

1. Introduction

Les applications des entreprises nécessitent en général l'accès à une base de données ou tout simplement un logiciel permettant de gérer facilement des fonctionnalités d'une base de données. Sachant que la base de données est quelque chose d'extérieure à l'application, c'est à partir de cette dernière (l'application) que la liaison se fera. Toutes les classes permettant de faire cela se trouvent dans le namespace **System.Data**. Communément, on appelle cela de l'**ADO.NET**.

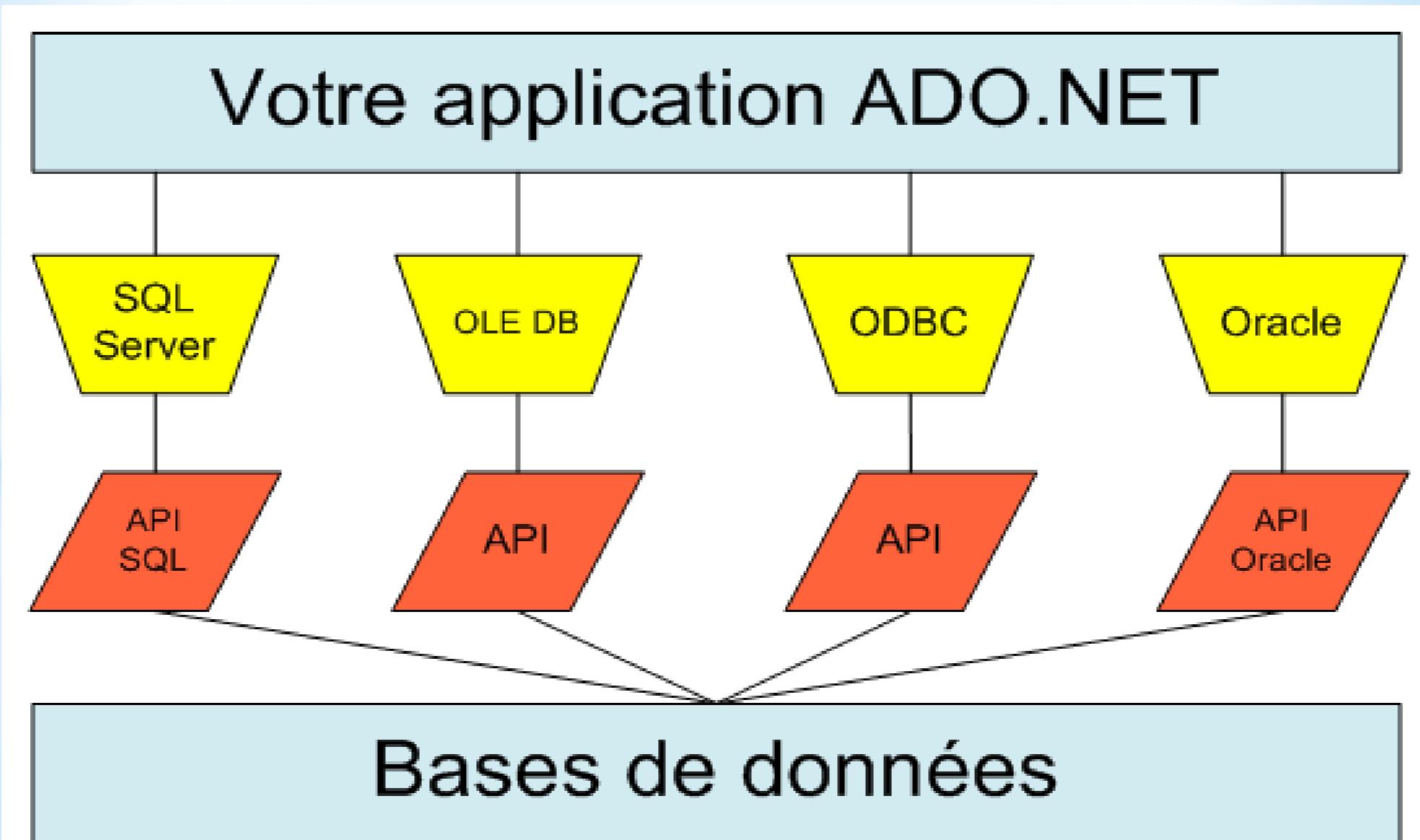
2. Présentation ADO.NET :

2.1 Les fournisseurs de données

Chaque **fournisseur de données** permet la communication avec un type de base de données au travers d'une **API**. Une **API** (Application Programming Interface) est l'interface qui permet l'accès de logiciel par un autre. Ces fournisseurs permettent de récupérer et de transférer des modifications entre l'application et une base de données. Toutes les classes permettant d'utiliser ces **fournisseurs** se trouvent dans l'espace de nom **System.Data**. Sur le Framework, il existe quatre types de fournisseurs :

- Sql Server
- OLE DB
- OBDC
- Oracle

2 . Présentation ADO.NET :
2.1 Les fournisseurs de données



2. Présentation ADO.NET :

2.1 Les fournisseurs de données :

Chaque **fournisseur** est relié à une **base de données propre**, c'est-à-dire qu'il est compatible à l'API de sa **base de données**. Cependant, les bases de données peuvent implémenter **plusieurs API** :

Fournisseur	Description
SQL Server	Les classes de ce fournisseur se trouvent dans l'espace de nom System.Data.SqlClient , chaque nom de ces classes est préfixé par Sql . Il permet SQL Server à accès au serveur sans utiliser d'autres couches logicielles le rendant plus performant.
OLE DB	Les classes de ce fournisseur se trouvent dans l'espace de nom System.Data.OleDb , chaque nom de ces classes est préfixé par OleDb . Ce fournisseur exige l'installation de MDAC (Microsoft Data Access Components). L'avantage de ce fournisseur est qu'il peut dialoguer avec n'importe quelle base de données le temps que le pilote OLE DB est installé dessus, mais par rapport à SQL server, OLE DB utilise une couche logicielle nommée OLE DB ; il requiert donc plus de ressources diminuant par conséquent les performances.
ODBC	Les classes de ce fournisseur se trouvent dans l'espace de nom System.Data.Obdc , chaque nom de ces classes sont préfixés par Obdc . Tout comme l'OLE DB, ODBC exige l'installation de MDAC . Il fonctionne avec le même principe qu'OLE DB mais au lieu d'utiliser une couche logicielle, il utilise le pilote OBDC.
ORACLE	Les classes de ce fournisseur se trouvent dans l'espace de nom System.Data.OracleClient , chaque nom de ces classes est préfixé par Oracle . Il permet simplement de se connecter à une source de données Oracle. (gestion d'accès aux bases Oracle)

2. Présentation ADO.NET :

2.1 Les fournisseurs de données

Ils proposent tous l'implémentation de quatre classes, de base, nécessaires pour le dialogue avec la base de données :

- La classe **Connection** : permet d'établir une connexion avec le serveur de base de données.
- La classe **Command** : permet de demander l'exécution d'une instruction ou d'un ensemble d'instructions SQL à un serveur.
- La classe **DataReader** : procure un accès en lecture seule et un défilement, en avant seulement, aux données.
- La classe **DataAdapter** : est utilisée pour assurer le transfert des données vers un système de cache local à l'application (le **DataSet**) et mettre à jour la base de données, en fonction des modifications effectuées localement dans le **DataSet**. Quelques autres classes sont disponibles pour, par exemple, la gestion des transactions, ou le passage de paramètres à une instruction SQL.

2. Présentation ADO.NET :

2.2 Rechercher les fournisseurs disponibles :

Afin que l'accès aux données soit sûr, les fournisseurs de données doivent être disponibles sur le poste de travail.

La classe **DbProviderFactories** (Espace de nom :**System.Data.Common**) propose la méthode partagée **GetFactoryClasses**, permettant d'énumérer les fournisseurs de données disponibles sur le poste de travail en question.

Rq: L'espace de noms **System.Data.Common** membre du **System.Data** contient des classes partagées par les fournisseurs de données .NET Framework.

Un Fournisseur de données .NET Framework décrit une collection de classes utilisées pour accéder à une source de données, notamment une base de données, dans l'espace managé.

Les classes dans **System.Data.Common** sont conçues pour permettre aux développeurs d'écrire du code ADO.NET qui fonctionnera sur tous les fournisseurs de données .NET Framework.

2. Présentation ADO.NET :

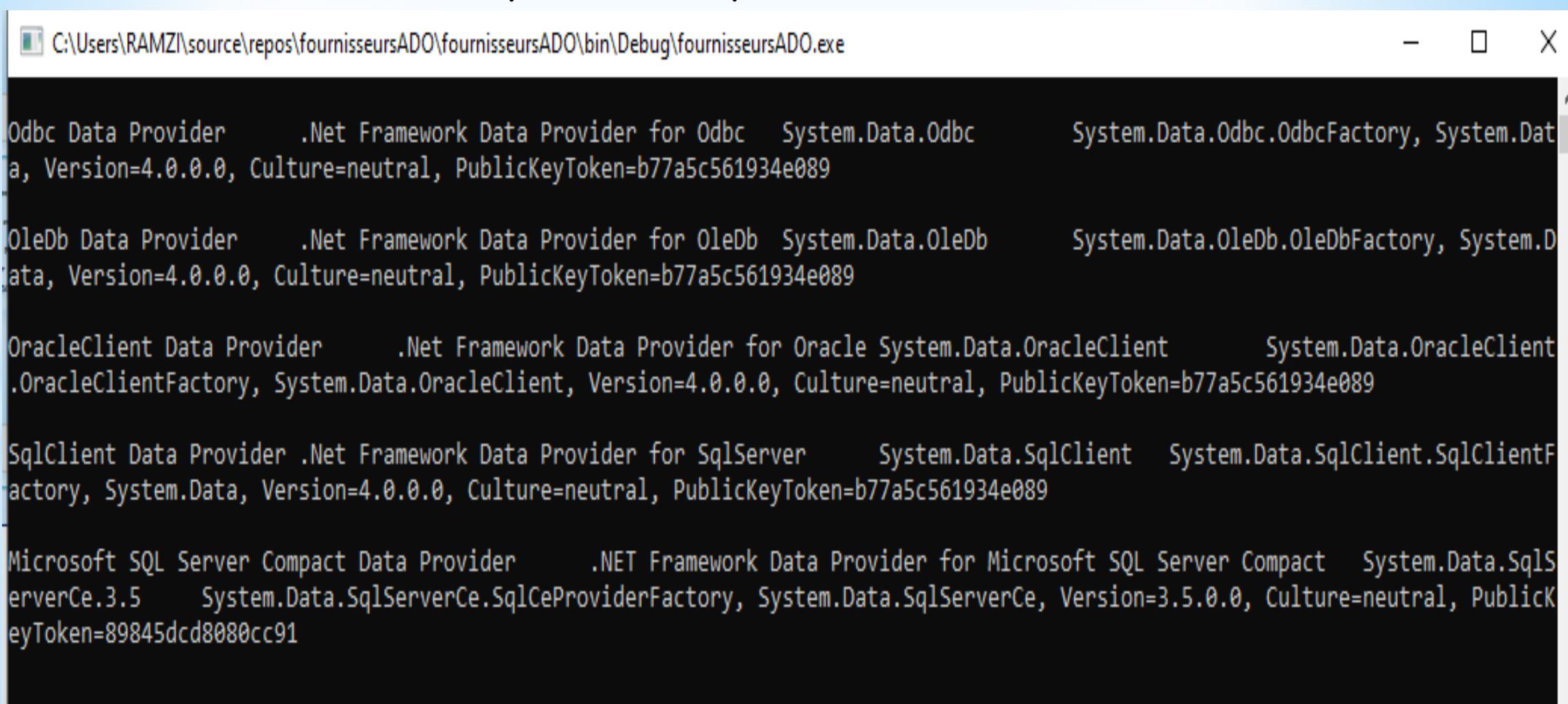
2.2 Rechercher les fournisseurs disponibles : Exemple :

```
using System.Data;
using System.Data.Common;

namespace fournisseursADO
{
    class Program
    {
        static void Main(string[] args)
        {
            //récupération de la liste des fournisseurs dans une datatable
            DataTable listeFournisseurs;
            listeFournisseurs = DbProviderFactories.GetFactoryClasses();
            foreach(DataColumn colonne in listeFournisseurs.Columns)
            {
                //Afficher les noms des colonnes du DataTable
                Console.Write(colonne.ColumnName + "\t");
            }
            Console.Write("\n\n");
            foreach(DataRow ligne in listeFournisseurs.Rows)
            {
                //Affiche chaque ligne
                foreach (DataColumn colonne in listeFournisseurs.Columns)
                {
                    //Affiche les cellules
                    object[] detLigne = ligne.ItemArray;
                    Console.Write(detLigne[colonne.Ordinal]+\t);
                }
                Console.Write("\n\n");
            }
            Console.ReadKey();
        }
    }
}
```

2. Présentation ADO.NET :

2.2 Rechercher les fournisseurs disponibles : Exemple :



```
C:\Users\RAMZI\source\repos\fournisseursADO\fournisseursADO\bin\Debug\fournisseursADO.exe

Odbc Data Provider      .Net Framework Data Provider for Odbc      System.Data.Odbc
      , Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089      System.Data.Odbc.OdbcFactory, System.Data

OleDb Data Provider     .Net Framework Data Provider for OleDb     System.Data.OleDb
      , Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089      System.Data.OleDb.OleDbFactory, System.Data

OracleClient Data Provider .Net Framework Data Provider for Oracle System.Data.OracleClient
      , Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089      System.Data.OracleClient
      .OracleClientFactory, System.Data.OracleClient, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089

SqlClient Data Provider .Net Framework Data Provider for SqlServer   System.Data.SqlClient
      , Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089      System.Data.SqlClient.SqlClientF
      actory, System.Data, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089

Microsoft SQL Server Compact Data Provider .NET Framework Data Provider for Microsoft SQL Server Compact
      , Version=3.5      System.Data.SqlServerCe.SqlCeProviderFactory, System.Data.SqlServerCe, Version=3.5.0.0, Culture=neutral, PublicKeyToken=89845dcd8080cc91      System.Data.SqlS
      erverCe.3.5
```

TP 1 : Contrôles avancés

Trouvez la liste des fournisseurs disponibles sur Votre Poste de travail

3. Architecture ADO.NET :

3.1 Présentation ADO.Net :

■ **ADO.Net** est une technologie, intégrée au **Framework .Net**, offrant des méthodes d'accès aux données. Elle dérive de la technologie **ADO.Net**, qui lui-même succéda à **ADO**. Cette bibliothèque de classes permet de récupérer, manipuler, ou mettre à jour des données provenant d'une source de données (base de données, fichier XML, etc...)

■ La grande révolution d'**ADO.Net** est qu'elle repose sur un modèle d'accès aux données **déconnectés par défaut**. Contrairement aux technologies précédentes tel que **ADO**, qui accédaient aux données de façon **connectés en continu**, trop coûteuse en **ressource** et peut recommandable pour des applications complexes.

■ **ADO.Net** peut se « découper » en 2 groupes principaux qui sont :

- Le groupe de données (**objets déconnectés**): stockant les données sur la machine locale
- Le fournisseur (**objets connectés**): gérant la communication entre le programme et la base de données

3. Architecture ADO.NET :

3.1 Présentation ADO.Net :

Deux composants d'ADO.NET permettent d'accéder à des données et de les manipuler :

- Fournisseurs de données .NET Framework
- Objet DataSet

□ Fournisseurs de données .NET Framework : Les fournisseurs de données .NET Framework sont des composants explicitement conçus pour la manipulation des données et un accès aux données rapide, **avant uniquement et en lecture seule**. L'objet **Connection** assure la connectivité avec une source de données. L'objet **Command** permet l'accès aux commandes de base de données pour **retourner des données, modifier des données, exécuter des procédures stockées et envoyer ou extraire des informations sur les paramètres**.

□ Le **DataReader** fournit un flux très performant de données en provenance de la source de données. Enfin, l'objet **DataAdapter** établit une passerelle entre l'objet **DataSet** et la source de données. Le **DataAdapter** utilise les objets **Command** pour exécuter des commandes SQL au niveau de la source de données afin d'une part de charger le **DataSet** de données, et d'autre part de répercuter dans la source de données les modifications apportées aux données contenues dans le **DataSet**.

3. Architecture ADO.NET :

3.1 Présentation ADO.Net 2.0 :

■ **DataSet** : Le **DataSet** ADO.NET est explicitement conçu pour un accès aux données **indépendant de toute source de données**. Il peut donc être utilisé avec plusieurs sources de données différentes, utilisé avec des données XML ou utilisé pour gérer des données locales de l'application. Le **DataSet** contient une collection d'un ou de plusieurs objets **DataTable** constitués de **lignes** et de **colonnes** de données, ainsi que des informations concernant les **contraintes de clé primaire**, de **clé étrangère** et des **informations relationnelles** sur les données contenues dans les objets **DataTable**.

