**) ;
parcourir (**Arbre.filsD**) ;
Traiter-3 (**Attribut n°3**) ;
Fsi



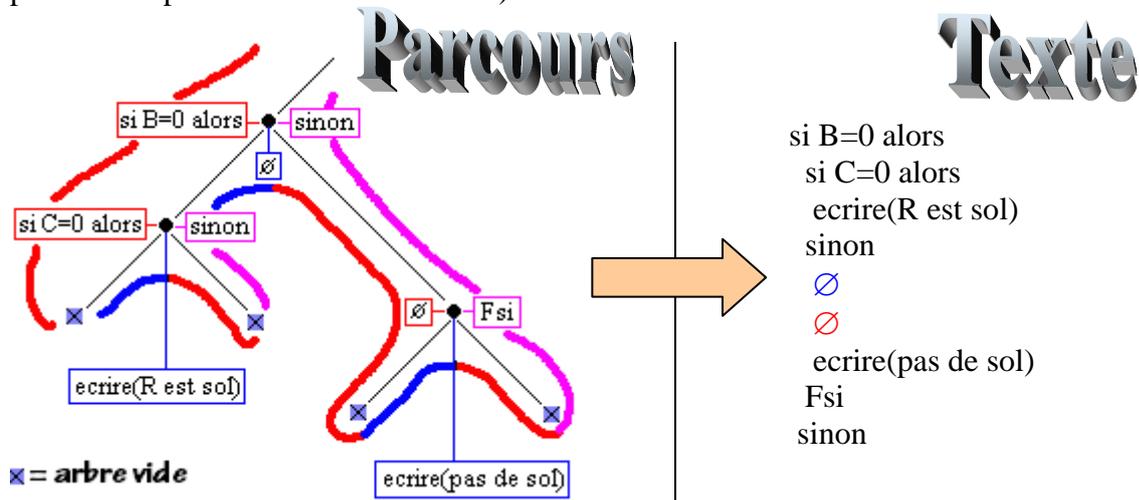
<p style="text-align: center;">Parcours</p> <p>si Arbre ≠ ∅ alors Traiter-1 (Attribut n°1) ; parcourir (Arbre.filsG) ; Traiter-2 (Attribut n°2) ; parcourir (Arbre.filsD) ; Traiter-3 (Attribut n°3) ; Fsi</p>	<pre> ecrire(X1); Fsi sinon ∅ ∅ Equation2 Fsi ∅ </pre>
--	--

Rappelons que le symbole ∅ représente la chaîne vide il est uniquement mis dans le texte dans le but de permettre le suivi du parcours de l'arbre.

Pour bien comprendre le parcours aux feuilles de l'arbre précédent, nous avons fait figurer ci-dessous sur un exemple, les **noeuds vides** de chaque feuille et le **parcours complet associé** :

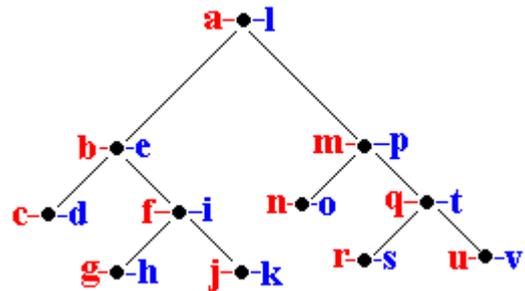


Le parcours partiel ci-haut produit le texte algorithmique suivant (le symbole ∅ est encore écrit pour la compréhension de la traversée) :



Exercice

Soit l'arbre ci-contre possédant 2 attributs par noeuds (un symbole de type caractère)



On propose le traitement en profondeur de l'arbre comme suit :
 L'attribut de gauche est écrit en descendant, l'attribut de droite est écrit en remontant, il n'y a pas d'attribut ni de traitement lors de l'examen du fils droit en venant du fils gauche.
 écrire la chaîne de caractère obtenue par le parcours ainsi défini.

Réponse :

abcdefghijklmnoqrstuvpl

Terminons cette revue des descriptions algorithmiques des différents parcours classiques d'arbre binaire avec le parcours en largeur (Cet algorithme nécessite l'utilisation d'une file du type Fifo dans laquelle l'on stocke les nœuds).

Algorithme de parcours en largeur

```

Largeur ( Arbre )

si Arbre ≠ ∅ alors
    ajouter racine de l'Arbre dans Fifo;
tantque Fifo ≠ ∅ faire
    prendre premier de Fifo;
    traiter premier de Fifo;
    si filsG de premier de Fifo ≠ ∅ alors
        ajouter filsG de premier de Fifo dans Fifo;
    Fsi
    si filsD de premier de Fifo ≠ ∅ alors
        ajouter filsD de premier de Fifo dans Fifo;
    Fsi
ftant
Fsi
    
```

2.6 Insertion, suppression, recherche dans un arbre binaire de recherche

Algorithme d'insertion dans un arbre binaire de recherche

placer l'élément *Elt* dans l'arbre *Arbre* par adjonctions successives aux feuilles

placer (*Arbre* *Elt*)

si *Arbre* = \emptyset **alors**

 créer un nouveau noeud contenant *Elt* ;

Arbre.Racine = ce nouveau noeud

sinon

 { - tous les éléments "info" de tous les noeuds du sous-arbre de gauche
 sont inférieurs ou égaux à l'élément "info" du noeud en cours (arbre)

 - tous les éléments "info" de tous les noeuds du sous-arbre de droite
 sont supérieurs à l'élément "info" du noeud en cours (arbre)

 }

si clef (*Elt*) \leq clef (*Arbre.Racine*) **alors**

placer (*Arbre.filsG* *Elt*)

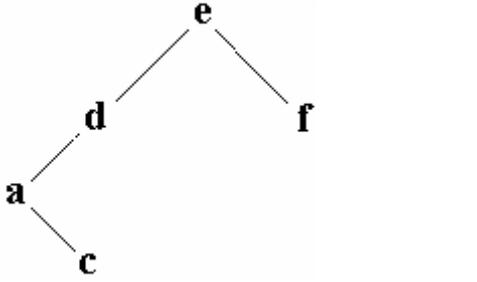
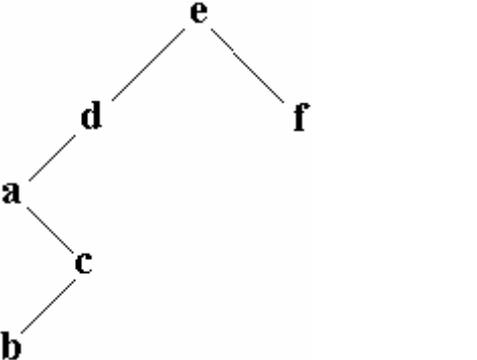
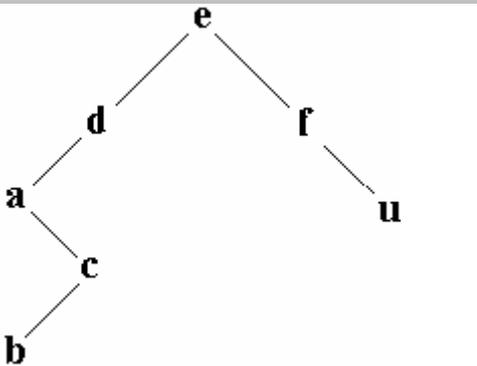
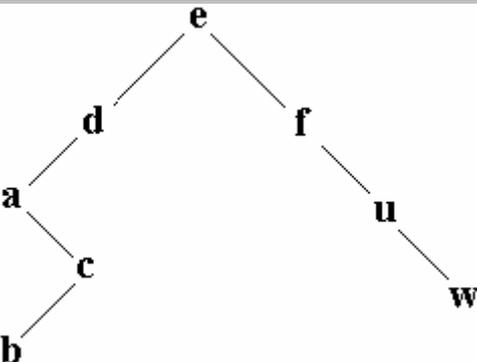
sinon

placer (*Arbre.filsD* *Elt*)

Fsi

Soit par exemple la liste de caractères alphabétiques : **e d f a c b u w**, que nous rangeons dans cet ordre d'entrée dans un arbre binaire de recherche. Ci-dessous le suivi de l'algorithme de placements successifs de chaque caractère de cette liste dans un arbre de recherche:

Insertions successives des éléments	Arbre de recherche obtenu
placer (racine , 'e') <i>e est la racine de l'arbre.</i>	e
placer (racine , 'd') <i>d < e donc fils gauche de e.</i>	<pre> e / d </pre>
placer (racine , 'f') <i>f > e donc fils droit de e.</i>	<pre> e / \ d f </pre>
placer (racine , 'a') <i>a < e donc à gauche, a < d donc fils gauche de d.</i>	<pre> e / \ d f / a </pre>

<p>placer (racine , 'e') <i>c < e donc à gauche, c < d donc à gauche, c > a donc fils droit de a.</i></p>	
<p>placer (racine , 'b') <i>b < e donc à gauche, b < d donc à gauche, b > a donc à droite de a, b < c donc fils gauche de c.</i></p>	
<p>placer (racine , 'u') <i>u > e donc à droite de e, u > f donc fils droit de f.</i></p>	
<p>placer (racine , 'w') <i>w > e donc à droite de e, w > f donc à droite de f, w > u donc fils droit de u.</i></p>	

Algorithme de recherche dans un arbre binaire de recherche

chercher l'élément *Elt* dans l'arbre *Arbre* :

Chercher (Arbre *Elt*) : Arbre

si Arbre = ∅ alors

Afficher *Elt* non trouvé dans l'arbre;

sinon

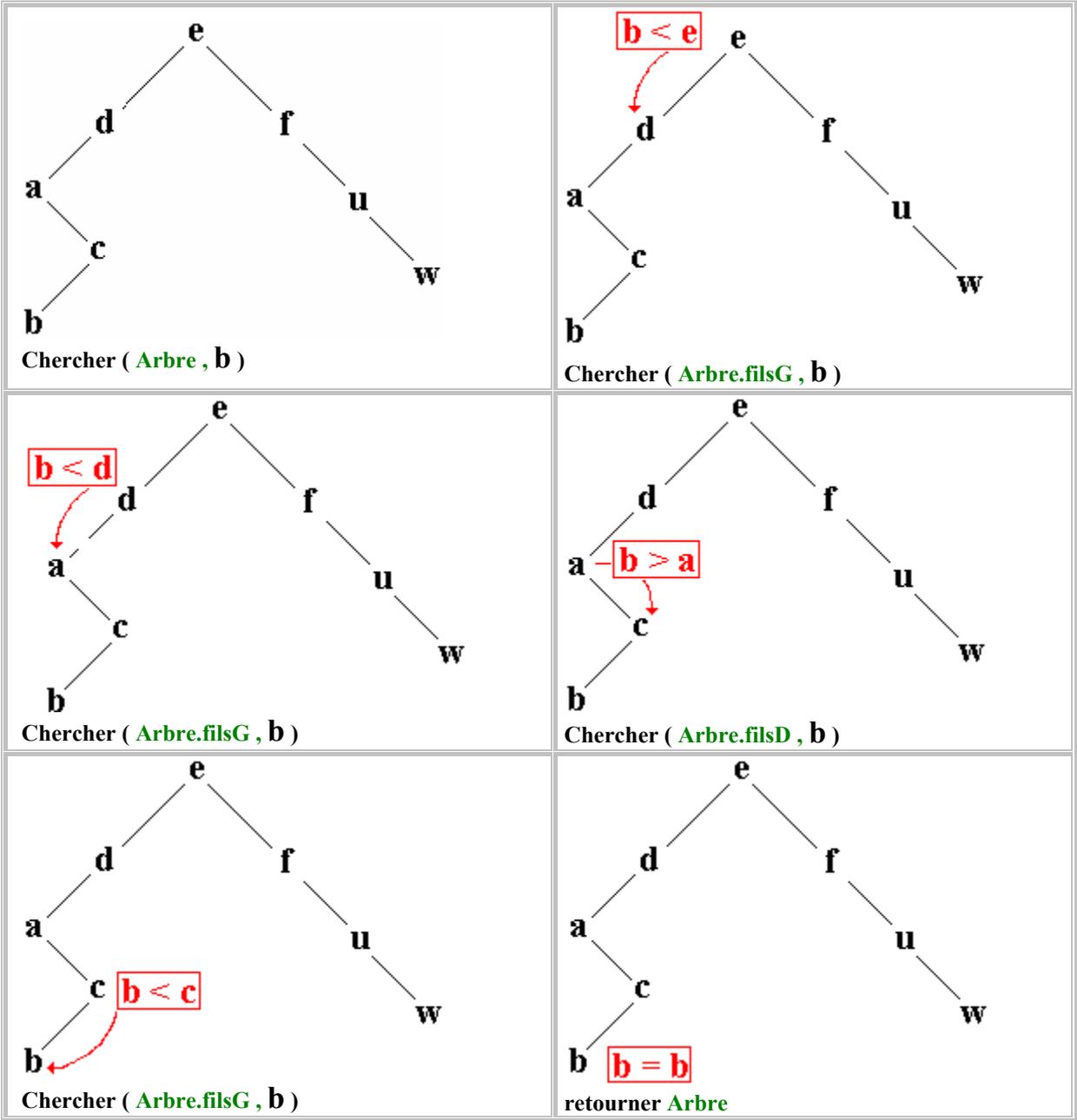
si clef (*Elt*) < clef (Arbre.Racine) alors

```

Chercher ( Arbre.filsG Elt ) //on cherche à gauche
sinon
  si clef ( Elt ) > clef ( Arbre.Racine ) alors
    Chercher ( Arbre.filsD Elt ) //on cherche à droite
  sinon retourner Arbre.Racine //l'élément est dans ce noeud
Fsi
Fsi
Fsi

```

Ci-dessous le suivi de l'algorithme de recherche du caractère **b** dans l'arbre précédent :



Algorithme de suppression dans un arbre binaire de recherche

Afin de pouvoir supprimer un élément dans un arbre binaire de recherche, il est nécessaire de pouvoir d'abord le localiser, ensuite supprimer le noeud ainsi trouvé et éventuellement procéder à la réorganisation de l'arbre de recherche.

Nous supposons que notre arbre binaire de recherche ne possède que des éléments tous distincts (pas de redondance).

```
supprimer l'élément Elt dans l'arbre Arbre :
Supprimer ( Arbre , Elt ) : Arbre
local noeudMax : Noeud
si Arbre =  $\emptyset$  alors
    Afficher Elt non trouvé dans l'arbre;
sinon
    si clef ( Elt ) < clef ( Arbre.Racine ) alors
        Supprimer ( Arbre.filsG , Elt ) //on cherche à gauche
    sinon
        si clef ( Elt ) > clef ( Arbre.Racine ) alors
            Supprimer ( Arbre.filsD , Elt ) //on cherche à droite
        sinon //l'élément est dans ce noeud
            si Arbre.filsG =  $\emptyset$  et Arbre.filsD  $\neq$   $\emptyset$  alors //sous-arbre gauche vide
                remplacer Arbre par son sous-arbre droit Arbre.filsD
            sinon
                si Arbre.filsD =  $\emptyset$  et Arbre.filsG  $\neq$   $\emptyset$  alors //sous-arbre droit vide
                    remplacer Arbre par son sous-arbre gauche Arbre.filsG
                sinon
                    si Arbre.filsD  $\neq$   $\emptyset$  et Arbre.filsG  $\neq$   $\emptyset$  alors //le noeud a deux descendants
                        noeudMax  $\leftarrow$  maxClef( Arbre.filsG ); //noeudMax = le max du fils gauche
                        clef ( Arbre.Racine )  $\leftarrow$  clef (noeudMax); //remplacer etiquette
                        Supprimer ( Arbre.filsG , clef (noeudMax) ) //on cherche à gauche
                    sinon //le noeud est alors une feuille
                        détruire ( Arbre )
            Fsi
        Fsi
    Fsi
Fsi
Fsi
Fsi
Fsi
```

Cet algorithme utilise l'algorithme récursif **maxClef** de recherche de la plus grandeclef dans l'arbre *Arbre* :

//par construction il suffit de descendre systématiquement toujours le plus à droite

```
maxClef ( Arbre ) : Arbre
si Arbre.filsD =  $\emptyset$  alors
    retourner Arbre.Racine //c'est le plus grand élément
sinon
```

```
maxClef ( Arbre.filsD )  
Fsi
```

Classe générique C# d'arbre binaire avec parcours en profondeur

Code C# d'une classe générique d'arbre binaire `ArbreBin<T0>` avec parcours en profondeur, traduit à partir des algorithmes précédents :

```
interface IArbreBin<T0> {  
    T0 Info { get; set; }  
}  
  
class ArbreBin<T0> : IArbreBin<T0>  
{  
    private T0 InfoLoc;  
    private ArbreBin<T0> fg;  
    private ArbreBin<T0> fd;  
  
    public ArbreBin(T0 s) : this() {  
        InfoLoc = s;  
    }  
  
    public ArbreBin()  
    {  
        InfoLoc = default(T0);  
        fg = default(ArbreBin<T0>);  
        fd = default(ArbreBin<T0>);  
    }  
  
    public T0 Info  
    {  
        get { return InfoLoc; }  
        set { InfoLoc = value; }  
    }  
  
    public ArbreBin<T0> filsG  
    {  
        get { return this.fg; }  
        set  
        {  
            if ( value != null )  
                fg = new ArbreBin<T0>(value.Info);  
            else  
                fg = default(ArbreBin<T0>);  
        }  
    }  
  
    public ArbreBin<T0> filsD  
    {  
        get { return this.fd; }  
        set  
        {  
            if ( value != null )  
                fd = new ArbreBin<T0>(value.Info);  
            else
```

```

        fd = default(ArbreBin<T0>);
    }
}

public void prefixe()
{
    Console.WriteLine(this.Info);
    if (fg != null)
        this.fg.prefixe();
    if (fd != null)
        this.fd.prefixe();
}

public void postfixe()
{
    if (fg != null)
        this.fg.postfixe();
    if (fd != null)
        this.fd.postfixe();
    Console.WriteLine(this.Info);
}

public void infixe()
{
    if (fg != null)
        this.fg.infixe();
    Console.WriteLine(this.Info);
    if (fd != null)
        this.fd.infixe();
}
}

```

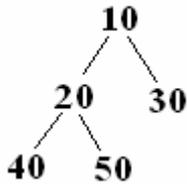
Cette classe C# permet de construire et de parcourir en profondeur selon les trois parcours précédemment étudié. Ci-après un exemple d'utilisation de cette classe T0 = int :

```

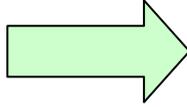
static void Main(string[] args)
{
    ArbreBin<int> treeRac = new ArbreBin<int>();
    treeRac.Info = 10;
    treeRac.filsG = new ArbreBin<int>(20);
    treeRac.filsD = new ArbreBin<int>(30);
    treeRac.filsG.filsG = new ArbreBin<int>(40);
    treeRac.filsG.filsD = new ArbreBin<int>(50);
    Console.WriteLine(" treeRac.info = " + treeRac.Info);
    Console.WriteLine(" treeRac.filsG.info = " + treeRac.filsG.Info);
    Console.WriteLine(" treeRac.filsD.info = " + treeRac.filsD.Info);
    Console.WriteLine(" treeRac.filsG.filsG.info = "+treeRac.filsG.filsG.Info);
    Console.WriteLine("--- parcours prefixe ---");
        treeRac.prefixe();
    Console.WriteLine("--- parcours postfixe ---");
        treeRac.postfixe();
    Console.WriteLine("--- parcours infixe ---");
        treeRac.infixe();
    Console.ReadLine();
}

```

La méthode Main construit cet arbre binaire :



Les parcours en profondeur donnent les résultats suivants :



```

treeRac.info = 10
treeRac.filsG.info = 20
treeRac.filsD.info = 30
treeRac.filsG.filsG.info = 40
--- parcours prefixe ---
10
20
40
50
30
--- parcours postfixe ---
40
50
20
30
10
--- parcours infixe ---
40
20
50
10
30
  
```

Classe générique C# d'arbre binaire avec parcours en largeur

On ajoute à la classe précédente le parcours en largeur, pour cela nous déclarons deux nouveaux champs privés dans la classe pour représenter la FIFO et le premier élément de cette FIFO. Pour la file FIFO, nous utilisons la classe générique de file de .Net `Queue<T0>`, le type d'élément T0 étant alors `ArbreBin<T>` :

```

private Queue<ArbreBin<T>> Fifo = new Queue<ArbreBin<T>>();
private ArbreBin<T> Premier;
  
```

<i>Algorithme</i>	<i>Méthode C#</i>
<p>Largeur (Arbre)</p> <p>si Arbre ≠ ∅ alors ajouter racine de l'Arbre dans Fifo; tantque Fifo ≠ ∅ faire prendre premier de Fifo; traiter premier de Fifo; si filsG de premier de Fifo ≠ ∅ alors ajouter filsG de premier de Fifo dans Fifo; Fsi si filsD de premier de Fifo ≠ ∅ alors ajouter filsD de premier de Fifo dans Fifo; Fsi ftant Fsi</p>	<pre> public void largeur() { Fifo.Enqueue(this); while (Fifo.Count != 0) { Premier = Fifo.Dequeue(); Console.WriteLine(Premier.Info); if (Premier.filsG != null) Fifo.Enqueue(Premier.filsG); if (Premier.filsD != null) Fifo.Enqueue(Premier.filsD); } } </pre>

Voici uniquement le code rajouté relatif au parcours en largeur, dans la classe `ArbreBin<T>`, le reste est identique au code défini dans les pages précédentes :

```

class ArbreBin<T0> : IArbreBin<T0>
{
    private T0 InfoLoc;
    private ArbreBin<T0> fg;
    private ArbreBin<T0> fd;
    public ArbreBin(T0 s) : this() { ... }
    public ArbreBin() { ... }
    public T0 Info { ... }
    public ArbreBin<T0> filsG { ... }
    public ArbreBin<T0> filsD { ... }
    public void prefixe() { ... }
    public void postfixe() { ... }
    public void infixe() { ... }

    private Queue<ArbreBin<T>> Fifo = new Queue<ArbreBin<T>>();
    private ArbreBin<T> Premier;
    public void largeur()
    {
        Fifo.Enqueue(this);
        while (Fifo.Count != 0)
        {
            Premier = Fifo.Dequeue();
            Console.WriteLine(Premier.Info);
            if (Premier.filsG != null)
                Fifo.Enqueue(Premier.filsG);
            if (Premier.filsD != null)
                Fifo.Enqueue(Premier.filsD);
        }
    }
}

```

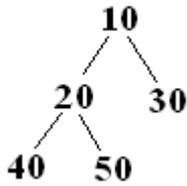
Ci-après, la reprise de l'exemple d'utilisation de cette classe avec T0 = int :

```

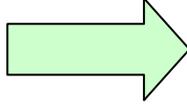
static void Main(string[] args)
{
    ArbreBin<int> treeRac = new ArbreBin<int>();
    treeRac.Info = 10;
    treeRac.filsG = new ArbreBin<int>(20);
    treeRac.filsD = new ArbreBin<int>(30);
    treeRac.filsG.filsG = new ArbreBin<int>(40);
    treeRac.filsG.filsD = new ArbreBin<int>(50);
    Console.WriteLine(" treeRac.info = " + treeRac.Info);
    Console.WriteLine(" treeRac.filsG.info = " + treeRac.filsG.Info);
    Console.WriteLine(" treeRac.filsD.info = " + treeRac.filsD.Info);
    Console.WriteLine(" treeRac.filsG.filsG.info = "+treeRac.filsG.filsG.Info);
    Console.WriteLine("--- parcours prefixe ---");
        treeRac.prefixe();
    Console.WriteLine("--- parcours postfixe ---");
        treeRac.postfixe();
    Console.WriteLine("--- parcours infixe ---");
        treeRac.infixe();
    Console.WriteLine("--- parcours Largeur ---");
        treeRac.largeur();
    Console.ReadLine();
}

```

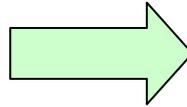
La méthode Main construit cet arbre binaire :



Les parcours en profondeur donnent les résultats suivants :



Le parcours en largeur donne le résultat suivant :



```

treeRac.info = 10
treeRac.filsG.info = 20
treeRac.filsD.info = 30
treeRac.filsG.filsG.info = 40
--- parcours prefixe ---
10
20
40
50
30
--- parcours postfixe ---
40
50
20
30
10
--- parcours infixe ---
40
20
50
10
30
--- parcours Largeur ---
10
20
30
40
50
  
```

Classe générique C# d'arbre binaire de recherche

On définit une nouvelle classe d'arbre binaire de recherche générique que nous nommons **ArbreBinRech<T>** dont chaque nœud est un arbre binaire de type **ArbreBin<T>**; on contraint le type T à implémenter l'interface **IComparable** afin de pouvoir travailler sur des données possédant une méthode **CompareTo** autorisant un test de comparaison de leurs valeurs :

1°) Méthode *placer* pour l'insertion dans un arbre binaire de recherche :

```

class ArbreBinRech<T> where T : IComparable<T>
{
    public ArbreBin<T> racine;

    public ArbreBinRech()
    {
        racine = null;
    }
    public ArbreBinRech(ArbreBin<T> tree)
    {
        racine = tree;
    }
}
  
```

Méthodes C#	Algorithme - rappel
<pre> public void placer(T elt) { if (racine == null) { racine = new ArbreBin<T>(elt); return; } placerR(racine, elt); } </pre>	<p>placer (Arbre Elt)</p> <p>si Arbre = ∅ alors creer un nouveau noeud contenant Elt ; Arbre.Racine = ce nouveau noeud</p> <p>sinon { - tous les éléments "info" de tous les noeuds du sous-arbre de gauche</p>

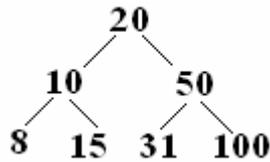
<pre>private void placerR(ArbreBin<T> tree, T elt) { if (elt.CompareTo(tree.Info) <= 0) { if (tree.filsG == null) tree.filsG = new ArbreBin<T>(elt); else this.placerR(tree.filsG, elt); } else { if (tree.filsD == null) tree.filsD = new ArbreBin<T>(elt); else this.placerR(tree.filsD, elt); } } }</pre>	<p><i>sont inférieurs ou égaux à l'élément "info" du noeud en cours (arbre)</i></p> <p><i>- tous les éléments "info" de tous les noeuds du sous-arbre de droite</i></p> <p><i>sont supérieurs à l'élément "info" du noeud en cours (arbre)</i></p> <p><i>}</i></p> <p>si <code>clef (Elt) ≤ clef (Arbre.Racine)</code> alors</p> <p>placer (Arbre.filsG Elt)</p> <p>sinon</p> <p>placer (Arbre.filsD Elt)</p> <p>Fsi</p>
---	--

}

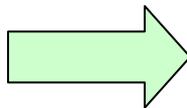
Ci-après, exemple d'utilisation de cette classe avec `T = int` qui implémente `IComparable` :

```
static void Main(string[] args)
{
    ArbreBinRech<int> treeRech = new ArbreBinRech<int>();
    treeRech.placer(20);
    treeRech.placer(50);
    treeRech.placer(100);
    treeRech.placer(10);
    treeRech.placer(15);
    treeRech.placer(8);
    treeRech.placer(31);
    Console.WriteLine("--- arbre recherche en Largeur ---");
    treeRech.racine.largeur();
    Console.WriteLine("--- arbre recherche en prefixe ---");
    treeRech.racine.prefixe();
    Console.WriteLine("--- arbre recherche en postfixe ---");
    treeRech.racine.postfixe();
    Console.WriteLine("--- arbre recherche en infixe ---");
    treeRech.racine.infixe();
    Console.Read();
}
```

La méthode Main précédente construit l'arbre binaire de recherche qui suit :



Les différents parcours donnent les résultats suivants :



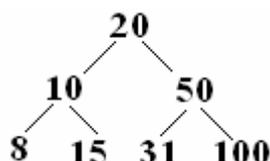
```

--- arbre recherche en Largeur ---
20
10
50
8
15
31
100
--- arbre recherche en prefixe ---
20
10
8
15
50
31
100
--- arbre recherche en postfixe ---
8
15
10
31
100
50
20
--- arbre recherche en infixe ---
8
10
15
20
31
50
100
  
```

2*) Méthode *chercher* pour la recherche dans un arbre binaire de recherche :

Méthodes C#	Algorithme - rappel
<pre> public ArbreBin<T> rechercher(T clef) { return rechercherR(racine, clef); } private ArbreBin<T> rechercherR(ArbreBin<T> tree, T clef) { if (tree == null) return null; if (clef.CompareTo(tree.Info) == 0) return tree; if (clef.CompareTo(tree.Info) < 0) return rechercherR(tree.filsG, clef); else return rechercherR(tree.filsD, clef); } </pre>	<p>chercher l'élément <i>Elt</i> dans l'arbre <i>Arbre</i> :</p> <p>Chercher (Arbre Elt) : Arbre</p> <p>si Arbre = ∅ alors Afficher <i>Elt</i> non trouvé dans l'arbre;</p> <p>sinon si clef (Elt) < clef (Arbre.Racine) alors Chercher (Arbre.filsG Elt) //on cherche à gauche</p> <p>sinon si clef (Elt) > clef (Arbre.Racine) alors Chercher (Arbre.filsD Elt) //on cherche à droite</p> <p>sinon retourner Arbre.Racine //l'élément est dans ce noeud</p> <p>Fsi Fsi Fsi</p>

On utilise l'arbre binaire de recherche précédent :

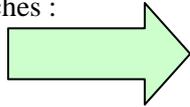


Dans la méthode Main précédente on ajoute le code suivant permettant de rechercher les nœuds de clefs fixées (31 et 78) :

```

Console.WriteLine("--- recherche de clef : 31 ---");
if (treeRech.rechercher(31) == null)
    Console.WriteLine(">>> 31 pas trouvée.");
else
    Console.WriteLine(">>> " + treeRech.rechercher(31).Info);
  
```

Résultats des recherches :



```
Console.WriteLine("--- recherche de clef : 78 ---");
if (treeRech.rechercher(78) == null)
    Console.WriteLine(">>> 78 pas trouvée.");
else
    Console.WriteLine(">>> " + treeRech.rechercher(78).Info);
```

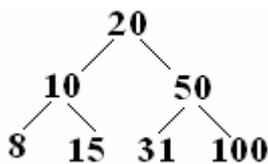
```
--- recherche de clef : 31 ---
>>> 31
--- recherche de clef : 78 ---
>>> 78 pas trouvée.
```

3*) Méthode *supprimer* pour la suppression dans un arbre binaire de recherche :

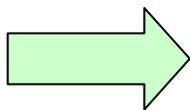
Méthodes C#	Algorithme - rappel
<pre>public void supprimer(T clef) { if (racine != null) supprimerR(racine, null, 'X', clef); } private void supprimerR(ArbreBin<T> tree, ArbreBin<T> parent, char branche, T clef) { ArbreBin<T> noeudMax; if (tree == null) return;//clef non trouvée else if (clef.CompareTo(tree.Info) < 0) { //on cherche à gauche : this.supprimerR(tree.filsG, tree, 'G', clef); } else if (clef.CompareTo(tree.Info) > 0) { //on cherche à droite : this.supprimerR(tree.filsD, tree, 'D', clef); } else //l'élément est dans ce nœud: if (tree.filsD == null && tree.filsG != null) { //sous-arbre droit vide //remplacer arbre par son sous-arbre gauche : tree.Info = tree.filsG.Info; tree.filsD = tree.filsG.filsD; tree.filsG = tree.filsG.filsG; } else if (tree.filsG == null && tree.filsD != null) { //sous-arbre gauche vide //remplacer arbre par son sous-arbre droit : tree.Info = tree.filsD.Info; tree.filsD = tree.filsD.filsD; tree.filsG = tree.filsD.filsG; } else if (tree.filsG != null && tree.filsD != null) { // le noeud a 2 fils //le max du fils gauche :</pre>	<p>Cette méthode suit très exactement l'algorithme proposé plus haut.</p> <p>La méthode <code>supprimerR</code> possède comme dans les deux traitements précédents les deux paramètres <code>ArbreBin<T> tree</code> et <code>T clef</code>, elle possède en plus 2 autres paramètres :</p> <ul style="list-style-type: none"> • <code>ArbreBin<T> parent</code> = la référence du parent du nœud actuel. • <code>char branche</code> = un caractère 'G' ou 'D' indiquant si le nœud actuel est un filsG ou un filsD du parent. <p>L'implantation du remplacement d'un nœud par son fils gauche s'effectue ainsi :</p> <pre>tree.Info = tree.filsG.Info; tree.filsD = tree.filsG.filsD; tree.filsG = tree.filsG.filsG;</pre> <p>L'implantation du remplacement d'un nœud par son fils droit s'effectue ainsi :</p> <pre>tree.Info = tree.filsD.Info; tree.filsD = tree.filsD.filsD; tree.filsG = tree.filsD.filsG;</pre>

<pre> noeudMax = maxClef(tree.filsG); tree.Info = noeudMax.Info; //← remplacer clef //on cherche à gauche : this.supprimerR(tree.filsG, tree, 'G', noeudMax.Info); } else // le noeud est une feuille: on le détruit { if (branche == 'D') parent.filsD = null; else if (branche == 'G') parent.filsG = null; } } private ArbreBin<T> maxClef(ArbreBin<T> tree) { if (tree.filsD == null) return tree; else return maxClef(tree.filsD); } </pre>	<p>L'implantation de la destruction du nœud contenant la clef est effectuée par la suppression dans le parent de la référence pointant vers ce nœud.</p> <p>Le caractère "branche" permet de savoir si ce nœud est le fils gauche (branche == 'G') ou bien le fils droit du parent (branche == 'D').</p> <p>fils gauche => parent.filsG = null; fils droit => parent.filsD = null;</p>
---	---

On utilise l'arbre binaire de recherche précédent :



Résultats de diverses suppressions :



Dans la méthode Main précédente on ajoute le code suivant permettant de rechercher les nœuds de clefs fixées (31 et 78) :

```

int clef = 20;
Console.WriteLine("--- suppression clef : " + clef + " ---");
treeRech.supprimer(clef);
treeRech.racine.largeur();

```

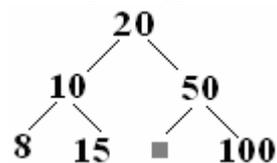
clef = 31

```

--- suppression clef : 31
20
10
50
8
15
100

```

arbre après suppression :



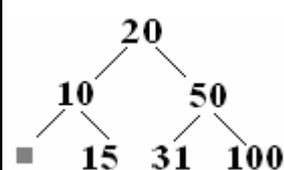
clef = 8

```

--- suppression clef : 8
20
10
50
15
31
100

```

arbre après suppression :



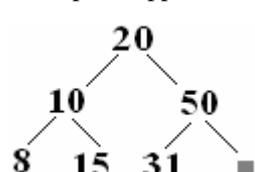
clef = 100

```

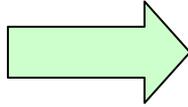
--- suppression clef : 100
20
10
50
8
15
31

```

arbre après suppression :



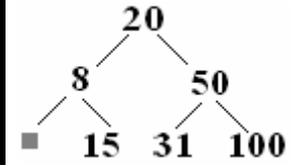
Résultats de diverses suppressions :



clef = 10

```
--- suppression clef : 10
20
8
50
15
31
100
```

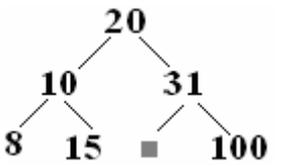
arbre après suppression :



clef = 50

```
--- suppression clef : 50
20
10
31
8
15
100
```

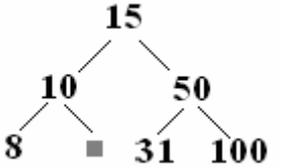
arbre après suppression :



clef = 20

```
--- suppression clef : 20
15
10
50
8
31
100
```

arbre après suppression :



Principes des bases de données



Plan général: 

- 1. Introduction et Généralités**
- 2. Le modèle de données relationnelles**
- 3. Principes fondamentaux d'une algèbre relationnelle**
- 4. SQL et Algèbre relationnelle**

1. Introduction et Généralités

1.1 Notion de système d'information

L'informatique est une science du traitement de l'information, laquelle est représentée par des données. Aussi, très tôt, on s'est intéressé aux diverses manières de pouvoir stocker des données dans des mémoires auxiliaires autres que la mémoire centrale. Les données sont stockées dans des périphériques dont les supports physiques ont évolué dans le temps : entre autres, d'abord des cartes perforées, des bandes magnétiques, des cartes magnétiques, des mémoires à bulles magnétiques, puis aujourd'hui des disques magnétiques, ou des CD-ROM ou des DVD.