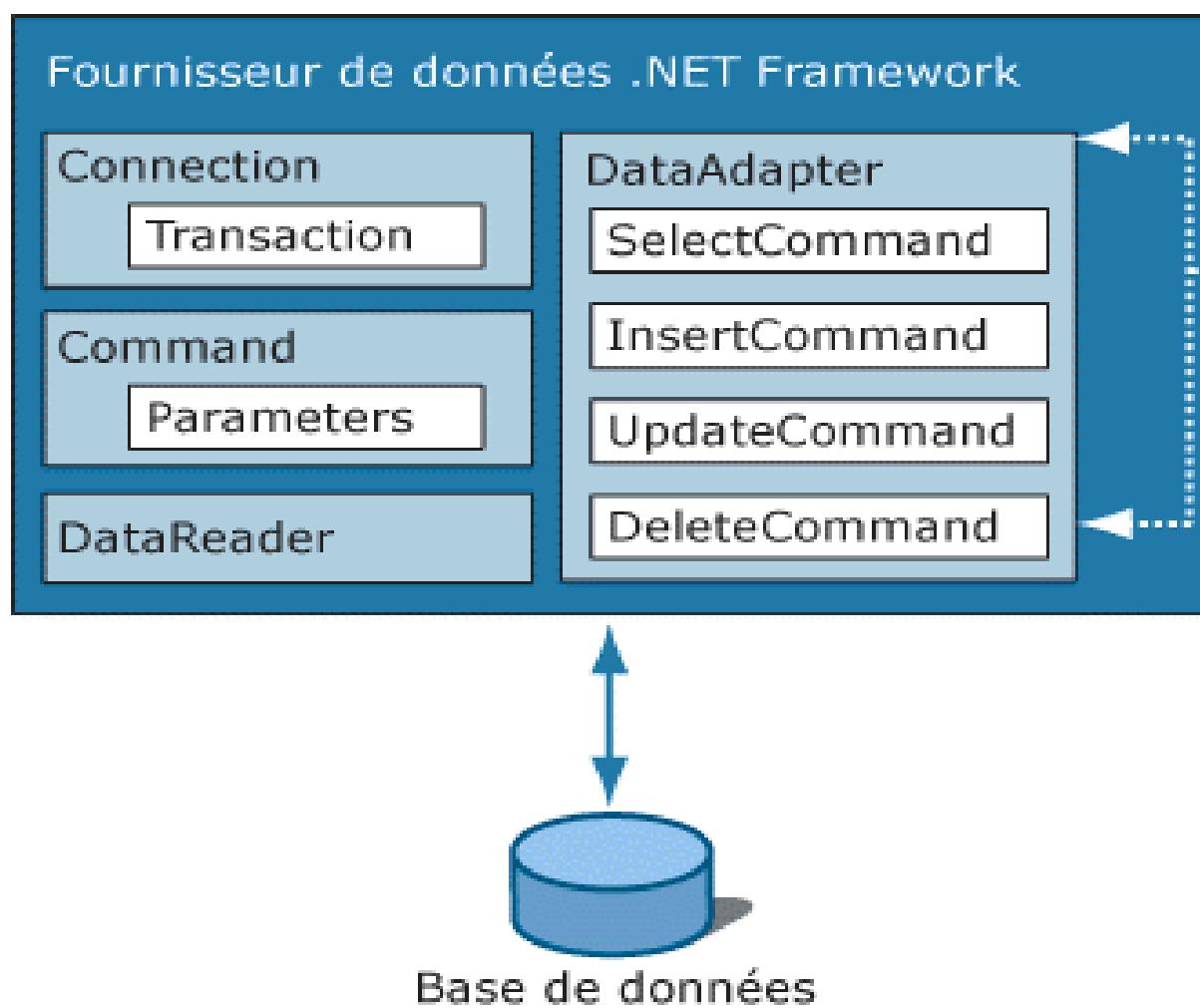
Le diagramme suivant illustre la relation entre un fournisseur de données .NET Framework et un **DataSet**.

3. Architecture ADO.NET :

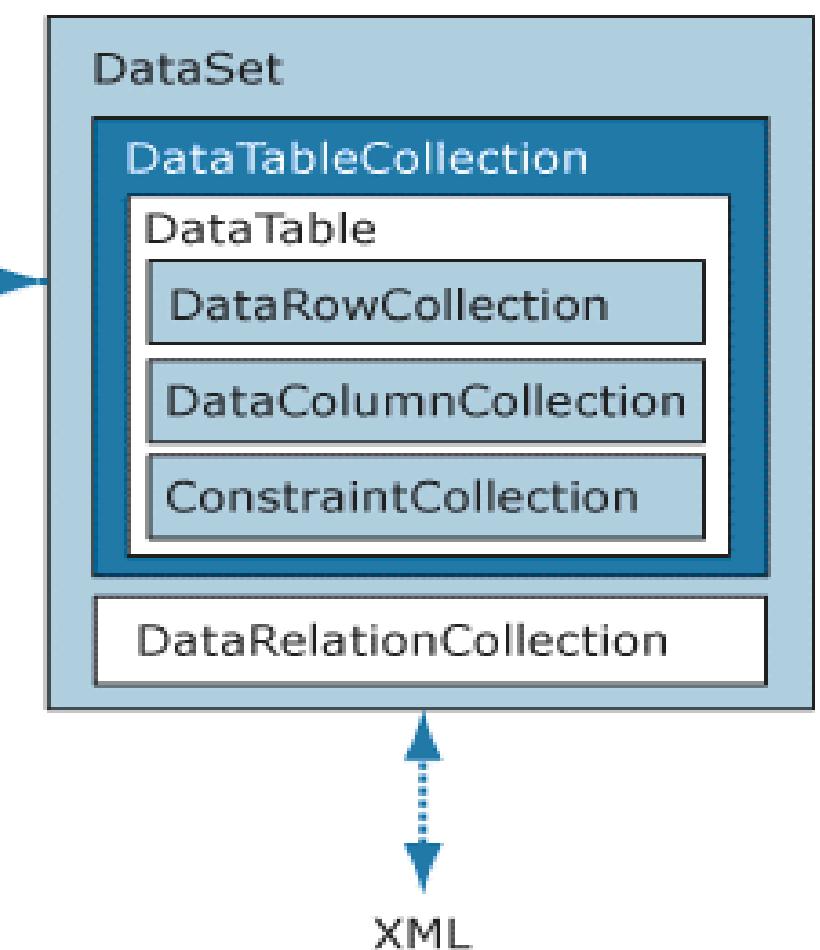
3.1 Présentation ADO.Net 2.0 :

Voici ci-dessous le schéma de l'architecture ADO.Net :

Objets connectés



Objets déconnectés



3. Architecture ADO.NET :

3.1 Présentation ADO.Net :

■ Mode connecté / Mode déconnecté :

L'ADO.NET permet de séparer les actions d'accès ou de modification d'une base de données. En effet, il est possible de manipuler une base de données sans être connecté à celle-ci, il suffit juste de se connecter pendant un court laps de temps afin de faire une mise à jour. Ceci est possible grâce au **DataSet**. C'est pourquoi, il existe deux types de fonctionnements :

- Le mode connecté
- Le mode déconnecté (Mode utilisé par Défaut)

Ci-après, la différence par avantages et inconvénients :

	Avantages	Inconvénients
Mode connecté	la connexion est permanente, les données sont toujours à jour. De plus il est facile de voir quels sont les utilisateurs connectés et sur quoi ils travaillent. Enfin, la gestion est simple, il y a connexion au début de l'application puis déconnexion à la fin.	surtout au niveau des ressources. En effet, tous les utilisateurs ont une connexion permanente avec le serveur. Même si l'utilisateur n'y fait rien la connexion gaspille beaucoup de ressource entraînant aussi des problèmes d'accès au réseau.
Mode déconnecté	possible de brancher un nombre important d'utilisateurs sur le même serveur. En effet, ils se connectent le moins souvent et durant la plus courte durée possible. De plus, avec cet environnement déconnecté, l'application gagne en performance par la disponibilité des ressources pour les connexions.	Les données ne sont pas toujours à jour, ce qui peut aussi entraîner des conflits lors des mises à jour. Il faut aussi penser à prévoir du code pour savoir ce que va faire l'utilisateur en cas de conflits.

3. Architecture ADO.NET :

3.1 Présentation ADO.Net :

■ Choix d'un DataReader (Mode connecté) / DataSet(Mode déconnecté):

Au moment de décider si votre application doit utiliser un **DataReader** ou un **DataSet**, pensez au type de fonctionnalité requis par votre application.

■ Utilisez un **DataSet** pour effectuer les opérations suivantes :

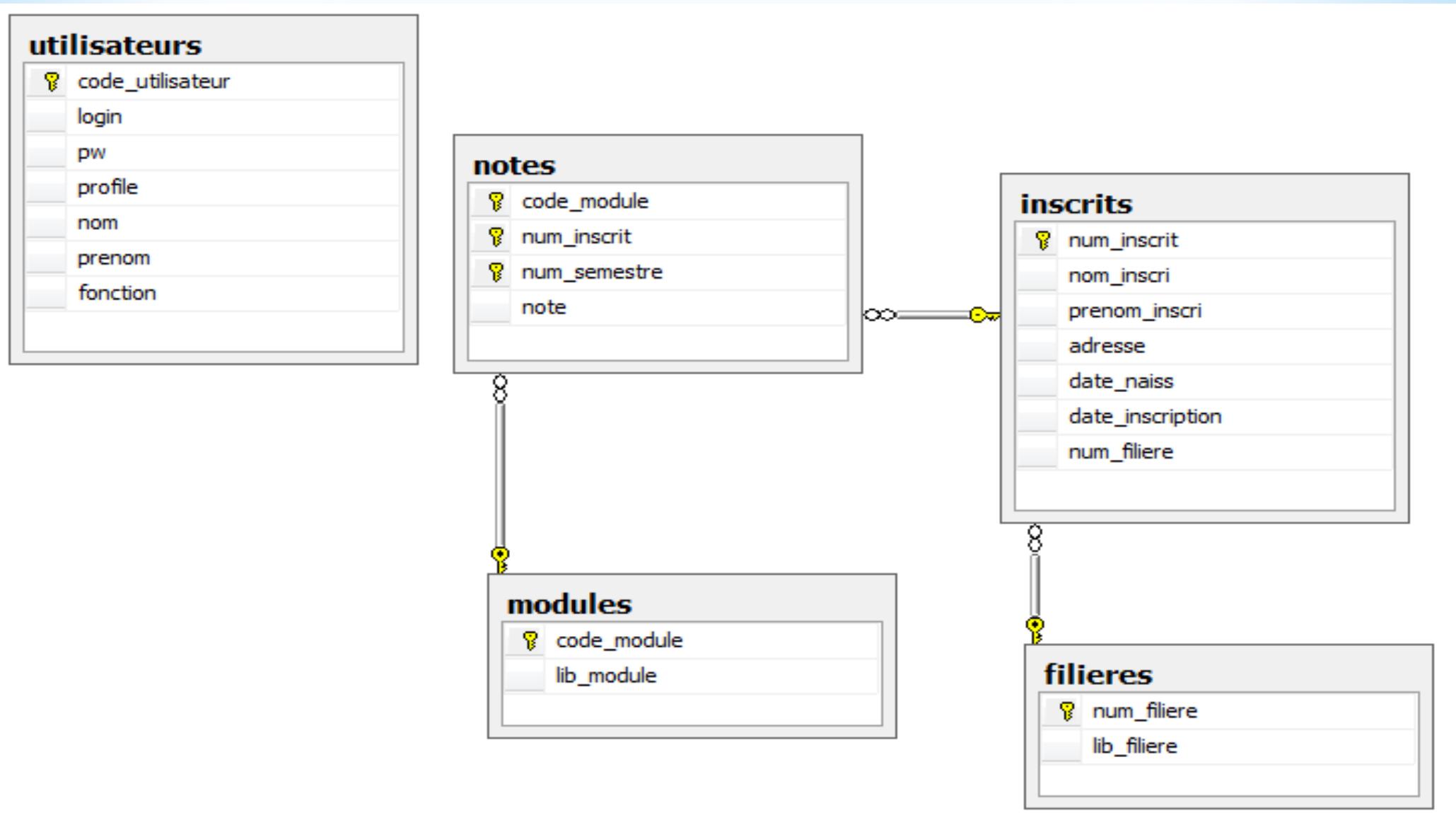
- Mettre **des données en cache localement** dans votre application afin de pouvoir les manipuler. Si vous devez uniquement lire les résultats d'une requête, le **DataReader** est le meilleur choix.
- Fournir un accès distant entre couches ou à partir d'un **service Web XML**.
- Interagir dynamiquement avec les données par le biais par exemple de la liaison à un contrôle Windows Forms ou de la combinaison et la mise en relation de données de diverses sources.
- Réaliser un **traitement complet des données sans qu'une connexion ouverte à la source de données soit nécessaire**, ce qui libère la connexion pour d'autres clients.

Si vous n'avez pas besoin de la fonctionnalité fournie par le **DataSet**, vous pouvez améliorer les performances de votre application en utilisant le **DataReader** pour retourner les données avec un accès en lecture seule et avant uniquement.

RQ : Bien que le **DataAdapter** utilise le **DataReader** pour remplir un **DataSet**, en utilisant le **DataReader**, vous pouvez gagner en performances car vous économiserez la mémoire qui serait utilisée par le **DataSet** et n'aurez pas à effectuer le traitement requis pour créer et remplir le **DataSet**.

Utilisation du Mode Connecté

Nous utiliseront le Serveur SQL Server 2008. La base de données utilisée sera **inscrits.mdb** qui permet de Gérer les inscrits ainsi que leur notes, dont le schéma est ci-dessous :



1. Connexion à une base de données :

Pour pouvoir travailler avec un serveur de base de données, une application doit établir une connexion réseau avec le **Serveur**, en utilisant le fournisseur **SQLServer** à travers la **classes** de ce fournisseur qui se trouvent dans l'espace de nom **System.Data.SqlClient..**

La classe **SqlConnection** est capable de gérer une connexion vers un Serveur SQL Server version 7.0 ou ultérieure.

Exemple : **System.Data.SqlClient.SqlConnection ctn;**

Puis, nous devons créer l'instance de la classe et l'initialiser en appelant un constructeur. L'initialisation va consister essentiellement à indiquer les paramètres utilisés pour établir la connexion avec le **Serveur**. Ces paramètres sont définis sous **forme d'une chaîne de caractères**. Ils peuvent être indiqués lors de l'appel du constructeur ou modifiés par la suite par la propriété **ConnectionString**.

a. Chaine de connexion :

Le format standard d'une **chaîne de connexion** est constitué d'une série de couples **mot clé/ valeur séparés par des points virgules**. Le signe = est utilisé pour l'affectation d'une valeur à un mot clé. Une **Exception** sera déclenchée si la chaîne de connexion contient une erreur. Les **mots clés suivants** sont disponibles pour une **chaîne de connexion** :

1. Connexion à une base de données :

a. Chaine de connexion :

■ Liste des différentes paramètres disponibles dans une chaine de connection:

Paramètre	Description
Connect Timeout	Indique le temps d'attente de connexion en seconde. Ce laps de temps dépassé, une exception est levée.
Data Source	Indique le nom ou l'adresse réseau du serveur.
Initial Catalog	Indique le nom de la base de données où l'application doit se connecter.
Integrated Security	Indique s'il faut un nom et un mot de passe. Si la valeur est sur False, un login et password doivent être fournis dans la chaîne de connexion. Si non le compte Windows de l'utilisateur est utilisé pour l'authentification.
Persist Security Info	Indique si le nom et le mot de passe est visible par la connexion.
Pwd	Indique le mot de passe associé au compte SQLServer.
User ID	Indique le nom du compte SQL Server.
Connection LifeTime	Indique la durée de vie d'une connexion dans un pool, la valeur 0 (zéro) correspond à l'infini.
Connection Reset	Indique si la connexion a été réinitialisée lors de son retour dans un pool.
Max Pool Size	Indique le nombre maximum de connexion dans un pool. Par défaut, le nombre maximum de connexions est 100.
Min Pool Size	Indique le nombre minimum de connexion dans un pool
Pooling	Indique si une connexion peut être sortie d'un pool.

1. Connexion à une base de données :

a. Chaine de connexion :

□ Une chaîne de connexion prend donc la forme minimale suivante :

Syntaxe d'une forme minimal de déclaration :

```
ctn = New SqlConnection()
```

```
ctn.ConnectionString ="Data Source=.\SQLServeur;Initial Catalog=Inscrits;Integrated Security=true;"
```

□ Pour ouvrir la connexion on utilise la méthode **open**, la méthode **close** permet de fermer la connexion.

1. Connexion à une base de données :

b. Pool de connexions :

Les **pools de connexions** permettent d'améliorer les performances d'une application, en évitant la création de connexions supplémentaires. Lorsqu'une connexion est ouverte, un pool de connexions est créé en se basant sur un algorithme basé, lui-même sur la chaîne de connexion. Chaque pool est donc associé à une chaîne de connexion particulière. Si une nouvelle connexion est ouverte et qu'il n'existe pas de pool correspondant exactement à sa chaîne de connexion, alors un nouveau pool est créé. Les pools de connexions ainsi créés existeront jusqu'à la fin de l'application. Lors de la création du pool, d'autres connexions peuvent être créées automatiquement pour satisfaire la valeur **Min Pool Size** indiquée dans la chaîne de connexion. D'autres connexions pourront, par la suite, être ajoutées au pool jusqu'à atteindre la valeur **Max Pool Size** de la chaîne de connexion.

c. Événements de connexion :

La classe **SQLConnection** propose deux événements vous permettant d'être prévenu lorsque l'état de la connexion change ou qu'un message d'information est envoyé par le serveur.