La notion de fichier est apparue en premier : le fichier regroupe tout d'abord des objets de même nature, des enregistrements. Pour rendre facilement exploitables les données d'un fichier, on a pensé à différentes méthodes d'accès (accès séquentiel, direct, indexé).

Toute application qui gère des systèmes physiques doit disposer de paramètres sémantiques décrivant ces systèmes afin de pouvoir en faire des traitements. Dans des systèmes de gestion de clients les paramètres sont très nombreux (noms, prénoms, adresse, n°Sécu, sport favori, est satisfait ou pas,...) et divers (alphabétiques, numériques, booléens, ...).

Dès que la quantité de données est très importante, les fichiers montrent leurs limites et il a fallu trouver un moyen de stocker ces données et de les organiser d'une manière qui soit facilement accessible.

Base de données (BD)

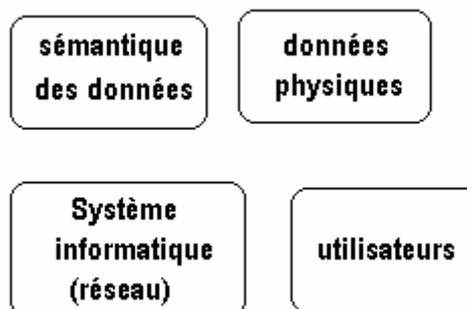
Une BD est composée de données stockées dans des mémoires de masse sous une forme structurée, et accessibles par des applications différentes et des utilisateurs différents. Une BD doit pouvoir être utilisée par plusieurs utilisateurs en "même temps".



Une base de données est structurée par définition, mais sa structuration doit avoir un caractère universel : il ne faut pas que cette structure soit adaptée à une application particulière, mais qu'elle puisse être utilisable par plusieurs applications distinctes. En effet, un même ensemble de données peut être commun à plusieurs systèmes de traitement dans un problème physique (par exemple la liste des passagers d'un avion, stockée dans une base de données, peut aussi servir au service de police à vérifier l'identité des personnes interdites de séjour, et au service des douanes pour associer des bagages aux personnes....).

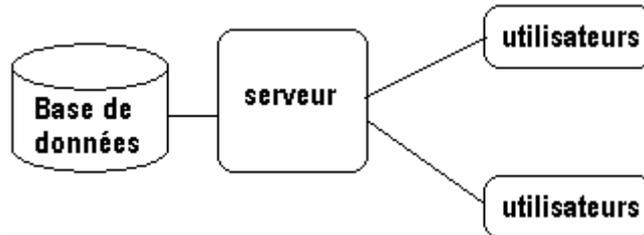
Système d'information

Dans une entreprise ou une administration, la structure sémantique des données, leur organisation logique et physique, le partage et l'accès à de grandes quantités de données grâce à un système informatique, se nomme un système d'information.



Les entités qui composent un système d'information

L'organisation d'un SI relève plus de la gestion que de l'informatique et n'a pas exactement sa place dans un document sur la programmation. En revanche la cheville ouvrière d'un système d'information est un outil informatique appelé un **SGBD** (système de gestion de base de données) qui repose essentiellement sur un système informatique composé traditionnellement d'une **BD** et d'un réseau de postes de travail consultant ou mettant à jour les informations contenues dans la base de données, elle-même généralement située sur un ordinateur-serveur.



Systeme de Gestion de Base de Données (SGBD)

Un SGBD est un ensemble de logiciels chargés d'assurer les fonctions minimales suivantes :

- Le maintien de la cohérence des données entre elles,
- le contrôle d'intégrité des données accédées,
- les autorisations d'accès aux données,
- les opérations classiques sur les données (consultation, insertion, modification, suppression)

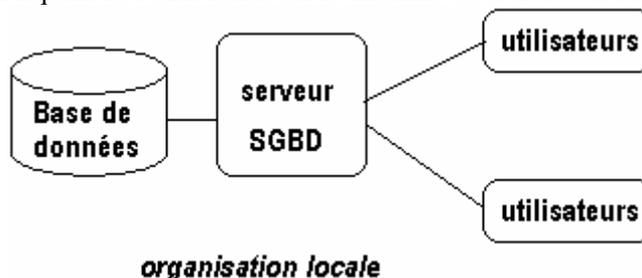
La cohérence des données est subordonnée à la définition de contraintes d'intégrité qui sont des règles que doivent satisfaire les données pour être acceptées dans la base. Les contraintes d'intégrité sont contrôlées par le moteur du SGBD :

- au niveau de chaque champ, par exemple le : prix est un nombre positif, la date de naissance est obligatoire.
- Au niveau de chaque table - voir plus loin la notion de clef primaire : deux personnes ne doivent pas avoir à la fois le même nom et le même prénom.
- Au niveau des relations entre les tables : contraintes d'intégrité référentielles.

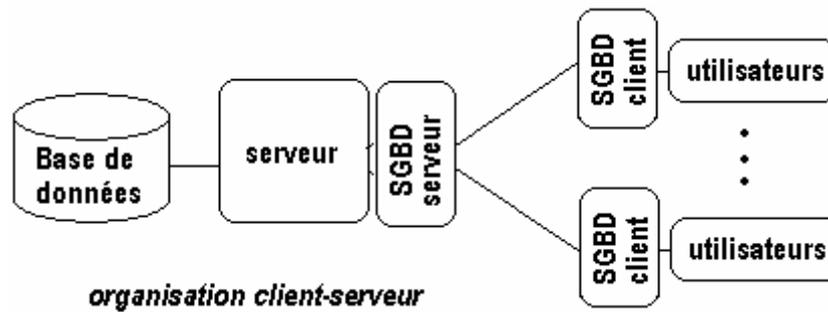
Par contre la redondance des données (formes normales) **n'est absolument pas vérifiée automatiquement** par les SGBD, il faut faire des requêtes spécifiques de recherche d'anomalies (dites post-mortem) **à postériori**, ce qui semble être une grosse lacune de ces systèmes puisque les erreurs sont déjà présentes dans la base !

On organise actuellement les SGBD selon deux modes :

L'organisation locale selon laquelle le SGBD réside sur la machine où se trouve la base de données :



L'organisation client-serveur selon laquelle sur le SGBD est réparti entre la machine serveur locale supportant la BD (partie SGBD serveur) et les machines des utilisateurs (partie SGBD client). Ce sont ces deux parties du SGBD qui communiquent entre elles pour assurer les transactions de données :



Le caractère généraliste de la structuration des données induit une description abstraite de l'objet BD (Base de données). Les applications étant indépendantes des données, ces dernières peuvent donc être manipulées et changées indépendamment du programme qui y accédera en implantant les méthodes générales d'accès aux données de la base, conformément à sa structuration abstraite.

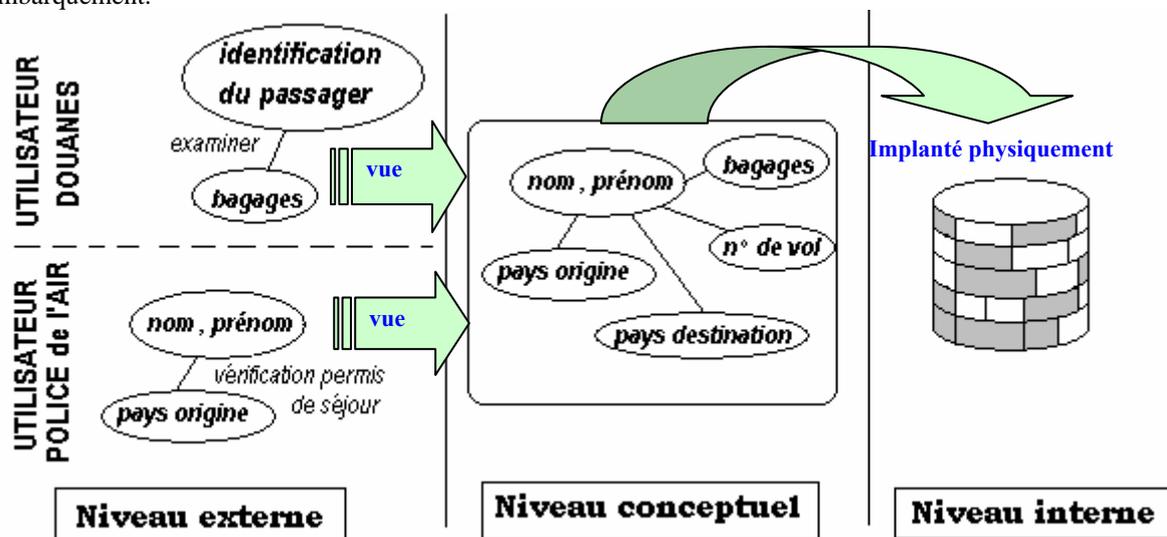
Une Base de Données peut être décrite de plusieurs points de vue, selon que l'on se place du côté de l'utilisateur ou bien du côté du stockage dans le disque dur du serveur ou encore du concepteur de la base.

Il est admis de nos jours qu'une **BD** est décrite en trois niveaux d'abstraction : un seul niveau a une existence matérielle physique et les deux autres niveaux sont une explication abstraite de ce niveau matériel.

Les 3 niveaux d'abstraction définis par l'ANSI depuis 1975

- ❑ **Niveau externe** : correspond à ce que l'on appelle une vue de la BD ou la façon dont sont perçues au niveau de l'utilisateur les données manipulées par une certaine application (vue abstraite sous forme de schémas)
- ❑ **Niveau conceptuel** : correspond à la description abstraite des composants et des processus entrant dans la mise en œuvre de la BD. Le niveau conceptuel est le plus important car il est le résultat de la traduction de la description du monde réel à l'aide d'expressions et de schémas conformes à un modèle de définition des données.
- ❑ **Niveau interne** : correspond à la description informatique du stockage physique des données (fichiers séquentiels, indexages, tables de hachage,...) sur le disque dur.

Figurons pour l'exemple des passagers d'un avion, stockés dans une base de données de la compagnie aérienne, sachant qu'en plus du personnel de la compagnie qui a une vue externe commerciale sur les passagers, le service des douanes peut accéder à un passager et à ses bagages et la police de l'air peut accéder à un passager et à son pays d'embarquement.

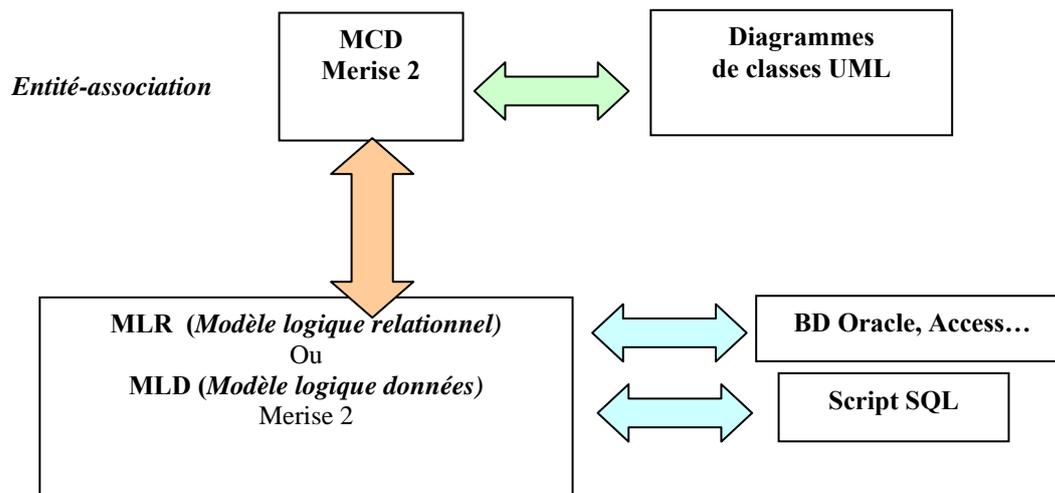


Le niveau conceptuel forme l'élément essentiel d'une BD et donc d'un SGBD chargé de gérer une BD, il est décrit avec un modèle de conception de données MCD avec la méthode française Merise qui est très largement répandu, ou bien par le formalisme des diagrammes de classes UML qui prend une part de plus en plus grande dans le formalisme de description conceptuelle des données (rappelons qu'UML est un langage de modélisation formelle, orienté objet et graphique ; Merise2 a intégré dans Merise ces concepts mais ne semble pas beaucoup être utilisé). Nous renvoyons le lecteur intéressé par cette partie aux très nombreux ouvrages écrits sur Merise ou sur UML.

Dans la pratique actuelle les logiciels de conception de BD intègrent à la fois la méthode Merise 2 et les diagrammes de classes UML. Ceci leur permet surtout la génération automatique et semi-automatique (paramétrable) de la BD à partir du modèle conceptuel sous forme de scripts (programmes simples) SQL adaptés aux différents SGBD du marché (ORACLE, SYBASE, MS-SQLSERVER,...) et les différentes versions de la BD ACCESS.

Les logiciels de conception actuels permettent aussi la rétro-génération (ou reverse engineering) du modèle à partir d'une BD existante, cette fonctionnalité est très utile pour reprendre un travail mal documenté.

En résumé pratique :

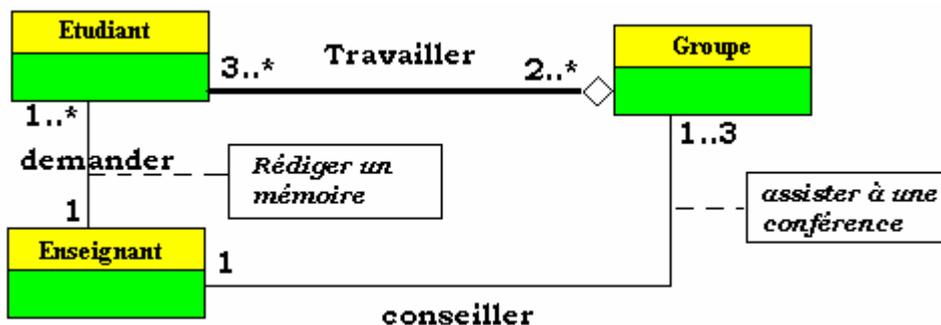


C'est en particulier le cas du logiciel français WIN-DESIGN dont une version démo est disponible à www.win-design.com et de son rival POWER-AMC (ex AMC-DESIGNOR).

Signalons enfin un petit logiciel plus modeste, très intéressant pour débiter avec version limitée seulement par la taille de l'exemple : CASE-STUDIO chez CHARONWARE. Les logiciels basés uniquement sur UML sont, à ce jour, essentiellement destinés à la génération de code source (Java, Delphi, VB, C++,...), les versions **Community** (versions logicielles libres) de ces logiciels ne permettent pas la génération de BD ni celle de scripts SQL. Les quelques schémas qui illustreront ce chapitre seront décrits avec le langage UML.

L'exemple ci-après schématise en UML le mini-monde universitaire réel suivant :

- ❑ un enseignant pilote entre 1 et 3 groupes d'étudiants,
- ❑ un enseignant demande à 1 ou plusieurs étudiants de rédiger un mémoire,
- ❑ un enseignant peut conseiller aux groupes qu'il pilote d'aller assister à une conférence,
- ❑ un groupe est constitué d'au moins 3 étudiants,
- ❑ un étudiant doit s'inscrire à au moins 2 groupes.



Si le niveau conceptuel d'une BD est assis sur un modèle de conceptualisation de haut niveau (Merise, UML) des données, il est ensuite fondamentalement traduit dans le Modèle Logique de représentation des Données (MLD). Ce dernier s'implémentera selon un modèle physique des données.

Il existe plusieurs MLD Modèles Logiques de Données et plusieurs modèles physiques, et pour un même MLD, on peut choisir entre plusieurs modèles physiques différents.

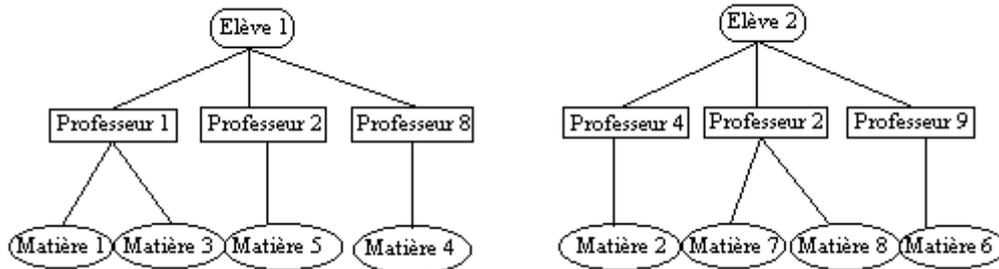
Il existe 5 grands modèles logiques pour décrire les bases de données.

Les modèles de données historiques

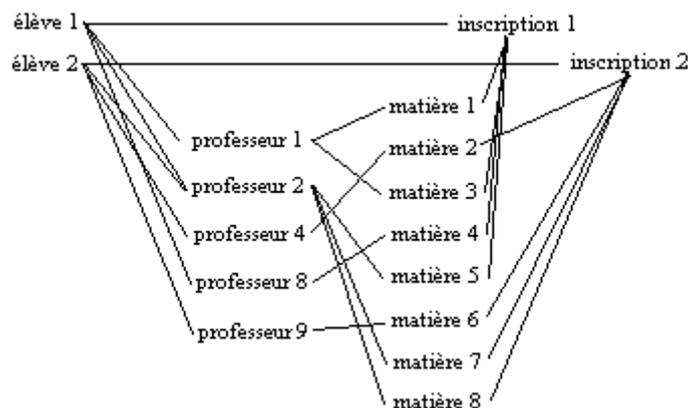
(Prenons un exemple comparatif où des élèves ont des cours donnés par des professeurs leur enseignant certaines

matières (les enseignants étant pluridisciplinaires)

- **Le modèle hiérarchique:** l'information est organisée de manière arborescente, accessible uniquement à partir de la racine de l'arbre hiérarchique. Le problème est que les points d'accès à l'information sont trop restreints.



- **Le modèle réseau:** toutes les informations peuvent être associées les unes aux autres et servir de point d'accès. Le problème est la trop grande complexité d'une telle organisation.



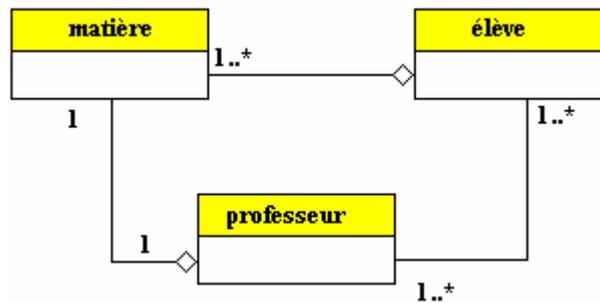
- **Le modèle relationnel:** toutes les relations entre les objets contenant les informations sont décrites et représentées sous la forme de tableaux à 2 dimensions.

élève 1	matière 1
élève 1	matière 3
élève 1	matière 4
élève 1	matière 5
élève 2	matière 2
élève 2	matière 6
élève 2	matière 7
élève 2	matière 8

professeur 1	matière 1
professeur 1	matière 3
professeur 2	matière 5
professeur 2	matière 7
professeur 2	matière 8
professeur 4	matière 2
professeur 8	matière 4
professeur 9	matière 6

Dans ce modèle, la gestion des données (insertion, extraction,...) fonctionne selon la théorie mathématique de l'algèbre relationnelle. C'est le modèle qui allie une grande indépendance vis à vis des données à une simplicité de description.

- **Le modèle par déduction :** comme dans le modèle relationnel les données sont décrites et représentées sous la forme de tableaux à 2 dimensions. La gestion des données (insertion, extraction,...) fonctionne selon la théorie mathématique du calcul dans la logique des prédicats. Il ne semble exister de SGBD commercial directement basé sur ce concept. Mais il est possible de considérer un programme Prolog (programmation en logique) comme une base de données car il intègre une description des données. Ce sont plutôt les logiciels de réseaux sémantiques qui sont concernés par cette approche (cf. logiciel AXON).
- **Le modèle objet :** les données sont décrites comme des classes et représentées sous forme d'objets, un modèle **relationnel-objet** devrait à terme devenir le modèle de base.



L'expérience montre que le modèle relationnel s'est imposé parce qu'il était le plus simple en terme d'indépendance des données par rapport aux applications et de facilité de représenter les données dans notre esprit. C'est celui que nous décrivons succinctement dans la suite de ce chapitre.

2. Le modèle de données relationnelles

Défini par EF Codd de la société IBM dès 1970, ce modèle a été amélioré et rendu opérationnel dans les années 80 sous la forme de SGBD-R (SGBD Relationnels). Ci-dessous une liste non exhaustive de tels SGBD-R :

- Access de Microsoft,
- Oracle,
- DB2 d'IBM,
- Interbase de Borland,
- SQL server de microsoft,
- Informix,
- Sybase,
- MySQL,
- PostgreSQL,

Nous avons déjà vu dans un précédent chapitre, la notion de relation binaire : une relation binaire R est un sous-ensemble d'un produit cartésien de deux ensembles finis E et F que nous nommerons domaines de la relation R :

$$R \subseteq E \times F$$

Cette définition est généralisable à n domaines, nous dirons que R est une relation n -aire sur les domaines E_1, E_2, \dots, E_n si et seulement si :

$$R \subseteq E_1 \times E_2 \dots \times E_n$$

Les ensembles E_k peuvent être définis comme en mathématiques : en extension ou en compréhension :

$$\left| \begin{array}{l} E_k = \{ 12, 58, 36, 47 \} \text{ en extension} \\ E_k = \{ x / (x \text{ est entier et } x \in [1, 20]) \} \text{ en compréhension} \end{array} \right.$$

Notation

si nous avons: $R = \{ (v_1, v_2 \dots, v_n) \}$,
 Au lieu d'écrire : $(v_1, v_2 \dots, v_n) \in R$, on écrira $R(v_1, v_2, \dots, v_n)$

Exemple de déclarations de relations :

Passager (nom, prénom, n° de vol, nombre de bagages), cette relation contient les informations utiles sur un passager d'une ligne aérienne.

Personne (nom, prénom), cette relation caractérise une personne avec deux attributs

Enseignement (professeur, matière) , cette relation caractérise un enseignement avec le nom de la matière et le professeur qui l'enseigne.

Schéma d'une relation

On appelle schéma de la relation $R : \mathbf{R}(a_1 : E_1, a_2 : E_2, \dots, a_n : E_n)$

Où $(a_1 , a_2 \dots , a_n)$ sont appelés les **attributs**, chaque attribut a_k indique comment est utilisé dans la relation R le domaine E_k , chaque attribut prend sa valeur dans le domaine qu'il définit, nous notons $val(a_k) = v_k$ où v_k est un élément (une valeur) quelconque de l'ensemble E_k (domaine de l'attribut a_k).

Convention : lorsqu'il n'y a pas de valeur associée à un attribut dans un n-uplet, on convient de lui mettre une valeur spéciale notée **null**, indiquant l'absence de valeur de l'attribut dans ce n-uplet.

Degré d'une relation

On appelle degré d'une relation, le nombre d'attributs de la relation.

Exemple de schémas de relations :

Passager (nom : chaîne, prénom : chaîne, n° de vol : entier, nombre de bagages : entier) relation de degré 4.

Personne (nom : chaîne, prénom : chaîne) relation de degré 2.

Enseignement (professeur : ListeProf, matière : ListeMat) relation de degré 2.

*Attributs : prenons le schéma de la relation **Enseignement***

Enseignement (professeur : ListeProf, matière : ListeMat). C'est une relation binaire (degré 2) sur les deux domaines ListeProf et ListeMat. L'attribut professeur joue le rôle d'un paramètre formel et le domaine ListeProf celui du type du paramètre.

Supposons que :

ListeProf = { Poincaré, Einstein, Lavoisier, Rimbault , Planck }

ListeMat = { mathématiques, poésie , chimie , physique }

L'attribut professeur peut prendre toutes valeurs de l'ensemble ListeProf :

Val(professeur) = Poincaré, ..., Val(professeur) = Rimbault

Si l'on veut dire que le poste d'enseignant de chimie n'est pas pourvu on écrira :

Le couple (**null** , chimie) est un couple de la relation **Enseignement**.

Enregistrement dans une relation

Un n-uplet $(val(a_1), val(a_2) \dots , val(a_n)) \in \mathbf{R}$ est appelé un enregistrement de la relation \mathbf{R} . Un enregistrement est donc constitué de valeurs d'attributs.

Dans l'exemple précédent (Poincaré , mathématiques), (Rimbault , poésie) , (**null** , chimie) sont trois enregistrements de la relation **Enseignement**.

Clef d'une relation

Si l'on peut caractériser d'une façon **bijective** tout n-uplet d'attributs $(a_1 , a_2 \dots , a_n)$ avec seulement un sous-ensemble restreint $(a_{k1} , a_{k2} \dots , a_{kp})$ avec $p < n$, de ces attributs, alors ce sous-ensemble est appelé une **clef** de la relation. Une relation peut avoir plusieurs clefs, nous choisissons l'une d'elle en la désignant comme **clef primaire de la relation**.

Clef minimale d'une relation

On a intérêt à ce que la clef **primaire soit minimale** en nombre d'attributs, car il est clair que si un sous-ensemble à p attributs $(a_{k1}, a_{k2}, \dots, a_{kp})$ est une clef, tout sous-ensemble à $p+m$ attributs dont les p premiers sont les $(a_{k1}, a_{k2}, \dots, a_{kp})$ est aussi une clef :

$(a_{k1}, a_{k2}, \dots, a_{kp}, a_0, a_1)$

$(a_{k1}, a_{k2}, \dots, a_{kp}, a_{10}, a_5, a_9, a_2)$ sont aussi des clefs etc...

Il n'existe aucun moyen méthodique formel général pour trouver une clef primaire d'une relation, il faut observer attentivement. Par exemple :

- Le code Insee est une clef primaire permettant d'identifier les personnes.
- Si le couple (nom, prénom) peut suffire pour identifier un élève dans une classe d'élèves, et pourrait être choisi comme clef primaire, il est insuffisant au niveau d'un lycée où il est possible que l'on trouve plusieurs élèves portant le même nom et le même premier prénom ex: (martin, jean).

Convention : on souligne dans l'écriture d'une relation dont on a déterminé une clef primaire, les attributs faisant partie de la clef.

Clef secondaire d'une relation

Tout autre clef de la relation qu'une clef primaire (minimale), exemple :

Si $(a_{k1}, a_{k2}, \dots, a_{kp})$ est un clef primaire de R

$(a_{k1}, a_{k2}, \dots, a_{kp}, a_0, a_1)$ et $(a_{k1}, a_{k2}, \dots, a_{kp}, a_{10}, a_5, a_9, a_2)$ sont des clefs secondaires.

Clef étrangère d'une relation

Soit $(a_{k1}, a_{k2}, \dots, a_{kp})$ un p -uplet d'attributs d'une relation R de degré n . [$R(a_1 : E_1, a_2 : E_2, \dots, a_n : E_n)$]

Si $(a_{k1}, a_{k2}, \dots, a_{kp})$ est une clef primaire d'une autre relation Q on dira que $(a_{k1}, a_{k2}, \dots, a_{kp})$ est une clef étrangère de R .

Convention : on met un # après chaque attribut d'une clef étrangère.

Exemple de clef secondaire et clef étrangère :

Passager (nom# : chaîne, prénom# : chaîne, n° de vol : entier, nombre de bagages : entier, n° client : entier) relation de degré 5.

Personne (nom : chaîne, prénom : chaîne, âge : entier, civilité : Etatcivil) relation de degré 4.

n° client est une clef primaire de la relation **Passager**.

(nom, n° client) est une clef secondaire de la relation **Passager**.

(nom, n° client, n° de vol) est une clef secondaire de la relation **Passager**....etc

(nom, prénom) est une clef primaire de la relation **Personne**, comme (nom#, prénom#) est aussi un couple d'attributs de la relation **Passager**, c'est une clef étrangère de la relation **Passager**.

On dit aussi que dans la relation **Passager**, le couple d'attributs (nom#, prénom#) réfère à la relation **Personne**.

Règle d'intégrité référentielle

Toutes les valeurs d'une clef étrangère ($V_{k1}, V_{k2}, \dots, V_{kp}$) se retrouvent comme valeur de la clef primaire de la relation référée (ensemble des valeurs de la clef étrangère est **inclus** au sens large dans l'ensemble des valeurs de la clef primaire)

Reprenons l'exemple précédent

(nom, prénom) est une clef étrangère de la relation **Passager**, c'est donc une clef primaire de la relation **Personne** : les domaines (liste des noms et liste des prénoms associés au nom doivent être strictement les mêmes dans **Passager** et dans **Personne**, nous dirons que la clef étrangère respecte la contrainte d'intégrité référentielle.

Règle d'intégrité d'entité

Aucun des attributs participant à une clef primaire ne peut avoir la valeur **null**.

Nous définirons la valeur **null**, comme étant une valeur spéciale n'appartenant pas à un domaine spécifique mais ajoutée par convention à tous les domaines pour indiquer qu'un champ n'est pas renseigné.

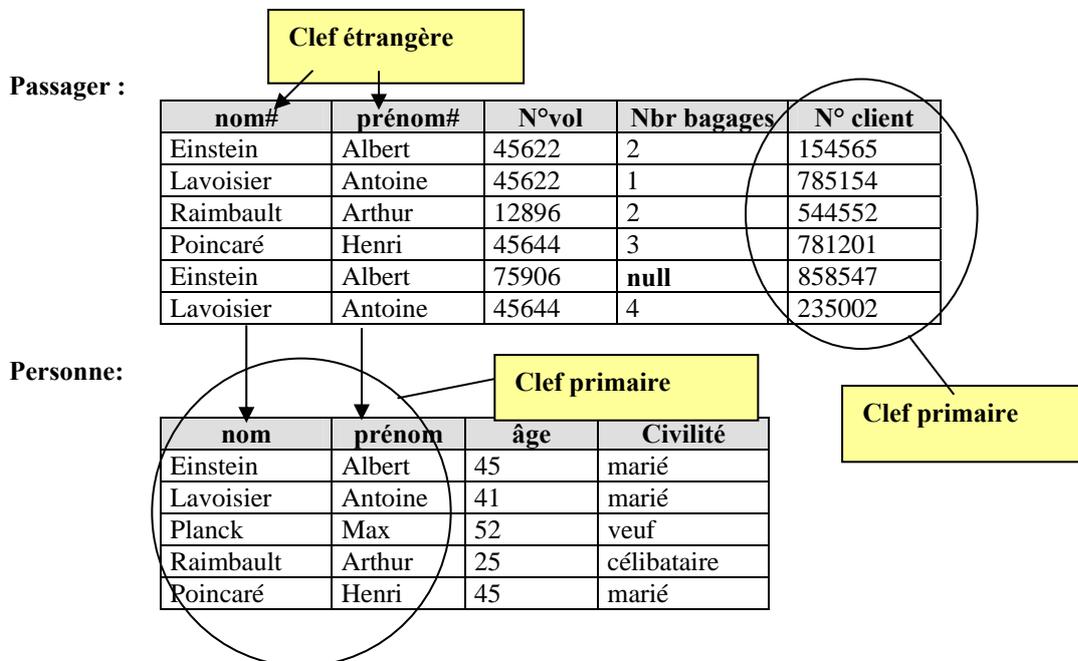
Représentation sous forme tabulaire

Reprenons les relations Passager et Personne et figurons un exemple pratique de valeurs des relations.

Passager (nom# : chaîne, prénom# : chaîne, n° de vol : entier, nombre de bagages : entier, n° client : entier).

Personne (nom : chaîne, prénom : chaîne, âge : entier, civilité : Etatcivil) relation de degré 4.

Nous figurons les tables de valeurs des deux relations



Nous remarquons que la compagnie aérienne attribue un numéro de client unique à chaque personne, c'est donc un bon choix pour une clef primaire.

Les deux tables (relations) ont deux colonnes qui portent les mêmes noms colonne **nom** et colonne **prénom**, ces deux colonnes forment une clef primaire de la table **Personne**, c'est donc une clef étrangère de **Passager** qui réfère **Personne**.

En outre, cette clef étrangère respecte la contrainte d'intégrité référentielle : la liste des valeurs de la clef étrangère dans **Passager** est incluse dans la liste des valeurs de la clef primaire associée dans **Personne**.