□ L'événement **StateChanged** : est déclenché lors d'un changement d'état de la connexion. Le gestionnaire de cet événement reçoit un paramètre de type **StateChangeEventArg** permettant d'obtenir, avec la propriété **CurrentState**, l'état actuel de la connexion et avec la propriété **OriginalState**, l'état de la connexion avant le déclenchement de l'événement..

1. Connexion à une base de données :

c. Événements de connexion :

Pour tester la valeur de ces propriétés, vous pouvez utiliser l'énumération `ConnectionState` qui possède différentes propriétés :

Propriété

`Broken`

Description

Permet de savoir si la connexion est interrompue, cette connexion peut se fermer puis se réouvrir.

`Closed`

Permet de savoir si l'objet connexion est fermé.

`Connecting`

Permet de savoir si l'objet connexion est en cours de connexion.

`Executing`

Permet de savoir si une commande est en train de s'exécuter.

`Fetching`

Permet de savoir si l'objet connexion est en train de récupérer des données.

`Open`

Permet de savoir si l'objet connexion est ouvert.

■ L'événement `InfoMessage` : déclenché lorsque le serveur vous informe d'une situation, anormale en utilisant le `e` de type `InfoMessageEventArgs`. A travers la propriété `Errors` de ce paramètre vous avez accès à des objets `SqlErrors`

1. Connexion à une base de données :

c. Événements de connexion :

Exemple :

▀ **StateChanged** : déclenché lors d'un changement d'état de la connexion en utilisant le paramètre e de type **StateChangeEventArgs** :

```
If(e.currentState == ConnectionState.open) {  
    messagebox.show("bravo ! connexion réussi");  
}
```

▀ **InfoMessage** : déclenché lorsque le serveur vous informe d'une situation, anormale en utilisant le e de type **InfoMessageEventArgs**. A travers la propriété Errors de ce paramètre vous avez accès à des objets **SqlErrors**

```
SqlClient.SqlError info;  
ForEach(info in e.Errors) {  
    Messagebox.show(info.Message);  
}
```

2. Exécution d'une Commande :

Après avoir établi une connexion vers un Serveur de la base de données, vous pouvez lui transmettre des instructions SQL. La classe **SqlCommandes** est utilisée pour demander au Serveur l'exécution de différents types de requêtes SQL. Cette Classe contient plusieurs méthode permettant l'exécution de différents types de requête SQL et peut être instanciée de façon classique, ou une instance peut être obtenu par la méthode **CreateCommande** de la connexion.

a. Crédation d'une commande :

□ Utilisation Constructure par défaut : Exemple :

```
SqlCommand cmd = New SqlCommand();
cmd.connection = ctn;
cmd.CommandText= "Select * From utilisateurs";
```

□ Utilisation d'un Constructeur surchargé : Exemple :

```
SqlCommand cmd = new sqlCommand("Select * From utilisateurs",ctn);
```

□ Utilisation de la méthode Create Command de la connexion: Exemple

```
SqlCommand cmd;
cmd=ctn.CreateCommand();
cmd.CommandText ="Select * From utilisateurs";
```

2. Exécution d'une Commande :

b. Lecture d'information :

Fréquemment l'instruction SQL d'une **SqlCommand** sélectionne un ensemble d'enregistrement, ou une valeur unique(résultat d'un calcul).

- o Une instruction SQL, renvoyant un ensemble d'enregistrements, doit être exécutée par la méthode : **ExecuteReader** : qui retourne un objet **DataReader** qui permet la lecture des informations en provenance de la base de données.
- o Si l'instruction renvoie qu'une Valeur unique, la méthode **ExecuteScalar** se charge de l'exécution et retourne elle-même la valeur.

□ **Exemple1 : Cas d'une valeur unique :** Le code suivant permet la récupération du nombre d'utilisateur avec un profile Administrateur :

```
using System.Data.SqlClient;

namespace fournisseursADO
{
    class Program
    {
        static void Main(string[] args)
        {
            string chaine = "Data Source=DESKTOP-V3HR419;Initial Catalog=inscrits;Integrated Security=True";
            SqlConnection ctn = new SqlConnection(chaine);
            SqlCommand cmd = new SqlCommand("Select count(code_utilisateur) from utilisateurs where profile='a'", ctn);
            ctn.Open();
            Console.WriteLine("le nombre d'utilisateur est {0} ", cmd.ExecuteScalar());
            ctn.Close();
            Console.ReadKey();
        }
    }
}
```

2. Exécution d'une Commande :

b. Lecture d'information :

□ Cas d'instructions renvoyant plusieurs enregistrements :

Il faut exécuter l'instruction **ExecuteReader** et Récupérer l'objet **DataReader**, il faut utiliser la méthode **Read** de la Classe **DataReader** , qui retourne une Valeur Booléen indique s'il reste un Enregistrement Suivant. Le Déplacement ni possible qu'au en avant appelé **Forward Only**.

Les informations contenues dans l'enregistrement courant sont accessibles par une des méthodes **Get...** de la classe **DataReader**. Ces méthodes permettent d'extraire les données de l'enregistrement et de les convertir dans un type de données .NET. Il en existe une version pour chaque type de données du Framework .NET. Il faut bien sûr que les informations présentes dans l'enregistrement, puissent être converties dans le type correspondant. Si la conversion est impossible, il y a déclenchement d'une Exception.

Les méthodes **Get...** attendent, comme paramètre, le numéro du champ à partir duquel elles récupèrent l'information.

Vous pouvez aussi utiliser la propriété, par défaut, **Item** du **DataReader** en indiquant le nom du champ concerné. Il n'y a pas, dans ce cas, de conversion et la valeur renvoyée est de type **Object**.

2. Exécution d'une Commande :**b. Lecture d'information :**

Exemple 1 : Cas d'instructions renvoyant plusieurs enregistrements : Exemple : Le code suivant affiche la liste de toutes les Utilisateurs disponibles dans la dataGridView:



L'utilisation d'une connexion par un DataReader s'effectue de manière exclusive. Pour que la connexion soit à nouveau disponible pour une autre commande, vous devez obligatoirement fermer le DataReader après son utilisation.

2. Exécution d'une Commande :**b. Lecture d'information :**

Exemple 1 : Cas d'instructions renvoyant plusieurs enregistrements : Exemple : Le code suivant affiche la liste de toutes les Utilisateurs disponibles dans la dataGridView:

```
string ch = ConfigurationManager.ConnectionStrings["chaine"].ConnectionString;
SqlConnection ctn = new SqlConnection(ch);
SqlCommand cmd = new SqlCommand();
cmd.CommandText = "select * from customers";
cmd.Connection = ctn;
ctn.Open();
SqlDataReader lecteur = cmd.ExecuteReader();
BindingSource src=new BindingSource();
src.DataSource = lecteur;
dgv_utilisateurs.DataSource = src;
lecteur.Close();
ctn.Close();
```

2. Exécution d'une Commande :

b. Lecture d'information :

□ **Exemple2 : Cas d'instructions renvoyant plusieurs enregistrements : Exemple :** Le code suivant affiche la liste de toutes les Utilisateurs disponibles :

```
using System.Data.SqlClient;

namespace fournisseursADO
{
    class Program
    {
        static void Main(string[] args)
        {
            string chaine = "Data Source=DESKTOP-V3HR419;Initial Catalog=inscrits;Integrated Security=True";
            SqlConnection ctn = new SqlConnection(chaine);
            SqlCommand cmd = new SqlCommand("Select * from utilisateurs", ctn);
            ctn.Open();
            SqlDataReader lecteur = cmd.ExecuteReader();
            while (lecteur.Read())
            {
                Console.WriteLine("code utilisateur : {0} \t login : {1} \t password : {2}", lecteur.GetInt32(0), lecteur.GetString(1),
lecteur.GetString(2));

            }
            lecteur.Close();
            ctn.Close();
            Console.ReadKey();
        }
    }
}
```

L'utilisation d'une connexion par un **DataReader** s'effectue de manière exclusive. Pour que la connexion soit à nouveau disponible pour une autre commande, vous devez obligatoirement fermer le **DataReader** après son utilisation.

Atelier 1

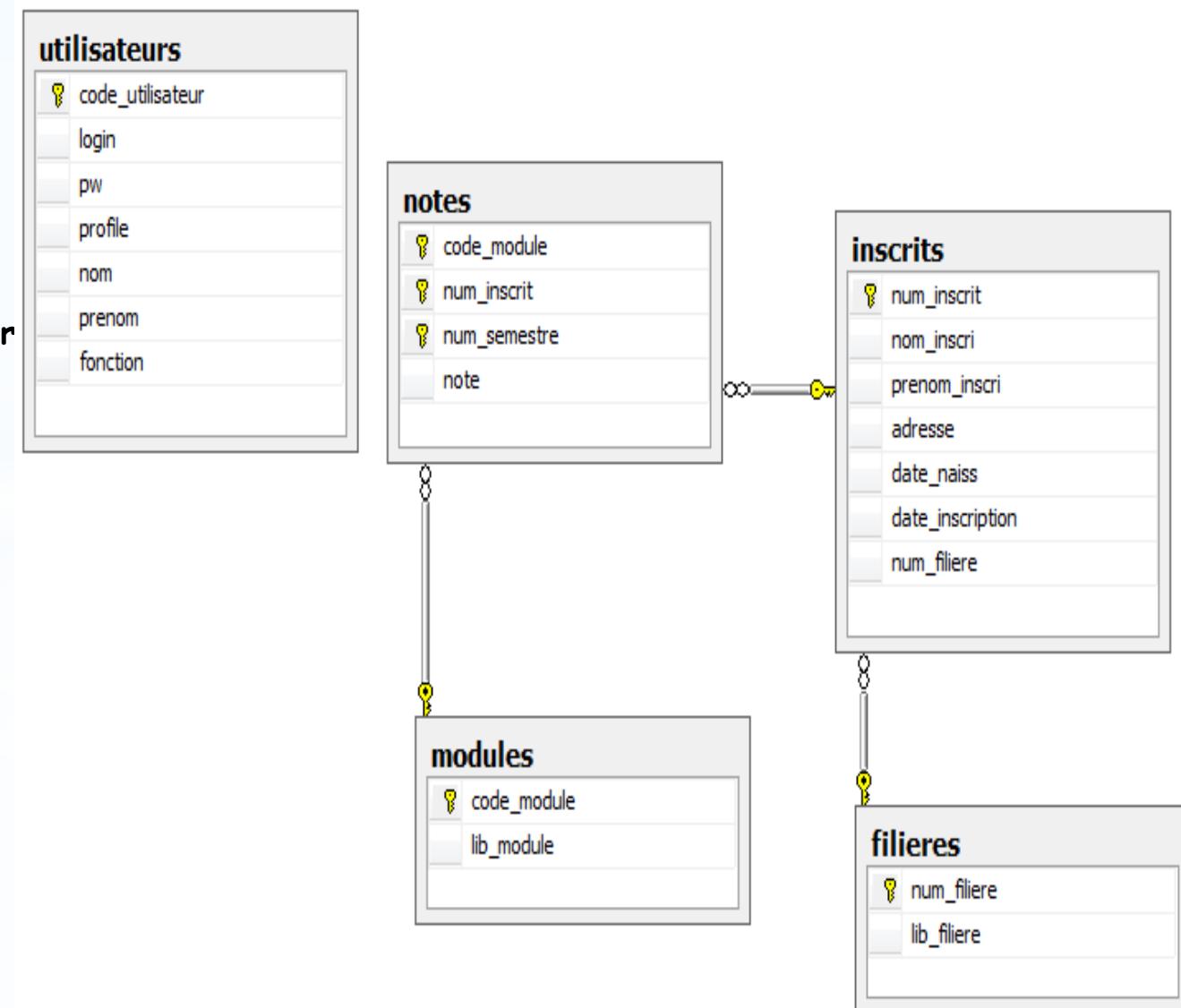
TP 1 : Les bases de l'accès aux données, Mode connecté

Création de la base de données : inscrits.mdb

Créez une base de données SqlServer en utilisant Sql Server nommée "inscrits.mdf" :

Utiliser SSMS et le langage SQL pour créer la Base de données Inscrits.mdb et respecter selon le schéma ci-dessous et en respectant les contraintes suivantes : Code utilisateur, code module, num_filiere avec une numérotation automatique, code_module de type entier et num_inscrit de type caractère 6 maximum. Profile doit être de type a : pour Administrateur, u : pour utilisateur limiter, o : opérateur.

Ajouter 3 utilisateurs à votre table Utilisateurs en utilisant SSMS et le langage SQL, avec les profiles suivants : 2 utilisateur avec un profile Administrateur, 1 utilisateur avec profile Utilisateur, 1 utilisateur avec profile Opérateur.



Atelier 1

TP 2 : L'accès à la base de données et Manipulation des données

La connexion à notre base de données doit être unique et centralisé pour toute le projet.

Pour pouvoir manipuler une Base de données SqlSever vous devez importer l'espace de nom : **System.Data.SqlClient**.

1. Connexion à la base de données Inscrits.mdb : utilisation d'une Variable objet globale ctn Il existe plusieurs méthodes pour accéder à une base de données. Nous allons utiliser le mode connecté. Nous reverrons le mode déconnecté par la suite, mais le principe qu'on va utilisé est le suivant :

1. Créer une nouvelle application winforms avec comme nom **generale.cs** et créer un objet **ctn** avec un niveau d'accessibilité public, qui sera accessible au niveau de toutes les formes de votre solution.

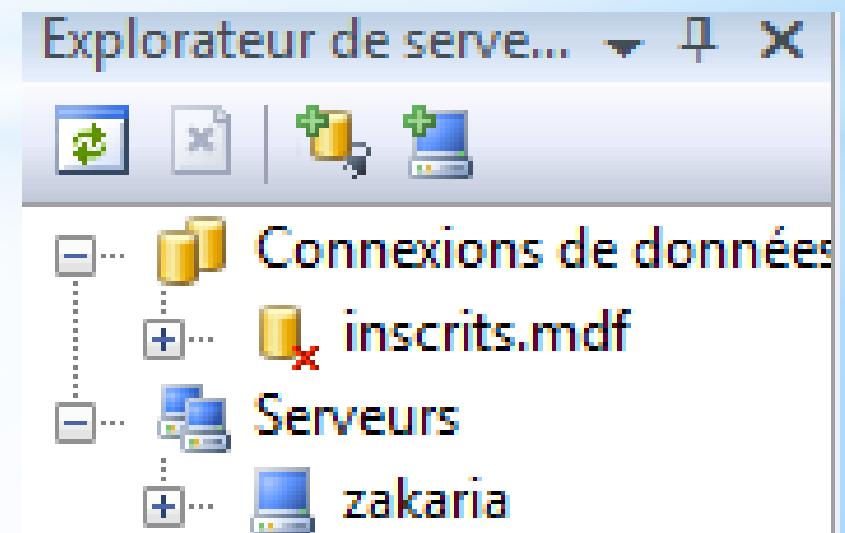
2. Ajouter un Event Handler (gestionnaire d'événement) dans l'événement Load de votre forme menu, qui permet de lier la procédure **onStateChange** que vous concevez à l'événement **OnStateChange**, cette procédure permettra d'afficher un message de confirmation de connexion avec la signature suivante :

```
onStateChange(ByVal Object sender, ByVal StateChangeEvent e)
```

3. Programmer l'ouverture de votre objet **ctn** et prévoyez la gestion des Exceptions qui peuvent surgir dans l'événement Load de votre Forme **FrmListUtilisateurs** (sub remplirTreeview).

RQ : Pour manipuler la structure et les données de votre Base de données depuis votre EDI Visual Studio, utiliser la Fenêtre Explorateur de Serveurs :(le SSMS intégré à Visual Studio).

Menu Affichage -> Explorateur de Serveurs -> Cliquez droit sur le nom de la connexion puis Actualiser s'il existe si non ajouté une nouvelle connexion vers votre base de données.



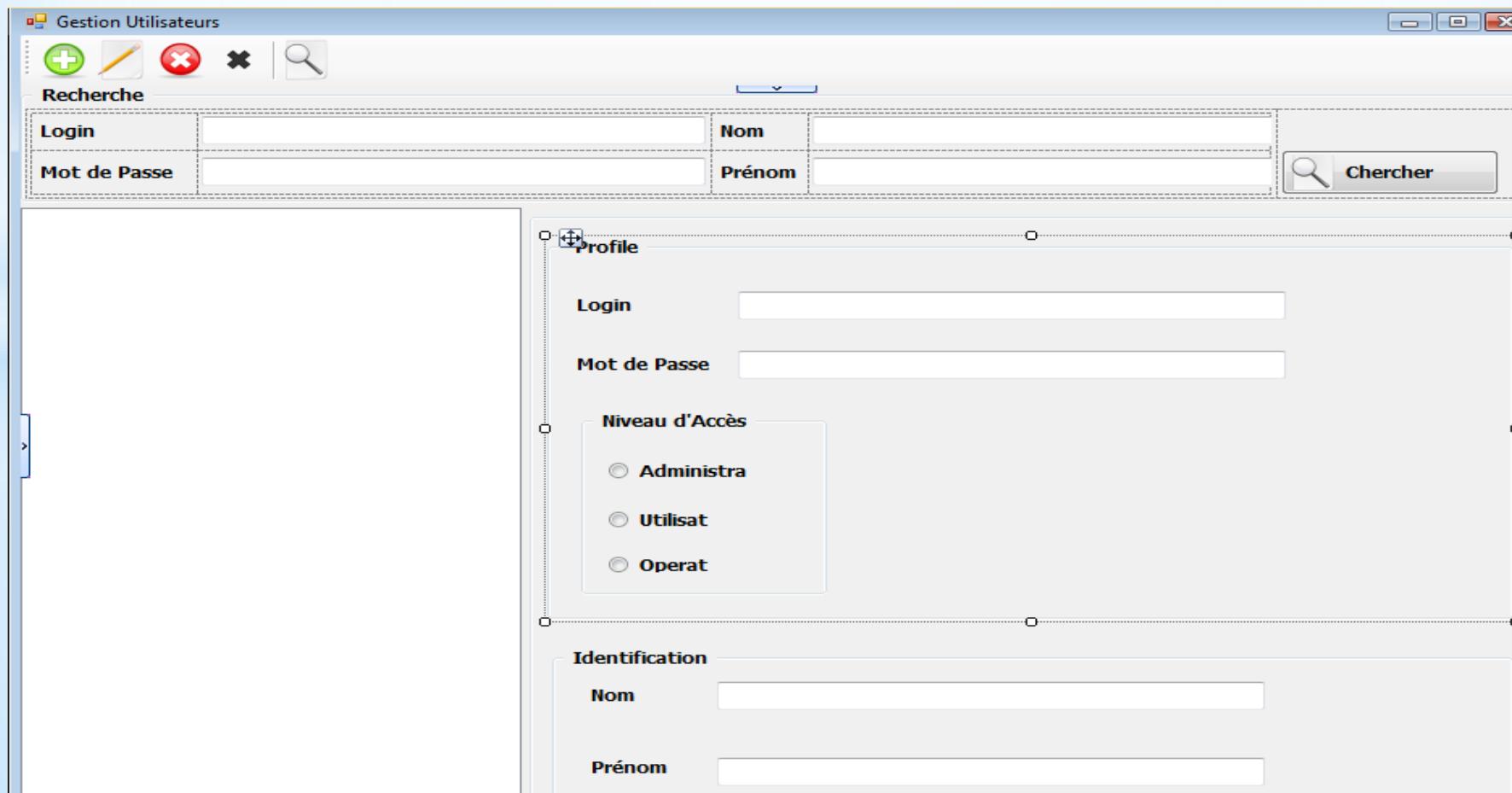
Atelier 1

TP 3 : L'accès à la base de données et Manipulation des données: lecture des données

Modifier le code de votre Formulaire **FrmAjoModUtilisateur** pour :

Remplir la liste des éléments de votre contrôle **TreeView_utilisateurs** avec la liste des utilisateurs de votre table utilisateurs issu de la base de données **Inscrits.mdf**.

1. Modifier l'interface de votre Formulaire **FrmAjoModUtilisateur** suite au figure cidessous:



Atelier 1

TP 3 : L'accès à la base de données et Manipulation des données: lecture des données

2. Ecrire le code qui permet de créer un nouveau Contrôle TreeView avec 3 noeuds : Administrateur, Utilisateur et Opérateur,

3. Créer un nouveau Objet Commande pour transmettre une instruction SQL, qui permet de sélectionner l'ensemble des utilisateurs de votre table utilisateurs, Il faut faire appelle à la méthode ExecuteReader pour exécuter la Commande SQL et Récupérer le résultat dans un objet DataReader, utiliser ensuite la méthode Read de la Classe DataReader avec une boucle For Each pour récupérer les différentes lignes (utilisateurs) de votre table utilisateurs. Ajouter pour chaque nœud de votre contrôle TreeView l'ensemble des noms et prénom des utilisateurs de ce type qui existe dans votre table utilisateurs.

RQ : Pour lire Les informations de chaque colonne dans l'enregistrement courant utiliser la méthode Get.

2. Exécution d'une Commande :

C. Modification des informations :

La modification des informations dans une base de données s'effectue principalement par les instructions SQL **INSERT**, **UPDATE**, **DELETE**. Ces instructions ne retournent pas d'enregistrements en provenance de la base de données. Pour utiliser ces instructions, vous devez créer une **SqlCommand**, puis demander l'exécution de cette commande par la méthode **ExecuteNonQuery**.

Cette méthode retourne le nombre d'enregistrements affectés par l'exécution de l'instruction SQL contenue dans la **SqlCommand**. Si la propriété **CommandText** contient plusieurs instructions SQL, alors la valeur renvoyée par la méthode **ExecuteNonQuery** correspond au nombre total de lignes affectées par toutes les instructions SQL de la **SqlCommand**.

2. Exécution d'une Commande :

C. Modification des informations :

Exemple : Le code suivant ajoute une un nouveau utilisateur dans la table Utilisateurs.

```
using System.Data.SqlClient;

namespace fournisseursADO
{
    class Program
    {
        static void Main(string[] args)
        {
            string chaine = "Data Source=DESKTOP-V3HR419;Initial Catalog=inscrits;Integrated
Security=True";
            SqlConnection ctn = new SqlConnection(chaine);
            SqlCommand cmd = new SqlCommand("insert into utilisateurs
values('adm','4444','a','soudy','youssef','commercial')", ctn);
            ctn.Open();
            Console.WriteLine("{0} lignes ajoutées dans la table", cmd.ExecuteNonQuery());
            ctn.Close();
            Console.ReadKey();
        }
    }
}
```

2. Exécution d'une Commande :

d. Utilisation de paramètres :

La manipulation d'instructions SQL peut être facilitée par la **création de paramètres**. Ils permettent de construire des instructions **SQL génériques**, pouvant facilement être réutilisées. Le principe de fonctionnement est semblable aux **procédures et fonctions** de C#. Une alternative à l'utilisation de paramètres pourrait être la **construction dynamique d'instruction SQL par concaténation de chaînes de caractères**.

Exemple : utilisant la technique de **concaténation de chaîne** de caractère qui permet de rechercher un Utilisateur par son Login et pw :

```
string chaine = "Data Source=.;Initial Catalog=inscrits;Integrated Security=True";
SqlConnection ctn = new SqlConnection(chaine);
SqlCommand cmd = new SqlCommand();
cmd.Connection = ctn;
Console.Write("saisir le login du client recherche : ");
string login = Console.ReadLine();
Console.Write("saisir le pw du client recherche : ");
string pw = Console.ReadLine();
cmd.CommandText = " SELECT * from Utilisateurs WHERE login= '" + login + "' and pw= '" + pw + "'";
ctn.Open();
SqlDataReader lecteur = cmd.ExecuteReader();
while(lecteur.Read()){
    Console.WriteLine("nom utilisateur:{0}", lecteur.GetString(4));
}
lecteur.Close();
ctn.Close();
```

2. Exécution d'une Commande :

d. Utilisation de paramètres :

La partie importante de ce code se situe lors de l'affectation d'une valeur à la propriété `CommandText`. Une instruction SQL correcte doit être construite par concaténation de chaînes de caractères. Si plusieurs informations doivent varier, il y a une multitude de concaténations à réaliser. Les erreurs classiques dans ces concaténations sont :

- o L'oubli d'un espace;
- o L'oubli des caractères ' ', pour encadrer une valeur de type chaîne de caractères;
- o Un nombre de caractère impair.

Toutes ces erreurs ont, pour même effet, la création d'une instruction SQL invalide qui sera rejetée à l'exécution par le serveur.

Par-dessus ces problèmes, l'utilisation des concaténations peut donner lieu à une vulnérabilité qui peut être exploitée par un utilisateur malveillant par attaque qu'on appelle 'injection SQL', cette attaque provoque des conséquences néfastes sur la base de données et la sécurité de l'application.

■ Utilisation de la collection `Parameters` de la class `SqlCommand` :

L'utilisation des paramètres simplifie considérablement l'écriture de ce type de requête. Les paramètres sont utilisés pour marquer un emplacement dans une requête où sera placé, au moment de l'exécution, une valeur littérale chaîne de caractères ou numérique. Les paramètres peuvent être nommés ou anonymes. Un paramètre anonyme est introduit dans une requête par le caractère ?. Les paramètres nommés sont spécifiés par le caractère @ suivi du nom du paramètre.

2. Exécution d'une Commande :

d. Utilisation de paramètres :

□ Utilisation de la collection **Parameters** de la class **SqlCommand** :

La requête de notre exemple précédent peut prendre les formes suivantes:

□ Exemple1: Utilisation des paramètres anonyme :

```
cmd.CommandText = "Select * from Utilisateurs where login = ? and pw = ?";
```

□ Exemple2 : Utilisation des paramètres nommés :

```
cmd.CommandText = "Select * from Utilisateurs where login = @login and pw = @pw";
```

RQ : L'exécution de la **SqlCommand** échoue maintenant si aucune information n'est fournie pour le ou les paramètres.

La **SqlCommand** doit avoir une **liste de valeurs** utilisées pour le remplacement des **paramètres**, au moment de l'exécution. Cette liste est stockée dans la collection **Parameters** de la **SqlCommand**.

Avant l'exécution de la **SqlCommand**, il faut donc créer les objets **SqlParameter** et les ajouter à la collection. Pour chaque **SqlParameter**, il faut fournir :

- Le **nom** du paramètre;
- La **valeur** du paramètre;
- La **direction** d'utilisation du paramètre.

2. Exécution d'une Commande :

d. Utilisation de paramètres :

■ Utilisation de la collection Parameters de la class SqlCommand :

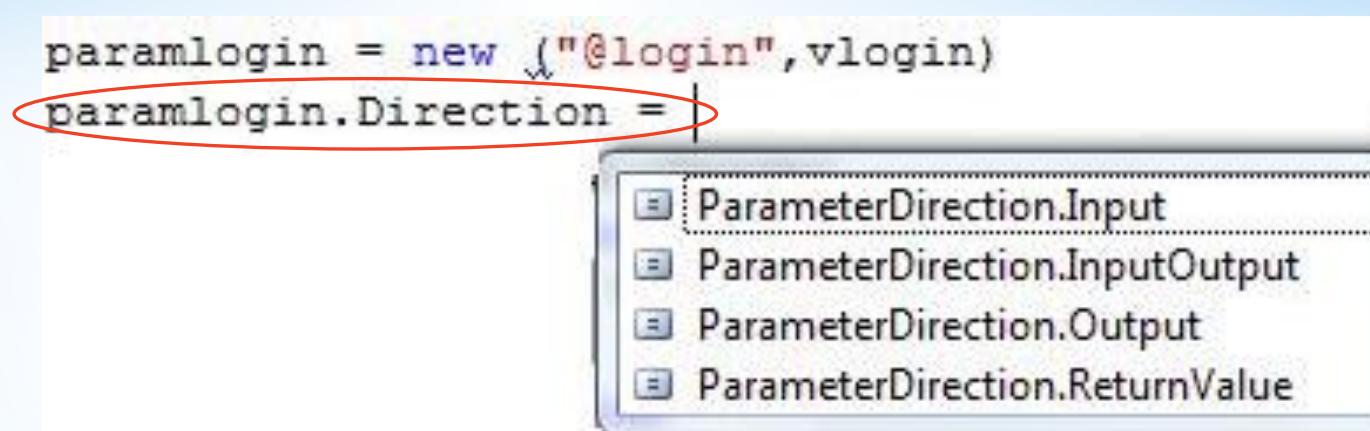
Les deux premiers informations (le nom et la valeur) sont indiquées lors de la construction de l'objet

Exemple :

```
String vlogin;  
SqlParameter paramLogin;  
paramLogin = New SqlParameter("@login", vlogin);
```

La direction d'utilisation indique si l'information contenue dans le paramètre, est passée au code SQL pour son exécution (Input) ou si c'est l'exécution du code SQL qui va modifier la valeur du paramètre (output) ou les deux (InputOutput). La propriété Direction de la classe SqlParameter indique le mode d'utilisation du paramètre.

Exemple :



2. Exécution d'une Commande :

d. Utilisation de paramètres :

■ Utilisation de la collection **Parameters** de la class **SqlCommand** :

Le paramètre est maintenant prêt à être ajouté à la collection **Parameters**.

- o Si la requête utilise les paramètres anonymes. Les paramètres doivent obligatoirement être ajoutés à la collection, dans l'ordre de leur apparition dans la requête.
- o Si les paramètres nommés sont utilisés, il n'est pas indispensable de respecter cette règle.

La **SqlCommand** est maintenant prête pour l'exécution. A noter qu'avec cette solution nous n'avons pas à nous soucier du type de valeur attendue par l'instruction SQL pour savoir si nous devons l'encadrer avec des caractères '. Si des paramètres sont utilisés en sortie de l'instruction SQL, ils ne seront Disponibles qu'après la fermeture du DataReader.

2. Exécution d'une Commande :

d. Utilisation de paramètres :

Exemple 1 : Une Commande SQL qui comprend une instruction Sql qui permet de chercher un Client par son Code et de retourner le nom, prénom et la fonction.

The screenshot shows a Windows application window titled "Form1". Inside the window, there is a label "Donner le code client" positioned above a text input field. To the right of the input field is a button labeled "Afficher". Below the input field, there are three empty text boxes labeled "Nom", "Prénom", and "Fonction" respectively, which are intended to display the results of the search. The overall layout is clean and functional, typical of a C# Windows Forms application.

2. Exécution d'une Commande :

d. Utilisation de paramètres :

Exemple 1 : Une Commande SQL qui comprend une instruction Sql qui permet de chercher un Client par son Code et de retourner le nom, prénom et la fonction.

```
private void btnAfficher_Click(object sender, EventArgs e)
{
    string codeUtilisateur = txtCode.Text;
    string chaine = "Data Source=.;Initial Catalog=inscrits;Integrated Security=True";
    SqlConnection ctn = new SqlConnection(chaine);
    SqlCommand cmd = new SqlCommand();
    cmd.Connection = ctn;
    cmd.CommandText = "SELECT nom, prenom, fonction FROM utilisateurs WHERE code_utilisateur = @code";
    SqlParameter paramCode = new SqlParameter("@code", codeUtilisateur);
    //paramCode.ParameterName = "@code";
    //paramCode.Value = codeUtilisateur;
    paramCode.Direction = ParameterDirection.Input;
    cmd.Parameters.Add(paramCode);
    ctn.Open();
    SqlDataReader lecteur = cmd.ExecuteReader();
    while (lecteur.Read())
    {
        lblNom.Text = lecteur.GetString(0);
        lblPrenom.Text = lecteur.GetString(1);
        lblFonction.Text = lecteur.GetString(2);
    }

    lecteur.Close();
    ctn.Close();
}
```

2. Exécution d'une Commande :**d. Utilisation de paramètres :**

Exemple 2 : Une Commande SQL qui comprend deux **instructions Sql** qui permettent de chercher un Client par son Code et de retourner le nbr de commande qu'il a passées.

```
string chaine = "Data Source=.;Initial Catalog=northwind;Integrated Security=True";
SqlConnection ctn = new SqlConnection(chaine);
SqlCommand cmd = new SqlCommand();
cmd.Connection = ctn;
Console.WriteLine("Saisir le code du client recherche : ");
string codeClient = Console.ReadLine();
cmd.CommandText = "select * from customers where customer_id = @code; select @nbCmd =
count(order_id) from orders where customer_id = @code";
SqlParameter paramCodeClient = new SqlParameter("@code", codeClient);
//paramCodeClient.ParameterName = "@code";
//paramCodeClient.Value = codeClient;
paramCodeClient.Direction = ParameterDirection.Input;
cmd.Parameters.Add(paramCodeClient);
int nbrCmd=0;
SqlParameter paramNbrCommandes = new SqlParameter("@nbCmd", nbrCmd);
//paramNbrCommandes.ParameterName = "@nbCmd";
paramNbrCommandes.Direction = ParameterDirection.Output;
cmd.Parameters.Add(paramNbrCommandes);
```

2. Exécution d'une Commande :

d. Utilisation de paramètres :

Exemple 2:

```
ctn.Open();
SqlDataReader lecteur = cmd.ExecuteReader();
while (lecteur.Read())
{
    Console.WriteLine("nom du client {0}", lecteur.GetString(2));
}
lecteur.Close();
Console.WriteLine("ce client a passé {0} commande(s)", paramNbrCommandes.Value);
ctn.Close();
Console.ReadKey();
```

Atelier 1

TP 4 : L'accès à la base de données et Manipulation des données: utilisation des paramètres

Modifier le code de votre FrmLogin qui permet de gérer la connexion à votre projet (Forme d'authentification) pour vérifier que le login et le mot de passe existent réellement dans la table utilisateurs de votre base de données inscrits.



Vous devez modifier le code de votre procédure click lié au bouton ok, pour cela on vous de mande d'utiliser une commande SQL avec deux paramètres nommer Login et Pw pour vérifier l'existence de ce compte de connexion dans la table utilisateurs.