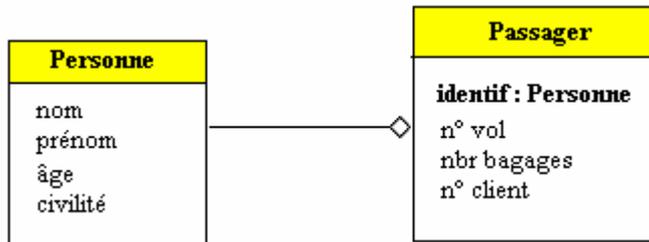


Diagramme UML modélisant la liaison Passager-Personne

Ne pas penser qu'il en est toujours ainsi, par exemple voici une autre relation **Passager2** dont la clef étrangère ne respecte pas la contrainte d'intégrité référentielle :

Clef étrangère, réfère Personne

Passager2 :

nom	prénom	N°vol	Nbr bagages	N° client
Einstein	Albert	45622	2	154565
Lavoisier	Antoine	45644	1	785154
Raimbault	Arthur	12896	2	544552
Poincaré	Henri	45644	3	781201
Einstein	Albert	75906	null	858547
Picasso	Pablo	12896	5	458023

En effet, le couple (Picasso, Pablo) n'est pas une valeur de la clef primaire dans la table **Personne**.

Principales règles de normalisation d'une relation

1^{ère} forme normale :

Une relation est dite en première forme normale si, chaque attribut est représenté par un identifiant unique (les valeurs ne sont pas des ensembles, des listes,...) .Ci-dessous une relation qui n'est pas en 1^{ère} forme normale car l'attribut **n° vol** est multivalué (il peut prendre 2 valeurs) :

nom	prénom	N°vol	Nbr bagage	N° client
Einstein	Albert	45622 , 75906	2	154565
Lavoisier	Antoine	45644, 45622	1	785154
Raimbault	Arthur	12896	2	544552
Poincaré	Henri	45644	3	781201
Picasso	Pablo	12896	5	458023

En pratique, il est très difficile de faire vérifier automatiquement cette règle, dans l'exemple proposé on pourrait imposer de passer par un masque de saisie afin que le N°vol ne comporte que 5 chiffres.

2^{ème} forme normale :

Une relation est dite en deuxième forme normale si, elle est **en première forme normale** et si chaque attribut qui n'est pas une clef, dépend entièrement de tous les attributs qui composent la clef primaire. La relation **Personne** (**nom** : chaîne, **prénom** : chaîne , âge : entier , civilité : Etatcivil) est en deuxième forme normale :

nom	prénom	âge	Civilité
Einstein	Albert	45	marié
Lavoisier	Antoine	41	marié
Planck	Max	52	veuf

Rimbault	Arthur	25	célibataire
Poincaré	Henri	45	marié

Car l'attribut âge ne dépend que du nom et du prénom, de même pour l'attribut civilité.

La relation **Personne3** (nom : chaîne, prénom : chaîne , âge : entier , civilité : Etatcivil) qui a pour clef primaire (nom , âge) n'est pas en deuxième forme normale :

<u>nom</u>	prénom	<u>âge</u>	Civilité
Einstein	Albert	45	marié
Lavoisier	Antoine	41	marié
Planck	Max	52	veuf
Rimbault	Arthur	25	célibataire
Poincaré	Henri	45	marié

Car l'attribut Civilité ne dépend que du nom et non pas de l'âge ! Il en est de même pour le prénom, soit il faut changer de clef primaire et prendre (nom, prénom) soit si l'on conserve la clef primaire (nom , âge) , il faut décomposer la relation **Personne3** en deux autres relations **Personne31** et **Personne32** :

<u>nom</u>	<u>âge</u>	Civilité
Einstein	45	marié
Lavoisier	41	marié
Planck	52	veuf
Rimbault	25	célibataire
Poincaré	45	marié

Personne31(nom : chaîne, âge : entier , civilité : Etatcivil) :
2^{ème} forme normale

<u>nom</u>	<u>âge</u>	prénom
Einstein	45	Albert
Lavoisier	41	Antoine
Planck	52	Max
Rimbault	25	Arthur
Poincaré	45	Henri

Personne32(nom : chaîne, âge : entier , prénom : chaîne) :
2^{ème} forme normale

En pratique, il est aussi très difficile de faire vérifier automatiquement la mise en deuxième forme normale. Il faut trouver un jeu de données représentatif.

3^{ème} forme normale :

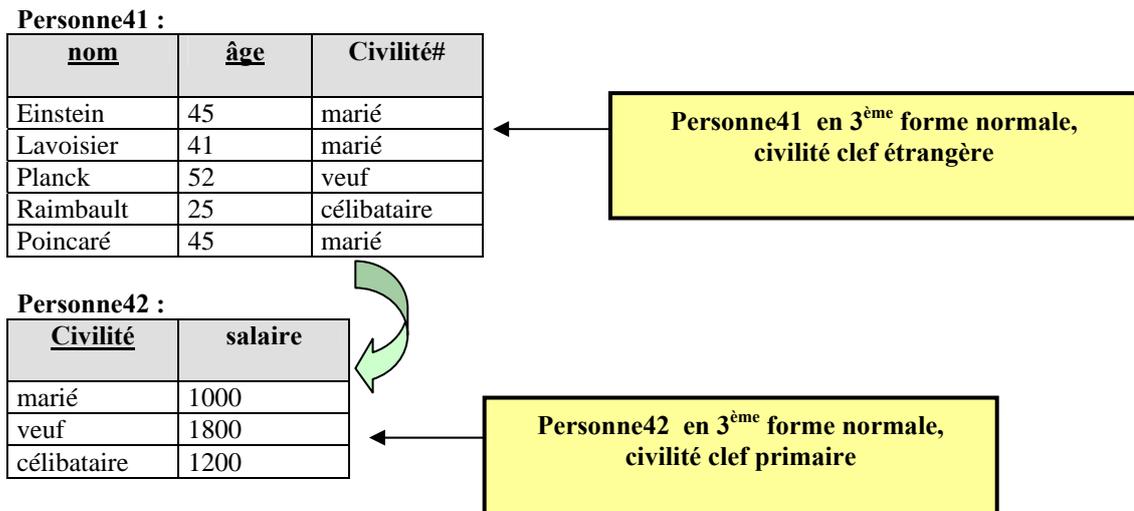
Une relation est dite en troisième forme normale si chaque attribut qui ne compose pas la clef primaire, dépend directement de son identifiant et à travers une dépendance fonctionnelle. Les relations précédentes sont toutes en forme normale. Montrons un exemple de relation qui n'est pas en forme normale. Soit la relation **Personne4** (nom : chaîne, âge : entier, civilité : Etatcivil, salaire : monétaire) où par exemple le salaire dépend de la clef primaire et que l'attribut civilité ne fait pas partie de la clef primaire (nom , âge) :

<u>nom</u>	<u>âge</u>	Civilité	salaire
Einstein	45	marié	1000
Lavoisier	41	marié	1000
Planck	52	veuf	1800
Rimbault	25	célibataire	1200
Poincaré	45	marié	1000

salaire = f (Civilité) =>
Pas 3^{ème} forme normale

L'attribut salaire dépend de l'attribut civilité, ce que nous écrivons salaire = f(civilité), mais l'attribut civilité ne fait pas partie de la clef primaire clef = (nom , âge), donc **Personne4** n'est pas en 3^{ème} forme normale :

Il faut alors décomposer la relation **Personne4** en deux relations **Personne41** et **Personne42** chacune en troisième forme normale:



En pratique, il est également très difficile de faire contrôler automatiquement la mise en troisième forme normale.

Remarques pratiques importantes pour le débutant :

- Les spécialistes connaissent deux autres formes normales. Dans ce cas le lecteur intéressé par l'approfondissement du sujet, trouvera dans la littérature, de solides références sur la question.
- Si la clef primaire d'une relation n'est composée que d'un seul attribut (choix conseillé lorsque cela est possible, d'ailleurs on trouve souvent des clefs primaires sous forme de numéro d'identification client, Insee,...) automatiquement, la relation est en 2^{ème} forme normale, car chaque autre attribut non clef étrangère, ne dépend alors que de la valeur unique de la clef primaire.
- Penser dès qu'un attribut est fonctionnellement dépendant d'un autre attribut qui n'est pas la clef elle-même à décomposer la relation (créer deux nouvelles tables).
- En l'absence d'outil spécialisé, il faut de la pratique et être très systématique pour contrôler la normalisation.

Base de données relationnelles BD-R:

Ce sont des données structurées à travers :

- Une famille de domaines de valeurs,
- Une famille de relations n-aires,
- Les contraintes d'intégrité sont respectées par toute clef étrangère et par toute clef primaire.
- Les relations sont en 3^{ème} forme normale. (à minima en 2^{ème} forme normale)

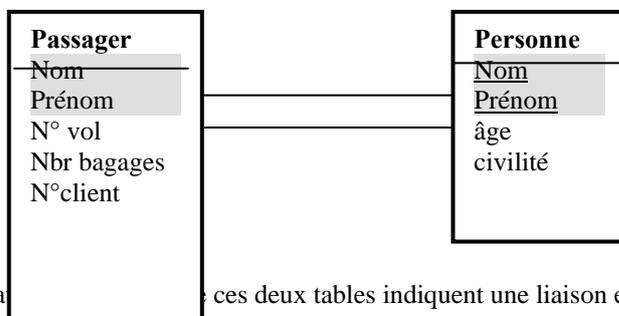
Les données sont accédées et manipulées grâce à un langage appelé **langage d'interrogation** ou

Système de Gestion de Base de Données relationnel :

C'est une famille de logiciels comprenant :

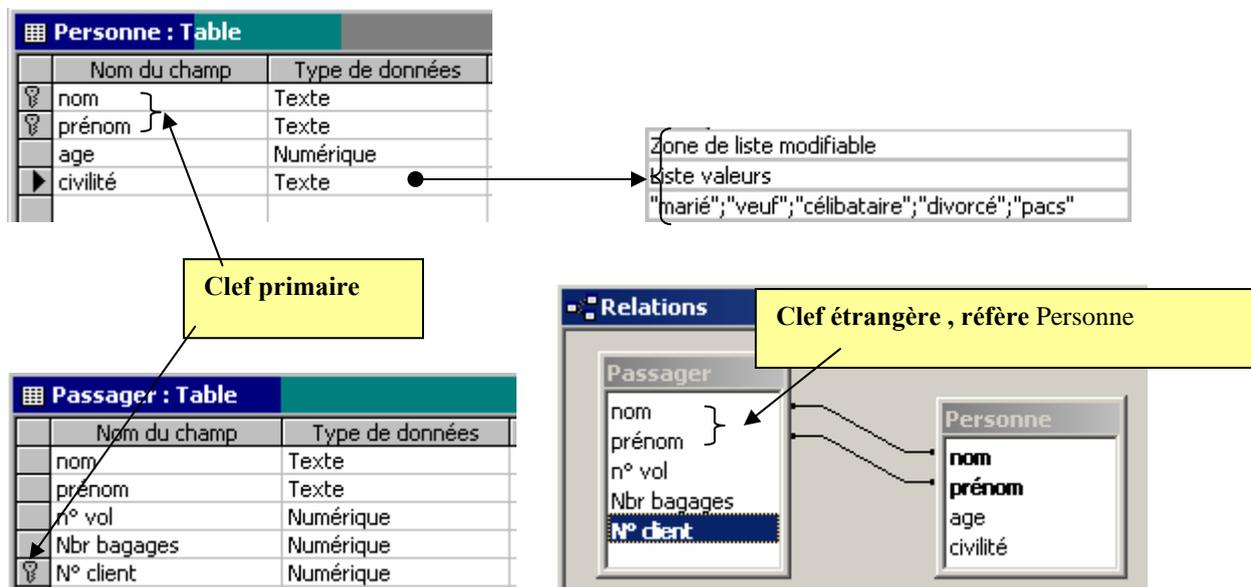
- Une BD-R.
- Un langage d'interrogation.
- Une gestion en interne des fichiers contenant les données et de l'ordonnement de ces données.
- Une gestion de l'interface de communication avec les utilisateurs.
- La gestion de la sécurité des accès aux informations contenues dans la BD-R.

Le schéma relation d'une relation dans une BD relationnelle est noté graphiquement comme ci-dessous :



Les attributs Nom et Prénom de ces deux tables indiquent une liaison entre les deux relations.

Voici dans le **SGBD-R Access**, la représentation des schémas de relation ainsi que la liaison sans intégrité des deux relations précédentes **Passager** et **Personne** :



Access et la représentation des enregistrements de chaque table :

Passager : Table					
	nom	prénom	n° vol	Nbr bagages	N° client
	Einstein	Albert	45622	2	154565
	Lavoisier	Antoine	45644	4	235002
	Raimbault	Arthur	12896	2	544552
	Poincaré	Henri	45644	3	781201
	Lavoisier	Antoine	45644	1	785154
	Einstein	Albert	75906	0	858547

Les enregistrements de la relation **Passager**

Personne : Table				
	nom	prénom	age	civilité
	Einstein	Albert	45	marié
	Lavoisier	Antoine	41	marié
	Planck	Max	52	veuf
	Poincaré	Henri	45	marié
	Raimbault	Arthur	25	célibataire

Les enregistrements de la relation **Personne**

Les besoins d'un utilisateur d'une base de données sont classiquement ceux que l'on trouve dans tout ensemble de données structurées : insertion, suppression, modification, recherche avec ou sans critère de sélection. Dans une BD-R, ces besoins sont exprimés à travers un langage d'interrogation. Historiquement deux classes de langages relationnels équivalentes en puissance d'utilisation et de fonctionnement ont été inventées : les langages **algébriques** et les langages des **prédicats**.

Un langage relationnel n'est pas un langage de programmation : il ne possède pas les structures de contrôle de base d'un langage de programmation (condition, itération, ...). **Très souvent il doit être utilisé comme complément à l'intérieur de programmes Delphi, Java, C#, ...**

Les langages d'interrogation prédicatifs sont des langages fondés sur la logique des prédicats du 1^{er} ordre, le plus ancien s'appelle **Query By Example QBE**.

Ce sont les langages algébriques qui sont de loin les plus utilisés dans les SGBD-R du commerce, le plus connu et le plus utilisé dans le monde se dénomme le **Structured Query Language** ou **SQL**. Un tel langage n'est qu'une implémentation en anglais d'opérations définies dans une algèbre relationnelle servant de modèle mathématique à tous les langages relationnels.

3. Principes fondamentaux d'une l'algèbre relationnelle

Une algèbre relationnelle est une famille d'opérateurs binaires ou unaires dont les opérands sont des **relations**. Nous avons vu que l'on pouvait faire l'union, l'intersection, le produit cartésien de relations binaires dans un chapitre précédent, comme les relations n-aires sont des ensembles, il est possible de définir sur elle une algèbre opératoire utilisant les opérateurs classiques ensemblistes, à laquelle on ajoute quelques opérateurs spécifiques à la manipulation des données.

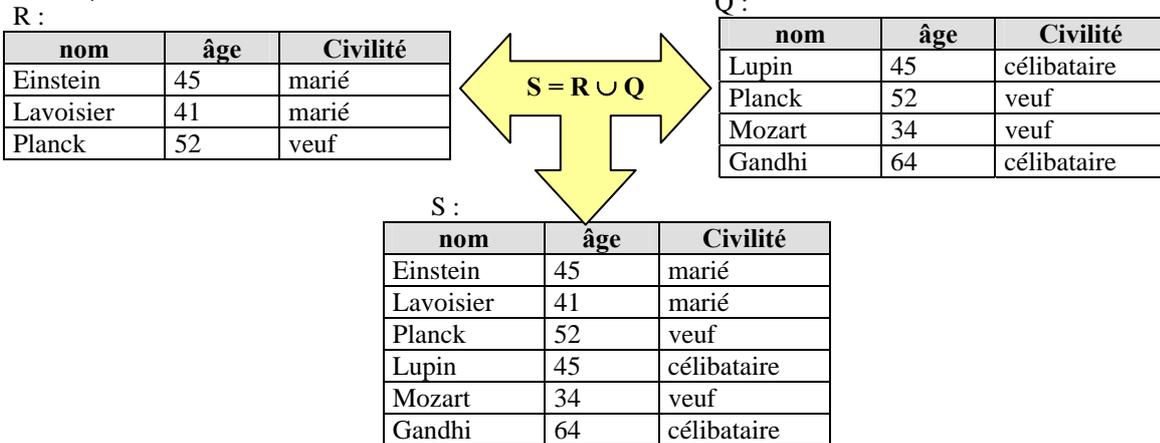
Remarque pratique :

La phrase "**tous les n-uples sont distincts, puisqu'éléments d'un même ensemble nommé relation**" se transpose en pratique en la phrase "**toutes les lignes d'une même table nommée relation, sont distinctes**" (même en l'absence de clef primaire explicite)".

Nous exhibons les opérateurs principaux d'une algèbre relationnelle et nous montrerons pour chaque opération, un exemple sous forme.

Union de 2 relations

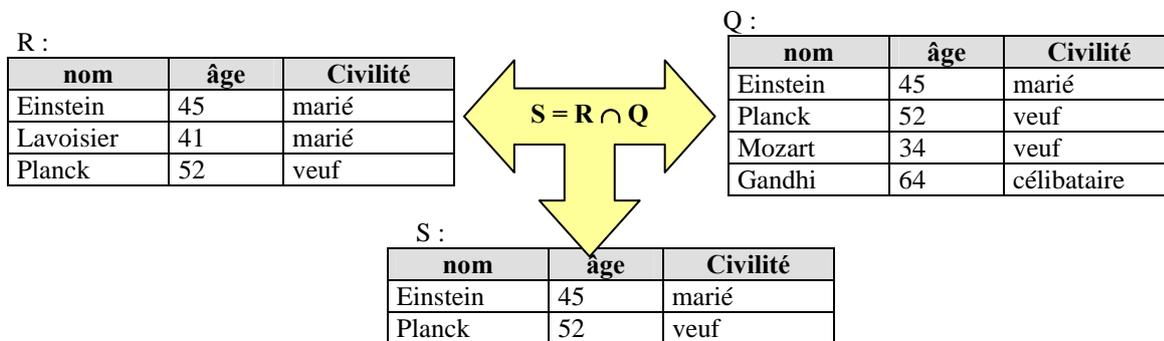
Soient R et Q deux relations de même domaine et de même degré on peut calculer la nouvelle relation $S = R \cup Q$ de même degré et de même domaine contenant les enregistrements différents des deux relations R et Q :



Remarque : (Planck, 52, veuf) ne figure qu'une seule fois dans la table $R \cup Q$.

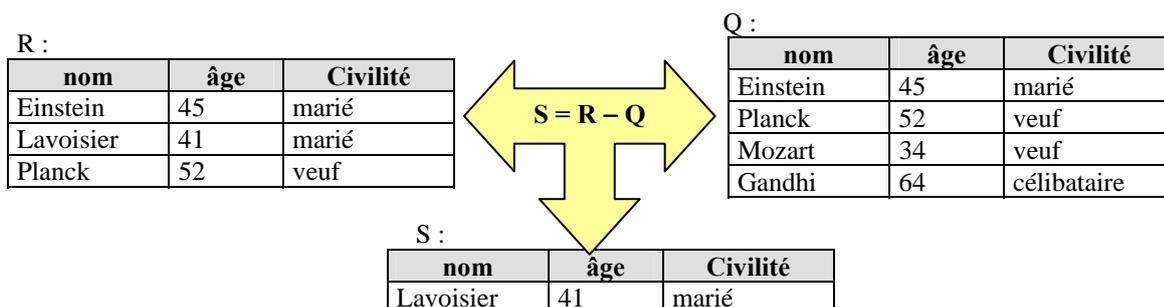
Intersection de 2 relations

Soient R et Q deux relations de même domaine et de même degré on peut calculer la nouvelle relation $S = R \cap Q$ de même degré et de même domaine contenant les enregistrements communs aux deux relations R et Q :



Différence de 2 relations

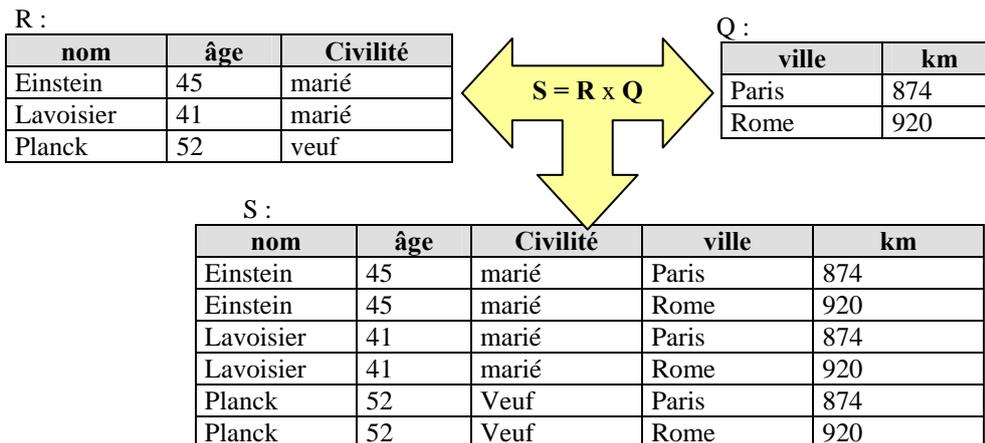
Soient R et Q deux relations de même domaine et de même degré on peut calculer la nouvelle relation $S = R - Q$ de même degré et de même domaine contenant les enregistrements qui sont présents dans R mais qui ne sont pas dans Q (on exclut de R les enregistrements qui appartiennent à $R \cap Q$) :



Produit cartésien de 2 relations

Soient R et Q deux relations de domaine et de degré quelconques ($\text{degré}(R)=n$, $\text{degré}(Q)=p$), avec $\text{Domaine}(R) \cap \text{Domaine}(Q) = \emptyset$ (pas d'attributs en communs).

On peut calculer la nouvelle relation $S = R \times Q$ de degré $n + p$ et de domaine égal à l'union des domaines de R et de Q contenant tous les couples d'enregistrements à partir d'enregistrements présents dans R et d'enregistrements présents dans Q :



Selection ou Restriction d'une relation

Soit R une relation, soit $R(a_1 : E_1, a_2 : E_2, \dots, a_n : E_n)$ le schéma de cette relation.

Soit $\text{Cond}(a_1, a_2, \dots, a_n)$ une expression booléenne classique (expression construite sur les attributs avec les connecteurs de l'algèbre de Boole et les opérateurs de comparaison $<$, $>$, $=$, $>=$, $<=$, $<>$)

On note $S = \text{select}(\text{Cond}(a_1, a_2, \dots, a_n), R)$, la nouvelle relation S construite ayant le même schéma que R soit $S(a_1 : E_1, a_2 : E_2, \dots, a_n : E_n)$, qui ne contient que les enregistrements de R qui satisfont à la condition booléenne $\text{Cond}(a_1, a_2, \dots, a_n)$.

R :

nom	âge	Civilité	ville	km
Einstein	45	marié	Paris	874
Mozart	32	marié	Rome	587
Gandhi	64	célibataire	Paris	258
Lavoisier	41	marié	Rome	124
Lupin	42	Veuf	Paris	608
Planck	52	Veuf	Rome	405

$\text{Cond}(a_1, a_2, \dots, a_n) = \{ \text{âge} > 42 \text{ et } \text{ville} = \text{Paris} \}$

S :

nom	âge	Civilité	ville	km
Einstein	45	marié	Paris	874
Gandhi	64	célibataire	Paris	258

Select ({ âge > 42 et ville=Paris}, R) signifie que l'on ne recopie dans S que les enregistrements de R constitués des personnes ayant séjourné à Paris et plus âgées que 42 ans.

Projection d'une relation

Soit R une relation, soit $R(a_1 : E_1, a_2 : E_2, \dots, a_n : E_n)$ le schéma de cette relation.

On appelle $S = \text{proj}(a_{k1}, a_{k2}, \dots, a_{kp})$ la projection de R sur un sous-ensemble restreint $(a_{k1}, a_{k2}, \dots, a_{kp})$ avec $p < n$, de ses attributs, la relation S ayant pour schéma le sous-ensemble des attributs $S(a_{k1} : E_{k1}, a_{k2} : E_{k2}, \dots, a_{kp} : E_{kp})$ et contenant les enregistrements différents obtenus en ne considérant que les attributs $(a_{k1}, a_{k2}, \dots, a_{kp})$.

Exemple

R :

nom	âge	Civilité	ville	km
Einstein	45	marié	Paris	874
Mozart	32	marié	Rome	587
Lupin	42	Veuf	Paris	464
Einstein	45	marié	Venise	981
Gandhi	64	célibataire	Paris	258
Lavoisier	41	marié	Rome	124
Lupin	42	Veuf	Paris	608
Planck	52	Veuf	Rome	405

$S1 = \text{proj}(\text{nom}, \text{civilité})$

nom	Civilité
Einstein	marié
Mozart	marié
Lupin	Veuf
Gandhi	célibataire
Lavoisier	marié
Planck	Veuf

$S2 = \text{proj}(\text{nom}, \text{âge})$

nom	âge
Einstein	45
Mozart	32
Lupin	42
Gandhi	64
Lavoisier	41
Planck	52

$S3 = \text{proj}(\text{nom}, \text{ville})$

nom	ville
Einstein	Paris
Mozart	Rome
Lupin	Paris
Einstein	Venise
Gandhi	Paris
Lavoisier	Rome
Planck	Rome

$S4 = \text{proj}(\text{ville})$

ville
Paris
Rome
Venise

Que s'est-il passé pour Mr Einstein dans $S2$?

Einstein	45	marié	Paris	874
Einstein	45	marié	Venise	981
Einstein		marié		

Lors de la recopie des enregistrements de R dans $S2$ on a ignoré les attributs âge, ville et km, le couple (Einstein, marié) ne doit se retrouver qu'une seule fois car une relation est un ensemble et ses éléments sont tous distincts.

Jointure de deux relations

Soient R et Q deux relations de domaine et de degré quelconques ($\text{degré}(R) = n$, $\text{degré}(Q) = p$), avec $\text{Domaine}(R) \cap \text{Domaine}(Q) = \emptyset$ (pas d'attributs en communs).

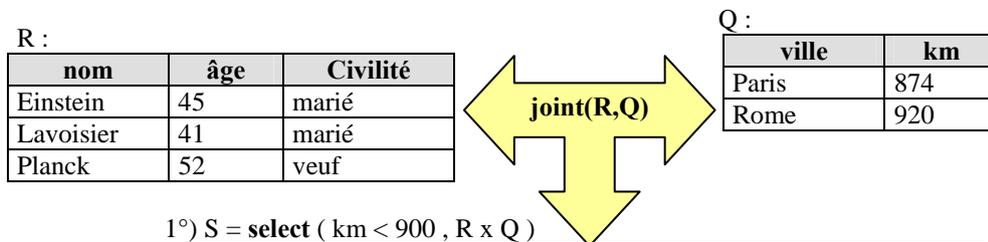
soit $R \times Q$ leur produit cartésien de degré $n + p$ et de domaine D union des domaines de R et de Q .

Soit un ensemble $(a_1, a_2, \dots, a_{n+p})$ d'attributs du domaine D de $R \times Q$.

La relation **joint**(R,Q) = **select** (Cond(a₁, a₂, ..., a_{n+p}), R x Q), est appelée jointure de R et de Q (c'est donc une sélection de certains attributs sur le produit cartésien).

Une jointure couramment utilisée en pratique, est celle qui consiste en la sélection selon une condition d'égalité entre deux attributs, les personnes de "l'art relationnel" la dénomment alors l'**équi-jointure**.

Exemples de 2 jointures :



1°) S = **select** (km < 900 , R x Q)

nom	âge	Civilité	ville	km
Einstein	45	marié	Paris	874
Lavoisier	41	marié	Paris	874
Planck	52	veuf	Paris	874

2°) S = **select** (km < 900 et Civilité = veuf, R x Q)

nom	âge	Civilité	ville	km
Planck	52	veuf	Paris	874

Nous nous plaçons maintenant du point de vue pratique, non pas de l'administrateur de BD mais de l'utilisateur uniquement concerné par l'extraction des informations contenues dans une BD-R.

Un SGBD permet de gérer une base de données. A ce titre, il offre de nombreuses fonctionnalités supplémentaires à la gestion d'accès simultanés à la base et à un simple interfaçage entre le modèle logique et le modèle physique : il sécurise les données (en cas de coupure de courant ou autre défaillance matérielle), il permet d'accéder aux données de manière confidentielle (en assurant que seuls certains utilisateurs ayant des mots de passe appropriés, peuvent accéder à certaines données), il ne permet de mémoriser des données que si elles sont du type abstrait demandé : on dit qu'il vérifie leur intégrité (des données alphabétiques ne doivent pas être enregistrées dans des emplacements pour des données numériques,...)

Actuellement, une base de données n'a pas de raison d'être sans son SGBD. Aussi, on ne manipule que des bases de données correspondant aux SGBD qui les gèrent : il vous appartient de choisir le SGBD-R qui vous convient (il faut l'acheter auprès de vendeurs qui généralement, vous le fournissent avec une application de manipulation visuelle, ou bien utiliser les SGBD-R qui vous sont livrés gratuitement avec certains environnements de développement comme Borland studio ou Visual Studio ou encore utiliser les produits gratuits comme mySql).

Lorsque l'on parle d'utilisateur, nous entendons l'application utilisateur, car l'utilisateur final n'a pas besoin de connaître quoique ce soit à l'algèbre relationnelle, il suffit que l'application utilisateur communique avec lui et interagisse avec le SGBD.

Une application doit pouvoir "parler" au SGBD : elle le fait par le moyen d'un langage de manipulation des données. Nous avons déjà précisé que la majorité des SGBD-R utilisent un langage relationnel ou de requêtes nommé SQL pour manipuler les données.

4. SQL et Algèbre relationnelle

Requête

Les requêtes sont des questions posées au SGBD, concernant une recherche de données contenues dans une ou plusieurs tables de la base.

Par exemple, on peut disposer d'une table définissant des clients (noms, prénoms, adresses, n° de client) et d'une autre table associant des numéros de clients avec des numéros de commande d'articles, et vouloir poser la question suivante : quels sont les noms des clients ayant passé des commandes ?

Une requête est en fait, une instruction de type langage de programmation, respectant la norme SQL, permettant de réaliser un tel questionnement. L'exécution d'une requête permet d'extraire des données en provenance de tables de la base de données : ces données réalisent ce que l'on appelle une **projection** de champs (en provenance de plusieurs tables). Le résultat d'exécution d'une requête est une table constituée par les réponses à la requête.

Le SQL permet à l'aide d'instructions spécifiques de manipuler des données présentes à l'intérieur des tables :

Instruction SQL	Actions dans la (les) table(s)
INSERT INTO <...>	Ajout de lignes
DELETE FROM <...>	Suppression de lignes
TRUNCATE TABLE <...>	Suppression de lignes
UPDATE <...>	Modification de lignes
SELECT <...> FROM <...>	Extraction de données

Ajout, suppression et modification sont les trois opérations typiques de la **mise à jour** d'une BD. L'extraction concerne la **consultation** de la BD.

Il existe de nombreuses autres instructions de création, de modification, de suppression de tables, de création de clefs, de contraintes d'intégrités référentielles, création d'index, etc... Nous nous attacherons à donner la traduction en SQL des opérateurs principaux de l'algèbre relationnelle que nous venons de citer.

Traduction en SQL des opérateurs relationnels

C'est l'instruction SQL "**SELECT** <...>**FROM** <...>" qui implante tous ces opérateurs. Tous les exemples utiliseront la relation R = TableComplete suivante et l'interpréteur SQL d'Access :

TableComplete : Table					
	nom	âge	civilité	ville	km
	Einstein	45	marié	Paris	874
	Mozart	32	marié	Rome	587
	Lupin	42	Veuf	Paris	464
	Einstein	45	marié	Venise	981
	Gandhi	64	célibataire	Paris	258
	Lavoisier	41	marié	Rome	124
	Lupin	42	Veuf	Paris	608
	Planck	52	Veuf	Rome	405

La relation initiale :
R=TableComplete

Projection d'une relation R

$$S = \text{proj}(a_{k1}, a_{k2}, \dots, a_{kp})$$

SQL : **SELECT DISTINCT** $a_{k1}, a_{k2}, \dots, a_{kp}$ **FROM R**

<p><i>Instruction SQL version opérateur algèbre :</i> SELECT DISTINCT nom , civilité FROM Tablecomplete</p> <p><i>Lancement de la requête SQL :</i></p> <div style="border: 1px solid black; padding: 5px;"> <p>RequêteS1 : Requête Sélection</p> <p>SELECT DISTINCT [nom], [civilité] FROM TableComplete;</p> </div> <p>Le mot DISTINCT assure que l'on obtient bien une nouvelle relation.</p>	<p><i>Table obtenue après requête :</i></p> <table border="1"> <thead> <tr> <th colspan="2">RequêteS1 : Requête Sélection</th> </tr> <tr> <th>nom</th> <th>civilité</th> </tr> </thead> <tbody> <tr><td>Einstein</td><td>marié</td></tr> <tr><td>Gandhi</td><td>célibataire</td></tr> <tr><td>Lavoisier</td><td>marié</td></tr> <tr><td>Lupin</td><td>Veuf</td></tr> <tr><td>Mozart</td><td>marié</td></tr> <tr><td>Planck</td><td>Veuf</td></tr> </tbody> </table>	RequêteS1 : Requête Sélection		nom	civilité	Einstein	marié	Gandhi	célibataire	Lavoisier	marié	Lupin	Veuf	Mozart	marié	Planck	Veuf				
RequêteS1 : Requête Sélection																					
nom	civilité																				
Einstein	marié																				
Gandhi	célibataire																				
Lavoisier	marié																				
Lupin	Veuf																				
Mozart	marié																				
Planck	Veuf																				
<p><i>Instruction SQL non relationnelle (tout même redondant) :</i> SELECT nom , civilité FROM Tablecomplete</p> <p><i>Lancement de la requête SQL :</i></p> <div style="border: 1px solid black; padding: 5px;"> <p>RequêteS1 : Requête Sélection</p> <p>SELECT [nom], [civilité] FROM TableComplete;</p> </div>	<p><i>Table obtenue après requête :</i></p> <table border="1"> <thead> <tr> <th colspan="2">RequêteS1 : Requête Sélé</th> </tr> <tr> <th>nom</th> <th>civilité</th> </tr> </thead> <tbody> <tr><td>Einstein</td><td>marié</td></tr> <tr><td>Mozart</td><td>marié</td></tr> <tr><td>Lupin</td><td>Veuf</td></tr> <tr><td>Einstein</td><td>marié</td></tr> <tr><td>Gandhi</td><td>célibataire</td></tr> <tr><td>Lavoisier</td><td>marié</td></tr> <tr><td>Lupin</td><td>Veuf</td></tr> <tr><td>Planck</td><td>Veuf</td></tr> </tbody> </table>	RequêteS1 : Requête Sélé		nom	civilité	Einstein	marié	Mozart	marié	Lupin	Veuf	Einstein	marié	Gandhi	célibataire	Lavoisier	marié	Lupin	Veuf	Planck	Veuf
RequêteS1 : Requête Sélé																					
nom	civilité																				
Einstein	marié																				
Mozart	marié																				
Lupin	Veuf																				
Einstein	marié																				
Gandhi	célibataire																				
Lavoisier	marié																				
Lupin	Veuf																				
Planck	Veuf																				
<p>Remarquons dans le dernier cas SELECT nom , civilité FROM Tablecomplete que la table obtenue n'est qu'une extraction de données, mais qu'en aucun cas elle ne constitue une relation puisqu'une relation est un ensemble et que les enregistrements sont tous distincts!</p>																					

Une autre projection sur la même table :

<p><i>Instruction SQL version opérateur algèbre :</i> SELECT DISTINCT nom , ville FROM Tablecomplete</p> <p><i>Lancement de la requête SQL :</i></p> <div style="border: 1px solid black; padding: 5px;"> <p>RequêteS2 : Requête Sélection</p> <p>SELECT DISTINCT [nom], [ville] FROM TableComplete;</p> </div>	<p><i>Table obtenue après requête :</i></p> <table border="1"> <thead> <tr> <th colspan="2">RequêteS2 : Requête Sélé</th> </tr> <tr> <th>nom</th> <th>ville</th> </tr> </thead> <tbody> <tr><td>Einstein</td><td>Paris</td></tr> <tr><td>Einstein</td><td>Venise</td></tr> <tr><td>Gandhi</td><td>Paris</td></tr> <tr><td>Lavoisier</td><td>Rome</td></tr> <tr><td>Lupin</td><td>Paris</td></tr> <tr><td>Mozart</td><td>Rome</td></tr> <tr><td>Planck</td><td>Rome</td></tr> </tbody> </table>	RequêteS2 : Requête Sélé		nom	ville	Einstein	Paris	Einstein	Venise	Gandhi	Paris	Lavoisier	Rome	Lupin	Paris	Mozart	Rome	Planck	Rome
RequêteS2 : Requête Sélé																			
nom	ville																		
Einstein	Paris																		
Einstein	Venise																		
Gandhi	Paris																		
Lavoisier	Rome																		
Lupin	Paris																		
Mozart	Rome																		
Planck	Rome																		

Sélection-Restriktion

$S = \text{select} (\text{Cond}(a_1, a_2, \dots, a_n), R)$

SQL : **SELECT * FROM R WHERE** Cond(a_1, a_2, \dots, a_n)

Le symbole * signifie toutes les colonnes de la table (tous les attributs du schéma)

Instruction SQL version opérateur algèbre :
SELECT * FROM Tablecomplete **WHERE**
 âge > 42 **AND** ville = Paris

Lancement de la requête SQL :

TableComplete Requête : Requête Sélection
SELECT * FROM TableComplete
WHERE [âge]>42 AND [ville]="Paris";

Table obtenue après requête :

TableComplete Requête : Requête Sélection					
	nom	âge	civilité	ville	km
	Einstein	45	marié	Paris	874
	Gandhi	64	célibataire	Paris	258

On a sélectionné toutes les personnes de plus de 42 ans ayant séjourné à Paris.