Voila le début du Code :

```
SqlCommand cmd;  
SqlDataReader lecteur;  
String login = string.empty;  
String pw = string.empty;  
SqlParameter paramLogin;  
SqlParameter paramPw;
```

Atelier 1

TP 4 : L'accès à la base de données et Manipulation des données: utilisation des paramètres

Respecter l'algorithme suivant :

Si on a saisi le login et mot de passe alors

 Déclaration d'un nouvel objet commande

 Déclaration objet datareader

 Déclaration des objets paramètres (nommer)

 Ouverture connexion

 Réglage des propriétés des paramètres

 Exécution commande

 Récupération résultat

 Si le login et le pw existent

 Affichage des différentes options du menu

 Si non

 Affichage d'un message

 Fin si

 Si non

 Affichage d'un message

 Fin si

 Fermeture objet dataReader

 Fermeture objet connexion

2. Exécution d'une Commande :

E. Exécution de procédure stockée :

□ Les **procédures stockées** sont des éléments d'une base de données correspondant à un ensemble d'instructions SQL, pouvant être exécutées par simple appel de leur nom. Ce sont des véritables programmes SQL pouvant **recevoir des paramètres et renvoyer des valeurs**. De plus, les procédures stockées sont **enregistrées dans le cache mémoire du serveur**, sous forme **compilée** lors de leur première exécution, ce qui accroît les performances pour les exécutions suivantes. Un autre avantage des procédures stockées est de centraliser sur le serveur de base de données tous les codes SQL d'une application. Si des modifications doivent être apportées dans les instructions SQL, vous n'aurez que des modifications à effectuer sur le serveur sans avoir à reprendre le code de l'application, donc sans avoir à régénérer et redéployer l'application.

□ L'appel à une procédure stockée, à partir de C#, est pratiquement similaire à l'exécution d'une instruction SQL.

- La propriété **CommandText** contient le **nom de la procédure stockée**.
- Vous devez également modifier la propriété **CommandType** avec la valeur **CommandType.StoredProcedure** pour indiquer que la propriété **CommandText** contient le nom d'une procédure stockée.

2. Exécution d'une Commande :

E. Exécution de procédure stockée :

¶ Comme pour une instruction SQL, une procédure stockée peut utiliser des paramètres en entrée ou en sortie. Il y a un troisième type de paramètre disponible pour les procédures stockées :

- o Le type **ReturnValue** : Ce type de paramètre sert à récupérer la valeur renvoyée par l'instruction **Return** de la procédure stockée .

Exemple 1 : Procédure Stocké avec un paramètre ReturnValue :

Pour tester ces nouvelles notions, nous allons utiliser la procédure stockée suivante, qui retourne le montant total de toutes les commandes passées par un client.

```
CREATE PROCEDURE sp_TotalClient
@code char(5) AS
declare @total money
select @total=sum(unit_price*quantity)
from orders inner join order_details on
orders.order_id=order_details.order_id where
customer_id=@code
return @total
GO
```

2. Exécution d'une Commande :

E. Exécution de procédure stockée :

Exemple 1 : Procédure Stocké avec un paramètre **ReturnValue** : Code utilisé pour exécuté la procédure stocké Coté C#

```
Console.WriteLine("Saisir le code client recherché : ");
string codeClient = Console.ReadLine();
string chaine = "Data Source=.;Initial Catalog=northwind;Integrated Security=True";
SqlConnection ctn = new SqlConnection(chaine);
SqlCommand cmd = new SqlCommand();
cmd.Connection = ctn;
cmd.CommandText = "sp_TotalClient";
cmd.CommandType = CommandType.StoredProcedure;
SqlParameter paramCodeClient = new SqlParameter("@code", codeClient);
paramCodeClient.Direction = ParameterDirection.Input;
cmd.Parameters.Add(paramCodeClient);
SqlParameter paramMantant = new SqlParameter("RETURN_VALUE", SqlDbType.Decimal);
paramMantant.Direction = ParameterDirection.ReturnValue;
cmd.Parameters.Add(paramMantant);
ctn.Open();
cmd.ExecuteNonQuery();
Console.WriteLine("Ce client a passé pour {0} Euros de commande", paramMantant.Value);
ctn.Close();
```

2. Exécution d'une Commande :

E. Exécution de procédure stockée :

Exemple 2 : Procédure Stocké avec 2 paramètres de type input : Code utilisé pour exécuté la procédure stocké Coté C#

```
CREATE PROCEDURE sp_authentification  
    @login varchar(30),  
    @pw varchar(30)  
AS  
SELECT * FROM utilisateurs  
WHERE login=@login AND pw=@pw  
GO
```

Code d'exécution coté C# :

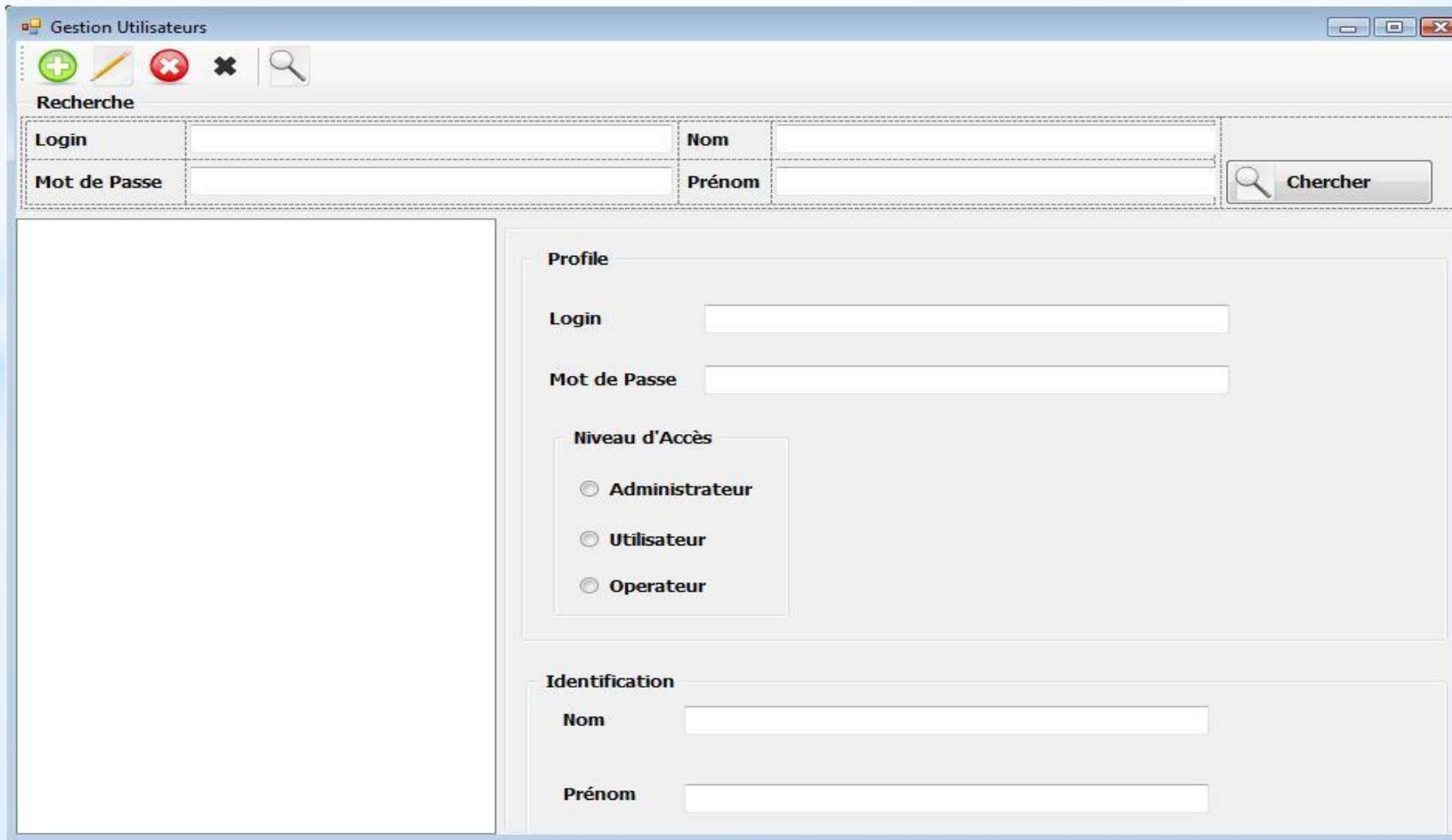
```
Console.WriteLine("Saisir le login : ");
string login = Console.ReadLine();
Console.WriteLine("Saisir le mot de passe : ");
string pw = Console.ReadLine();
string chaine = "Data Source=.;Initial Catalog=inscrits;Integrated Security=True";
SqlConnection ctn = new SqlConnection(chaine);
SqlCommand cmd = new SqlCommand();
cmd.Connection = ctn;
cmd.CommandText = "sp_authentification";
cmd.CommandType = CommandType.StoredProcedure;
SqlParameter paramLogin = new SqlParameter("@login", login);
paramLogin.Direction = ParameterDirection.Input;
cmd.Parameters.Add(paramLogin);
SqlParameter paramPW = new SqlParameter("@pw", pw);
paramPW.Direction = ParameterDirection.Input;
cmd.Parameters.Add(paramPW);
ctn.Open();
SqlDataReader lecteur= cmd.ExecuteReader();
while (lecteur.Read())
{
    Console.WriteLine("Le nom de cet utilisateur est {0}", lecteur[4]);
}
lecteur.Close();
ctn.Close();
```

Atelier 1

TP 5 : L'accès à la base de données et Manipulation des données: utilisation des procédures stockées

A présent on va essayer d'utiliser le mode connecté pour compléter le codage des différentes options offertes par la Tool Strip (barre d'outils) de notre Formulaire FrmAjoModUtilisateur : Ajoute, Modification, Suppression et la Recherche d'un utilisateur.

1. Bouton Rechercher : Commencer par modifier votre Interface pour ajouter une zone permettant de saisir les informations concernant la Recherche : Ajouter un contrôle SplitContainer (nommer le SplitContainerPrinci) qui divisera votre Formulaire en 2 parties Horizontales : remplissez la première Panel1 par un contrôle Groupe box et des contrôle TableLayoutPanel et votre deuxièmes Panel2 par le reste des contrôles déjà créées :



Atelier 1

TP 5 : L'accès à la base de données et Manipulation des données: utilisation des procédures stockées

2. la Panel1 (Recherche) ne doit pas apparaître au démarrage de votre formulaire, Ajouter le Code lié au clic sur le bouton rechercher pour faire apparaître ou disparaître cette zone.

3. Ajouter la procédure stockée authentification qui permet de rechercher un utilisateur par son login et son mot de passe.

4. Ajouter le code lié à l'événement click de votre Bouton Chercher suite au petit algorithme suivant : (on va limiter la recherche rien qu'avec le login et le mot de passe)

Déclaration objet command

Déclaration objets paramètres (login et mot de passe pour la procédure stockée)

Déclaration objet DataReader

Si on a saisi le login et le mot de passer alors

Création nouveau objet commande

Réglage propriétés connexion objet commande(procédure stocké)

Réglage propriétés des paramètres

Ouverture connexion

Exécution objet commande et récupération résultat

Si objet reader contient une ligne alors

Remplir zone texte boxe (utiliser procedure remplitTextBoxe)

Si non

Afficher message

Finsi

Fermer lecteur

Fermer connexion

Sinon

Afficher message

Fin si

```
[-] Create procedure [authentification]
    [-] @login varchar(30), @pw varchar(30)
        as
            [-] select * from utilisateurs
                where login=@login and pw =@pw
```

Atelier 1

TP 5 : L'accès à la base de données et Manipulation des données: utilisation des procédures stockées

5. Programmer le reste des boutons pour réaliser : Ajout, la modification et la suppression d'un Utilisateur.

Utiliser une Procédure stocké ou une commande paramétrée pour afficher toutes les informations de l'utilisateur sélectionné sur le TreeView, puis ajouter le code qui permet en cliquant sur un des bouton (Ajout, Modification, Suppression) de réaliser l'opération attendue.

Exemple : Algorithme pour ajouter un nouvel Utilisateur :

Déclarations

 objet command

.....

Si toute les zones sont renseignées alors

S'il n'y pas d'utilisateur ayant le même login, mot de passe, nom et prénom alors

 Création nouveau objet commande

 Réglage propriétés connexion objet commande (propriétés : ctn et CommandText)

 Ouverture connexion

 Exécution objet commande (en utilisant la méthode ExecuteNonQuery)

 Affichage message de confirmation d'ajout

 Si non

 Affichage message d'alerte d'existence d'utilisateur

 Finsi

 Si non

 Affichage message d'Alerte (renseignement de toutes les zone obligatoire)

 Fin Si

 Fermeture Connexion

RQ : Essayer d'optimiser votre code en utilisant des fonctions ou des procédures non liées.

Utilisation du Mode Déconnecté

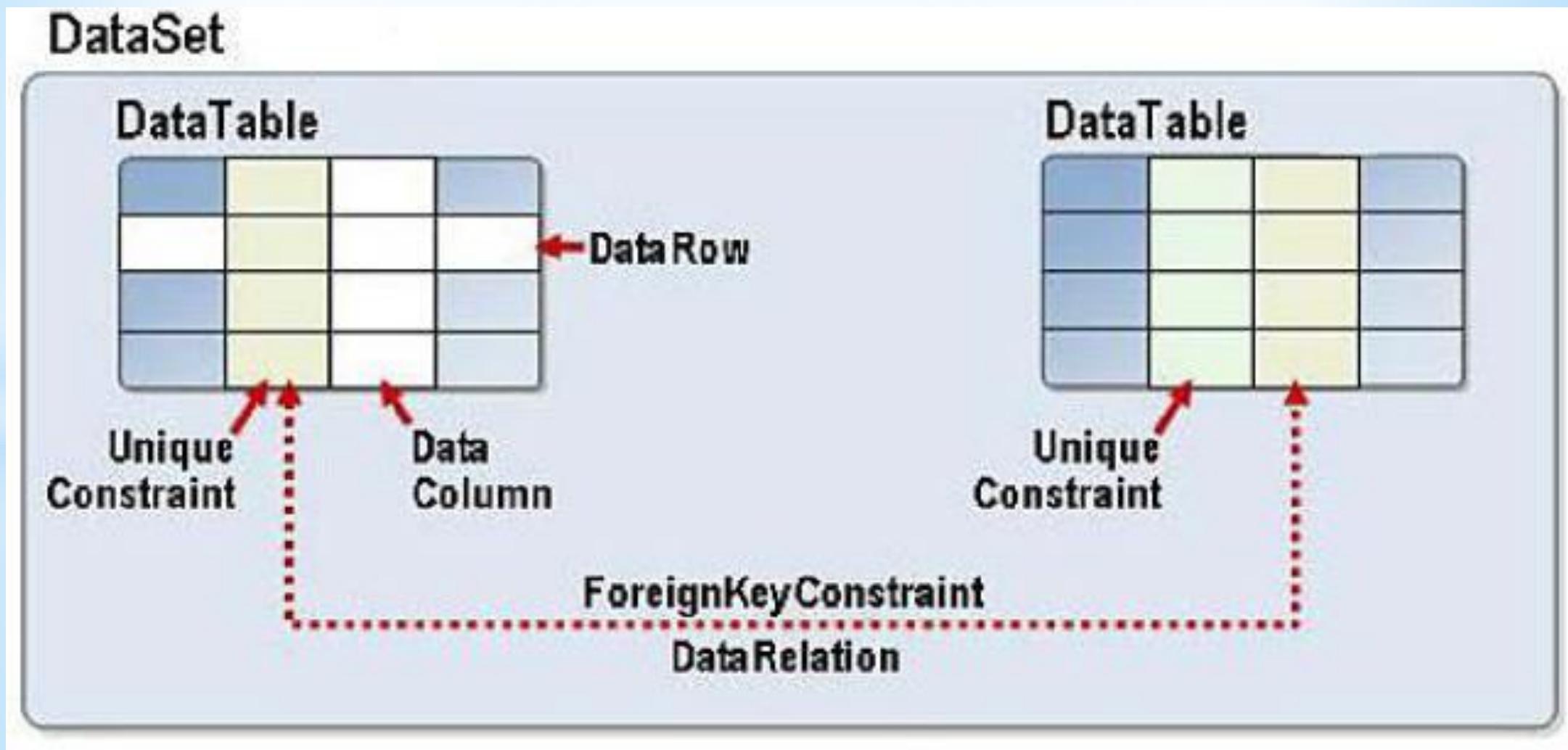
1. Introduction

Dans un **mode non connecté**, la liaison avec le serveur de base de données n'est pas permanente. Il faut donc **conserver localement les données sur lesquelles on souhaite travailler**. L'idée est de recréer, à l'aide de différentes classes, une **organisation similaire à celle d'une base de données**. Les principales **classes** sont représentées sur le schéma suivant:

- **DataSet** : C'est le **conteneur de plus haut niveau**, il joue le même rôle que la base de données.
- **DataTable** : Comme son nom l'indique, c'est l'équivalent d'une **table de base de données**.
- **DataRow** : Cette classe joue le rôle d'un **enregistrement (ligne)**.
- **DataColumn** : Cette classe remplace un **champ (colonne)** d'une table.
- **UniqueConstraint** : C'est l'équivalent de **la clé primaire d'une table**.
- **ForeignKeyConstraint** : C'est l'équivalent de la **clé étrangère**.
- **DataRelation** : Représente un lien **parent/enfant** entre deux **DataTable**.

1. Introduction

Le schéma ci-dessous représente cette organisation.



1. Remplir un **DataSet** à partir d'une base de données :

Pour pouvoir travailler localement avec les données, nous devons les rapatrier depuis la base de données dans un **DataSet**. Chaque fournisseur de données fournit une classe **DataAdapter**, assurant le dialogue entre la base de données et un **DataSet**. Tous les échanges se font par l'intermédiaire de cette classe, aussi bien de la base vers le **DataSet** que du **DataSet** vers la base pour la mise à jour des données.

Le **DataAdapter** utilisera une connexion pour contacter le Serveur et une ou plusieurs objets commandes pour le traitement des données.

1. Remplir un **DataSet** à partir d'une base de données :

a. Utilisation d'un **DataAdapter**: première chose à réaliser est de :

- Créer une instance de la classe **SQLDataAdapter**.

- Configurer le **DataAdapter** afin de lui indiquer quelles données nous souhaitons rapatrier à partir de la base de données :

- La propriété **SelectCommand** doit référencer un objet **Command**, contenant l'instruction SQL chargée de sélectionner les données. L'objet **Command** utilisé, peut également appeler une procédure stockée. La seule contrainte est que l'instruction SQL exécutée par l'objet **Command** soit une instruction **SELECT**. La classe **DataAdapter** contient également les propriétés **InsertCommand**, **DeleteCommand** et **UpdateCommand** référençant les objets **Command**, utilisés lors de la mise à jour de la base de données. Tant que nous ne souhaitons pas effectuer de mise à jour de la base, ces propriétés sont facultatives.

- La méthode **Fill** de la classe **DataAdapter** est ensuite utilisée pour remplir le **DataSet** avec le résultat de l'exécution de la commande **SelectCommand**. Cette méthode attend, comme paramètre, le **DataSet** qu'elle doit remplir et un objet **DataTable** ou une chaîne de caractères utilisée pour nommer la **DataTable** dans le **DataSet**.

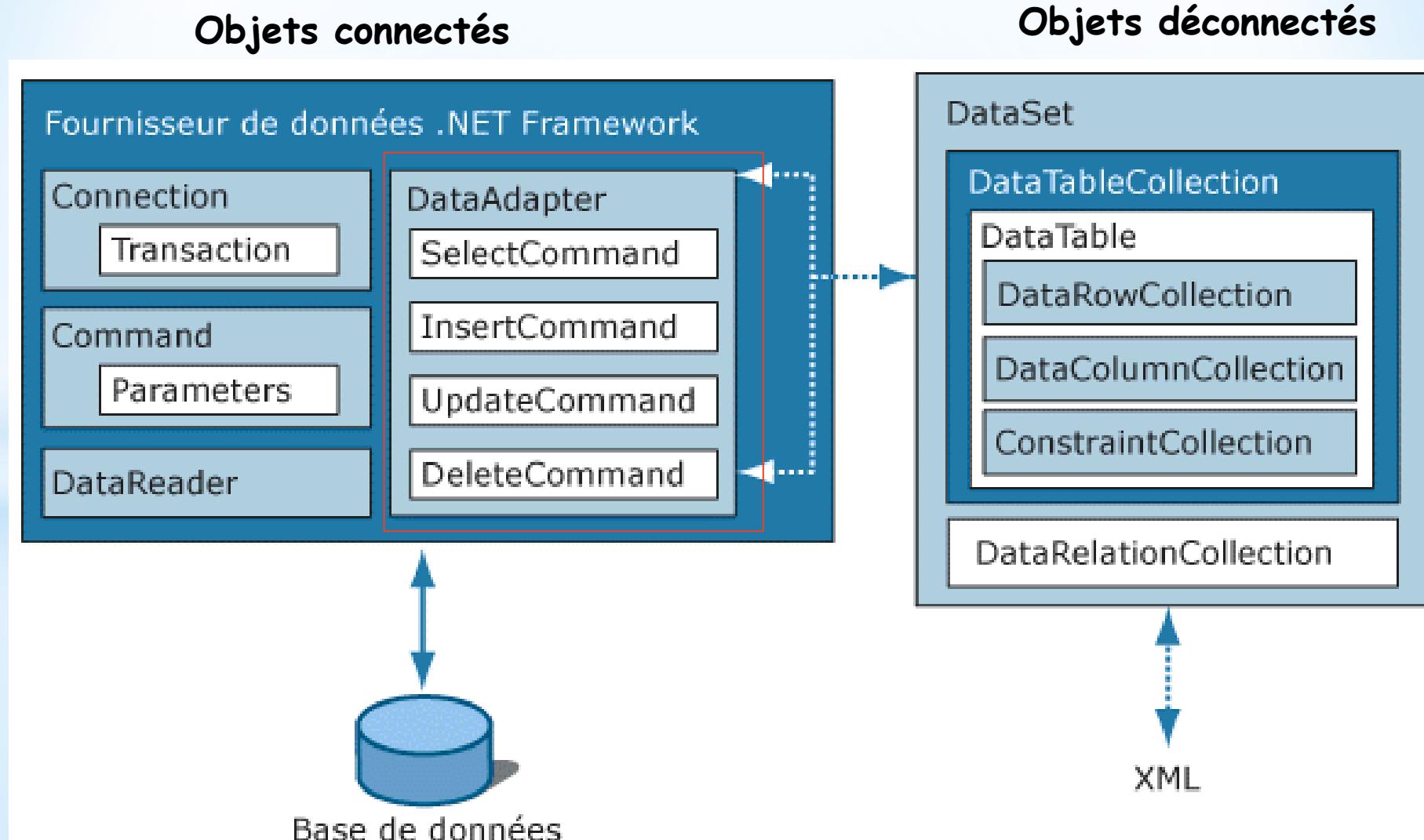
1. Remplir un DataSet à partir d'une base de données :

Le **DataAdapter** utilise, en interne, un objet **DataReader** pour obtenir le nom des et le type des champs pour créer la **DataTable** dans le **Dataset** et ensuite le remplir avec les données.

RQ : La **DataTable** et les **DataColumn** sont créés uniquement s'ils n'existent pas déjà, sinon la méthode **Fill** utilise la structure existante.

Si une **DataTable** est créée, elle est ajoutée à la collection **Tables** du **DataSet**. Le type de données des **DataColumn** est défini en fonction des mappages prévus par le fournisseur de données, entre les types de la base de données et les types .NET.

3. Utilisation d'un DataAdapter par une DataSet :



1. Remplir un **DataSet** à partir d'une base de données :

Exemple : remplir un **DataSet** avec les **code**, **nom**, **adresse** et **ville** de la **tables clients** : on suppose l'existence toujours de notre objet connexion qui comprend la chaîne de connexion vers notre base de données :



1. Remplir un **DataSet** à partir d'une base de données :

```
string chaine = "Data Source=.;Initial Catalog=northwind;Integrated Security=True";
SqlConnection ctn = new SqlConnection(chaine);
SqlCommand cmd = new SqlCommand();
cmd.Connection = ctn;
cmd.CommandText = "SELECT customer_id, contact_name, address, city from customers";
DataSet ds = new DataSet();
SqlDataAdapter da = new SqlDataAdapter();
da.SelectCommand = cmd;
da.Fill(ds, "customers");
 dgvClients.DataGridViewAutoSizeColumnsMode=DataGridViewAutoSizeColumnsMode.Fill;
dgvClients.DataSource = ds.Tables["customers"];
```

1. Remplir un DataSet à partir d'une base de données :

- o Dans ce code, la **connexion n'a pas été ouverte et fermée explicitement**. En effet, la méthode **Fill** ouvre la connexion si elle n'est pas déjà ouverte et dans ce cas, la referme également à la fin de son exécution. Toutefois, si vous avez besoin d'utiliser plusieurs fois la méthode **Fill**, il est plus efficace de gérer vous-même l'ouverture et la fermeture de connexion.
- o Un **DataSet** peut contenir plusieurs **DataTable** créées à partir de **DataAdapter différents**. Les données peuvent même provenir de bases de données différentes, voire de types de serveurs différents.
- o Lorsque le **DataAdapter** construit la **DataTable**, les noms des champs de la base sont utilisés pour nommer les **DataColumn**. Il est possible de personnaliser ces noms en créant des objets **DataTableMapping** et en les ajoutant à la collection **TableMappings** du **DataAdapter**.

1. Remplir un DataSet à partir d'une base de données :

- o Ces objets **DataTableMapping** contiennent eux mêmes des objets **DataColumnMapping** utilisés par la méthode **Fill**, comme traducteurs entre les noms des champs dans la base et les noms des **Datacolumn** dans le **DataSet**.
- o Dans ce cas, lors de l'appel de la méthode **Fill** nous devons lui indiquer le nom du **DataTableMapping** à utiliser. Si, pour un ou plusieurs champs, il n'y a pas de mappage disponible, alors le nom du champ dans la base est utilisé comme nom pour la **DataColumn** correspondante.

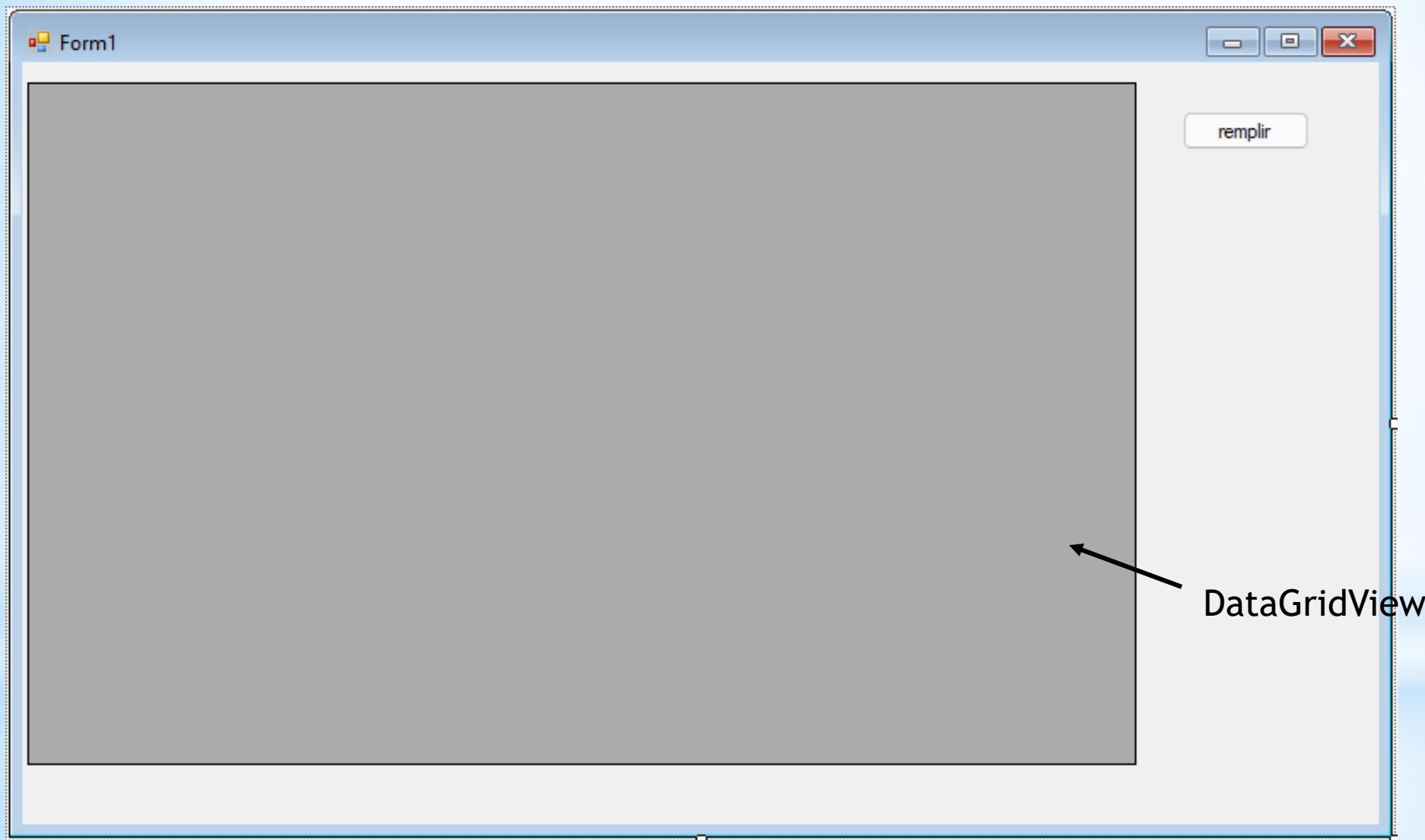
1. Remplir un **DataSet** à partir d'une base de données :

Exemple : utiliser cette technique pour traduire les champs et le nom de la table **Customers** et affiche le nom des **Datacolumn** du **DataTable**

```
string chaine = "Data Source=.;Initial Catalog=northwind;Integrated Security=True";
SqlConnection ctn = new SqlConnection(chaine);
SqlCommand cmd = new SqlCommand();
cmd.Connection = ctn;
cmd.CommandText = "SELECT customer_id,contact_name,address, city from customers";
DataSet ds = new DataSet();
SqlDataAdapter da = new SqlDataAdapter();
da.SelectCommand = cmd;
DataTableMapping mappage = new DataTableMapping("customers","clients");
mappage.ColumnMappings.Add("customer_id", "codeClient");
mappage.ColumnMappings.Add("contact_name", "nom");
mappage.ColumnMappings.Add("address", "adresse");
mappage.ColumnMappings.Add("city", "ville");
da.TableMappings.Add(mappage);
da.Fill(ds, "customers");
foreach(DataColumn dc in ds.Tables["clients"].Columns)
{
    Console.WriteLine(dc.ColumnName + "\t\t");
}
Console.WriteLine("\n");
foreach(DataRow dr in ds.Tables["Clients"].Rows)
{
    foreach (var dc in dr.ItemArray)
    {
        Console.WriteLine(dc.ToString() + "\t\t");
    }
    Console.WriteLine();
}
```

Nous obtenons à l'affichage :
CodeClient Nom Adresse Ville

Même exemple avec le mode graphique en utilisant la forme suivante :



Voici le code dans l'événement click du bouton:

```
string chaine = "Data Source=.;Initial Catalog=northwind;Integrated Security=True";
SqlConnection ctn = new SqlConnection(chaine);
SqlCommand cmd = new SqlCommand();
cmd.Connection = ctn;
cmd.CommandText = "SELECT customer_id,contact_name,address, city from customers";
DataSet ds = new DataSet();
SqlDataAdapter da = new SqlDataAdapter();
da.SelectCommand = cmd;
DataTableMapping mappage = new DataTableMapping("customers", "clients");
mappage.ColumnMappings.Add("customer_id", "codeClient");
mappage.ColumnMappings.Add("contact_name", "nom");
mappage.ColumnMappings.Add("address", "adresse");
mappage.ColumnMappings.Add("city", "ville");
da.TableMappings.Add(mappage);
da.Fill(ds, "customers");
dgvClients.AutoSizeColumnsMode=DataGridViewAutoSizeColumnsMode.Fill;
dgvClients.DataSource = ds.Tables["clients"];
```

b. Ajout de contraintes existantes à un DataSet :

La méthode **Fill** ne fait que transférer, vers le **DataSet**, les données en provenance de la base. Bien souvent, des **contraintes de clés primaires** sont utilisées dans la base de données et, par défaut, la méthode **Fill** ne les rapatrie pas dans le **DataSet**.

Pour pouvoir récupérer ces **contraintes** dans le **DataSet**, il y a deux solutions possibles:

- o Modifier la propriété **MissingSchemaAction** du **DataAdapter** avec la valeur **MissingSchemaAction.AddWithKey**.

Exemple: `da.MissingSchemaAction= MissingSchemaAction.AddWithKey`

- o Procéder en deux étapes en appelant d'abord la méthode **FillSchema** du **DataAdapter** pour créer la structure complète de la **DataTable**, puis ensuite appeler la méthode **Fill** pour remplir la **DataTable** avec les données.

Exemple :

```
da.FillSchema(ds, SchemaType.Mapped, "Customers");
```

```
da.Fill(ds, "Customers");
```

Le deuxième paramètre de la méthode **FillSchema** indique si le mappage doit être pris en compte ou si les informations issues de la base sont utilisées.

b. Ajout de contraintes existantes à un DataSet :

Il est important d'ajouter les contraintes de clés primaires car la méthode Fill va se comporter différemment si elles existent ou pas.

- Si les contraintes existent au niveau du DataSet, lorsque la méthode Fill importe un enregistrement depuis la base, elle vérifie s'il n'existe pas déjà une ligne avec la même valeur de clé primaire dans la DataTable. Si c'est le cas, elle met uniquement à jour les champs de la ligne existante. Si, par contre, il n'y a pas de ligne avec une valeur de clé primaire identique, alors la ligne est créée dans la DataTable.
- S'il n'y a pas de contrainte de clé primaire sur la DataTable, la méthode Fill ajoute tous les enregistrements en provenance de la base. Il risque dans ce cas d'y avoir des doublons dans la DataTable. Ceci est particulièrement important lorsque la méthode Fill doit être appelée plusieurs fois, par exemple, obtenir les données modifiées par une autre personne dans la base de données.

2. Configurer un DataSet sans base de données

Il n'est pas nécessaire de disposer d'une base de données pour pouvoir utiliser des **DataSet**. Ils peuvent servir d'alternative à l'utilisation de tableaux pour la gestion interne des données d'une application. Dans ce cas, toutes les opérations effectuées automatiquement par le **DataAdapter** devront être réalisées manuellement par le code. Ceci inclut notamment la création des **DataTable** avec leurs **DataColumn**.