Combinaison d'opérateur projection-restriktion

Projection distincte et sélection :
SELECT DISTINCT nom , civilité, âge **FROM**
 Tablecomplete **WHERE** âge >= 45

Lancement de la requête SQL :

TableComplete Requête : Requête Sélection
SELECT DISTINCT [nom], [civilité], [âge]
FROM TableComplete
WHERE [âge]>=45 ;

Table obtenue après requête :

TableComplete Requête : Requête Sélection			
	nom	civilité	âge
	Einstein	marié	45
	Gandhi	célibataire	64
	Planck	Veuf	52

On a sélectionné toutes les personnes d'au moins 45 ans et l'on ne conserve que leur nom, leur civilité et leur âge.

Intersection, union, différence,

$S = R \cap Q$

SQL : **SELECT * FROM R INTERSECT** SELECT * FROM Q

$S = R \cup Q$

SQL : **SELECT * FROM R UNION** SELECT * FROM Q

$S = R - Q$

SQL : **SELECT * FROM R MINUS** SELECT * FROM Q

Produit cartésien

$$S = R \times Q$$

SQL : SELECT * FROM R, Q

Afin de ne pas présenter un exemple de table produit trop volumineuse, nous prendrons comme opérandes du produit cartésien, deux tables contenant peu d'enregistrements :

Personne : Table				Employeur : Table	
nom	prénom	age	civilité	nom	n°Insee
Einstein	Albert	45	marié	Université	12112472
Lavoisier	Antoine	41	marié	Collège	25478550
				Institut	54578559

Produit cartésien :

SELECT * FROM Employeur , Personne

Employeur Requête : Requête Sélection						
Personne.nom	prénom	age	civilité	Employeur.nom	n°Insee	
Einstein	Albert	45	marié	Université	12112472	
Lavoisier	Antoine	41	marié	Université	12112472	
Einstein	Albert	45	marié	Collège	25478550	
Lavoisier	Antoine	41	marié	Collège	25478550	
Einstein	Albert	45	marié	Institut	54578559	
Lavoisier	Antoine	41	marié	Institut	54578559	

Nous remarquons qu'en apparence l'attribut **nom** se retrouve dans le domaine des deux relations ce qui semble contradictoire avec l'hypothèse "Domaine(R) ∩ Domaine(Q) = ∅ (pas d'attributs en communs). En fait ce n'est pas un attribut commun puisque les valeurs sont différentes, il s'agit plutôt de deux attributs différents qui ont la même identification. Il suffit de préfixer l'identificateur par le nom de la relation (Personne.nom et Employeur.nom).

Combinaison d'opérateurs : projection - produit cartésien

SELECT Personne.nom , prénom, civilité, n°Insee FROM Employeur , Personne

On extrait de la table produit cartésien uniquement 4 colonnes :

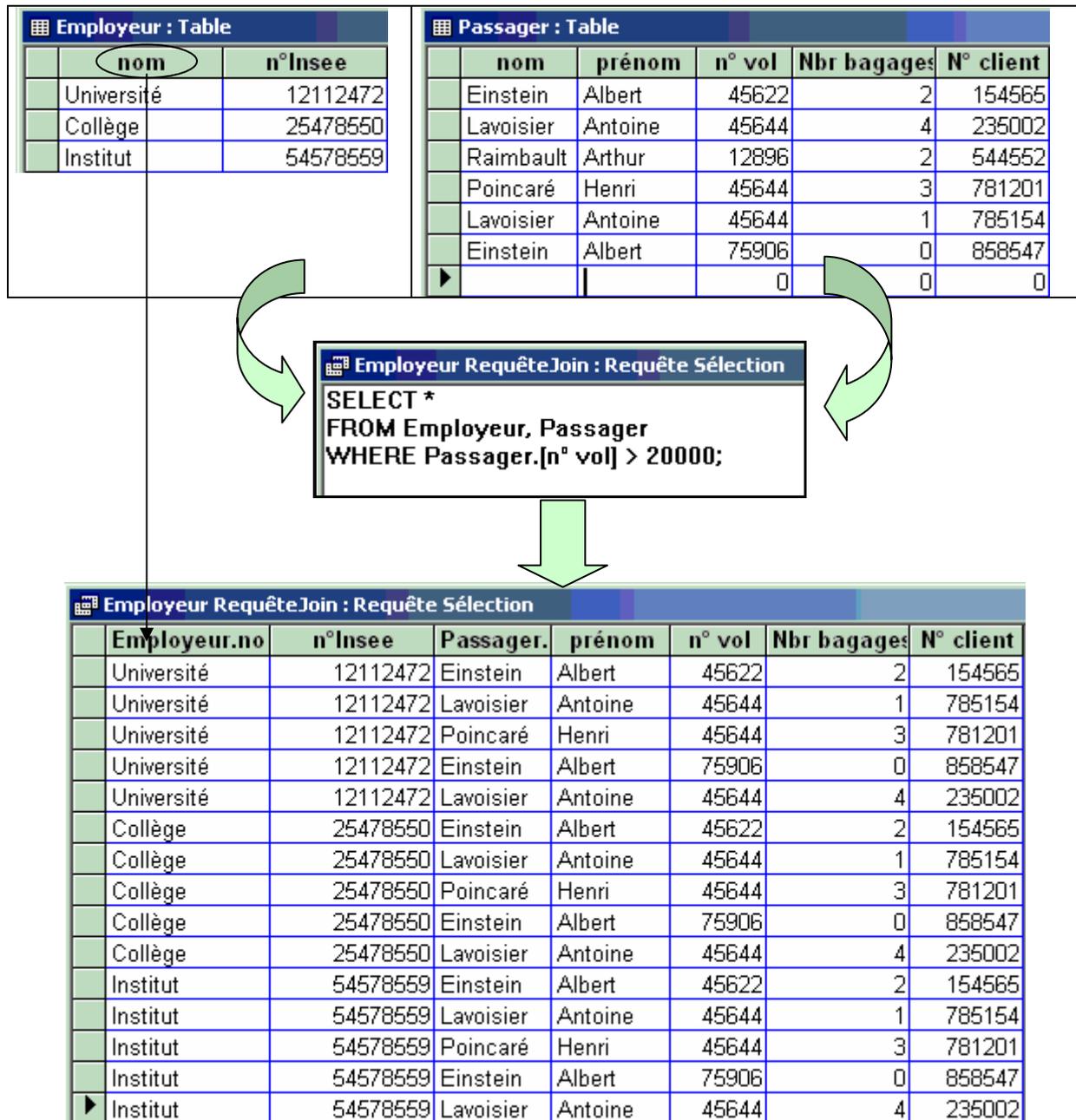
Employeur Requête : Requête Sélection				
nom	prénom	civilité	n°Insee	
Einstein	Albert	marié	12112472	
Lavoisier	Antoine	marié	12112472	
Einstein	Albert	marié	25478550	
Lavoisier	Antoine	marié	25478550	
Einstein	Albert	marié	54578559	
Lavoisier	Antoine	marié	54578559	

Jointure de deux relations

Soient R et Q deux relations de domaine et de degré quelconques ($\text{degré}(R) = n$, $\text{degré}(Q) = p$), avec $\text{Domaine}(R) \cap \text{Domaine}(Q) = \emptyset$ (pas d'attributs en communs).

La jointure $\text{joint}(R,Q) = \text{select}(\text{Cond}(a_1, a_2, \dots, a_{n+p}), R \times Q)$.

SQL : `SELECT * FROM R,Q WHERE Cond(a1, a2, ..., an+p)`



Remarque pratique importante

Le langage SQL est plus riche en fonctionnalités que l'algèbre relationnelle. En effet SQL intègre des possibilités de calcul (numériques et de dates en particulier).

Soit une table de tarifs de produit avec des prix hors taxe:

Tarifs : Table	
Article	PrixHT
chaise	10,00 €
table	100,00 €
Téléviseur	1 000,00 €

1°) Usage d'un opérateur multiplicatif : calcul de la nouvelle table des tarifs TTC abondés de la TVA à 20% sur le prix hors taxe.

Tarifs : Table	
Article	PrixHT
chaise	10,00 €
table	100,00 €
Téléviseur	1 000,00 €

Requête SQL : Tarifs TTC

```
SELECT Article , PrixHT*1.20 AS PrixTTC
FROM Tarifs;
```

Tarifs TTC : Requête Sélection	
Article	PrixTTC
chaise	12
table	120
Téléviseur	1200

2°) Usage de la fonction intégrée SUM : calcul du total des prix HT.

Tarifs : Table	
Article	PrixHT
chaise	10,00 €
table	100,00 €
Téléviseur	1 000,00 €

Requête SQL : Total HT

```
SELECT SUM(PrixHT) AS Total
FROM Tarifs;
```

Total HT : Requête Sélection	
Total	
1 110,00 €	

ADO .Net : données relationnelles de .Net 2.0



Plan général: 

1. Le modèle DataSet dans ADO.Net 2.0

- DataSet
- DataTable
- DataColumn
- DataRow
- DataRelation

2. Création et utilisation d'un DataSet

- Création d'un DataSet
- Création de 2 DataTable
- Définition du schéma de colonnes de chaque table par des DataColumn.
- Définition d'une clef primaire.
- Ajout des 2 tables au DataSet.
- Ajout d'une relation entre les deux tables.
- Exemple de création de données dans le DataSet
- Affichage des données d'un DataSet
- Sauvegarde des données du DataSet aux formats XML et XSL

Introduction

Nous avons vu dans un chapitre précédent que les fichiers simples peuvent être accédés par des flux, dès que l'organisation de l'information dans le fichier est fortement structurée les flux ne sont plus assez puissants pour fournir un accès souple aux données en particulier pour des applications à architecture multi-tiers (client/serveur, internet,...) et spécialement aux bases de données.

ADO .NET est un regroupement de types (classes, interfaces, ...) dans l'espace de nom **System.Data** construits par Microsoft afin de manipuler des données structurées dans le .NET Framework.

Le modèle ADO .NET du .NET Framework fournit au développeur un ensemble d'éléments lui permettant de travailler sur des données aussi bien en mode connecté qu'en mode déconnecté (ce dernier mode est le mode préférentiel d' ADO .NET car c'est celui qui est le plus adapté aux architectures multi-tiers). ADO .NET est indépendant du mode de stockage des données : les classes s'adaptent automatiquement à l'organisation

ADO .NET permet de traiter des données situées dans des bases de données selon le modèle relationnel mais il supporte aussi les données organisées selon le modèle hiérarchique.

ADO .NET échange toutes ses informations au format XML

L'entité la plus importante d' ADO .NET permettant de gérer les données en local dans une mémoire cache complètement déconnectée de la source de données (donc indépendante de cette source) est le DataSet en fait la classe **System.Data.DataSet** et la collection de classes qui lui sont liées.

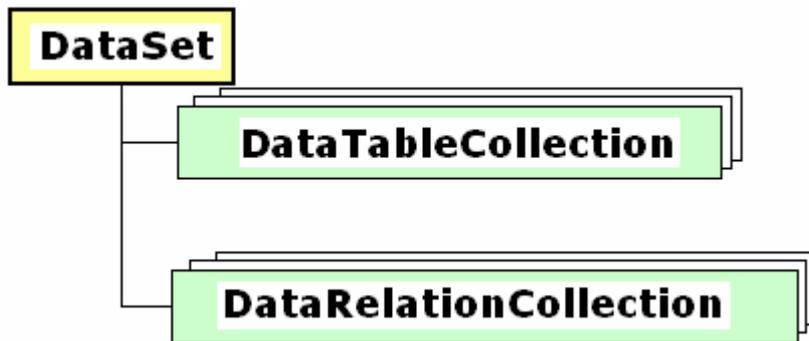
1. Le modèle DataSet dans ADO.Net 2.0

Le principe de base du DataSet est de se connecter à une source de données (par exemple une base de données) de charger toutes ses tables avec leur relations, puis ensuite de travailler en mode déconnecté sur ces tables en mémoire et enfin se reconnecter pour effectuer la mise à jour éventuelle des données.

Les classes mises en jeu lors d'une telle opération sur les données sont les suivantes :

System.Data.DataSet	
System.Data.DataTable	System.Data.DataTableCollection
System.Data.DataColumn	System.Data.DataColumnCollection
System.Data.DataRow	System.Data.DataRowCollection
System.Data.DataRelation	System.Data.DataRelationCollection
System.Data.DataConstraint	System.Data.ConstraintCollection
System.Data.DataView	

Un **DataSet** est en fait une collection de tables représentée par un objet de collection de la classe **DataTableCollection** et une collection de relations entres ces tables représentée par un objet de collection de la classe **DataRelationCollection**.



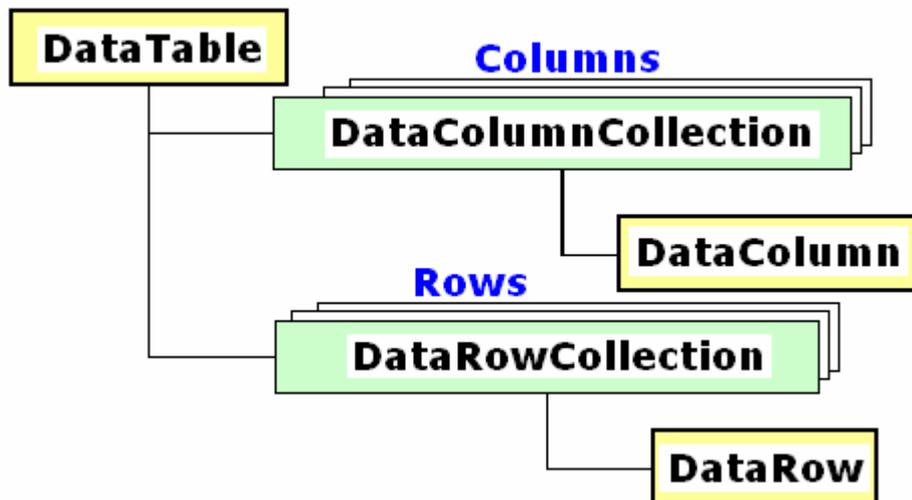
Une **DataTableCollection** est une collection (famille) d'une ou plusieurs DataTable qui représentent chacune une table dans le DataSet.

Une table générale est une entité composée de :

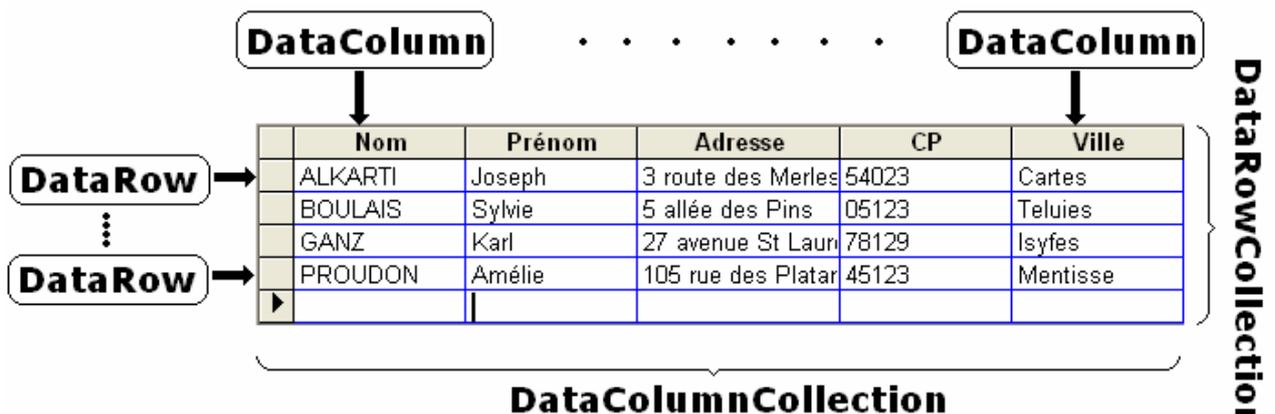
- Colonnes
- Lignes

Un objet de classe **DataTable** est composé en particulier des objets suivants :

- Une propriété **Columns** = Collection de colonnes (collection d'objets **DataColumn**)
- Une propriété **Rows** = Collection de lignes (collection d'objets **DataRow**)

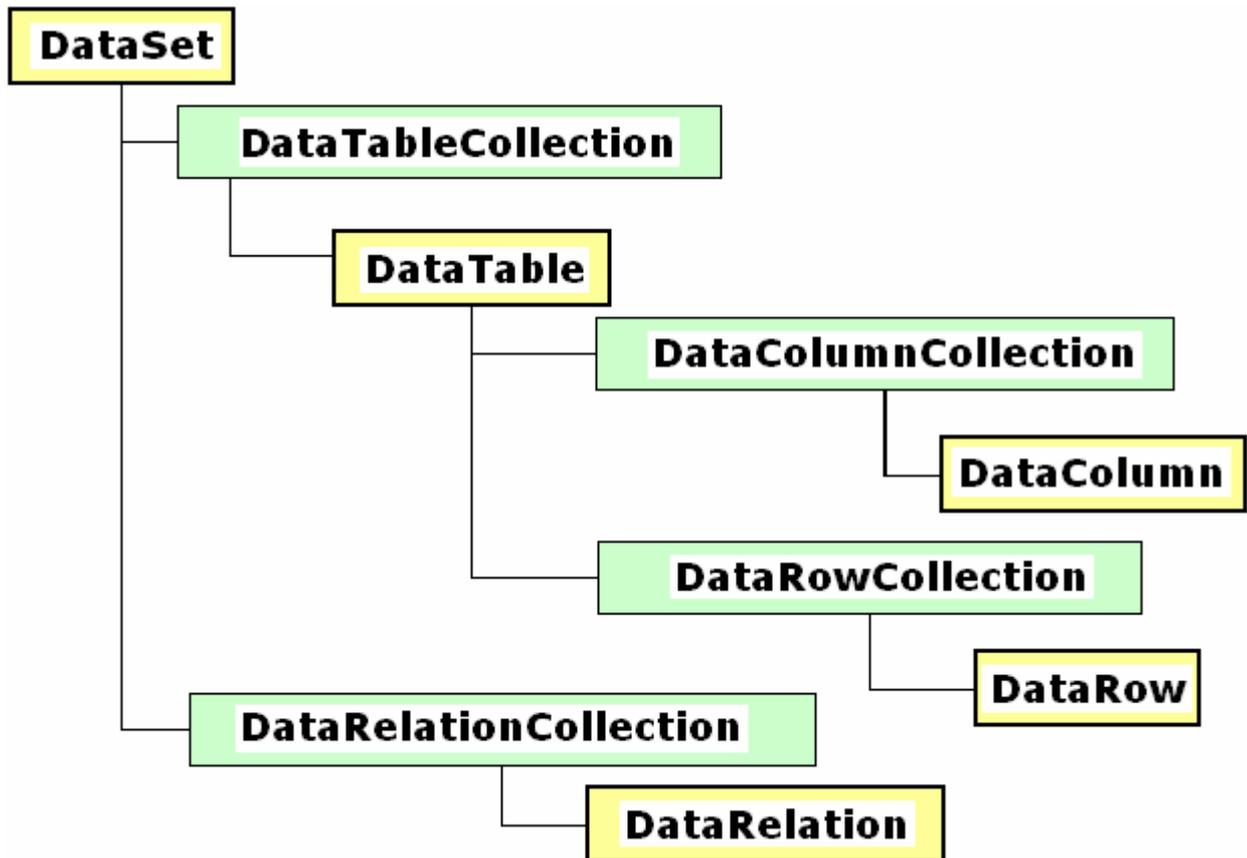


Le schéma général de la table d'un objet **DataTable** est donné par la famille des colonnes (l'objet Columns) chaque objet **DataColumn** de la collection représente une colonne de la table :

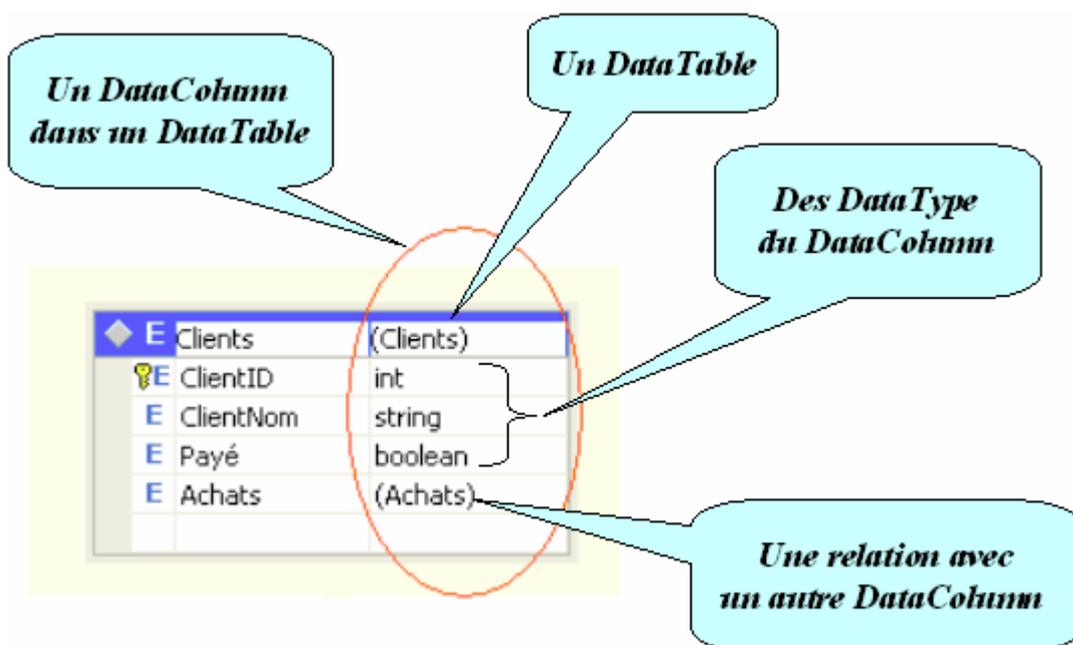


Pour résumer, un **DataSet** contient pour l'essentiel deux types d'objets :

- 1°) des objets **DataTable** inclus dans une **DataTableCollection**
- 2°) des objets **DataRelation** inclus dans une **DataRelationCollection**

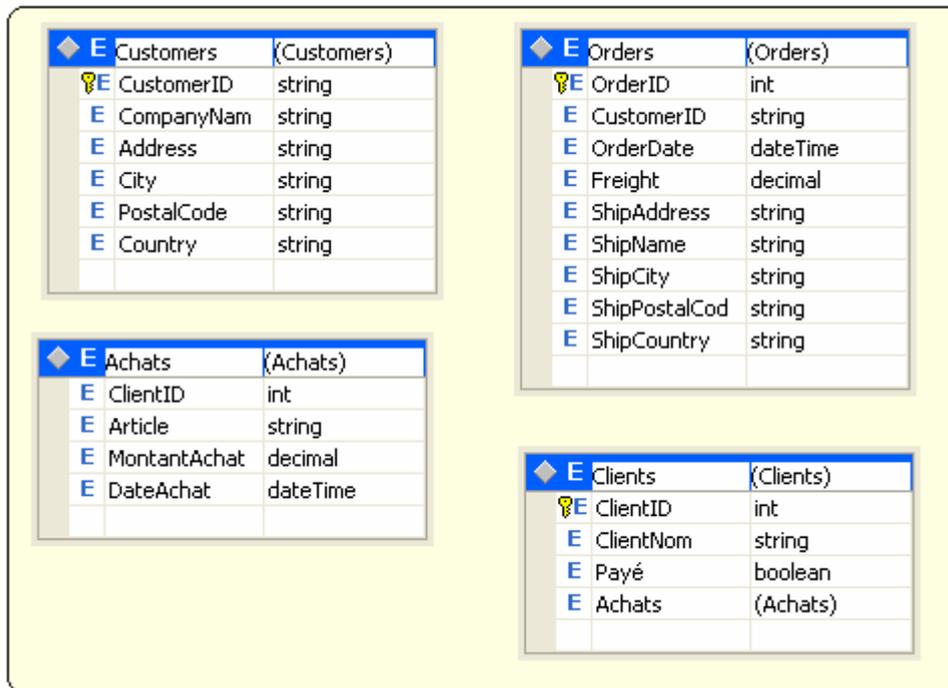


Une **DataRelationCollection** est une collection d'une ou plusieurs **DataRelation** qui représentent chacune une relation entre deux **DataTable** dans le **DataSet**.

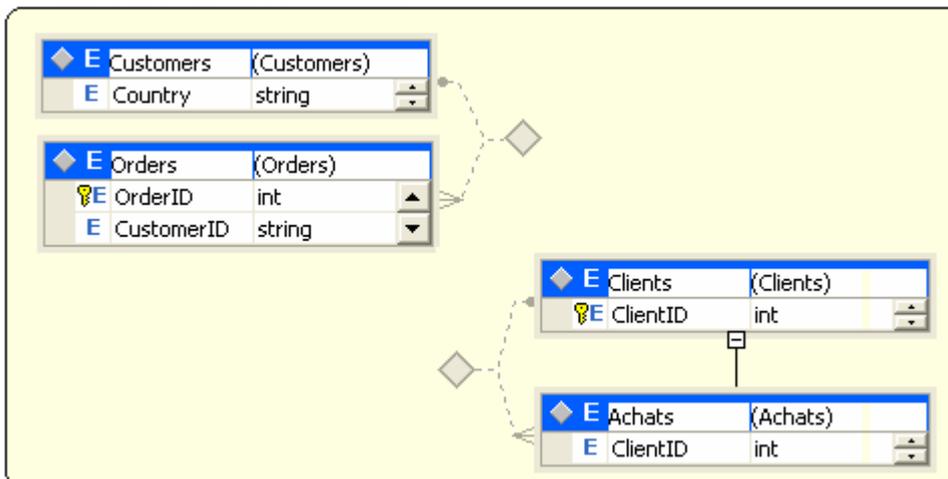


Voici une représentation fictive mais figurative d'un **DataSet** contenant 4 **DataTable** et 2 **DataRelation** :

La **DataTableCollection** du DataSet :



La **DataRelationCollection** du DataSet :



1. Création et utilisation d'un DataSet

Nous passons maintenant à la pratique d'utilisation d'un DataSet dans un programme C# de création de données selon la démarche suivante :

Phase de mise en place :

- ❑ Création d'un DataSet (un magasin).
- ❑ Création de 2 DataTable (une table client, une table achats).
- ❑ Définition du schéma de colonnes de chaque table par des DataColumn.
- ❑ Définition d'une clef primaire.
- ❑ Ajout des 2 tables au DataSet.
- ❑ Ajout d'une relation entre les deux tables.

Phase de manipulation des données proprement dites :

- ❑ Ajouter, supprimer ou modifier des données ligne par ligne dans chaque table.

Création d'un DataSet (un magasin) :

```
private DataSet unDataSet = new DataSet();  
unDataSet.DataSetName = "Magasin";
```

Création des deux tables :

```
private DataTable tabClients = new DataTable( "Clients" );  
private DataTable tabAchats = new DataTable( "Achats" );
```

Définition du schéma de 3 colonnes de la table Clients :

```
DataColumn colClientID = new DataColumn( "ClientID", typeof(int) );  
DataColumn colClientNom = new DataColumn( "ClientNom" );  
DataColumn colPayé = new DataColumn( "Payé", typeof(bool) );  
tabClients.Columns.Add(colClientID);  
tabClients.Columns.Add(colClientNom);  
tabClients.Columns.Add(colPayé);
```

Définition du schéma de 4 colonnes de la table Achats :

```
DataColumn colID = new DataColumn("ClientID", typeof(int));  
DataColumn colArticle = new DataColumn("Article", typeof(string));  
DataColumn colDateAchat = new DataColumn( "DateAchat", typeof(DateTime));  
DataColumn colMontantAchat = new DataColumn("MontantAchat", typeof(decimal));  
tabAchats.Columns.Add(colID);  
tabAchats.Columns.Add(colArticle);  
tabAchats.Columns.Add(colMontantAchat);  
tabAchats.Columns.Add(colDateAchat);
```

Définition d'une clef primaire composée de 2 colonnes :

```
tabClients.PrimaryKey = new DataColumn[] { colClientID, colClientNom};
```

Ajouter les 2 tables au DataSet :

```
unDataSet.Tables.Add(tabClients);
unDataSet.Tables.Add(tabAchats);
```

Ajouter une relation "Liste des achats" entre les 2 tables :

```
// Créer une DataRelation entre colClientID et colID
DataRelation dr = new DataRelation("Liste des achats", colClientID , colID);

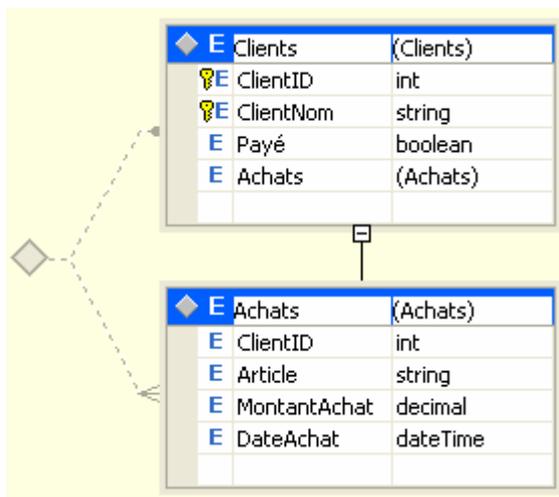
// ajouter cette relation dans le DataSet
unDataSet.Relations.Add(dr);

// imbrication de relation écrite dans le fichier xml:
dr.Nested=true;
```

La validation de toutes les modifications apportées au DataSet depuis son chargement s'effectue grâce à la méthode **AcceptChanges()**, on termine donc le code par la ligne :

```
unDataSet.AcceptChanges();
```

Le DataSet ainsi construit peut maintenant travailler en lecture et en écriture sur une structure de données construite sous forme de deux tables Clients et Achats reliées entre elles comme figuré ci-dessous :



Exemple de création de données dans le DataSet :

```
enum typArticle { vélo, télévision, radio, cuisinière }
```

```
//-- remplissage direct des tables
// Déclaration d'un référence de ligne de données
DataRow newRow1;
// Création de 6 lignes de clients dans la table "Clients"
for(int i = 1; i <= 6; i++) {
    // Création de la ligne newRow1 possédant le même schéma que tabClients
    newRow1 = tabClients.NewRow();
    newRow1["ClientID"] = 1000+i;
```

```

newRow1["ClientNom "] = "...";
newRow1["Payé "] = false;
// Ajouter la ligne newRow1 à la table tabClients
tabClients.Rows.Add(newRow1);
}
// Remplissage de la colonne "ClientNom" de chaque ligne de clients créée
tabClients.Rows[0]["ClientNom"] = "Legrand";
tabClients.Rows[1]["ClientNom"] = "Fischer";
tabClients.Rows[2]["ClientNom"] = "Dupont";
tabClients.Rows[3]["ClientNom"] = "Durand";
tabClients.Rows[4]["ClientNom"] = "Lamiel";
tabClients.Rows[5]["ClientNom"] = "Renoux";

// Remplissage de la colonne "ClientNom" de chaque ligne de clients créée
tabClients.Rows[0]["Payé"] = true;
tabClients.Rows[1]["Payé"] = true;
tabClients.Rows[2]["Payé"] = false;
tabClients.Rows[3]["Payé"] = false;
tabClients.Rows[4]["Payé"] = true;
tabClients.Rows[5]["Payé"] = false;

// Déclaration d'un référence de ligne de données
DataRow newRow2;

/* pour chacun des 6 clients remplissage aléatoire des 4 colonnes selon
un maximum de 7 lignes d'achats différents.
*/
Random gen = new Random();
int max ;
do{max = gen.Next(8);} while(max==0);
for(int i = 1; i <= 6; i++){
    for(int j = 1; j < max; j++){
        // création d'une nouvelle ligne d'achats
        newRow2 = tabAchats.NewRow();
        newRow2["ClientID"]= 1000+i;
        newRow2["DateAchat"]= new DateTime(2005, gen.Next(12)+1, gen.Next(29)+1);
        newRow2["MontantAchat"] = Math.Round(gen.NextDouble()*1000,2);
        newRow2["Article"]= Enum.GetName(typeof(typArticle),gen.Next(5));
        // Ajouter la ligne à la table Achats
        tabAchats.Rows.Add(newRow2);
    }
do{max = gen.Next(8);} while(max==0);
}

```

Affichage des données d'un DataSet

Voici affiché dans un composant visuel de NetFramework1.1 et NetFramework 2.0, le DataGrid, la table des Clients avec ses trois colonnes remplies par le programme précédent :

Affichage de la table des Clients			
	ClientID	ClientNom	Payé
▶	1001	Legrand	<input checked="" type="checkbox"/>
▶	1002	Fischer	<input checked="" type="checkbox"/>
▶	1003	Dupont	<input type="checkbox"/>
▶	1004	Durand	<input type="checkbox"/>
▶	1005	Lamiel	<input checked="" type="checkbox"/>
▶	1006	Renoux	<input type="checkbox"/>
*			

Voici affiché dans un autre composant visuel DataGrid, la table des Achats avec ses quatre colonnes remplies par le programme précédent :

Affichage de la table des Achats (euros HT)				
	ClientID	Article	MontantAchat	DateAchat
▶	1001	télévison	653,1	24/07/2005
	1001	radio	866,27	24/06/2005
	1001	cuisinière	58,23	14/02/2005
	1002	radio	443,34	12/06/2005
	1002	vélo	679,42	10/02/2005
	1002	radio	111,33	23/03/2005
	1002	vélo	788,25	22/07/2005
	1002	cuisinière	877,68	03/07/2005

Le composant DataGrid destiné à afficher une table de données, est remplacé dans la version 2.0 par le DataGridView, mais est toujours présent et utilisable, nous verrons illustrerons l'utilisation d'un DataGridView plus loin dans ce document avec des données provenant d'une Base de Données.

```
private System.Windows.Forms.DataGrid DataGridSimple;
```

Voici deux façons de lier ce composant visuel à la première table (la table Clients) du DataSet du programme de gestion du Magasin :

```
// Lie directement par le nom de la table le DataGrid au DataSet.  
DataGridSimple.SetDataSource(unDataSet, "Clients");
```

```
// Lie indirectement par le rang, le DataGrid au DataSet sur la première table :  
DataGridSimple.SetDataSource(unDataSet, unDataSet.Tables[0].TableName);
```

Enfin pour terminer la description des actions pratiques d'un DataSet, indiquons qu'il est possible de sauvegarder le schéma structuré (relation, clef primaire, tables,...) d'un DataSet dans un fichier de schémas au format XSL; il est aussi possible de sauvegarder toutes les données et leur structuration dans un fichier au format XML.