La première opération à réaliser est de créer une instance de la classe **DataTable**. Le constructeur attend, comme paramètre, le nom de la **DataTable**. Ce nom est ensuite utilisé pour identifier la **DataTable** dans la collection **Tables** du **DataSet**. Après sa création, la **DataTable** ne contient aucune structure. Nous devons donc créer une ou plusieurs **DataColumn** et les ajouter à la collection **Columns** de la **DataTable**.

Les **DataColumn** peuvent être créées, en utilisant un des constructeurs de la classe, ou automatiquement lors de l'ajout à la collection **Columns**. La première solution fournit plus de souplesse puisqu'elle permet la configuration de nombreuses propriétés de la **DataColumn** au moment de sa création. Vous devez au minimum indiquer un nom et un type de données pour la **DataColumn**.

```
DataTable col = new DataColumn("Ht", Type.GetType("decimal"));
table.Columns.Add(col);
table.Columns.Add("Tva", Type.GetType("decimal"));
```

Une **DataColumn** peut également être construite comme étant une expression basée sur une ou plusieurs autres **DataColumn**. Vous devez, dans ce cas, indiquer lors de la création de la **DataColumn**, l'expression servant au calcul de sa valeur. Le type de données généré par l'expression doit bien sûr être compatible avec le type de données de la **DataColumn**. Vous devez également être vigilant dans la conception de l'expression, en respectant la casse et en veillant à ne pas créer de référence circulaire entre les **DataColumn**.

```
table.Columns.Add("Ttc", Type.GetType("decimal"), "Ht * (1 + Tva / 100)");
```

Pour assurer l'unicité des valeurs d'une **DataColumn**, il est possible d'utiliser un type de **Datacolumn** auto incrémentés. La propriété **AutoIncrement** de cette **DataColumn** doit être positionnée sur **true**. Vous pouvez également modifier le pas d'incrémentation avec la propriété **AutoIncrementStep** et la valeur de départ avec la propriété **AutoIncrementSeed**. La valeur contenue dans cette **DataColumn** est calculée automatiquement lors de l'ajout d'une ligne à une **DataTable** en fonction de ces propriétés et des lignes existantes déjà dans la **DataTable**.

Ce type de **Datacolumn** est généralement utilisé comme clé primaire d'une **DataTable**. Vous avez la possibilité de définir la clé primaire d'une **DataTable** en fournissant à la propriété **PrimaryKey** un tableau contenant les différentes **DataColumn** devant composer la clé primaire. Les **DataColumn** concernées verront certaines de leurs propriétés automatiquement modifiées. La propriété **unique** sera positionnée sur **true** et la propriété **AllowDBNull** sur **false**. Si la clé primaire est constituée de plusieurs **DataColumn**, seule la propriété **AllowDBNull** sera modifiée sur ces **DataColumn**.

```
col = new DataColumn("Numero", Type.GetType("int"));
col.AutoIncrement = true;
col.AutoIncrementSeed = 1000;
col.AutoIncrementStep = 1;
table.Columns.Add(col);
table.PrimaryKey = new DataColumn[] {col};
```


3. Manipuler les données dans un DataSet :

a. Lecture des données :

Quelle que soit la méthode utilisée pour remplir un **DataSet**, le but de toute application est de manipuler les données présentes dans le **DataSet**. La classe **DataTable** contient de nombreuses propriétés et méthodes facilitant la manipulation des données.

■ Lecture des données à partir d'un **DataSet** :

○ Il faut tout d'abord obtenir une référence sur la **DataTable** contenant les données, puis nous pouvons parcourir la collection **Rows** de la **DataTable**. Cette collection est une instance de la classe **DataRowCollection**. Elle dispose de :

- La propriété **Item**, par défaut, permettant l'accès à une ligne particulière par un index.
- La propriété **count** : permet de connaître le nombre de lignes disponibles.
- Pour gérer le déplacement dans le jeu de résultats, vous devez le prévoir explicitement dans votre code. La méthode **GetEnumerator** met à notre disposition une instance de classe implémentant l'interface **IEnumerator**. Par cette instance de classe, nous avons accès aux méthodes **MoveNext** et **Reset** ainsi qu'à la propriété **Current**.

3. Manipuler les données dans un DataSet :

a. Lecture des données :

Ces trois éléments permettent de parcourir facilement toutes les lignes de la **DataTable**. Chaque ligne correspond à une instance de la classe **DataRow**. Cette classe possède également une propriété **Item**, par défaut, fournissant un accès aux différents champs de la **DataRow**. Chaque champ peut être obtenu par son nom ou par son index.

Exemple : affichage de la liste des Clients

```
string chaine = "Data Source=.;Initial Catalog=northwind;Integrated Security=True";
SqlConnection ctn = new SqlConnection(chaine);
SqlCommand cmd = new SqlCommand();
cmd.Connection = ctn;
cmd.CommandText = "SELECT customer_id,contact_name,address, city from customers";
DataSet ds = new DataSet();
SqlDataAdapter da = new SqlDataAdapter();
da.SelectCommand = cmd;
da.Fill(ds, "customers");
foreach(DataColumn dc in ds.Tables["customers"].Columns)
{
    Console.WriteLine(dc.ColumnName + "\t");
}
Console.WriteLine("\n");
//On récupère l'énumérateur sur les lignes de la DataTable
IEnumerator en = ds.Tables["customers"].Rows.GetEnumerator();
//On se place sur le début de la table par sécurité
en.Reset();
//on boucle tant que la méthode MoveNext nous indique qu'il reste des lignes
while (en.MoveNext())
{
    Console.WriteLine(((DataRow)en.Current)["customer_id"] + "\t" + ((DataRow)en.Current)["contact_name"] + "\t" +
    ((DataRow)en.Current)["address"] + "\t" + ((DataRow)en.Current)["city"] + "\n");
}
```

Atelier 2

TP1 : Configurer et Manipuler des données un DataSet

1. Modifier l'interface de votre Formulaire FrmAjoModModule suite au figure ci-dessous :

Ajout Modification Modules

Menu Module

Ajouter/ Supprimer Module

Nom Module :

Ajouter Supprimer Modifier Annuler Valider

<< Précédent Suivant >> Nouveau Fermer

Atelier 2

TP1 : Configurer et Manipuler des données un DataSet

2. Ajoute à Travers SSMS et le langage SQL les 4 modules suivantes :

Algorithme

Programmation Structuré

Langage SQL

Analyse MERISE

3. Ecrire le code qui permet de déclarer un DataSet, d'utiliser un objet SqlDataAdapter et configurer sa propriété SelectCommand pour remplir votre DataSet avec les données de la table Modules avec la contrainte du clé primaire.

4. Ajouter le Code aux boutons Précédent et Suivant en utilisant une référence sur la DataTable contenant les données et La propriété Item qui permettant l'accès à une ligne particulière par un index la propriété count qui permet de connaitre le nombre de lignes d'une table. Puis parcourir la collection Rows de la DataTable pour remplir votre textBox par le contenu de votre colonne.

Atelier 2

TP1 : Configurer et Manipuler des données un DataSet

Algorithme générale :

Déclaration et création objet DataSet ds (accessibilité public)

Déclaration objet SqlDataAdapter da (accessibilité public)

Déclaration objet command cmd

instancier nouveau objet da

instancier nouveau objet cmd

configurer votre cmd :

propriété connection

propriété commandText : l'instruction Sql de type Select à utiliser pour remplir votre ds)

configurer da (propriété SelectCommand)

modifier la propriété MissingSchemaAction avec la valeur AddWithKey

remplire votre ds avec les informations de la table cible (appelle à la méthode Fill)

Exemple :(utilisation de la méthode count et la propriété item de la classe DataRow pour parcourir une collection Rows d'une DataTable)

3. Manipuler les données dans un DataSet :

c. Etat et Version d'une DataRow :

La classe **DataRow** est capable de suivre les différentes modifications apportées aux données qu'elle contient. La propriété **RowState** permet de contrôler les modifications apportées à la ligne. (appelée Etat) Cinq valeurs définies dans une énumération sont possibles pour cette propriété :

- **Unchanged** : La ligne n'a pas changé depuis le remplissage du **DataSet** par la méthode **Fill** ou la validation des modifications par la méthode **AcceptChanges**.
- **Added** : La ligne a été ajoutée mais les modifications n'ont pas encore été validées par la méthode **AcceptChanges**.
- **Modified** : Un ou plusieurs champs de la ligne ont été modifiés.
- **Deleted** : La ligne a été effacée mais les modifications n'ont pas encore été validées par la méthode **AcceptChanges**.
- **Detached** : La ligne a été créée mais ne fait pas encore partie de la collection **Rows** d'une **DataTable**.

3. Manipuler les données dans un DataSet :

c. Etat et Version d'une DataRow :

Les **différentes versions d'une ligne** sont également disponibles.

Lorsque vous accédez aux valeurs contenues dans une ligne, vous pouvez spécifier la **version qui vous intéresse**. Pour cela, l'**énumération DataRowVersion** propose quatre valeurs:

- o **Current** : Version actuelle de la ligne. Cette version n'existe pas pour une ligne dont l'état est **Deleted**.
- o **Default** : Version par défaut de la ligne. Pour une ligne dont l'état est **Added**, **Modified**, **Unchanged**, cette version est équivalente à la version **Current**. Pour une ligne dont l'état est **Deleted**, cette version est équivalente à la version **Original**. Pour une ligne dont l'état est **Detached**, cette version est égale à la version **Proposed**.
- o **Original** : Version originale de la ligne. Pour une ligne dont l'état est **Added**, cette version n'existe pas.
- o **Proposed** : Version transitoire disponible pendant une opération de modification de la ligne ou pour une ligne ne faisant pas partie de la collection **Rows** d'une **DataTable**.

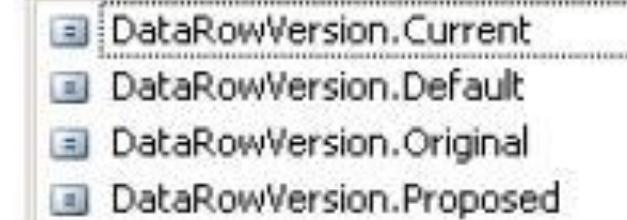
L'indication de la version désirée doit être spécifiée, lors de l'accès à un champ particulier d'une **DataRow**. Pour cela, il faut utiliser l'une des constantes précédentes, à la suite du nom ou de l'index du champ, lors de l'utilisation de la propriété **Item**, par défaut, de la **DataRow**.

3. Manipuler les données dans un DataSet :

c. Etat et Version d'une DataRow :

L'indication de la **version désirée** doit être spécifiée, lors de l'accès à un champ particulier d'une **DataRow**. Pour cela, il faut utiliser l'une des **constantes précédentes**, à la suite du nom ou de l'index du champ, lors de l'utilisation de la propriété **Item**, par défaut, de la **DataRow** .

Exemple :



```
ds.Tables("Customers").Rows(1).("ContactName",
```

Ces différentes versions seront utilisées, lors de la mise à jour de la base de données, pour par exemple gérer les accès concurrents.

3. Manipuler les données dans un DataSet :

d. Ajout de données :

L'ajout d'une ligne à une **DataTable** s'effectue simplement en ajoutant une **DataRow** à la collection **Rows** d'une **DataTable**. Il faut, au préalable, créer une instance de la classe **DataRow**. Il n'y a pas de constructeur disponible pour la classe **DataRow**. En effet, lorsque nous avons besoin d'une nouvelle instance d'une **DataRow**, nous ne voulons pas une **DataRow** quelconque mais une **DataRow** spécifique au schéma de notre **DataTable**. C'est pour cette raison que c'est à elle qu'est confié le soin de créer l'instance dont nous avons besoin par l'intermédiaire de la méthode **NewRow**.

□ Exemple d'ajout d'une lignes : **DataRow nouvelleLigne;**

```
nouvelleLigne = ds.Tables ["Customers"] .NewRow ();
```

'L'état de cette ligne est, pour l'instant, **Detached**. Nous pouvons 'ensuite ajouter des données dans cette nouvelle ligne.

```
nouvelleLinge["Name"] = "Ahmed";
```

'il nous reste à ajouter la ligne à la collection **Rows**

```
ds.tables ["customers"] .Rows .Add(nouvelleLinge);
```

'L'état de cette nouvelle ligne est maintenant **Added**

3. Manipuler les données dans un DataSet :

e. Modification de données :

La modification des données contenues dans une ligne est réalisée, simplement, en affectant aux champs correspondants, les valeurs souhaitées. Ces valeurs sont stockées dans la **version Current** de la ligne. L'état de la ligne est alors **Modified**. Cette solution présente un petit inconvénient. Si plusieurs champs d'une ligne doivent être modifiés, il peut y avoir pendant la modification, des états transitoires qui violent une ou plusieurs contraintes placées sur la **DataTable**.

C'est, par exemple, le cas s'il y a sur la **DataTable**, une **contrainte de clé primaire** placée sur deux **DataColumn**. Ceci a pour effet de déclencher une exception.

Pour pallier ce problème, nous pouvons demander temporairement l'arrêt de la vérification des contraintes pour cette ligne. La méthode **BeginEdit** passe la ligne en mode édition et suspend donc la vérification des contraintes pour cette ligne. Les valeurs affectées aux champs ne sont pas stockées dans la version **Current** de la ligne mais dans la version **Proposed**.

Lorsque toutes les modifications sont effectuées sur la ligne, vous pouvez les valider ou les annuler en appelant la méthode **EndEdit** ou la méthode **CancelEdit**.

3. Manipuler les données dans un DataSet :

e. Modification de données :

- Vous pouvez également vérifier les valeurs, en gérant l'événement `ColumnChanged` de la `DataTable`. Dans le gestionnaire d'événements, vous recevez un argument de type `DataColumnChangeEventArgs` permettant de savoir quelle `DataColumn` a été modifiée (`args.Column.ColumnName`), la valeur proposée pour cette `DataColumn` (`args.ProposedValue`) et permettant d'annuler les modifications (`args.row.CancelEdit`).
- En cas de validation avec la méthode `EndEdit`, la version `Proposed` de la ligne est recopiée dans la version `Current` et l'état de la ligne devient `Modified`. Si, par contre, vous annulez les modifications avec la méthode `CancelEdit`, la version `Current` n'est pas modifiée et l'état de la ligne est `inchangé`. Dans tous les cas, après l'appel d'une de ces deux méthodes, la vérification des contraintes est réactivée.

3. Manipuler les données dans un DataSet :

e. Modification de données :

Exemple : Code permettant de modifier code postal du client en vérifiant que celui-ci est bien numérique : p282

```
string codeClient = string.Empty;
string codePostal = string.Empty;
string chaine = "Data Source=.;Initial Catalog=northwind;Integrated Security=True";
SqlConnection ctn = new SqlConnection(chaine);
SqlCommand cmd = new SqlCommand();
cmd.Connection = ctn;
Console.WriteLine("Saisir le code client à modifier : ");
codeClient = Console.ReadLine();
cmd.CommandText = "SELECT * from customers where customer_id=@code";
SqlParameter paramCodeClient = new SqlParameter("@code", codeClient);
paramCodeClient.Direction = ParameterDirection.Input;
cmd.Parameters.Add(paramCodeClient);
DataSet ds = new DataSet();
SqlDataAdapter da = new SqlDataAdapter(cmd);
da.FillSchema(ds, SchemaType.Source, "customers");
da.Fill(ds, "customers");
DataTable table = ds.Tables["customers"];
table.Rows[0].BeginEdit();
Console.WriteLine("Saisir le nouveau code postal du client : ");
codePostal = Console.ReadLine();
table.Rows[0]["postal_code"] = codePostal;
table.Rows[0].EndEdit();
Console.WriteLine("Le nouveau code postal est : {0}", table.Rows[0]["postal_code"]);
```

3. Manipuler les données dans un DataSet :

e. Suppression de données :

Deux solutions différentes sont disponibles. Vous pouvez effacer une ligne ou supprimer une ligne. La nuance est subtile entre ces deux solutions:

- La suppression d'une donnée se fait avec la méthode **Remove** qui retire définitivement la **DataRow** de la collection **Rows** de la **DataTable**. Cette suppression est définitive.
- La méthode **Delete** ne fait que marquer la ligne pour la supprimer ultérieurement. L'état de la ligne passe à **Deleted** et ce n'est qu'au moment de la validation des modifications que la ligne est réellement supprimée de la collection **Rows** de la **DataTable**. Si les modifications sont annulées, la ligne reste dans la collection **Rows**.
- La méthode **Remove** est une méthode de la collection **Rows** (elle agit directement sur son contenu), la méthode **Delete** est une méthode de la classe **DataRow** (elle ne fait que changer une propriété de la ligne).