Sauvegarde des données du DataSet aux formats XML et XSL :

```
// Stockage uniquement du schéma général du DataSet dans un fichier séparé  
unDataSet.WriteXmlSchema("Donnees.xml");
```

```
// Stockage à la fois dans un même fichier du schéma et des données  
unDataSet.WriteXml("Donnees.xml", XmlWriteMode.WriteSchema);
```

Fichier "Donnees.xml" obtenu au format XSL

```
<?xml version="1.0" standalone="yes"?>
```

```

<xs:schema id="NewDataSet" xmlns="" xmlns:xs="http://www.w3.org/2001/XMLSchema"
xmlns:msdata="urn:schemas-microsoft-com:xml-msdata">
  <xs:element name="NewDataSet" msdata:IsDataSet="true" msdata:Locale="fr-FR">
    <xs:complexType>
      <xs:choice maxOccurs="unbounded">
        <xs:element name="Clients">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="ClientID" type="xs:int" />
              <xs:element name="ClientNom" type="xs:string" />
              <xs:element name="Payé" type="xs:boolean" minOccurs="0" />
              <xs:element name="Achats" minOccurs="0" maxOccurs="unbounded">
                <xs:complexType>
                  <xs:sequence>
                    <xs:element name="ClientID" type="xs:int" minOccurs="0" />
                    <xs:element name="Article" type="xs:string" minOccurs="0" />
                    <xs:element name="MontantAchat" type="xs:decimal" minOccurs="0" />
                    <xs:element name="DateAchat" type="xs:dateTime" minOccurs="0" />
                  </xs:sequence>
                </xs:complexType>
              </xs:element>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:choice>
    </xs:complexType>
    <xs:unique name="Constraint1">
      <xs:selector xpath="./Clients" />
      <xs:field xpath="ClientID" />
    </xs:unique>
    <xs:unique name="Constraint2" msdata:PrimaryKey="true">
      <xs:selector xpath="./Clients" />
      <xs:field xpath="ClientID" />
      <xs:field xpath="ClientNom" />
    </xs:unique>
    <xs:keyref name="Liste_x0020_des_x0020_achats" refer="Constraint1" msdata:IsNested="true">
      <xs:selector xpath="./Achats" />
      <xs:field xpath="ClientID" />
    </xs:keyref>
  </xs:element>
</xs:schema>

```

Fichier "Donnees.xml" obtenu format XML

```

<?xml version="1.0" standalone="yes"?>
<NewDataSet>
  <xs:schema id="NewDataSet" xmlns="" xmlns:xs="http://www.w3.org/2001/XMLSchema"
xmlns:msdata="urn:schemas-microsoft-com:xml-msdata">
    <xs:element name="NewDataSet" msdata:IsDataSet="true" msdata:Locale="fr-FR">
      <xs:complexType>
        <xs:choice maxOccurs="unbounded">
          <xs:element name="Clients">
            <xs:complexType>
              <xs:sequence>
                <xs:element name="ClientID" type="xs:int" />
                <xs:element name="ClientNom" type="xs:string" />
                <xs:element name="Payé" type="xs:boolean" minOccurs="0" />
                <xs:element name="Achats" minOccurs="0" maxOccurs="unbounded">
                  <xs:complexType>

```

```

    <xs:sequence>
      <xs:element name="ClientID" type="xs:int" minOccurs="0" />
      <xs:element name="Article" type="xs:string" minOccurs="0" />
      <xs:element name="MontantAchat" type="xs:decimal" minOccurs="0" />
      <xs:element name="DateAchat" type="xs:dateTime" minOccurs="0" />
    </xs:sequence>
  </xs:complexType>
</xs:element>
</xs:sequence>
</xs:complexType>
</xs:element>
</xs:choice>
</xs:complexType>
<xs:unique name="Constraint1">
  <xs:selector xpath="//Clients" />
  <xs:field xpath="ClientID" />
</xs:unique>
<xs:unique name="Constraint2" msdata:PrimaryKey="true">
  <xs:selector xpath="//Clients" />
  <xs:field xpath="ClientID" />
  <xs:field xpath="ClientNom" />
</xs:unique>
<xs:keyref name="Liste_x0020_des_x0020_achats" refer="Constraint1" msdata:IsNested="true">
  <xs:selector xpath="//Achats" />
  <xs:field xpath="ClientID" />
</xs:keyref>
</xs:element>
</xs:schema>
<Clients>
  <ClientID>1001</ClientID>
  <ClientNom>Legrand</ClientNom>
  <Payé>true</Payé>
  <Achats>
    <ClientID>1001</ClientID>
    <Article>cuisinière</Article>
    <MontantAchat>456.74</MontantAchat>
    <DateAchat>2005-12-12T00:00:00.0000000+01:00</DateAchat>
  </Achats>
</Clients>
<Clients>
  <ClientID>1002</ClientID>
  <ClientNom>Fischer</ClientNom>
  <Payé>true</Payé>
  <Achats>
    <ClientID>1002</ClientID>
    <Article>radio</Article>
    <MontantAchat>297.58</MontantAchat>
    <DateAchat>2005-02-25T00:00:00.0000000+01:00</DateAchat>
  </Achats>
  <Achats>
    <ClientID>1002</ClientID>
    <Article>télévison</Article>
    <MontantAchat>715.1</MontantAchat>
    <DateAchat>2005-07-19T00:00:00.0000000+02:00</DateAchat>
  </Achats>
  <Achats>
    <ClientID>1002</ClientID>
    <Article>télévison</Article>
    <MontantAchat>447.55</MontantAchat>
    <DateAchat>2005-08-16T00:00:00.0000000+02:00</DateAchat>
  </Achats>
</Clients>

```

```

</Achats>
<Achats>
  <ClientID>1002</ClientID>
  <Article>cuisinière</Article>
  <MontantAchat>92.64</MontantAchat>
  <DateAchat>2005-09-23T00:00:00.0000000+02:00</DateAchat>
</Achats>
<Achats>
  <ClientID>1002</ClientID>
  <Article>cuisinière</Article>
  <MontantAchat>171.07</MontantAchat>
  <DateAchat>2005-01-23T00:00:00.0000000+01:00</DateAchat>
</Achats>
</Clients>
<Clients>
  <ClientID>1003</ClientID>
  <ClientNom>Dupont</ClientNom>
  <Payé>>false</Payé>
  <Achats>
    <ClientID>1003</ClientID>
    <Article>aspirateur</Article>
    <MontantAchat>445.89</MontantAchat>
    <DateAchat>2005-02-11T00:00:00.0000000+01:00</DateAchat>
  </Achats>
</Clients>
<Clients>
  <ClientID>1004</ClientID>
  <ClientNom>Durand</ClientNom>
  <Payé>>false</Payé>
  <Achats>
    <ClientID>1004</ClientID>
    <Article>télévison</Article>
    <MontantAchat>661.47</MontantAchat>
    <DateAchat>2005-11-15T00:00:00.0000000+01:00</DateAchat>
  </Achats>
</Clients>
<Clients>
  <ClientID>1005</ClientID>
  <ClientNom>Lamiel</ClientNom>
  <Payé>>true</Payé>
</Clients>
<Clients>
  <ClientID>1006</ClientID>
  <ClientNom>Renoux</ClientNom>
  <Payé>>false</Payé>
  <Achats>
    <ClientID>1006</ClientID>
    <Article>cuisinière</Article>
    <MontantAchat>435.17</MontantAchat>
    <DateAchat>2005-06-22T00:00:00.0000000+02:00</DateAchat>
  </Achats>
  <Achats>
    <ClientID>1006</ClientID>
    <Article>cuisinière</Article>
    <MontantAchat>491.3</MontantAchat>
    <DateAchat>2005-12-25T00:00:00.0000000+01:00</DateAchat>
  </Achats>
  <Achats>
    <ClientID>1006</ClientID>
    <Article>cuisinière</Article>

```

```
<MontantAchat>388.81</MontantAchat>
<DateAchat>2005-10-13T00:00:00.0000000+02:00</DateAchat>
</Achats>
<Achats>
  <ClientID>1006</ClientID>
  <Article>radio</Article>
  <MontantAchat>864.93</MontantAchat>
  <DateAchat>2005-07-23T00:00:00.0000000+02:00</DateAchat>
</Achats>
</Clients>
</NewDataSet>
```

ADO .Net et SQL serveur 2005



Plan général: 

1. Rappel sur l'installation de SQL Serveur 2005

2. Accès lecture en mode connecté à une BD

Exemple de lecture en mode connecté dans une BD Access

Exemple de lecture en mode connecté dans une BD SQL serveur 2005

mode déconnecté : Affichage avec le DataGridView

1. **DataGridView** lié directement en lecture et écriture à un **DataTable**
2. Construire les lignes et les colonnes d'un **DataGridView** par programme
3. **DataGridView** lié à un **DataSet**

mode déconnecté : modifications de données à partir d'un DataGridView lié à un DataSet lui-même connecté à une BD

Exercices :

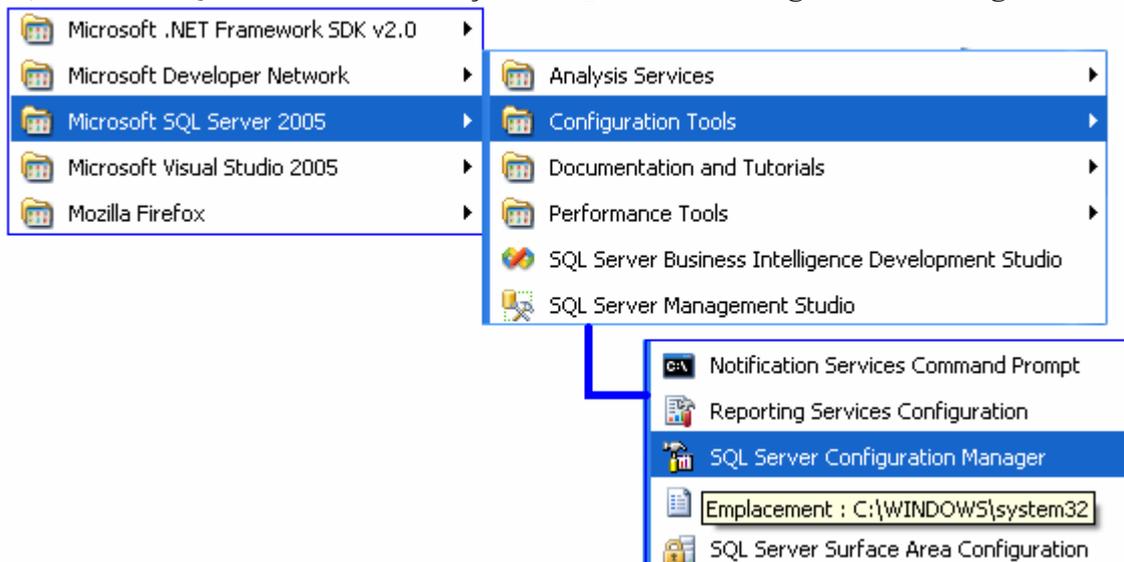
- **Gestion simplifiée d'un petit magasin**
- **Amélioration de la gestion du petit magasin : clef étrangère et delete/update en cascade**

1. Rappel sur l'installation de SQL Serveur 2005

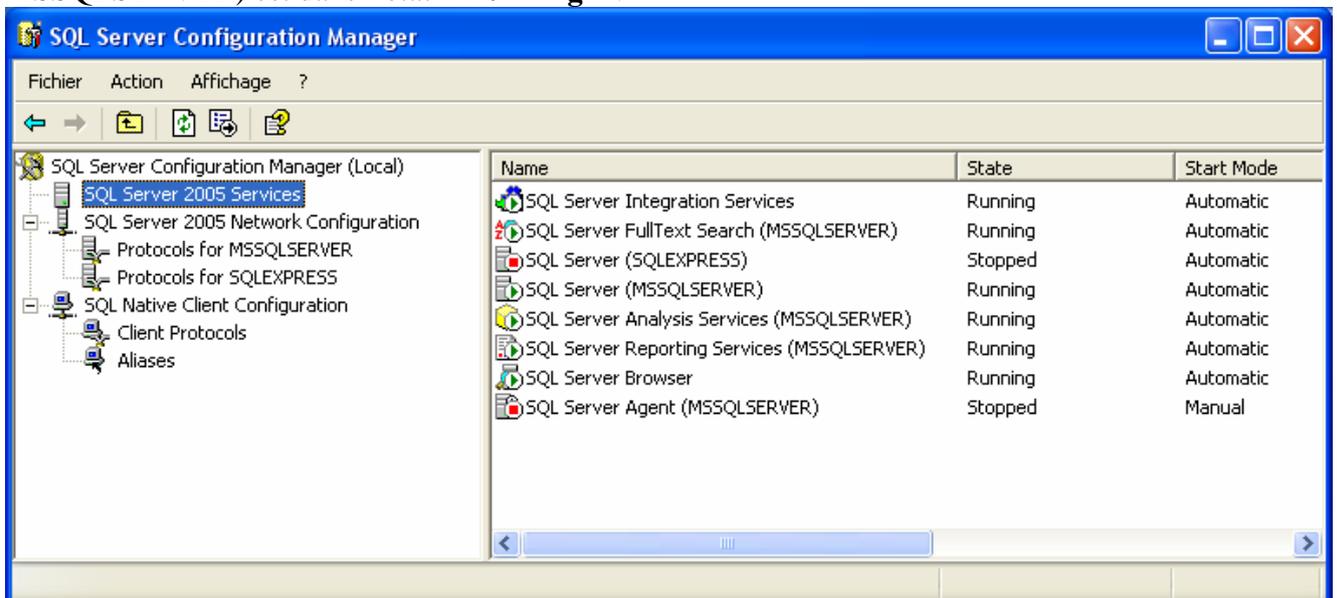
Installation de SQL server 2005 :

- désinstaller toute version précédente de SQL server
- installer SQL server 2005 éd. Developer (2 x CD-ROM).

1°) Vérifiez SQL server 2005 en lançant « **SQL server configuration manager** » :

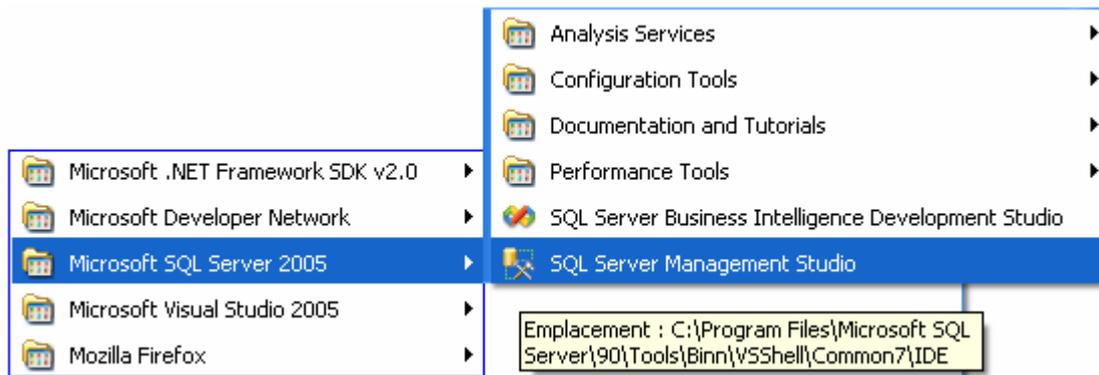


Le gestionnaire de configuration doit vous indiquer que SQL server (nommé ici MSSQLSERVER) est dans l'état « **Running** » :



La figure précédente montre qu'il est possible d'installer sur la même machine "SQL server express" (version gratuite) et SQL server 2005 (version payante).

2°) Accéder à une BD SQL server 2005 en lançant « **SQL server Management Studio** » :



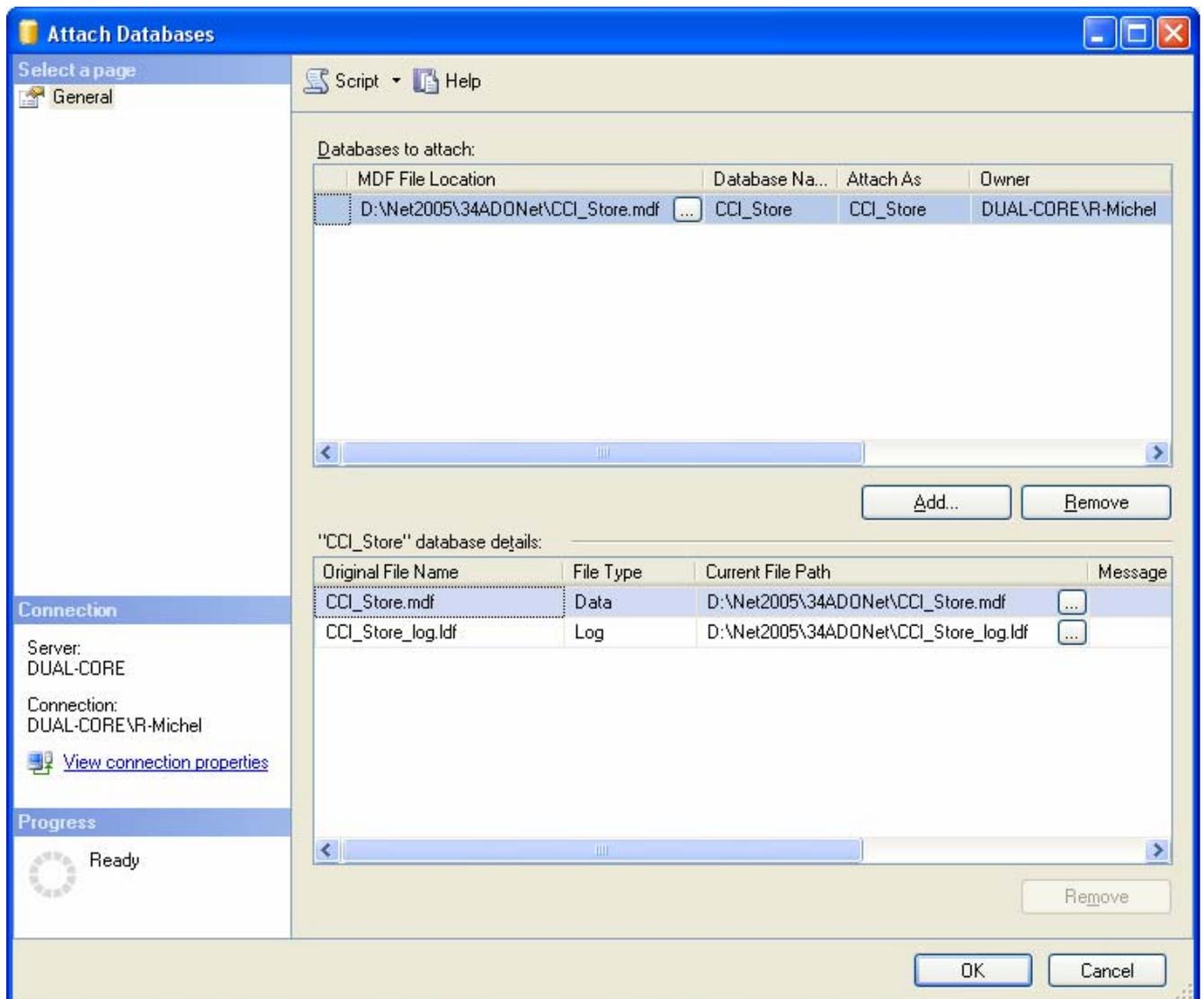
Invite de connexion à suivre



3°) Attacher une BD type *.mdf à SQL server 2005 (bouton droit de souris sur Databases) :



Sélectionner le fichier de la BD déjà existante (ici « CCI_Store.mdf ») :

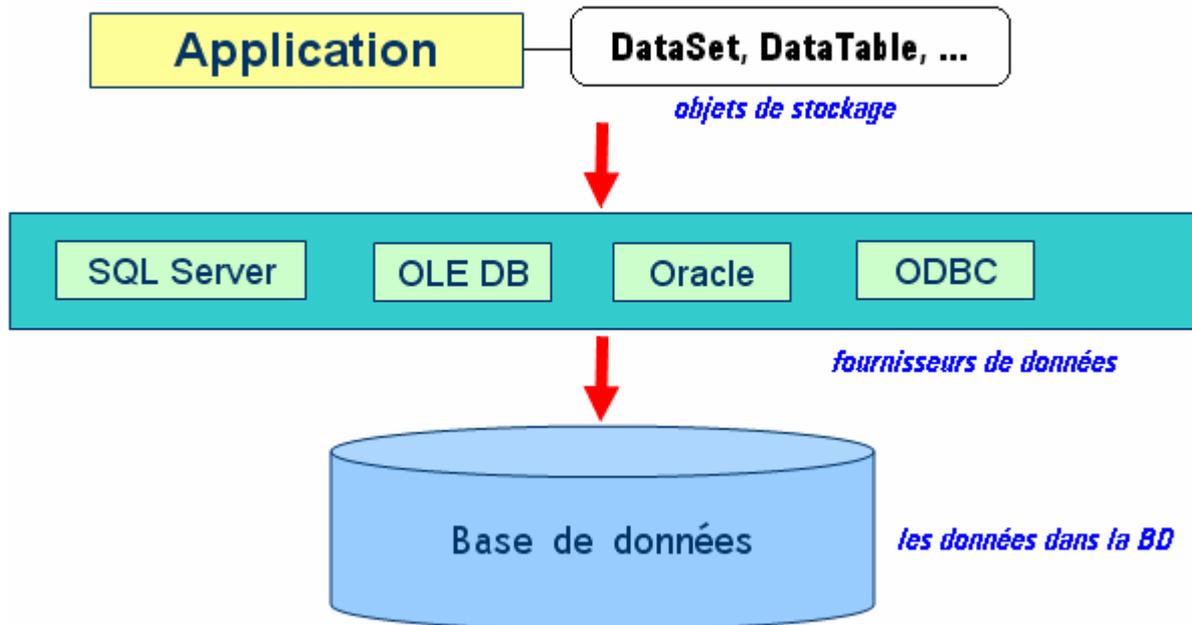


Pour manipuler SQL serveur 2005, voir le manuel fournit par Microsoft ou bien des ouvrages spécialisés sur le sujet. Par la suite, après avoir installé SQL serveur 2005 sur notre machine, nous avons pour objectif de montrer comment accéder à une base de données en local à partir d'un programme écrit en C# en utilisant les outils d'ADO.Net.

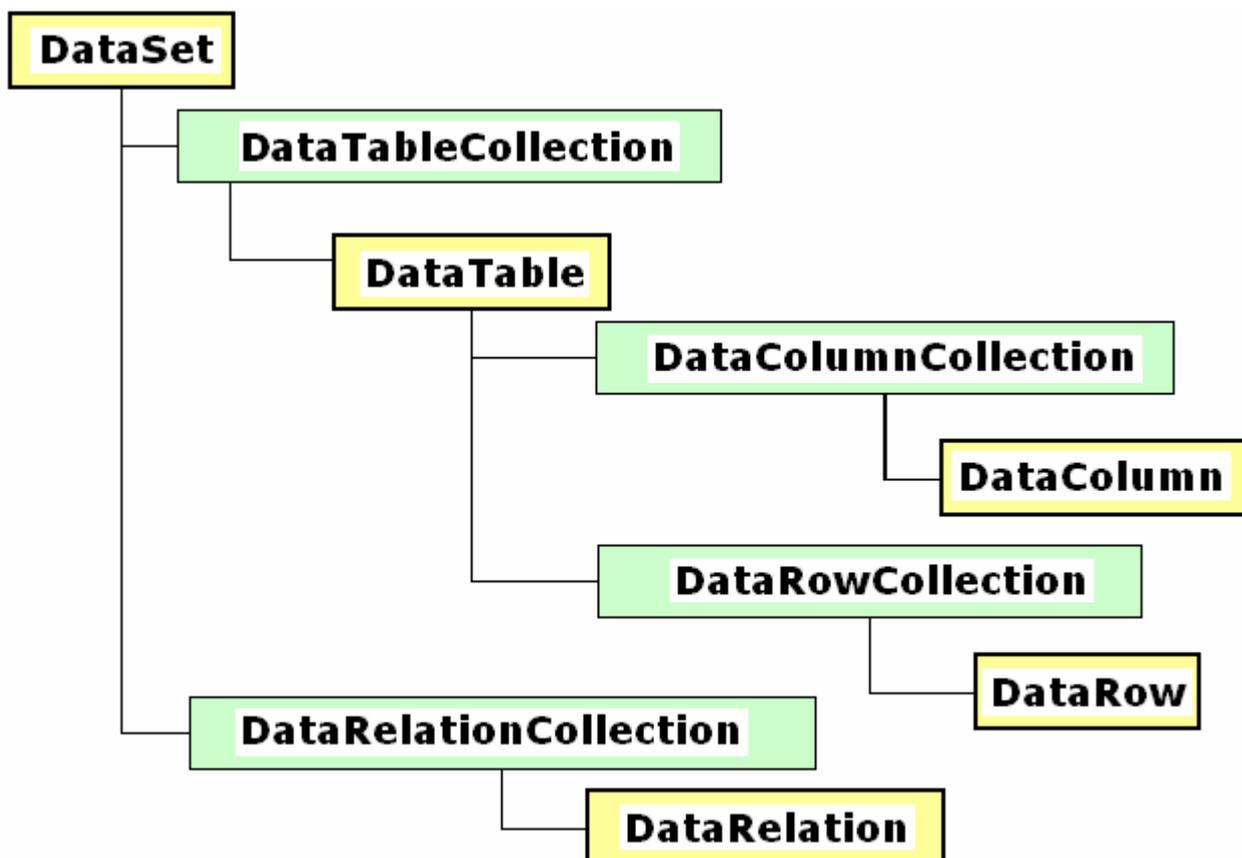
2. Accès lecture en mode connecté à une BD

Rappelons qu'ADO.Net 2.0 est composé de deux ensembles d'outils principaux de gestion des données :

- Les **objets de stockage de données** : objets déconnectés stockant les données sur la machine locale.
- Les **objets fournisseurs** (objets connectés directement) : gérant la communication entre le programme et la base de données.



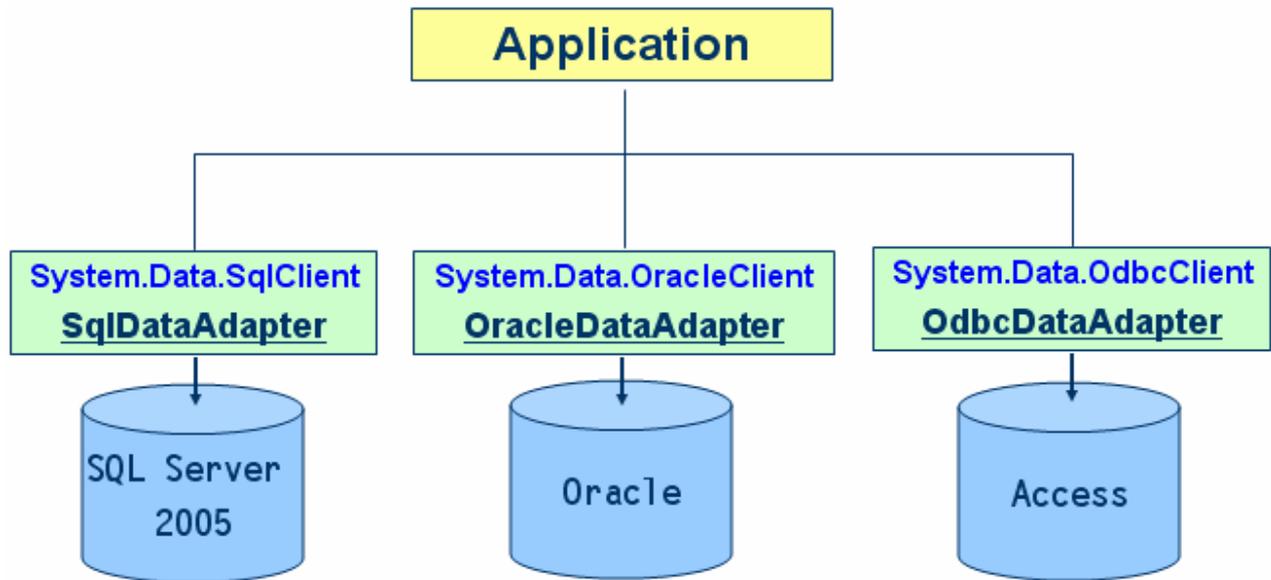
Nous avons exploré dans le chapitre précédent les possibilités offertes par les objets de données comme principalement le DataSet dont nous rappelons l'organisation ci-dessous :



Nous explorons dans ce chapitre, les objets du type fournisseurs et leurs utilisations dans un programme.

Ado .Net met propose plusieurs classes de **fournisseurs** permettant de gérer l'accès à diverses sources de données, toutes situées dans le namespace **System.Data** :

- **System.Data.SqlClient** : accès aux bases de données SQL Serveur
- **System.Data.Odbc** : accès aux bases de données gérées par Odbc
- **System.Data.OracleClient** : accès aux bases de données Oracle
-

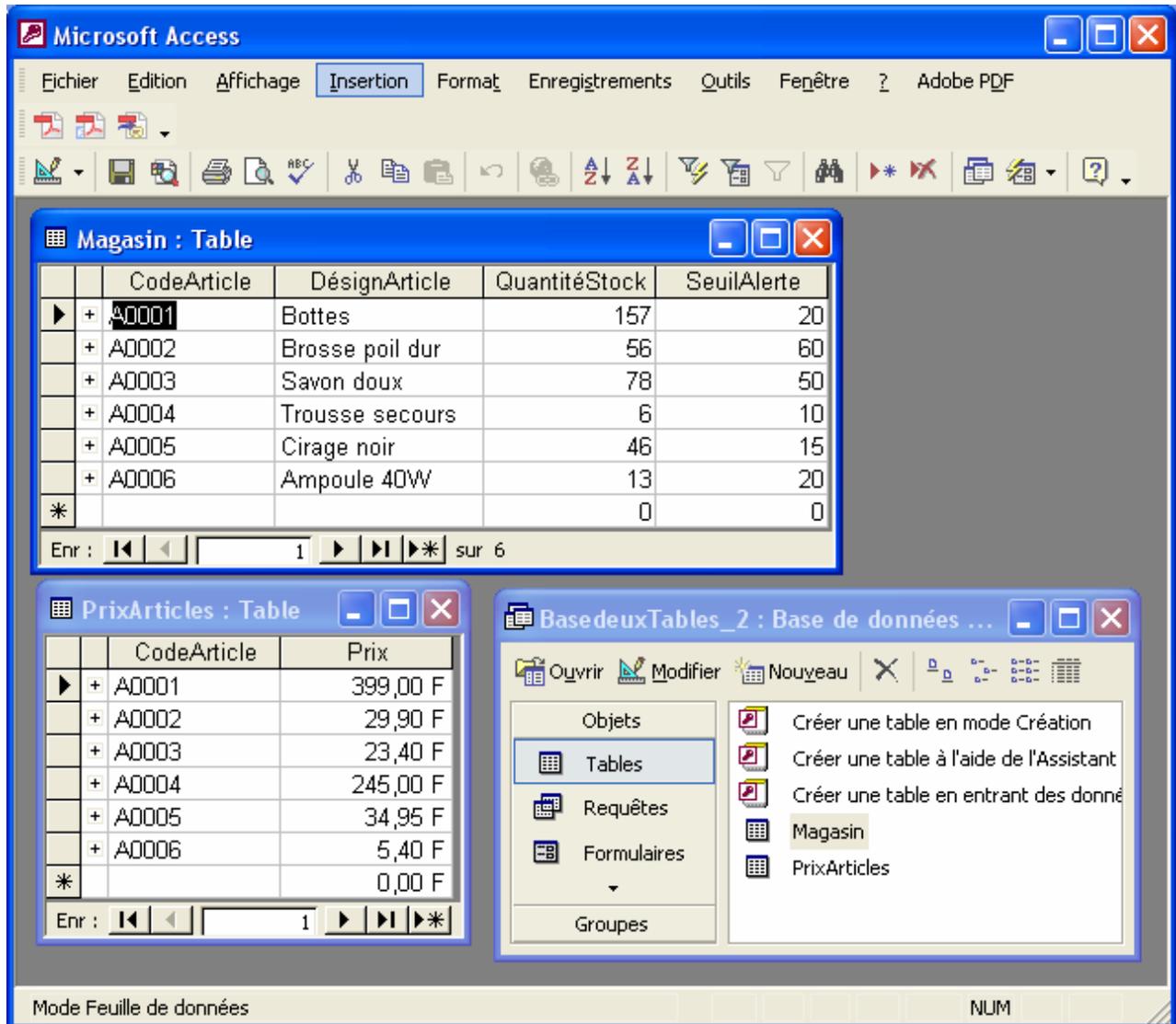


Ci-dessous les principaux objets d'ADO.Net 2.0 en liaison avec les BD :

Objets	Fonctionnalités
OdbcConnection SqlConnection OracleConnection	Assurent la gestion des connexions physiques vers la base de données du type concerné (ODBC, SQL server, Oracle,)
OdbcCommand SqlCommand OracleCommand	Assure le fonctionnement des commandes SQL (SELECT, INSERT, UPDATE, DELETE) vers la base de données. L'exécution de la commande (aussi appelée requête) s'effectue grâce à 4 méthodes qui se dénomment Executexxx selon le type de traitement à effectuer (par ex : ExecuteReader sert au mode connecté).
OdbcDataReader SqlDataReader OracleDataReader	Mode connecté : permet de lire les résultats des requêtes renvoyés par l'exécution de la commande SQL (en lecture seule). Ce mode de travail dans lequel le programme est connecté directement sur la BD, est déconseillé dès que l'application est complexe (par exe : modifications fréquentes).
DataSet	Mode déconnecté : permet de lire et de travailler les résultats des requêtes renvoyés par l'exécution de la commande SQL indépendamment de la BD. Le programme effectue un accès à la BD et range les données et leurs relations stockées en mémoire centrale dans un objet DataSet en cache. Ce mode est conseillé car l'application travaille alors sur une "image" des données et de leurs relations. Il faut ensuite effectuer une validation des modifications apportées aux données dans le cache.

Exemple de lecture en mode connecté dans une BD Access

Lancement de la lecture d'une table " **Magasin** " dans une BD Access nommée " **BasedeuxTables_2.mdb** ":



Nous nous proposons d'écrire un programme lisant et affichant le contenu de cette BD, enregistrement par enregistrement.

Le schéma général qui est adopté pour une telle action est le suivant :

- **Création d'un objet de connexion physique à la BD.**
- **Ouverture de la connexion physique à la BD.**
- **Création d'un objet de commande avec une requête.**
- **Exécution de la commande en mode connecté.**
- **Lecture et affichage des champs résultats de la commande.**
- **Fermeture de la connexion physique à la BD.**
- **Libération des ressources utilisées.**

Code source d'un programme affichant les résultats d'une requête dans un textBox :

```

// chemin vers la BD (Access):
string urlAccess = "Driver={Microsoft Access Driver (*.mdb)};"
+ @"DBQ=C:\BasedeuxTables_2.mdb";

// objet de connexion :
OdbcConnection connexion;
connexion = new OdbcConnection(urlAccess);

try
{
// ouvrir la connexion :
connexion.Open();

// préparation de la requête effective :
string Requete = "SELECT * FROM Magasin";
OdbcCommand cmdAccess = new OdbcCommand(Requete, connexion);

// lecture de plusieurs lignes :
OdbcDataReader lignesRead;
lignesRead = cmdAccess.ExecuteReader(CommandBehavior.CloseConnection);
while (lignesRead.Read())
{
    textBoxAffiche.AppendText(lignesRead.GetString(0) + " " +
        lignesRead.GetString(1) + " " +
        lignesRead.GetString(2) + " " +
        lignesRead.GetString(3) + "\r\n");
}

// fermer la connexion :
connexion.Close();
}
catch (Exception ex)
{
    Console.WriteLine("Désolé : " + ex.Message);
    textBoxAffiche.Clear();
    textBoxAffiche.AppendText("Désolé : " + ex.Message);
}

// libérer les ressources :
connexion.Dispose();
connexion = null;

```

Création d'un objet de connexion physique à la BD : BasedeuxTables_2.mdb

Ouverture de la connexion physique à la BD.

Création d'un objet de commande avec une requête.

Exécution de la commande en mode connecté.

Lecture et affichage des 4 champs résultats de la commande.



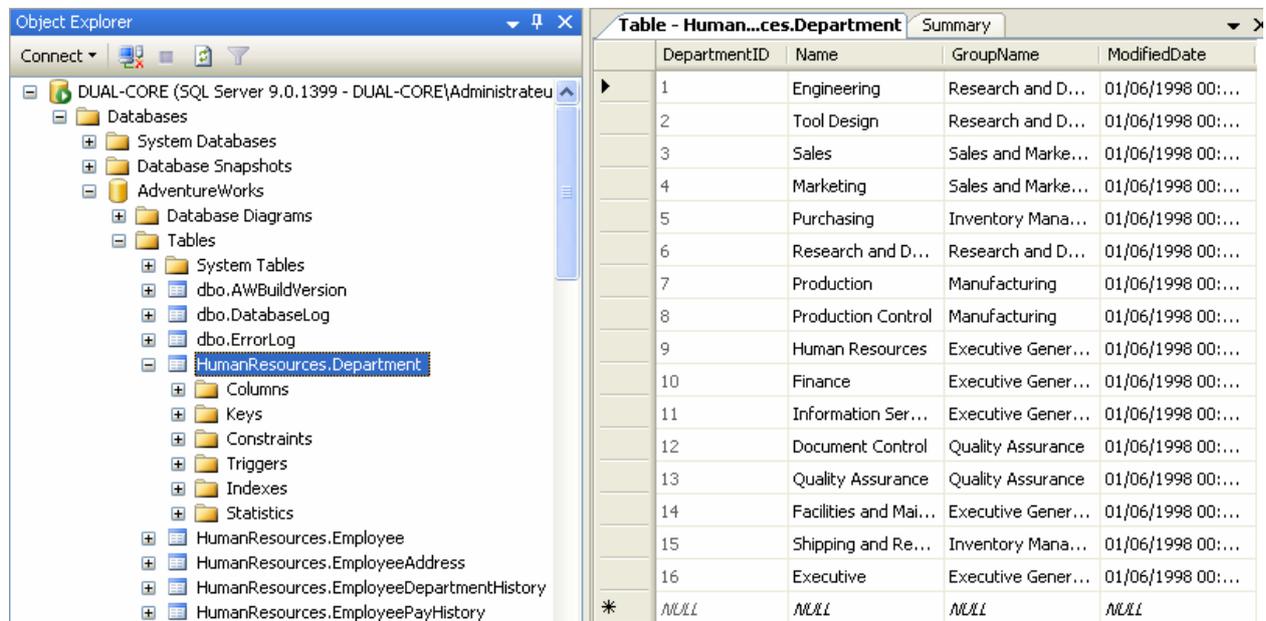
```

// libérer les ressources :
connexion.Dispose();
connexion = null;

```

Exemple de lecture en mode connecté dans une BD SQL serveur 2005

Reprenons la même action sur la BD de démonstration gratuite fournie par Microsoft avec SQL serveur 2005 "AdventureWorks.mdf" et lançons la lecture des 16 enregistrements de la table "HumanResources.Department":



DepartmentID	Name	GroupName	ModifiedDate
1	Engineering	Research and D...	01/06/1998 00:...
2	Tool Design	Research and D...	01/06/1998 00:...
3	Sales	Sales and Marke...	01/06/1998 00:...
4	Marketing	Sales and Marke...	01/06/1998 00:...
5	Purchasing	Inventory Mana...	01/06/1998 00:...
6	Research and D...	Research and D...	01/06/1998 00:...
7	Production	Manufacturing	01/06/1998 00:...
8	Production Control	Manufacturing	01/06/1998 00:...
9	Human Resources	Executive Gener...	01/06/1998 00:...
10	Finance	Executive Gener...	01/06/1998 00:...
11	Information Ser...	Executive Gener...	01/06/1998 00:...
12	Document Control	Quality Assurance	01/06/1998 00:...
13	Quality Assurance	Quality Assurance	01/06/1998 00:...
14	Facilities and Mai...	Executive Gener...	01/06/1998 00:...
15	Shipping and Re...	Inventory Mana...	01/06/1998 00:...
16	Executive	Executive Gener...	01/06/1998 00:...
*	NULL	NULL	NULL

// chemin vers la BD (SQL serveur 2005):

```
string urlSqlServer = @"Data Source=(local);Initial Catalog=AdventureWorks;" + "Integrated Security=SSPI;";
```

Le mécanisme reste le même que celui qui a été utilisé pour la BD Access, seuls les objets permettant la connexion, la commande, la lecture sont différents mais opèrent d'une manière identique. Ci-dessous les lignes de code qui diffèrent selon le type de BD utilisée :

```
// objet de connexion :
SqlConnection connexion;
connexion = new SqlConnection(urlSqlServer);
```

```
....
// préparation de la requête effective :
string Requete = "SELECT * FROM HumanResources.Department";
SqlCommand cmdSqlServ = new SqlCommand (Requete, connexion);
```

```
....
//lecture de plusieurs lignes :
SqlDataReader lignesRead;
lignesRead = cmdSqlServ.ExecuteReader(CommandBehavior.CloseConnection);
textBoxAffiche.Clear();
while (lignesRead.Read())
{
    textBoxAffiche.AppendText(Convert.ToString(lignesRead.GetInt16(0)) + " -- " +
        lignesRead.GetString(1) + " -- " +
        lignesRead.GetString(2) + " -- " +
        lignesRead.GetDateTime(3).ToString() + "\r\n");
}
```

Code source complet du programme affichant les résultats de la requête dans un textBox :

```
// chemin vers la BD (SQL serveur 2005):
string urlSqlServer = @"Data Source=(local);Initial Catalog=AdventureWorks;" + "Integrated Security=SSPI;";

// objet de connexion :
SqlConnection connexion;
connexion = new SqlConnection(urlSqlServer);

Console.WriteLine();

// ouvrir la connexion :
connexion.Open();

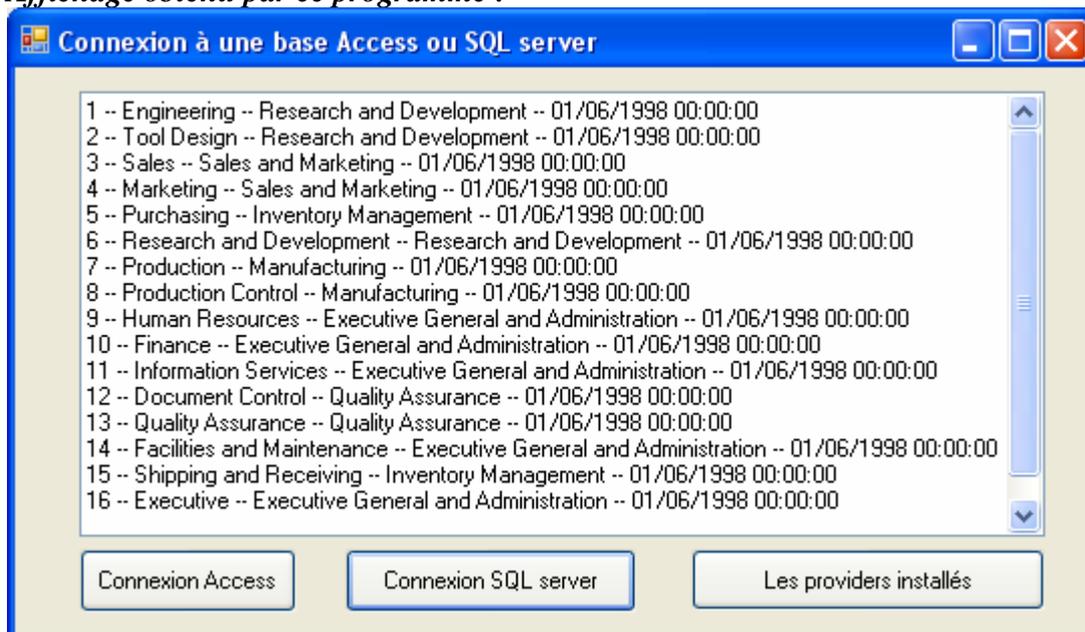
// traitement de la BD :
string Requete = "SELECT * FROM HumanResources.Department";
SqlCommand cmdSqlServ = new SqlCommand(Requete, connexion);

//lecture de plusieurs lignes :
SqlDataReader lignesRead;
lignesRead = cmdSqlServ.ExecuteReader(CommandBehavior.CloseConnection);
textBoxAffiche.Clear();
while (lignesRead.Read())
{
    textBoxAffiche.AppendText(Convert.ToString(lignesRead.GetInt16(0)) + "-- " +
        lignesRead.GetString(1) + "-- " +
        lignesRead.GetString(2) + "-- " +
        lignesRead.GetDateTime(3).ToString() + "\r\n");
}

// fermer la connexion :
connexion.Close();

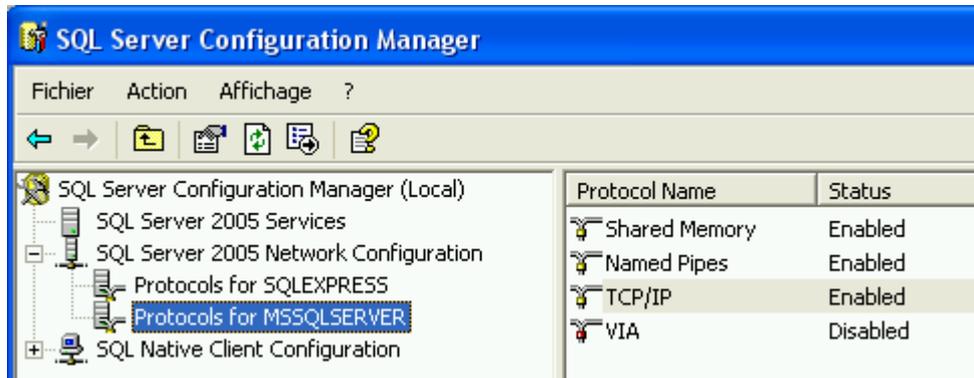
// libérer les ressources :
connexion.Dispose();
connexion = null;
```

Affichage obtenu par ce programme :

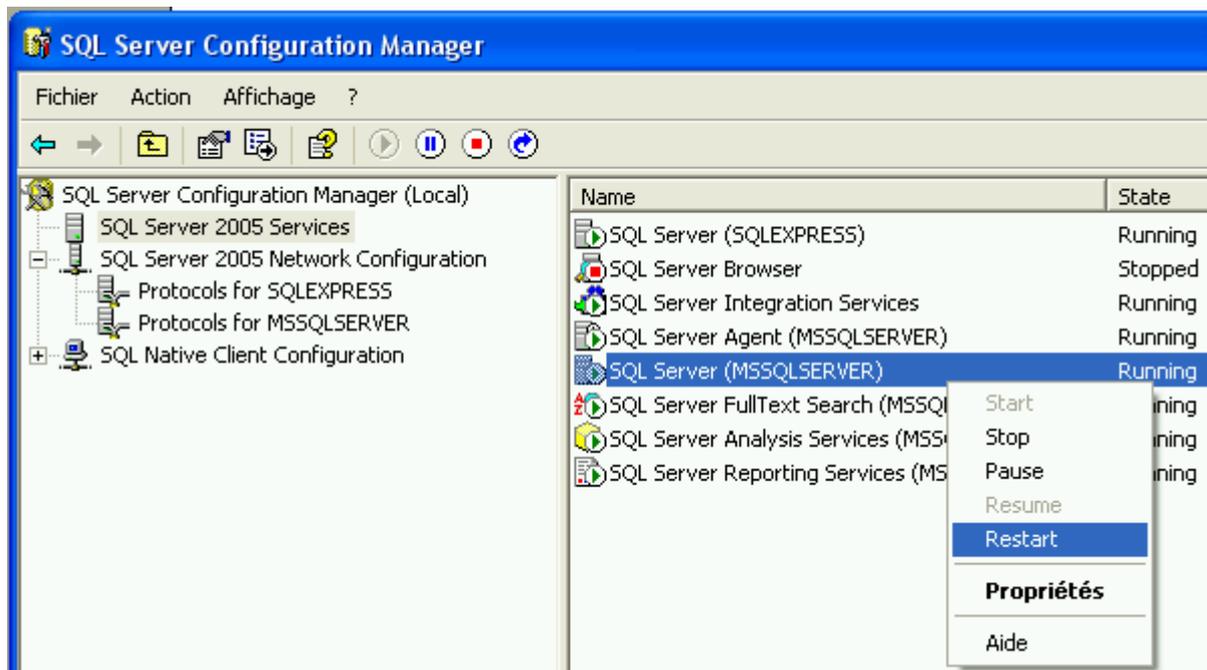


Remarque pratique de paramétrage de SQL serveur 2005 :

Si dans une utilisation pratique des exemples traités ci-après vous obtenez un message d'erreur indiquant que la connexion TCP/IP n'a pas pu avoir lieu, vérifiez que le statut du protocole TCP/IP est « **enabled** » en exécutant le programme SQL Server Config.Manager de SQL serveur 2005 :



N'oubliez pas ensuite de redémarrer SQL Server, à partir du même SQL Server Config. :



mode déconnecté : Affichage avec le DataGridView

Un **DataGridView** permet d'afficher des données provenant de différentes sources grâce à sa propriété de liaison de données (data-bind) **DataSource** en lecture ou écriture, qui est une référence vers l'objet qui contient les données à afficher.

1°) On peut lier un **DataGridView** directement en lecture et écriture à un **DataTable** :

```
DataGridView dataGridView1;.....
```

```
// création d'un DataTable :
```

```
DataTable table = new DataTable("Personnes");  
table.Columns.Add("Numéro", typeof(string));  
table.Columns[0].Unique = true;  
table.Columns.Add("Nom", typeof(string));  
table.Columns.Add("Prénom", typeof(string));  
table.Columns.Add("age", typeof(byte));  
table.Columns.Add("Revenus", typeof(double));
```

```
/* Data-bind du dataGridView1:
```

```
* après cette instruction le dataGridView1 est lié à la table :  
* toute ligne entrée manuellement dans le dataGridView1 est  
* automatiquement ajoutée dans la table.  
* */
```

```
dataGridView1.DataSource = table;//on relie à l'objet table
```

	Numéro	Nom	Prénom	age	Revenus
▶*					

//affichage de la table (vide pour l'instant)

```
/* Réciproquement, toute donnée ajoutée à la table est immédiatement  
* visualisée dans le dataGridView1.
```

```
* */
```

```
table.Rows.Add(888, "ffff", "xxxxx", 88, 8888);  
table.Rows.Add(999, "ggg", "yyyy", 99, 9999);
```

	Numéro	Nom	Prénom	age	Revenus
	888	ffff	xxxxx	88	8888
	999	ggg	yyyy	99	9999
▶*					

//affichage de la table (avec les deux lignes qui viennent d'être ajoutées)

2°) On peut construire les lignes et les colonnes d'un **DataGridView** par programme :

```
DataGridView dataGridView1;.....
```

```
// chargement par programme du dataGridView :
```

```
dataGridView1.ColumnCount = 5;  
dataGridView1.Columns[0].Name = "un";
```

```

dataGridView1.Columns[1].Name = "deux";
dataGridView1.Columns[2].Name = "trois";
dataGridView1.Columns[3].Name = "quatre";
dataGridView1.Columns[4].Name = "cinq";
dataGridView1.Rows.Add(123, "aaa", "bbb", 25, 1235.58);
dataGridView1.Rows.Add("xxxx", true, -23.6, "rrrr", 45);
dataGridView1.Rows.Add(4);
/* Remarque : le type n'est pas fixé tout est converti en string. */

```

	un	deux	trois	quatre	cinq
▶	123	aaa	bbb	25	1235,58
	xxxx	True	-23,6	rrrr	45

//affichage des lignes et des colonnes entrées par programme

3°) DataGridView lié à un DataSet :

Le principe d'utilisation des outils déconnectés est simple avec ADO.net, il est fondé sur une architecture MVC (modèle-vue-contrôleur) dans laquelle un **DataSet** est connecté à une BD et gère le modèle des données en mémoire centrale, un **DataGridView** est alors relié à une table du **DataSet**, il est ainsi chargé de gérer la vue et les interactions événementielles avec l'utilisateur.

Un **DataSet**, comme nous l'avons déjà vu dans un chapitre précédent permet de lire et d'écrire directement des données au format XML, il permet aussi par l'intermédiaire d'un objet du type **SqlDataAdapter** ou **OdbcDataAdapter** ou **OracleDataAdapter**..., de lire et d'écrire dans une BD de type SQL serveur, ODBC, Oracle... Le **DataGridView** sert alors à afficher et à modifier les données d'une table du **DataSet** :



D'une manière générale, un **xxxDataAdapter** est utilisé lors d'un échange avec un **DataSet** ou un **DataTable** (d'un **DataSet** ou non), un objet **xxxDataAdapter** est chargé d'assurer la liaison entre la BD physique et le **DataSet** ou le **DataTable** :

- Il contient des propriétés de commande SQL :
 - **public xxxCommand** SelectCommand { get; set; },
 - **public xxxCommand** InsertCommand { get; set; },
 - **public xxxCommand** UpdateCommand { get; set; },
 - **public xxxCommand** DeleteCommand { get; set; }, permettant de lancer ces commandes à partir du **DataTable** sur la BD.

- Il contient réciproquement des méthodes permettant de charger un **DataTable** ou un **DataSet** contenant un ou plusieurs **DataTable** (**public int** Fill(**DataTable** dataTable), **public int** Fill(**DataSet** dataSet),....)

Un **xxxDataAdapter** ouvre et ferme la connexion automatiquement (Open et Close).

DataGridView lié à un **DataSet** avec fichier XML

Nous montrons ici comment sauvegarder tout le contenu d'un **DataGridView** nommé dataGridView1 dans un fichier nommé "Table.xml" en utilisant un **DataSet** nommé dsTable.

a) Utilisons le **DataTable** nommé "Personnes" que nous avons créé précédemment :

```
DataTable table = new DataTable("Personnes");
table.Columns.Add("Numéro", typeof(string));
table.Columns[0].Unique = true;
table.Columns.Add("Nom", typeof(string));
table.Columns.Add("Prénom", typeof(string));
table.Columns.Add("age", typeof(byte));
table.Columns.Add("Revenus", typeof(double));
table.Rows.Add(888, "ffff", "xxxxx", 88, 8888);
table.Rows.Add(999, "ggg", "yyyy", 99, 9999);
```

b) Ajoutons cette table "Personnes" au **DataSet** dsTable :

```
DataSet dsTable = new DataSet();// création d'un DataSet
dsTable.Tables.Add(table); //ajout de la table "Personnes" au DataSet
```

c) Le dataGridView1 est lié au **DataSet** nommé dsTable par sa propriété **DataSource** et plus précisément à la première table du **DataSet** par sa propriété **DataMember** :

```
dataGridView1.DataSource = dsTable;
dataGridView1.DataMember = dsTable.Tables[0].TableName;
```

d) Le **DataSet** possède des méthodes permettant de sauvegarder les données brutes au format XML (méthode **WriteXml**), il peut aussi sauvegarder sa structure générale sous forme d'un schéma au format XML (méthode **WriteXmlSchema**). On peut dès lors, sauvegarder tout le contenu du dataGridView1 dans un fichier nommé "Table.xml" et le schéma du **DataSet** dans un fichier nommé "schemaTable.xml", en utilisant les méthodes **WriteXml** et **WriteXmlSchema** :

```
dsTable.WriteXmlSchema("schemaTable.xml");
dsTable.WriteXml("Table.xml");
```

Code d'un click sur un bouton permettant de sauvegarder au format XML :

```
DataSet dsTable = new DataSet();// création d'un DataSet
```

```
Button buttonSaveDataSet :  // bouton de sauvegarde au format XML
```

```

private void boutonSaveDataSet_Click(object sender, EventArgs e)
{
    // sauvegarde au format XML par DataSet du contenu du dataGridView :
    if (dsTable.Tables.Count != 0) // si le DataSet contient quelque chose !
    {
        dataGridView1.DataSource = dsTable;
        dataGridView1.DataMember = dsTable.Tables[0].TableName;
        dsTable.WriteXmlSchema("schemaTable.xml");
        dsTable.WriteXml("Table.xml");
        textBox1.Text = "nbr lignes = " +
            dsTable.Tables[0].Rows.Count.ToString();
    }
}

```

Le dataGridView1 a été chargé par le code a) , b) et c) :

	Numéro	Nom	Prénom	age	Revenus
	888	ffff	xxxxx	88	8888
	999	ggg	yyyy	99	9999
▶*					



Le click sur le bouton engendre deux fichiers au format XML :



Voici le contenu du fichier "schemaTable.xml" :

```

<?xml version="1.0" standalone="yes" ?>
- <xs:schema id="Temporaire" xmlns="" xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns:xm
- <xs:element name="Temporaire" msdata:IsDataSet="true" msdata:UseCurrentLocale="true">
  - <xs:complexType>
    - <xs:choice minOccurs="0" maxOccurs="unbounded">
      - <xs:element name="Personnes">
        - <xs:complexType>
          - <xs:sequence>
            <xs:element name="Numéro" type="xs:string" minOccurs="0" />
            <xs:element name="Nom" type="xs:string" minOccurs="0" />
            <xs:element name="Prénom" type="xs:string" minOccurs="0" />
            <xs:element name="age" type="xs:unsignedByte" minOccurs="0" />
            <xs:element name="Revenus" type="xs:double" minOccurs="0" />
          </xs:sequence>
        </xs:complexType>
      </xs:element>
    </xs:choice>
  </xs:complexType>
- <xs:unique name="Constraint1">
  <xs:selector xpath="./Personnes" />
  <xs:field xpath="Numéro" />
</xs:unique>
</xs:element>
</xs:schema>

```

On remarquera que la contrainte d'unicité pour les données de la colonne 0 est présente dans le schéma.

Voici le contenu du fichier "Table.xml" :

```
<?xml version="1.0" standalone="yes" ?>
- <Temporaire>
  - <Personnes>
    <Numéro>888</Numéro>
    <Nom>ffff</Nom>
    <Prénom>xxxxx</Prénom>
    <age>88</age>
    <Revenus>8888</Revenus>
  </Personnes>
  - <Personnes>
    <Numéro>999</Numéro>
    <Nom>ggg</Nom>
    <Prénom>yyyy</Prénom>
    <age>99</age>
    <Revenus>9999</Revenus>
  </Personnes>
</Temporaire>
```

Nous montrons maintenant comment récupérer les données et leur structure dans un **DataGridView** nommé dataGridView1 à partir des fichiers précédents "Table.xml" et "schemaTable.xml".

Le **DataSet** possède des méthodes permettant de lire les données brutes au format XML (méthode **ReadXml**), et de lire sa structure générale stockée au format XML (méthode **ReadXmlSchema**). On peut dès lors, afficher dans le dataGridView1 le contenu des fichiers nommés "Table.xml" et "schemaTable.xml", en utilisant les méthodes **ReadXml** et **ReadXmlSchema** puis en liant le dataGridView1 à la table du **DataSet** :

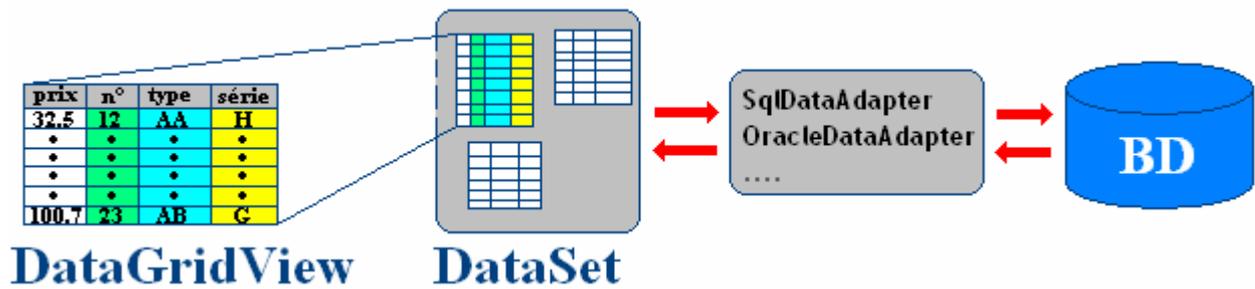
Code d'un click sur un bouton permettant de charger les fichiers XML :



Button buttonLoadDataSet : // bouton de chargement du fichier XML

```
private void buttonLoadDataSet_Click(object sender, EventArgs e)
{
  // récupération au format XML par DataSet du contenu du dataGridView :
  RAZ_DataGridView();// on efface
  dsTable = new DataSet("Temporaire");//RAZ du DataSet
  dsTable.ReadXmlSchema("schemaTable.xml");
  dsTable.ReadXml("Table.xml");
  dataGridView1.DataSource = dsTable;
  dataGridView1.DataMember = dsTable.Tables[0].TableName;
}
```

DataGridView lié à un DataSet connecté à une BD



Le principe reste le même que celui qui a été décrit précédemment du moins en ce qui concerne la liaison du DataGridView avec le DataSet.

La connexion aux données ne s'effectue pas directement comme avec des données au format XML, mais à travers un fournisseur de connexion à la BD adapté au type de la BD (SqlDataAdapter pour une BD SQL serveur, OracleDataAdapter pour une BD Oracle, ...)

Exemple de code pour affichage de la table "HumanResources.Department" dans la BD "AdventureWorks.mdf" de Microsoft :

Nous avons précédemment lancé la lecture des 16 enregistrements de cette table en mode connecté, ici nous montrons comment afficher cette même table en mode déconnecté.

The screenshot shows the Object Explorer on the left with the AdventureWorks database selected. The right pane displays the 'Table - Human...ces.Department' view with the following data:

DepartmentID	Name	GroupName	ModifiedDate
1	Engineering	Research and D...	01/06/1998 00:...
2	Tool Design	Research and D...	01/06/1998 00:...
3	Sales	Sales and Marke...	01/06/1998 00:...
4	Marketing	Sales and Marke...	01/06/1998 00:...
5	Purchasing	Inventory Mana...	01/06/1998 00:...
6	Research and D...	Research and D...	01/06/1998 00:...
7	Production	Manufacturing	01/06/1998 00:...
8	Production Control	Manufacturing	01/06/1998 00:...
9	Human Resources	Executive Gener...	01/06/1998 00:...
10	Finance	Executive Gener...	01/06/1998 00:...
11	Information Ser...	Executive Gener...	01/06/1998 00:...
12	Document Control	Quality Assurance	01/06/1998 00:...
13	Quality Assurance	Quality Assurance	01/06/1998 00:...
14	Facilities and Mai...	Executive Gener...	01/06/1998 00:...
15	Shipping and Re...	Inventory Mana...	01/06/1998 00:...
16	Executive	Executive Gener...	01/06/1998 00:...
*	NULL	NULL	NULL

```
private void loadDataGridViewFromDataSet()
{ /*----- Liaison par DataSet -----*/

    //--pour SQL server Ed.Developer 2005 :
    string urlSqlServer = @"Data Source=(local);Initial Catalog=AdventureWorks;"
        +"Integrated Security=SSPI;";

    // objet de connexion :
    SqlConnection connexion = new SqlConnection(urlSqlServer);
```

```
try  
{
```

```
// unSqlDataAdapter : objet de communication et d'échange de données entre DataSet et la BD  
SqlDataAdapter unSqlAdapter = new SqlDataAdapter("Select * From HumanResources.Department ",  
connexion);
```

```
//Remplissage du DataSet avec la table Person.Address :  
unSqlAdapter.Fill(unDataSet);
```

```
// Liaison du dataGridView1 à la table du DataSet :  
dataGridView1.DataSource = unDataSet.Tables[0];
```

```
}  
catch (SqlException ex)  
{  
    MessageBox.Show(ex.Message, "Erreur SQL", MessageBoxButtons.OK, MessageBoxIcon.Exclamation);  
}  
}
```

Résultat obtenu dans le DataGridView :



	DepartmentID	Name	GroupName	ModifiedDate
▶	1	Engineering	Research and D...	01/06/1998
	2	Tool Design	Research and D...	01/06/1998
	3	Sales	Sales and Market...	01/06/1998
	4	Marketing	Sales and Market...	01/06/1998
	5	Purchasing	Inventory Manag...	01/06/1998
	6	Research and D...	Research and D...	01/06/1998
	7	Production	Manufacturing	01/06/1998
	8	Production Control	Manufacturing	01/06/1998
	9	Human Resources	Executive Gener...	01/06/1998

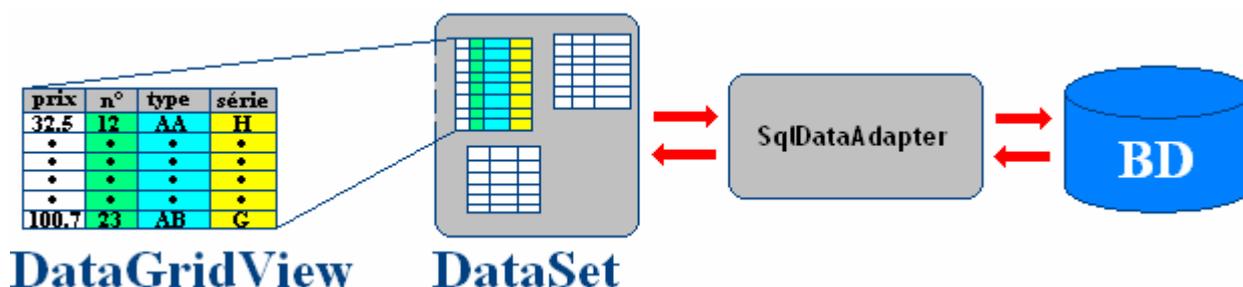
mode déconnecté : modifications de données à partir d'un DataGridView lié à un DataSet lui-même connecté à une BD

Le principe d'utilisation des outils déconnectés est simple avec ADO.net, il est fondé sur une architecture MVC (modèle-vue-contrôleur), nous proposons un exemple de travail en lecture et écriture dans une BD à partir de modifications effectuées dans un DataGridView lié à cette BD :

Soit une BD SQL serveur nommée "Boutique" possédant une table "Articles" :

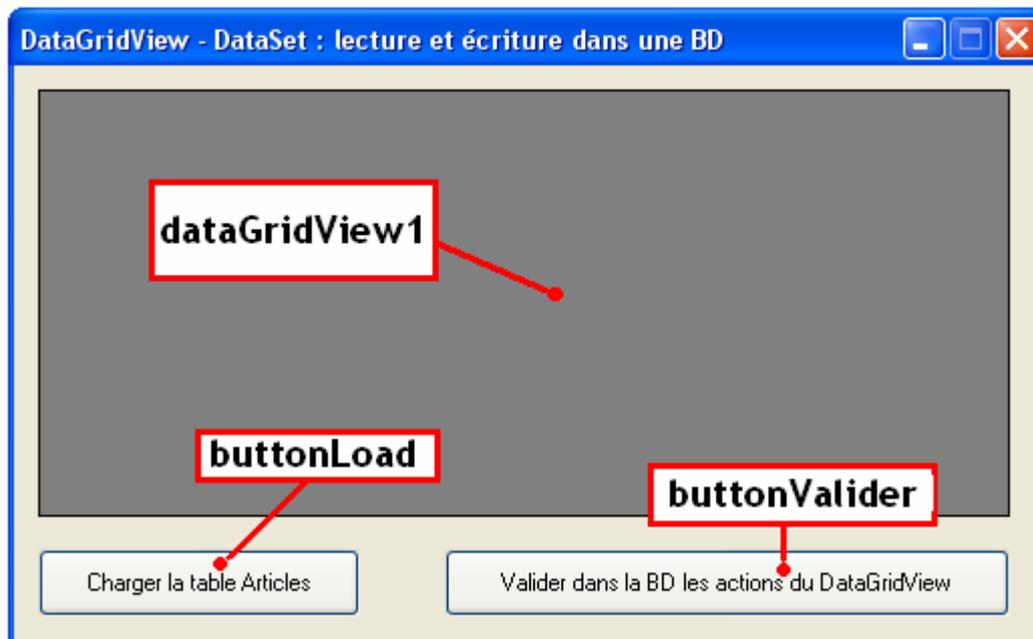
idArticle	genreArticle	quantité	prix
100	cahier	106	1,2000
101	fourchette	57	0,9000
102	cahier	120	2,0000
103	serviette	28	1,4000
104	verre	42	8,0000
105	assiette	108	2,0000
106	chemise	420	8,5000
107	couteau	83	1,1000
108	pantalon	27	45,0000
109	ski	25	12,5000
110	craie	1249	0,0400
111	savon	1000	100,0000
*	NULL	NULL	NULL

Un DataSet charge la table "Articles" de la BD SQL serveur nommée "Boutique.mdf" à travers un objet de communication SqlDataAdapter :



On se propose d'écrire une IHM simple contenant :

- un DataGridView nommé `dataGridView1` affichant et autorisant les modifications de cellules,
- un bouton `buttonLoad`, de chargement de la table "Articles" à partir de la BD (à travers un DataSet lui-même connectée à la BD par un SqlDataAdapter),
- un bouton `buttonValider`, de sauvegarde et mise à jour dans la BD des modifications apportées dans le `dataGridView1`.



```

public partial class FormDatSetBD : Form
{
    public FormDatSetBD()
    {
        InitializeComponent();
    }
    string urlSqlServer = @"Data Source=(local);Initial Catalog=Boutique;" + "Integrated Security=SSPI;";
    DataSet undatSet = new DataSet();
    // objet de connexion SqlConnection :
    SqlConnection connexion;

    //objet de communication et d'échange de données
    SqlDataAdapter dataAdapteur = null;

```

Chargement des données de la table "Articles" dans le dataGridView1 :

```

private void buttonLoad_Click(object sender, EventArgs e)
{
    // initialisation de l'objet de connexion SqlConnection :
    connexion = new SqlConnection(urlSqlServer);

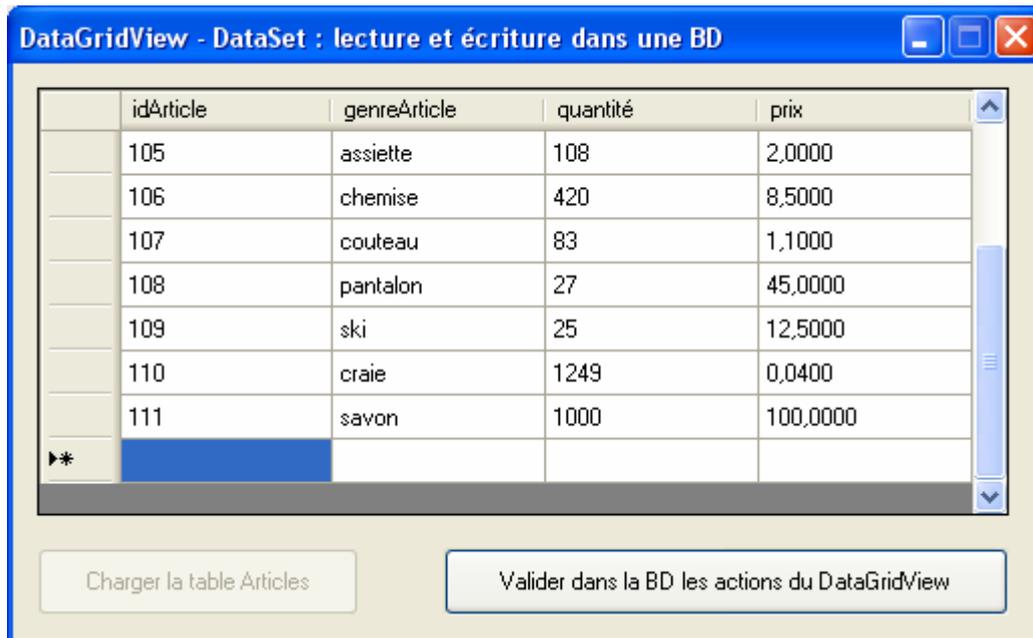
    //objet de communication et d'échange branché sur la table Articles
    dataAdapteur = new SqlDataAdapter("Select * From Articles", connexion);
    // conseillé par Microsoft pour accéder à l'information de clef primaire
    dataAdapteur.MissingSchemaAction = MissingSchemaAction.AddWithKey;

    //Céation et remplissage d'une table 'LesArticles' dans le DataSet :
    dataAdapteur.Fill(undatSet, "LesArticles");

    //liaison de données de la table "LesArticles" avec le dataGridView1 :
    dataGridView1.DataSource = undatSet;
    dataGridView1.DataMember = undatSet.Tables[0].TableName;

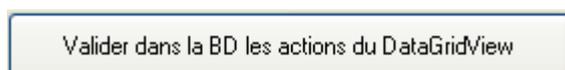
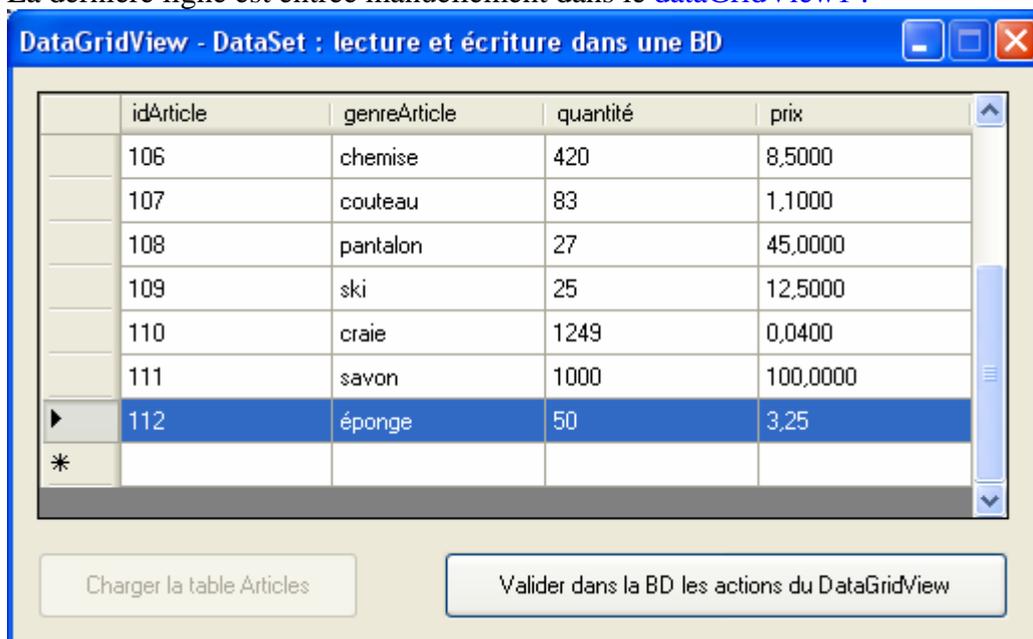
    buttonLoad.Enabled = false;
    buttonValider.Enabled = true;
}

```



Modification par envoi automatique d'une commande Transact-SQL :

La dernière ligne est entrée manuellement dans le `dataGridView1` :



//ce bouton valide les données dans la BD

```
private void buttonValider_Click(object sender, EventArgs e)
{
    // construction et lancement de la commande Transact-SQL insert, update ou delete:
    SqlCommandBuilder builder = new SqlCommandBuilder(dataAdapteur);
    dataAdapteur.Update(undatSet, undatSet.Tables[0].TableName);
    /* validation effective des changements apportés (mettre après la commande Transact-SQL)
```

```

* undatSet.AcceptChanges(); est automatiquement lancé par la méthode Update du dataAdapteur.
* */
buttonLoad.Enabled = true;
}

```

La méthode Update lance une commande SQL de type INSERT, après cette validation, la table "Articles" de la BD contient physiquement la nouvelle ligne :

	idArticle	genreArticle	quantité	prix
	100	cahier	106	1,2000
	101	fourchette	57	0,9000
	102	cahier	120	2,0000
	103	serviette	28	1,4000
	104	verre	42	8,0000
	105	assiette	108	2,0000
	106	chemise	420	8,5000
	107	couteau	83	1,1000
	108	pantalon	27	45,0000
	109	ski	25	12,5000
	110	craie	1249	0,0400
	111	savon	1000	100,0000
	112	éponge	50	3,2500
*	NULL	NULL	NULL	NULL

Supprimons l'article craie dans le DataGridView :

	idArticle	genreArticle	quantité	prix
	106	chemise	420	8,5000
	107	couteau	83	1,1000
	108	pantalon	27	45,0000
	109	ski	25	12,5000
	110	craie	1249	0,0400
	111	savon	1000	100,0000
	112	éponge	50	3,2500
*				

Ensuite nous cliquons sur le bouton Valider, la méthode Update lance alors une commande SQL du type DELETE :

```

SqlCommandBuilder builder = new SqlCommandBuilder(dataAdapteur);
dataAdapteur.Update(undatSet, undatSet.Tables[0].TableName);

```

Voici le nouvel état de la BD après cette validation :

Table - dbo.Articles		Summary		
	idArticle	genreArticle	quantité	prix
	100	cahier	106	1,2000
	101	fourchette	57	0,9000
	102	cahier	120	2,0000
	103	serviette	28	1,4000
	104	verre	42	8,0000
	105	assiette	108	2,0000
	106	chemise	420	8,5000
	107	couteau	83	1,1000
	108	pantalon	27	45,0000
▶	109	ski	25	12,5000
	111	savon	1000	100,0000
	112	éponge	50	3,2500
*	NULL	NULL	NULL	NULL

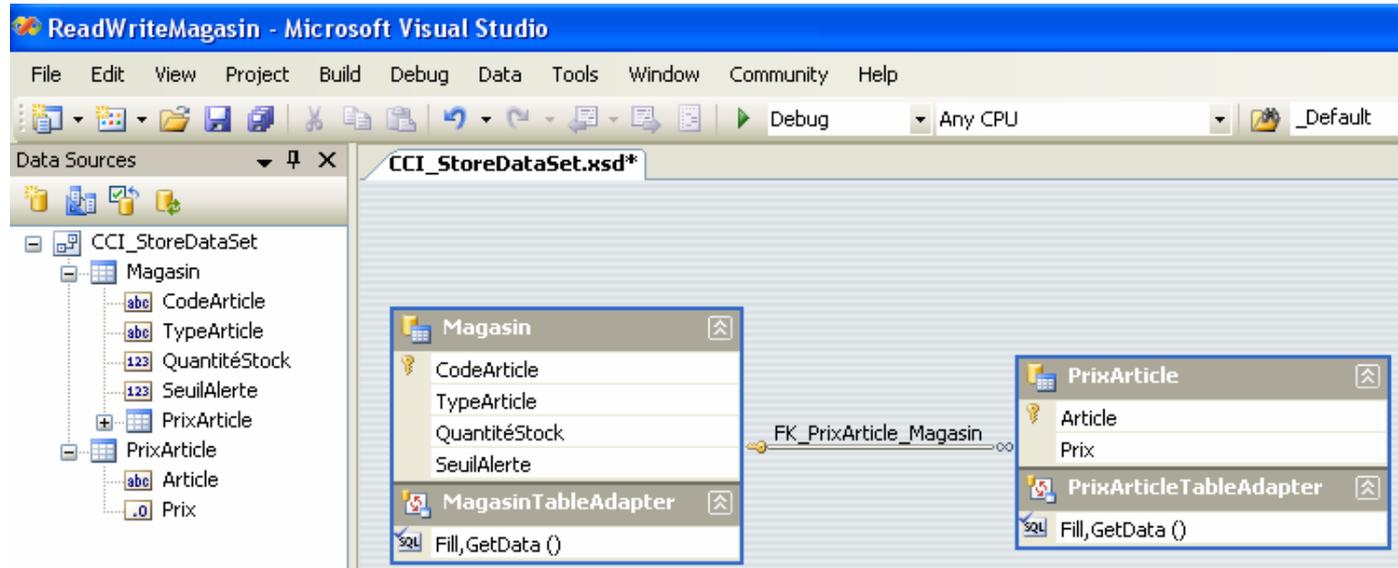
La ligne 110, craie, ... a bien été effacée de la BD.

En conclusion :

Toute modification (insertion d'une ou plusieurs lignes, effacement d'une ou plusieurs lignes, changement des données incluses dans ou plusieurs cellules) est notifiée par le [DataGridView](#) au [DataSet](#) qui l'envoie au [SQLDataAdapter](#) qui lui-même génère les commandes [Transact-SQL](#) adéquates vers la BD afin que les mises à jours soient effectives.

Exercice : Gestion simplifiée d'un petit magasin

Soit une BD SQL serveur 2005 de gestion de stock nommée "CCI_Store.mdf" et comportant 2 tables liées entre elles :

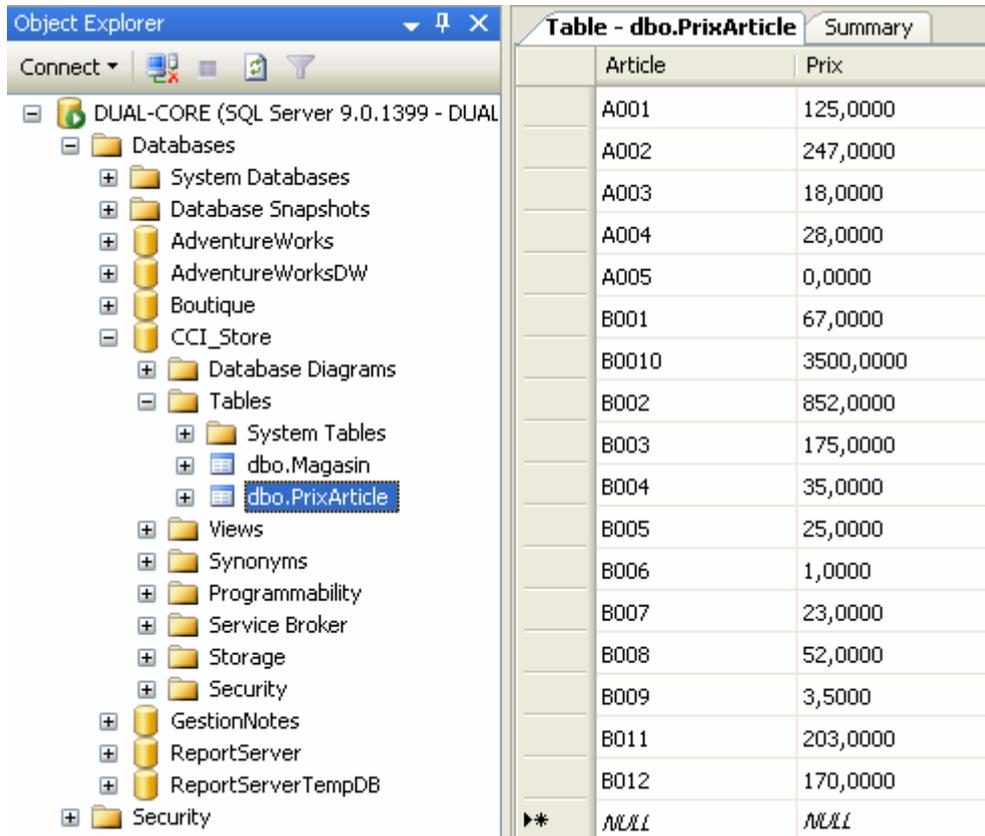


La table « Magasin » :

Object Explorer		Table - dbo.Magasin			
		Summary			
	CodeArticle	TypeArticle	QuantitéStock	SeuilAlerte	
▶	A001	vélo	250	15	
	A002	ski	90	5	
	A003	clavier	200	20	
	A004	chemise	47	22	
	A005	aaa	257	200	
	B001	chaise	478	12	
	B0010	piano	10	2	
	B002	home cinéma	50	5	
	B003	hi-fi	70	12	
	B004	pantalon	128	25	
	B005	chemise	256	24	
	B006	cahier	54	20	
	B007	reveil	52	13	
	B008	lecteur dvd	95	20	
	B009	cendrier	86	14	
	B011	écran	25	5	
	B012	disque 300Go	30	6	
*	NULL	NULL	NULL	NULL	

La colonne **CodeArticle** de la table **Magasin** est liée à la colonne **Article** de la table **PrixArticle**, c'est une **clef étrangère** de la table **Magasin** qui réfère à la table **PrixArticle**

La table « **PrixArticle** » :



Article	Prix
A001	125,0000
A002	247,0000
A003	18,0000
A004	28,0000
A005	0,0000
B001	67,0000
B0010	3500,0000
B002	852,0000
B003	175,0000
B004	35,0000
B005	25,0000
B006	1,0000
B007	23,0000
B008	52,0000
B009	3,5000
B011	203,0000
B012	170,0000
▶*	NULL

Programmation en « mode déconnecté » :

Implémentons en C#, les actions suivantes sur la BD "**CCI_Store.mdf**" :

- 1°) Le contenu de n'importe laquelle des cellules peut être modifié par l'utilisateur.
- 2°) Dès que le contenu d'une cellule est modifié la BD est mise-à-jour immédiatement.
- 3°) Dès que l'on clique dans une cellule, le contenu de la table est rafraîchi à partir de celui de la BD.
- 4°) L'application permet d'obtenir deux fichiers XML à partir de la BD :
 - 4.1°) La totalité de la BD au format XML.
 - 4.2°) Le schéma de la BD au format XSD.

On utilise un DataSet.

L'IHM de la gestion de stock affiche les deux tables et du XML obtenu à partir de la BD :

Table : Magasin

	CodeArticle	TypeArticle	QuantitéStock	SeuilAlerte
	A001	vélo	250	15
	A002	ski	90	5
▶	A003	ballon	200	20
	B001	téléviseur	85	10
	B002	home cinéma	50	5
	B003	hi-fi	70	12
	B004	lecteur dvd	100	20
	B005	caméscope	50	5

Table : PrixArticle

	Article	Prix
▶	A001	125,0000
	A002	247,0000
	A003	18,0000
	B001	256,0000
	B002	852,0000
	B003	175,0000
	B004	198,0000
	B005	720,0000

Schéma XML du DataSet Contenu de la BD au format XML

```
<?xml version="1.0" encoding="utf-16"?>
<xs:schema id="NewDataSetCCI" xmlns="" xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns:msdata="urn:schemas-microsoft-com:xml-msdata">
  <xs:element name="NewDataSetCCI" msdata:IsDataSet="true" msdata:UseCurrentLocale="true">
    <xs:complexType>
      <xs:choice minOccurs="0" maxOccurs="unbounded">
        <xs:element name="magasin">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="CodeArticle">
                <xs:simpleType>
                  <xs:restriction base="xs:string">
                    <xs:maxLength value="10" />
                  </xs:restriction>
                </xs:simpleType>
              </xs:element>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:choice>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Une solution

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
using System.Data.SqlClient;
using System.Data.OleDb;

namespace ReadWriteMagasin
{
    public partial class FStock : Form
    {
        private System.Data.DataSet dataSetStoreCCI;
        private string urlSqlServer = null;
        private SqlConnection connexion = null;
        private SqlDataAdapter magasinSqlAdapter = null;
        private SqlDataAdapter prixSqlAdapter = null;

        public FStock()
        {
            InitializeComponent();
            dataSetStoreCCI = new DataSet();//new CCI_StoreDataSet();
            dataSetStoreCCI.DataSetName = "NewDataSetCCI";
        }

        private DataTable loadTable(string nomTable, out SqlDataAdapter dataAdapteur)
        {
            DataTable table = new DataTable();
            //objet de communication et d'échange de données entre DataTable et la source de données (ici la BD)
            dataAdapteur = new SqlDataAdapter("Select * From " + nomTable, connexion);
            dataAdapteur.MissingSchemaAction = MissingSchemaAction.AddWithKey;

            //Remplissage de la table 'nomTable' :
            dataAdapteur.Fill(table);
            table.TableName = nomTable;
            return table;
        }

        private void displayTableMagasin()
        {
            //Remplissage du DataTable avec la table Magasin :
            DataTable table = loadTable("magasin", out magasinSqlAdapter);
            magasinSqlAdapter.Fill(table);

            //visualisation de la table "magasin" dans le dataGridViewMagasin :
            dataGridViewMagasin.DataSource = table;
            dataSetStoreCCI.Merge(table);
            dataSetStoreCCI.WriteXml("dsStoreMagasin.xml");
        }
    }
}
```

```

private void displayTablePrixArticle()
{
    //Remplissage du DataTable avec la table PrixArticle :
    DataTable table = loadTable("PrixArticle", out prixSqlAdapter);
    prixSqlAdapter.Fill(table);

    //visualisation de la table "PrixArticle" dans le dataGridViewPrix :
    dataGridViewPrix.DataSource = table;
    dataSetStoreCCI.Merge(table);
    dataSetStoreCCI.WriteXml("dsStorePrix.xml");
}

private void boutonCharger_Click(object sender, EventArgs e)
{
    try
    {
        //--plusieurs chaînes de chemin de connexion possibles équivalents pour SQL server 2005 :
        //-- (le serveur de l'auteur se nomme DUAL-CORE et se trouve à l'adresse réseau 10.5.8.1) :
        //string urlSqlServer = @"Data Source=(local);Initial Catalog=CCI_Store;" + "Integrated Security=SSPI;";
        //string urlSqlServer = @"Data Source=localhost;Initial Catalog=CCI_Store;" + "Integrated Security=SSPI;";
        //string urlSqlServer = @"Data Source=127.0.0.1;Initial Catalog=CCI_Store;" + "Integrated Security=SSPI;";
        //string urlSqlServer = @"Data Source=10.5.8.1;Initial Catalog=CCI_Store;" + "Integrated Security=SSPI;";
        //string urlSqlServer = @"Data Source=DUAL-CORE;Initial Catalog=CCI_Store;" + "Integrated Security=SSPI;";
        urlSqlServer = @"Data Source=127.0.0.1;Initial Catalog=CCI_Store;" + "Integrated Security=SSPI;";

        // objet de connexion SqlConnection :
        connexion = new SqlConnection(urlSqlServer);

        displayTableMagasin();
        displayTablePrixArticle();

        // visualisation du schéma XML du dataSetStoreCCI
        toolStripButtonXML.Enabled = true;
        richTextBoxSchemaXML.Text = dataSetStoreCCI.GetXmlSchema();

        //par sécurité : validation effective des changements apportés
        dataSetStoreCCI.AcceptChanges();
    }
    catch (SqlException ex)
    {
        MessageBox.Show(ex.Message, "Erreur SQL", MessageBoxButtons.OK, MessageBoxIcon.Exclamation);
    }
    finally
    {
        if (connexion != null)
            connexion.Close();
    }
}

private void boutonSaveXML_Click(object sender, EventArgs e)
{
    string FileName = "storeCCI.xml";
    dataSetStoreCCI.WriteXml(FileName, XmlWriteMode.IgnoreSchema);
    dataSetStoreCCI.WriteXmlSchema(FileName.Replace(".xml", ".xsl"));
    richTextBoxAllXML.Text = dataSetStoreCCI.GetXml();
}

private void boutonEffacer_Click(object sender, EventArgs e)
{
    dataGridViewMagasin.DataSource = null;
}

```

```

dataGridViewPrix.DataSource = null;
toolStripButtonXML.Enabled = false;
}

/* ----- modifications de données dans la base ----- */

private void dataGridViewMagasin_CellValueChanged(object sender, DataGridViewCellEventArgs e)
{ // modifications dans la table Magasin (sauf nouvelle ligne)
if (e.RowIndex < dataSetStoreCCI.Tables[0].Rows.Count) //si pas nouvelle ligne
{
textBoxNomColMagasin.Text = dataSetStoreCCI.Tables[0].Columns[e.ColumnIndex].ColumnName;
textBoxValColMagasin.Text = Convert.ToString(dataGridViewMagasin.CurrentRow.Cells[e.ColumnIndex].Value);

//modification du dataSetStoreCCI :
dataSetStoreCCI.Tables[0].Rows[e.RowIndex][e.ColumnIndex]=
dataGridViewMagasin[e.ColumnIndex,e.RowIndex].Value;

// construction et lancement de la commande Transact-SQL:
SqlCommandBuilder builder = new SqlCommandBuilder(magasinSqlAdapter);
magasinSqlAdapter.Update(dataSetStoreCCI, "magasin");

// visualiser la commande Transact-SQL:
Console.WriteLine(builder.GetUpdateCommand().CommandText);

//par sécurité : validation effective des changements apportés
dataSetStoreCCI.AcceptChanges();
}
}

private void dataGridViewPrix_CellValueChanged(object sender, DataGridViewCellEventArgs e)
{
if (e.RowIndex < dataSetStoreCCI.Tables[1].Rows.Count)
{
textBoxNomColPrix.Text = dataSetStoreCCI.Tables[1].Columns[e.ColumnIndex].ColumnName;
textBoxValColPrix.Text = Convert.ToString(dataGridViewPrix.CurrentRow.Cells[e.ColumnIndex].Value);

//modification du dataSetStoreCCI :
dataSetStoreCCI.Tables[1].Rows[e.RowIndex][e.ColumnIndex] =
dataGridViewPrix[e.ColumnIndex, e.RowIndex].Value;

// construction et lancement de la commande Transact-SQL:
SqlCommandBuilder builder = new SqlCommandBuilder(prixSqlAdapter);
prixSqlAdapter.Update(dataSetStoreCCI, "PrixArticle");

// visualiser la commande Transact-SQL:
Console.WriteLine(builder.GetUpdateCommand().CommandText);

//par sécurité : validation effective des changements apportés
dataSetStoreCCI.AcceptChanges();
}
}

private void dataGridViewMagasin_CellClick(object sender, DataGridViewCellEventArgs e)
{
// rafraîchissement des données présentes dans la base :
displayTableMagasin();
if (e.RowIndex < dataSetStoreCCI.Tables[0].Rows.Count) //si pas nouvelle ligne
if (e.RowIndex >= 0 & e.ColumnIndex >= 0) //car si ColumnIndex=-1, RowIndex=-1 on click alors dans les en-
têtes
dataGridViewMagasin.CurrentRow.Cells[e.ColumnIndex] =
dataGridViewMagasin.Rows[e.RowIndex].Cells[e.ColumnIndex];
}
}

```

```

}

private void dataGridViewPrix_CellClick(object sender, DataGridViewCellEventArgs e)
{
    // rafraîchissement des données présentes dans la base :
    displayTablePrixArticle();
    if (e.RowIndex < dataSetStoreCCI.Tables[1].Rows.Count) //si pas nouvelle ligne
    if (e.RowIndex >= 0 & e.ColumnIndex >= 0) //car si ColumnIndex=-1, RowIndex=-1 on click alors dans les en-
têtes
        dataGridViewPrix.CurrentRow = dataGridViewPrix.Rows[e.RowIndex];
}
}
}

```

Lors de l'exécution du programme précédent

Tentons de changer la référence B004 en référence B008 déjà existante :

Table : PrixArticle	
Article	Prix
A003	18,0000
B001	256,0000
B002	852,0000
B003	175,0000
B004	198,0000
B005	720,0000
B007	23,0000
B008	12,0000

Table : PrixArticle	
Article	Prix
B002	852,0000
B003	175,0000
B008	198,0000
B005	720,0000
B007	23,0000
B008	12,0000
*	

.Net lance une exception du type **ConstraintException** :

Column "Article" est contrainte à l'unicité, la valeur B008 est déjà présente.

```

private void dataGridViewPrix_CellValueChanged(object sender, DataGridViewCellEventArgs e)
{
    if (e.RowIndex < dataSetStoreCCI.Tables[1].Rows.Count) //si pas nouvelle ligne
    {
        textBoxNomColPrix.Text = dataSetStoreCCI.Tables[1].Columns[e.ColumnIndex].ColumnName;
        textBoxValColPrix.Text = Convert.ToString(dataGridViewPrix.CurrentRow.Cells[e.ColumnIndex].Value);

        //modification du dataSetStoreCCI :
        dataSetStoreCCI.Tables[1].Rows[e.RowIndex][e.ColumnIndex] = dataGridViewPrix.CurrentRow.Cells[e.ColumnIndex].Value;

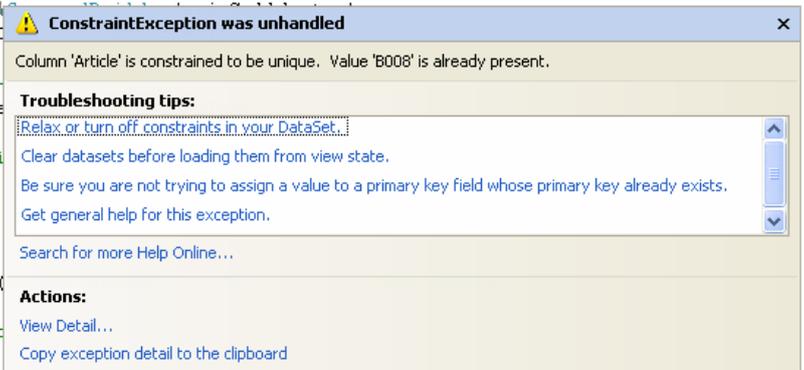
        // construction et lancement de la commande Transact-SQL:
        SqlCommandBuilder builder = new SqlCommandBuilder(prixSqlDataAdapter);
        prixSqlDataAdapter.Update(dataSetStoreCCI);

        // visualiser la commande Transact-SQL:
        Console.WriteLine(builder.GetUpdateCommand().CommandText);

        //par sécurité : validation effective
        dataSetStoreCCI.AcceptChanges();
    }
}

private void dataGridViewMagasin_CellClick(object sender, DataGridViewCellEventArgs e)
{
    // rafraîchissement des données présentes dans la base :
    displayTableMagasin();
    if (e.RowIndex < dataSetStoreCCI.Tables[1].Rows.Count) //si pas nouvelle ligne
    if (e.RowIndex >= 0 & e.ColumnIndex >= 0) //car si ColumnIndex=-1, RowIndex=-1 on click alors dans les en-
têtes
        dataGridViewMagasin.CurrentRow = dataGridViewMagasin.Rows[e.RowIndex];
}
}
}

```



Tentons de créer une nouvelle référence B009 à la place de la référence B004 déjà existante :

Table : PrixArticle		
	Article	Prix
	A003	18,0000
	B001	256,0000
	B002	852,0000
	B003	175,0000
▶	B004	198,0000
	B005	720,0000
	B007	23,0000
	B008	12,0000

Table : PrixArticle		
	Article	Prix
	B002	852,0000
	B003	175,0000
✎	B009	198,0000
	B005	720,0000
	B007	23,0000
	B008	12,0000
*		

.Net lance une exception du type **SQLException** :

UPDATE créé un conflit avec la clef étrangère de nom FK_PrixArticle_Magasin...

```
private void dataGridViewPrix_CellValueChanged(object sender, DataGridViewCellEventArgs e)
{
    if (e.RowIndex < dataSetStoreCCI.Tables[1].Rows.Count) //si pas nouvelle ligne
    {
        textBoxNomColPrix.Text = dataSetStoreCCI.Tables[1].Columns[e.ColumnIndex].ColumnName;
        textBoxValColPrix.Text = dataSetStoreCCI.Tables[1].Rows[e.RowIndex].Cells[e.ColumnIndex].Value;

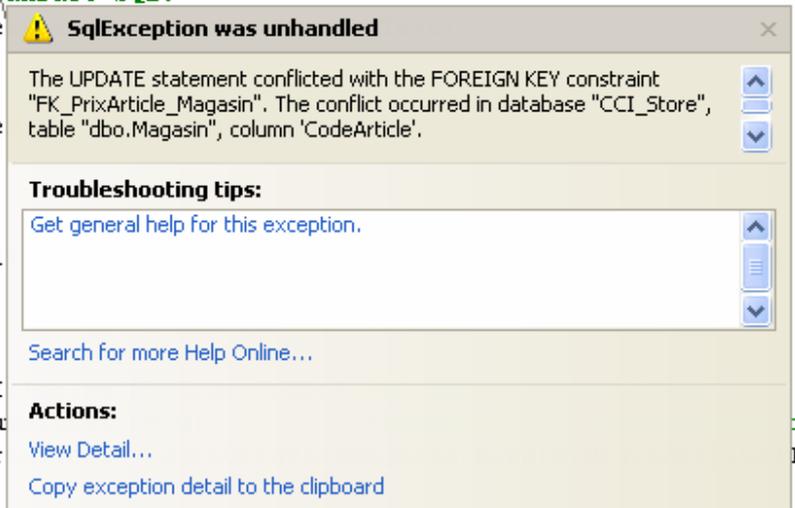
        //modification du dataSetStoreCCI :
        dataSetStoreCCI.Tables[1].Rows[e.RowIndex][e.ColumnIndex] = dataGridViewPrix[e.Co

        // construction et lancement de la commande Transact-SQL:
        SqlCommandBuilder builder = new SqlCommandBuilder(prixSqlAdapter);
        prixSqlAdapter.Update(dataSetStoreCCI, "PrixArticle");

        // visualiser la commande Transact-SQL:
        Console.WriteLine(builder.Ge

        //par sécurité : validation
        dataSetStoreCCI.AcceptChange
    }
}

private void dataGridViewMagasin_Cel
{
    // rafraîchissement des données
    displayTableMagasin();
    if (e.RowIndex < dataSetStoreCCI
        if (e.RowIndex >= 0 & e.Colu
            dataGridViewMagasin.Curr
    }
}
```



Si nous tentons de créer une nouvelle référence B009 à la place de la référence B004 déjà existante, mais cette fois-ci dans la table **Magasin** :

Table : Magasin

	CodeArticle	TypeArticle	QuantitéStock	SeuilAlerte
	A003	ballon	200	20
	B001	téléviseur	32	10
	B002	home cinéma	50	5
	B003	hi-fi	70	12
▶	B004	lecteur dvd	100	20
	B005	caméscope	50	5
	B006	piano	56	8
	B007	chapeau	250	40

.Net lance le même type **SqlException** :

UPDATE créé un conflit avec la clef étrangère de nom FK_PrixArticle_Magasin...

```
private void dataGridViewMagasin_CellValueChanged(object sender, DataGridViewCellEventArgs e)
{
    // modifications dans la table Magasin (sauf nouvelle ligne)
    if (e.RowIndex < dataSetStoreCCI.Tables[0].Rows.Count) //si pas nouvelle ligne
    {
        textBoxNomColMagasin.Text = dataSetStoreCCI.Tables[0].Columns[e.ColumnIndex].ColumnName;
        textBoxValColMagasin.Text = Convert.ToString(dataGridViewMagasin.CurrentCell.Value);

        //modification du dataSetStoreCCI :
        dataSetStoreCCI.Tables[0].Rows[e.RowIndex][e.ColumnIndex] = dataGridViewMagasin[e.Col

        // construction et lancement de la commande Transact-SQL:
        SqlCommandBuilder builder = new SqlCommandBuilder(magasinSqlAdapter);
        magasinSqlAdapter.Update(dataSetStoreCCI, "magasin");

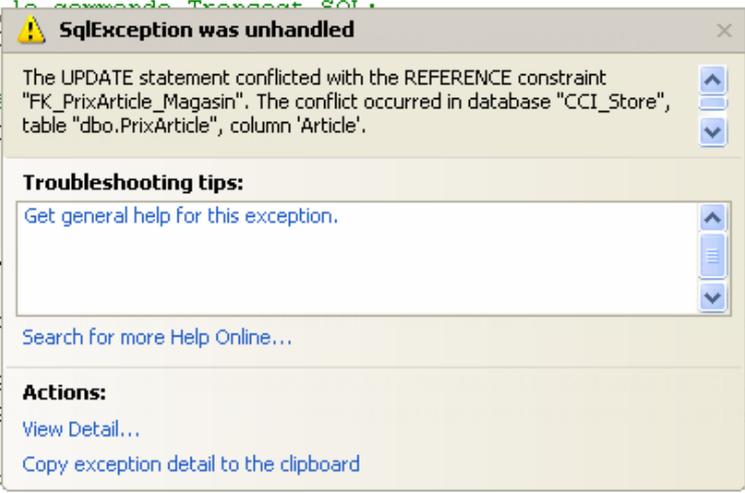
        // visualiser la commande Transact SQL:
        Console.WriteLine

        //par sécurité
        dataSetStoreCCI

    }
}

private void dataGridViewMagasin_CellValueChanged(object sender, DataGridViewCellEventArgs e)
{
    if (e.RowIndex < dataSetStoreCCI.Tables[0].Rows.Count) //si pas nouvelle ligne
    {
        textBoxNomColMagasin.Text = dataSetStoreCCI.Tables[0].Columns[e.ColumnIndex].ColumnName;
        textBoxValColMagasin.Text = Convert.ToString(dataGridViewMagasin.CurrentCell.Value);

        //modification du dataSetStoreCCI :
        dataSetStoreCCI.Tables[0].Rows[e.RowIndex][e.ColumnIndex] = dataGridViewMagasin[e.Col
    }
}
```



La colonne CodeArticle de la table Magasin est liée à la colonne Article de la table PrixArticle, c'est une clef étrangère de la table Magasin qui réfère à la table PrixArticle, dans les deux derniers cas nous n'avons pas respecté

la règle d'intégrité référentielle (*toutes les valeurs d'une clef étrangère doivent se retrouver comme valeur de la clef primaire de la relation référée*) :



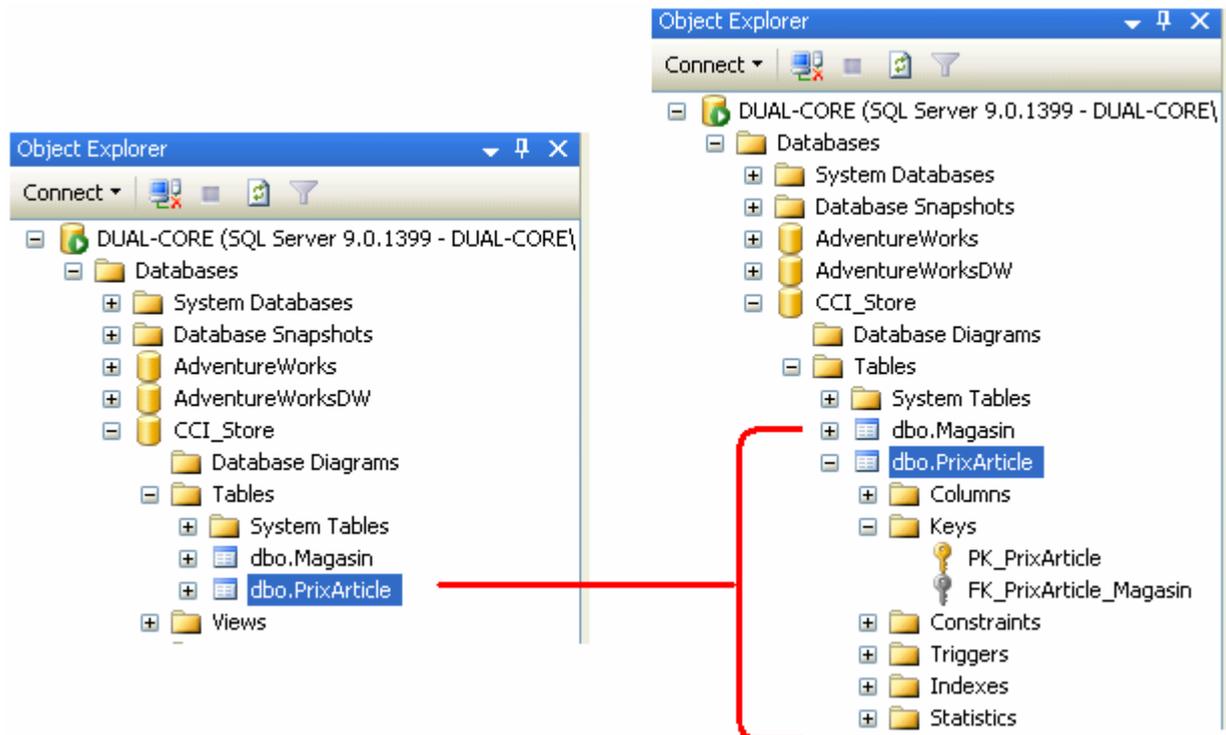
.Net renvoie les exceptions associées aux problèmes d'intégrités.

Amélioration de la gestion du petit magasin : clef étrangère et **delete/update** en cascade

Soit les deux tables de l'exercice sur un petit magasin :

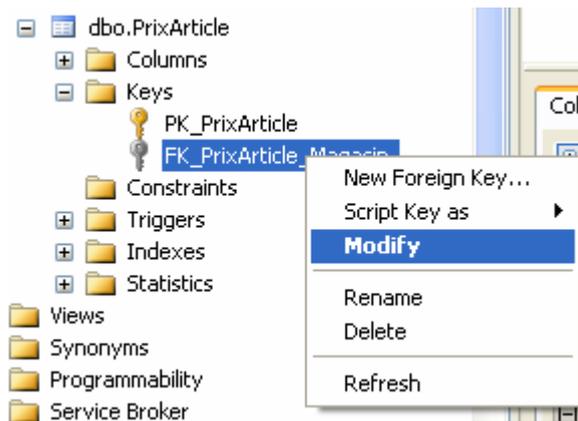


Microsoft SQL Server Management Studio permet de visualiser les deux clefs primaires de chacune des tables, et le fait que FK_PrixArticle_Magasin est une clef étrangère :

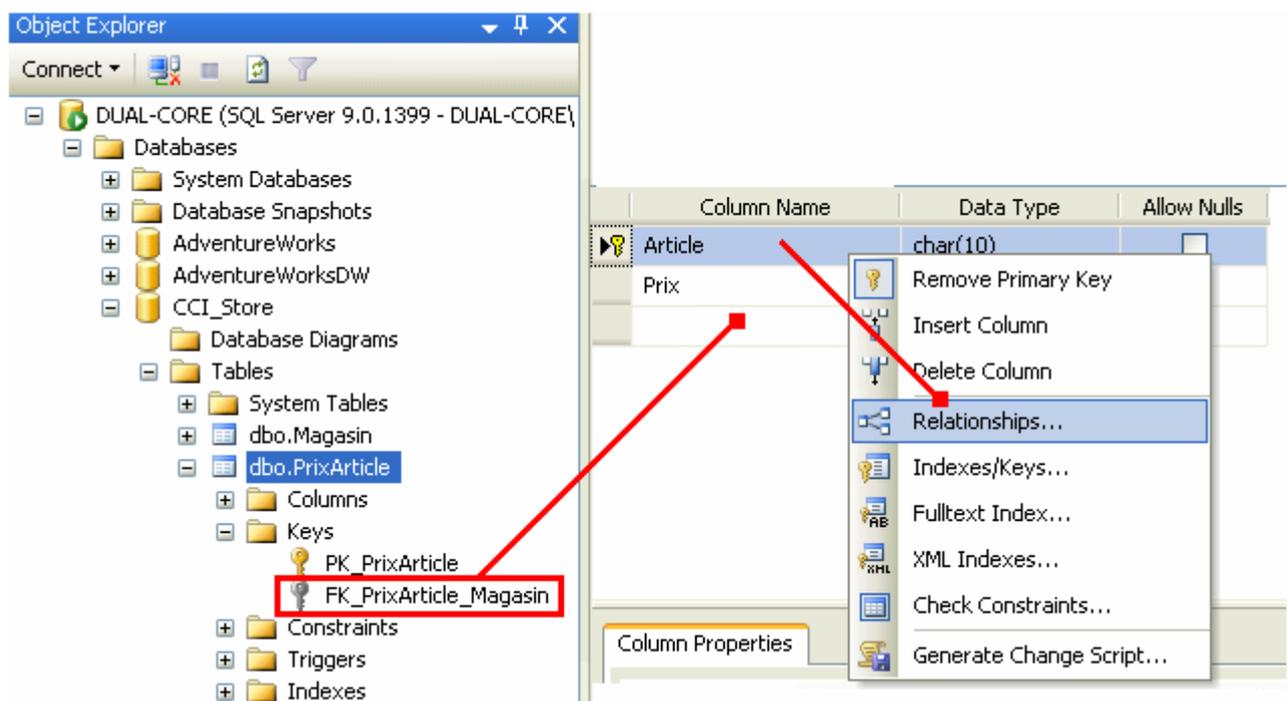


Il est possible avec SQL Server Management Studio et Visual Studio, de programmer visuellement la propagation de la mise à jour automatiquement de la clef étrangère aux relations aux quelles elle réfère (ici à la clef primaire de la table PrixArticle).

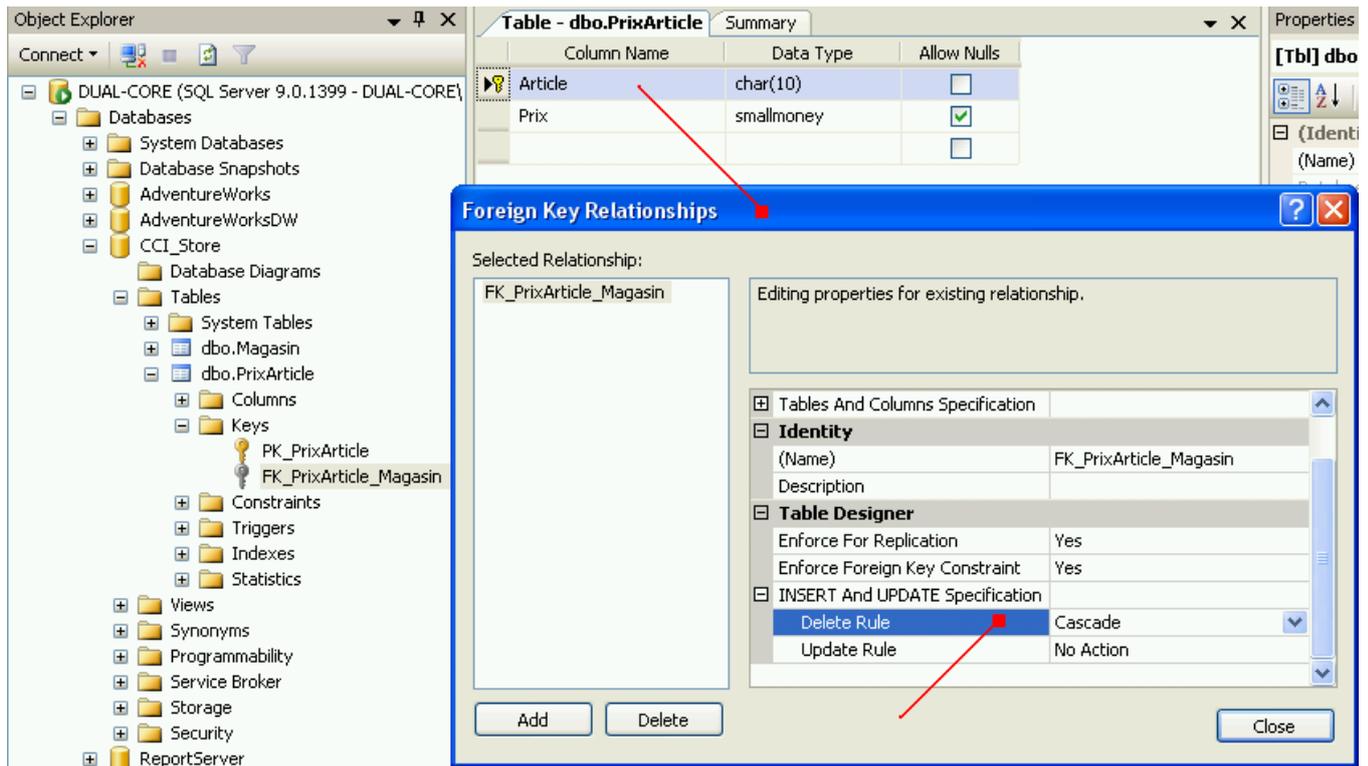
1°) Il faut soit sélectionner la clef étrangère FK_PrixArticle_Magasin et faire apparaître le menu pop-up associé par un click droit de souris et lancer la commande **Modify** :



2°) Soit effectuer la même opération (click droit de souris – commande Modify) sur le nom la table elle-même et ensuite faire apparaître à partir d'un click droit se souris sur la clef primaire la commande Relationship :

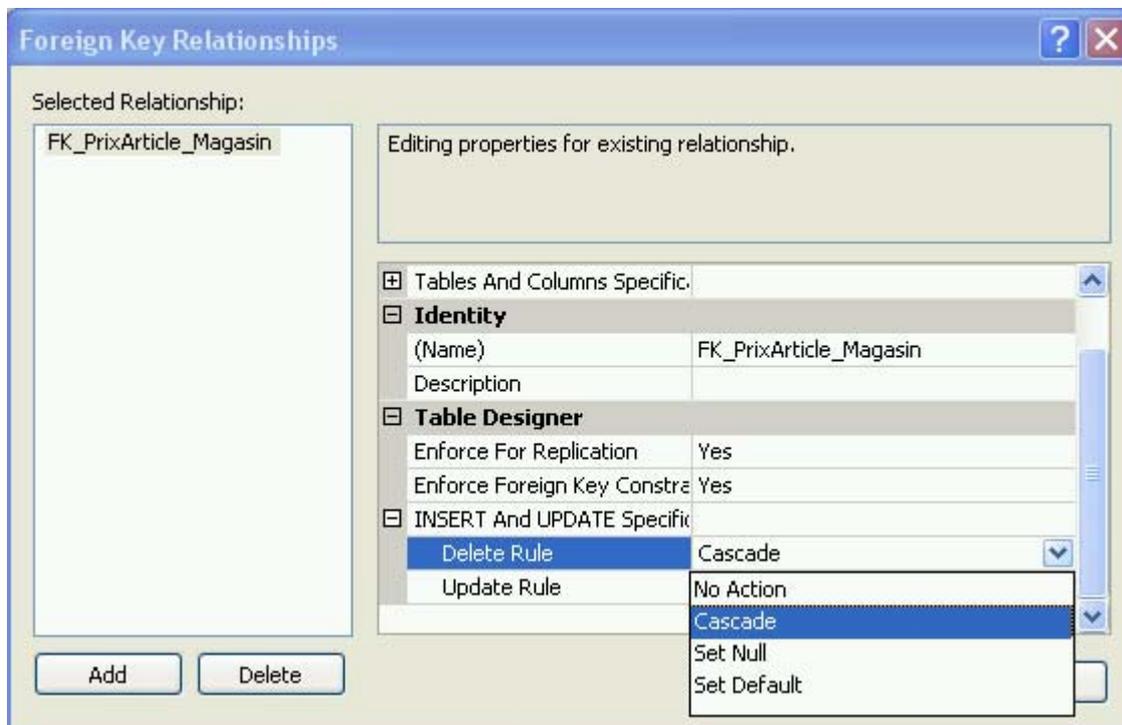


Ces deux opérations conduisent au même résultat : faire apparaître une fenêtre d'édition de propriétés pour les relations actuelles. Nous pouvons alors dans cet éditeur de propriétés, modifier les spécifications des commandes **UPDATE** et **DELETE**.



Pour chacune des commandes **UPDATE** et **DELETE** 4 choix sont possibles (**No Action** est le choix par défaut).

Nous choisissons de propager un **DELETE** dans la table Magasin à la table PrixArticle référencée par la clef étrangère en utilisant le choix **Cascade** :



Dans le programme C#, la commande DELETE dans la table magasin :

```
sendCommandTransact_SQL("supprimer", "magasin");
```

produit un DELETE sur la table Magasin et produit en cascade un DELETE sur la même ligne de la table prixArticle.

Implantation a effectuer

Sécurisez le programme de petit magasin comme suit :

- Chaque modification d'un ou plusieurs champs de la table Magasin est immédiatement et automatiquement validée dans la BD, sauf dans le cas de la création d'une nouvelle ligne où la validation des modifs sera proposée par l'activation d'un bouton.
- En empêchant toute modification du code Article dans la table PrixArticle à partir de l'IHM.
- En autorisant les modifications d'un ou plusieurs champs de la table Magasin, lorsqu'il s'agit du champ CodeArticle autoriser la propagation de la modification à la table PrixArticle.
- En permettant de supprimer avec confirmation, par un click droit de souris, une ligne entière de la table Magasin la propagation de la modification à la table PrixArticle.
- Lors de la création d'une nouvelle entrée dans la table Magasin, prévoir de ne pas mettre à jour le prix de l'article (qui est automatiquement mis à 0.00 €) tant que la ligne entière n'a pas été sauvegardée dans la BD.

The screenshot shows a software window titled "Gestion de stock : mode déconnecté". The interface includes a menu bar with "Charger les tables", "Effacer les tables", and "Convertir la BD au format XML". Below the menu bar, there are two data tables side-by-side.

Table : Magasin

	CodeArticle	TypeArticle	QuantitéStock	SeuilAlerte
	B005	chemise	256	24
	B006	cahier	54	20
	B007	reveil	52	13
	B008	lecteur dvd	100	20
	B009	cendrier	86	14
	B011	écran	13	5
▶	B012			
*				

Table : PrixArticle

	Article	Prix
▶	A001	125,0000
	A002	247,0000
	A003	18,0000
	B001	256,0000
	B0010	3500,0000
	B002	852,0000
	B003	175,0000
	B004	35,0000

At the bottom of the window, there is a form with a "CodeArticle" label and a text box containing "B012". To the right of this text box is a button labeled "Sauver modifs dans la BD".

Une solution de l'implantation

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
using System.Data.SqlClient;

namespace ModifMagasin
{
    public partial class FModifStock : Form
    {
        private System.Data.DataSet dataSetStoreCCI;
        private string urlSqlServer = null;
        private SqlConnection connexion = null;
        private SqlDataAdapter magasinSqlAdapter = null;
        private SqlDataAdapter prixSqlAdapter = null;
        //--pour SQL server Ed.Developpeur 2005 :
        //string urlSqlServer = @"Data Source=(local);Initial Catalog=CCI_Store;" + "Integrated Security=SSPI;";
        //string urlSqlServer = @"Data Source=localhost;Initial Catalog=CCI_Store;" + "Integrated Security=SSPI;";
        //string urlSqlServer = @"Data Source=127.0.0.1;Initial Catalog=CCI_Store;" + "Integrated Security=SSPI;";
        //string urlSqlServer = @"Data Source=10.0.0.1;Initial Catalog=CCI_Store;" + "Integrated Security=SSPI;";
        //string urlSqlServer = @"Data Source=DUAL-CORE;Initial Catalog=CCI_Store;" + "Integrated Security=SSPI;";

        public FModifStock()
        {
            InitializeComponent();
            dataSetStoreCCI = new DataSet();
            dataSetStoreCCI.DataSetName = "NewDataSetCCI";
        }

        private DataTable loadTable(string nomTable, out SqlDataAdapter dataAdapteur)
        {
            DataTable table = new DataTable();
            //objet de communication et d'échange de données entre DataTable et la source de données (ici la BD)
            dataAdapteur = new SqlDataAdapter("Select * From " + nomTable, connexion);
            dataAdapteur.MissingSchemaAction = MissingSchemaAction.AddWithKey;

            //Remplissage de la table 'nomTable' :
            dataAdapteur.Fill(table);
            table.TableName = nomTable;
            return table;
        }

        private void displayTableMagasin()
        {
            try
            {
                urlSqlServer = @"Data Source=127.0.0.1;Initial Catalog=CCI_Store;" + "Integrated Security=SSPI;";
                // objet de connexion SqlConnection :
                connexion = new SqlConnection(urlSqlServer);

                //Remplissage du DataTable avec la table Magasin :
                DataTable table = loadTable("magasin", out magasinSqlAdapter);
                magasinSqlAdapter.Fill(table);

                //visualisation de la table "magasin" dans le dataGridViewMagasin :
                dataGridViewMagasin.DataSource = table;
                dataSetStoreCCI.Merge(table);
                dataSetStoreCCI.WriteXml("dsStoreMagasin.xml");
            }
        }
    }
}
```

```

catch (SqlException ex)
{
    MessageBox.Show(ex.Message, "Erreur SQL", MessageBoxButtons.OK, MessageBoxIcon.Exclamation);
}
finally
{
    if (connexion != null)
        connexion.Close();
}
}

private void displayTablePrixArticle()
{
    try
    {
        urlSqlServer = @"Data Source=127.0.0.1;Initial Catalog=CCI_Store;" + "Integrated Security=SSPI;";
        // objet de connexion SqlConnection :
        connexion = new SqlConnection(urlSqlServer);
        //Remplissage du DataTable avec la table PrixArticle :
        DataTable table = loadTable("PrixArticle", out prixSqlAdapter);
        prixSqlAdapter.Fill(table);

        //visualisation de la table "PrixArticle" dans le dataGridViewPrix :
        dataGridViewPrix.DataSource = table;
        dataGridViewPrix.Columns[0].ReadOnly = true;
        dataSetStoreCCI.Merge(table);
        dataSetStoreCCI.WriteXml("dsStorePrix.xml");
    }
    catch (SqlException ex)
    {
        MessageBox.Show(ex.Message, "Erreur SQL", MessageBoxButtons.OK, MessageBoxIcon.Exclamation);
    }
    finally
    {
        if (connexion != null)
            connexion.Close();
    }
}

private void buttonCharger_Click(object sender, EventArgs e)
{
    displayTableMagasin();
    displayTablePrixArticle();

    // visualisation du schéma XML du dataSetStoreCCI
    toolStripButtonXML.Enabled = true;
    richTextBoxSchemaXML.Text = dataSetStoreCCI.GetXmlSchema();
}

private void buttonSaveXML_Click(object sender, EventArgs e)
{
    string FileName = "storeCCI.xml";
    dataSetStoreCCI.WriteXml(FileName, XmlWriteMode.IgnoreSchema);
    dataSetStoreCCI.WriteXmlSchema(FileName.Replace(".xml", ".xsl"));
    richTextBoxAllXML.Text = dataSetStoreCCI.GetXml();
}

private void buttonEffacer_Click(object sender, EventArgs e)
{
    dataGridViewMagasin.DataSource = null;
    dataGridViewPrix.DataSource = null;
    toolStripButtonXML.Enabled = false;
}

private void sendCommandTransact_SQL(string idCommand, string nomTable)
{
    SqlDataAdapter SqlDataAdapteur;

```

```

if (nomTable == "magasin")
    SqlDataAdapter = magasinSqlAdapter;
else
    SqlDataAdapter = prixSqlAdapter;

// construction et lancement de la commande Transact-SQL insert, update ou delete:
SqlCommandBuilder builder = new SqlCommandBuilder(SqlDataAdapter);
SqlDataAdapter.Update(dataSetStoreCCI, nomTable);

// visualiser la commande Transact-SQL:
switch (idCommand)
{
    case "insérer": Console.WriteLine(builder.GetInsertCommand().CommandText); break;
    case "modifier": Console.WriteLine(builder.GetUpdateCommand().CommandText); break;
    case "supprimer": Console.WriteLine(builder.GetDeleteCommand().CommandText); break;
}
}

/* ----- modifications de données dans la base ----- */

/*
* Lorsque l'on entre une nouvelle ligne à la fin (nouvel article) dans
* la table "magasin", il y a création d'une nouvelle ligne avec la même
* clef article et un prix à 0,00€ dans la table "prixarticle".
*/
private void dataGridViewMagasin_CellValueChanged(object sender, DataGridViewCellEventArgs e)
{
    // modifications dans la table Magasin
    textBoxNomColMagasin.Text = dataSetStoreCCI.Tables[0].Columns[e.ColumnIndex].ColumnName;
    textBoxValColMagasin.Text = Convert.ToString(dataGridViewMagasin.CurrentCell.Value);
    if (e.RowIndex <= dataSetStoreCCI.Tables[0].Rows.Count - 1) // mode modification de données d'une ligne existante
    {
        //modification du dataSetStoreCCI :
        dataSetStoreCCI.Tables[0].Rows[e.RowIndex][e.ColumnIndex] = dataGridViewMagasin[e.ColumnIndex,
e.RowIndex].Value;

        // construction et lancement de la commande Transact-SQL:
        sendCommandTransact_SQL("modifier", "magasin");
        // <=>
        //SqlCommandBuilder builder = new SqlCommandBuilder(magasinSqlAdapter);
        //magasinSqlAdapter.Update(dataSetStoreCCI, "magasin");

        // visualiser la commande Transact-SQL:
        //Console.WriteLine(builder.GetUpdateCommand().CommandText);

        //par sécurité : validation effective des changements apportés(mettre après la commande Transact-SQL)
        dataSetStoreCCI.AcceptChanges();
        if (e.ColumnIndex == 0) //on vient de changer un code Article, alors on réaffiche les données
        {
            dataSetStoreCCI = new DataSet();
            dataSetStoreCCI.DataSetName = "NewDataSetCCI";
            displayTableMagasin();
            displayTablePrixArticle();
        }
    } // sinon création d'une nouvelle ligne en fin de tableau
    else
    {
        //modification de la table magasin du dataSetStoreCCI :
        if (dataSetStoreCCI.Tables[0].Rows.Count < dataGridViewMagasin.RowCount)
        {
            // on crée la ligne dans la table magasin
            DataRow Line = dataSetStoreCCI.Tables[0].NewRow();
            Line[e.ColumnIndex] = dataGridViewMagasin[e.ColumnIndex, e.RowIndex].Value;
            dataSetStoreCCI.Tables[0].Rows.Add(Line);
            // construction et lancement de la commande Transact-SQL:
            sendCommandTransact_SQL("modifier", "magasin");
        }
    }
}

```

```

// on crée la ligne correspondante dans la table PrixArticle
if (dataGridViewMagasin[e.ColumnIndex, e.RowIndex].Value != DBNull.Value)
{
    DataRow newLine = dataSetStoreCCI.Tables[1].NewRow();
    newLine[0] = Line[0];
    newLine[1] = 0;
    dataSetStoreCCI.Tables[1].Rows.Add(newLine);
    // construction et lancement de la commande Transact-SQL:
    sendCommandTransact_SQL("modifier", "PrixArticle");
    displayTablePrixArticle();
}
}
buttonSave.Enabled = true;
dataGridViewPrix.ReadOnly = true;// aucune opération acceptée tant que la validation n'a pas eu lieu
}
}

private void dataGridViewPrix_CellValueChanged(object sender, DataGridViewCellEventArgs e)
{
    textBoxNomColPrix.Text = dataSetStoreCCI.Tables[1].Columns[e.ColumnIndex].ColumnName;
    textBoxValColPrix.Text = Convert.ToString(dataGridViewPrix.CurrentCell.Value);
    if (e.ColumnIndex != 0)
        if ((e.RowIndex < dataSetStoreCCI.Tables[1].Rows.Count - 1)
            | e.RowIndex == dataSetStoreCCI.Tables[1].Rows.Count - 1)// mode modification de données d'une ligne existante
        {
            // modification du dataSetStoreCCI :
            dataSetStoreCCI.Tables[1].Rows[e.RowIndex][e.ColumnIndex] = dataGridViewPrix[e.ColumnIndex,
e.RowIndex].Value;

            // construction et lancement de la commande Transact-SQL:
            sendCommandTransact_SQL("modifier", "PrixArticle");
            // <=>
            //SqlCommandBuilder builder = new SqlCommandBuilder(prixSqlAdapter);
            //prixSqlAdapter.Update(dataSetStoreCCI, "PrixArticle");

            // visualiser la commande Transact-SQL:
            //Console.WriteLine(builder.GetUpdateCommand().CommandText);

            // validation effective des changements apportés (mettre après la commande Transact-SQL)
            dataSetStoreCCI.AcceptChanges();
            displayTablePrixArticle();

        }
        // sinon création d'une nouvelle ligne interdite
    else
    {
        //rien pour l'instant...
    }
}

private void dataGridViewMagasin_MouseDown(object sender, MouseEventArgs e)
{
    // on supprime une ligne sélectionnée par click droit de souris sur cette ligne :
    int nbrRow = dataGridViewMagasin.SelectedRows.Count;
    int numRows = dataGridViewMagasin.CurrentCell.RowIndex;
    if (nbrRow != 0 & e.Button == MouseButtons.Right)
    {
        if (MessageBox.Show(this, "Confirmez-vous la suppression de cet article de la Base ?", "Suppression de la ligne
CodeArticle : "
+ dataGridViewPrix.Rows[numRows].Cells[0].Value + "demandée.",
MessageBoxButtons.YesNo) == DialogResult.Yes)
        {
            //dataSetStoreCCI.Tables[0].Rows.RemoveAt(numRows); // ne produit pas de DELETE du magasinSqlAdapter
            dataSetStoreCCI.Tables[0].Rows[numRows].Delete();
            dataGridViewMagasin.DataSource = dataSetStoreCCI.Tables[0];
            //dataGridViewMagasin.Update();

            // construction et lancement de la commande Transact-SQL:
            /* DELETE-UPDATE en cascade dans la définition de la clef étrangère "FK_PrixArticle_Magasin"

```

```

    * dans la table PrixArticle.(cf ClefEtrangereCascade.doc)
    */
    sendCommandTransact_SQL("supprimer", "magasin");
    // <=>
    //SqlCommandBuilder builder = new SqlCommandBuilder(magasinSqlAdapter);
    //magasinSqlAdapter.Update(dataSetStoreCCI, "magasin");

    // visualiser la commande Transact-SQL:
    //Console.WriteLine(builder.GetUpdateCommand().CommandText);

    // validation effective des changements apportés (mettre après la commande Transact-SQL)
    dataSetStoreCCI.AcceptChanges();
    dataSetStoreCCI = new DataSet();
    dataSetStoreCCI.DataSetName = "NewDataSetCCI";
    displayTableMagasin();
    displayTablePrixArticle();
    }
}

private void boutonSave_Click(object sender, EventArgs e)
{
    // construction et lancement de la commande Transact-SQL insert, update ou delete:
    SqlCommandBuilder builder = new SqlCommandBuilder(magasinSqlAdapter);
    magasinSqlAdapter.Update(dataSetStoreCCI, "magasin");

    builder = new SqlCommandBuilder(prixSqlAdapter);
    prixSqlAdapter.Update(dataSetStoreCCI, "PrixArticle");

    // validation effective des changements apportés (mettre après la commande Transact-SQL)
    dataSetStoreCCI.AcceptChanges();
    displayTableMagasin();
    displayTablePrixArticle();
    boutonSave.Enabled = false;
    dataGridViewPrix.ReadOnly = false;
}

private void FStock_FormClosing(object sender, FormClosingEventArgs e)
{
    //boutonSave_Click(sender, new EventArgs());
}
}

```