Exemple :

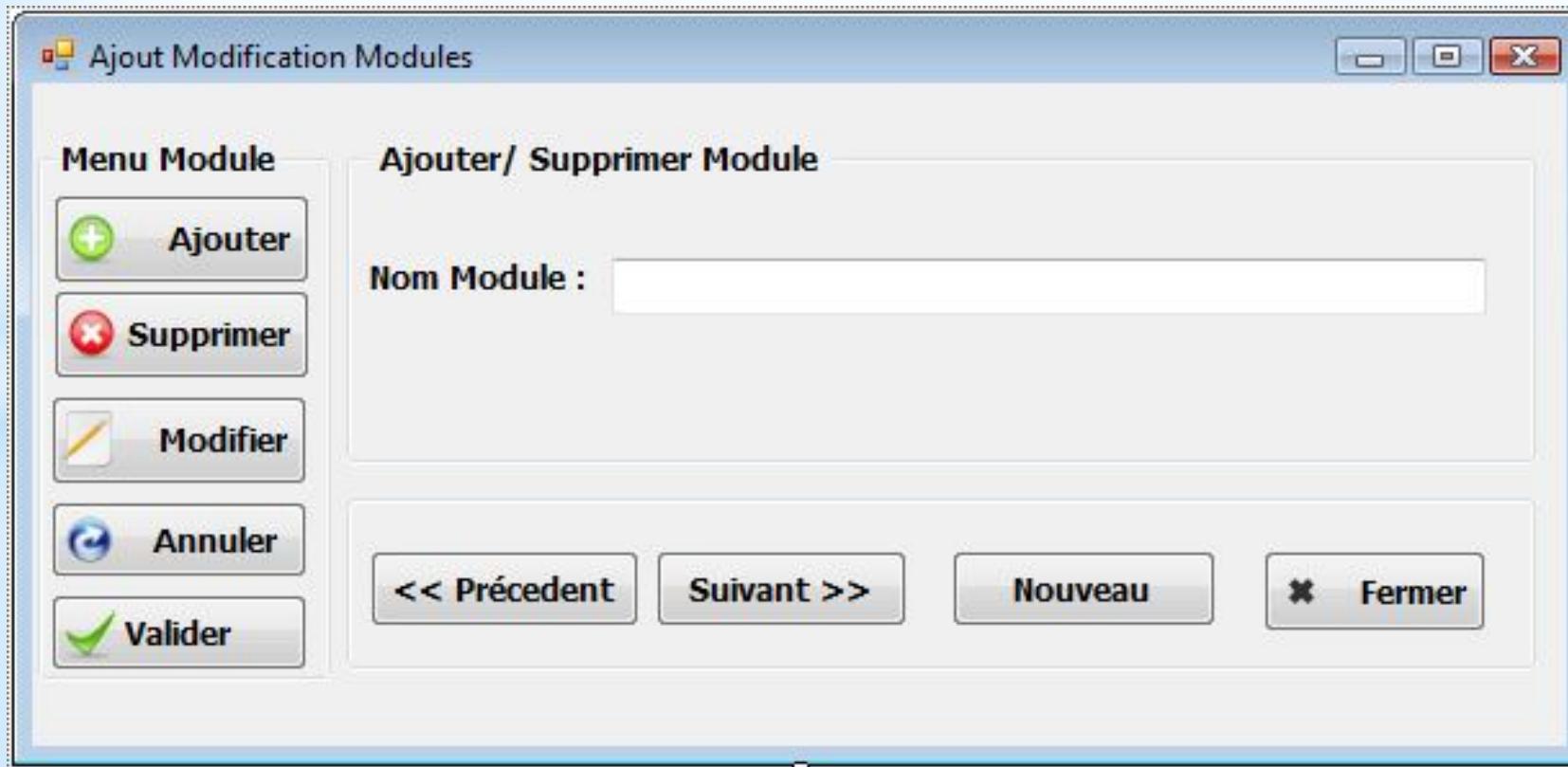
`Table.Rows[1].Delete();` 'efface la ligne

`Table.Rows.Remove[table.Rows[1]];` 'supprime la ligne

Atelier 2

TP2 : Ajout / Modification / Suppression Des données dans un DataSet

1 .A partir des différentes exemples utiliser dans le cours je vous demande d'Ecrir le code des boutons Ajouter Supprimer qui permettront respectivement d'ajouter, modifier et supprimer des lignes dans la DataTable Module de Votre DataSet ds.



2. Faites des ajouts, modification et des suppression arrêter votre programme et relancer, vérifier si vraiment les modification apportées sur votre DataTable est retenu.

3. Manipuler les données dans un DataSet :

g. Valider ou Annuler les modifications au niveau DataSet :

Jusqu'à présent, les modifications effectuées sur une ligne sont temporaires, il est encore possible de revenir à la version précédente,

ou au contraire de valider de façon définitive les modifications dans les lignes (mais encore il faut valider au niveau de la base de données).

■ Les méthodes `AcceptChanges` ou `RejectChanges` permettent respectivement la validation ou l'annulation des modifications au niveau du DataSet. Elles peuvent s'appliquer sur une `DataRow` individuelle, une `DataTable` ou un `DataSet` entier.

■ Lorsque la méthode `AcceptChanges` est exécutée, les actions suivantes sont réalisées :

o La méthode `EndEdit` est appelée implicitement pour la ligne. Si l'état de la ligne était `Added` ou `Modified`, il devient `Unchanged` et la version `Current` est recopiée dans la version `Original`. Si l'état de la ligne était `Deleted`, alors la ligne est supprimée.

■ La méthode `RejectChanges` exécute les actions suivantes :

o La méthode `CancelEdit` est appelée implicitement pour la ligne : Si l'état de la ligne était `Deleted` ou `Modified`, il revient à `Unchanged` et la version `Original` est recopiée dans la version `Current`. Si l'état de la ligne était `Added`, alors elle est supprimée.

3. Manipuler les données dans un DataSet :

g. Valider ou Annuler les modifications au niveau DataSet :

■ S'il existe des contraintes de clé étrangère, l'action de la méthode `AcceptChanges` ou `RejectChanges` est propagée aux lignes enfants en fonction de la propriété `AcceptRejectRule` de la contrainte.

4. Mettre à jour la base de données :

a. Mise à jour de la base de données :

Tout le travail effectué sur les données avec les méthodes vues précédemment est irrémédiablement perdu à la fermeture de l'application, si nous ne prenons pas soin de sauvegarder les données.

Dans la majorité des cas, les données proviennent d'une base de données, il faut donc la mettre à jour avec les modifications contenues dans un `DataSet`, une `DataTable` ou une `DataRow`.

Le `DataAdapter` a été utilisé pour remplir le `DataSet`, c'est également à lui que l'on va faire appel pour mettre à jour la base de données. Comme la méthode `Fill`, la méthode `Update` va utiliser des instructions SQL pour le dialogue avec la base de données. En fonction des besoins, elle utilisera l'instruction contenue dans la commande `InsertCommand`, `UpdateCommand` ou `DeleteCommand`.

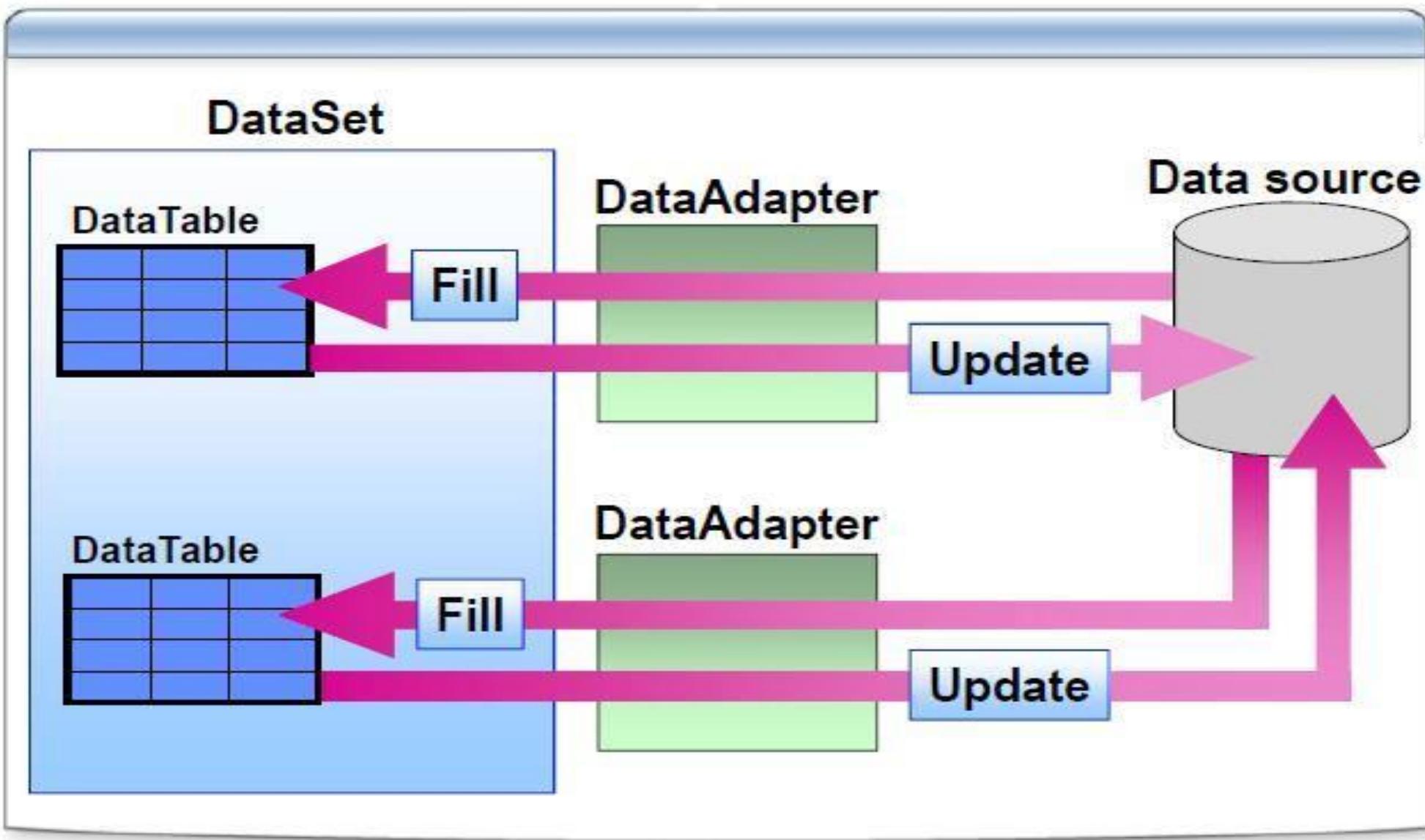
La méthode **Select** permet d'obtenir un tableau de **DataRow** correspondant à un critère spécifique. C'est ce tableau de **DataRow** qui est passé comme paramètre à la méthode **Update**.

L'exemple suivant réalise les suppressions, les modifications puis les ajouts dans la base de données.

```
DataRow[] lignes;  
// recupere les lignes supprimees et demande la mise a jour de la base  
lignes = table.Select(null, null, DataViewRowState.Deleted);  
da.Update(table);  
// recupere les lignes modifiees et demande la mise a jour de la base  
lignes = table.Select(null, null, DataViewRowState.ModifiedCurrent);  
da.Update(table);  
// recupere les lignes ajoutees et demande la mise a jour de la base  
lignes = table.Select(null, null, DataViewRowState.Added);  
da.Update(table);
```

RQ : Cet exemple suppose bien sur que les commandes **InsertCommand**, **DeleteCommand**, **UpdateCommand** soient définies au préalable.

DataAdapter



4 . Mettre à jour la base de données :

a. Mise à jour de la base de données :

Si la méthode **Update** a besoin d'une commande et qu'elle n'est pas disponible, alors une exception est générée. La méthode **Update** parcourt les lignes de la **DataTable** qu'elle doit mettre à jour et, en fonction de l'état de la ligne (**Added**, **Deleted**, **Modified**), appelle la commande **InsertCommand**, **DeleteCommand**, **UpdateCommand**.

b. Génération automatique de commandes :

Les commandes chargées de la mise à jour de la base peuvent être générées automatiquement par un objet **SqlCommandBuilder**. Pour fonctionner correctement, le **SqlCommandBuilder** a quelques exigences:

- La propriété **SelectCommand** doit être définie pour le **DataAdapter** car c'est à partir de cette instruction SQL qu'il va générer les instructions **INSERT**, **UPDATE**, **DELETE**.
- La clé primaire doit être disponible dans la **DataTable**.
- Les données ne doivent pas provenir d'une jointure entre plusieurs tables.

Si une ou plusieurs de ces exigences ne sont pas respectées, il y a déclenchement d'une exception lors de la génération des commandes.

4 . Mettre à jour la base de données :

b. Génération automatique de commandes

Les commandes sont générées en respectant les critères suivants:

❑ **InsertCommand** : Insère une ligne dans la base pour toutes les lignes dont l'état est **Added**. Tous les champs, hormis les champs **identity**, **expression** ou **TimeStamp** sont mis à jour.

❑ **UpdateCommand** : Met à jour dans la base toutes les lignes dont l'état est **Modified**. Tous les champs sont mis à jour sauf les champs **identity**, **expression** ou **TimeStamp**.

❑ **DeleteCommand** : Efface de la base toutes les lignes dont l'état est **Deleted**.

Les commandes générées sont disponibles via les méthodes **GetInsertCommand**, **GetUpdateCommand**, **GetDeleteCommand**.

4 . Mettre à jour la base de données :**b. Génération automatique de commandes :**

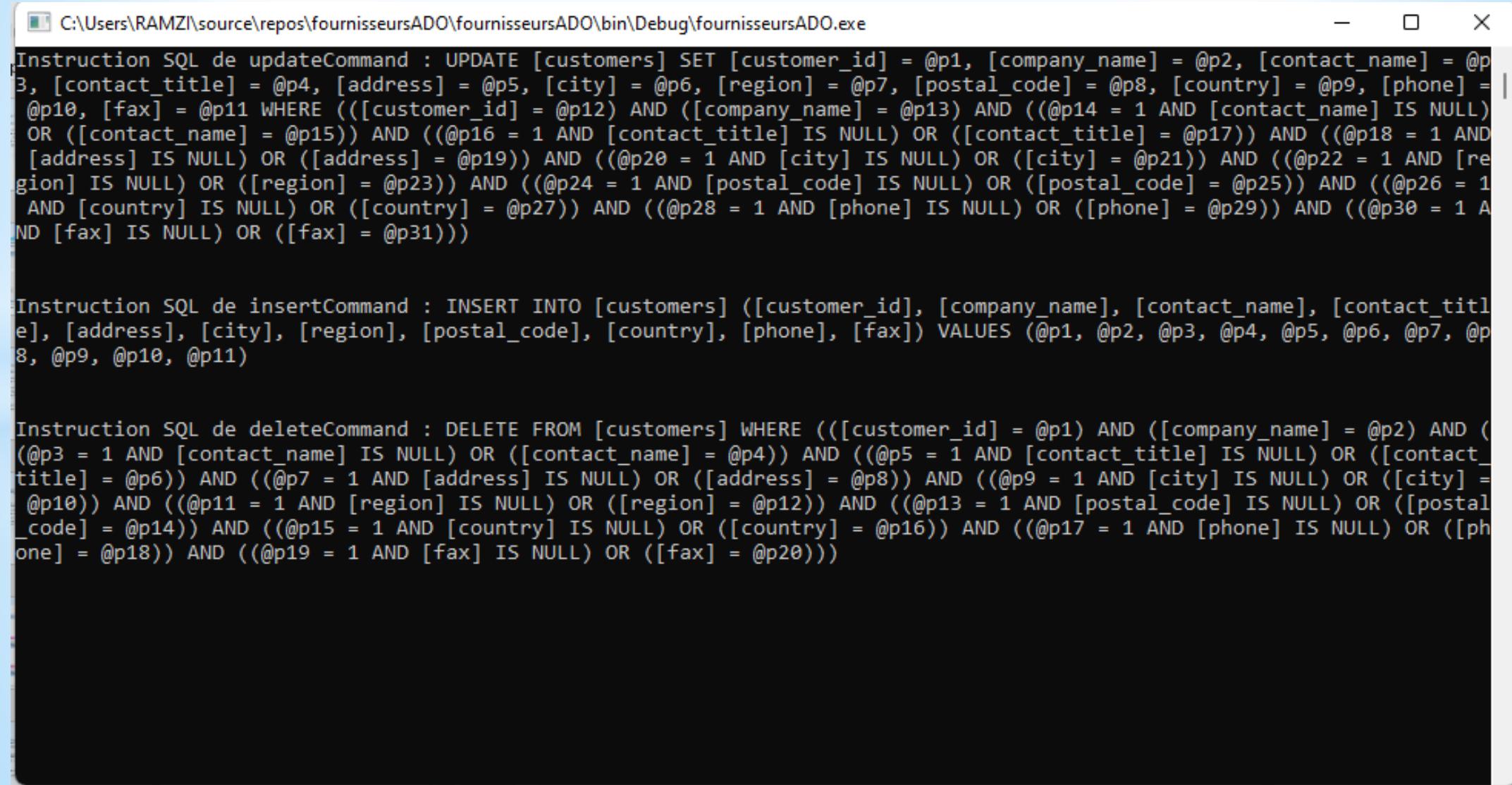
Exemple : affiche les instructions SQL des trois commandes générées automatiquement pour la table Inscrits

```
string chaine = "Data Source=.;Initial Catalog=northwind;Integrated Security=True";
SqlConnection ctn = new SqlConnection(chaine);
SqlCommand cmd = new SqlCommand("Select * from customers", ctn);
ctn.Open();
DataSet ds = new DataSet();
SqlDataAdapter da = new SqlDataAdapter(cmd);
da.Fill(ds, "customers");
DataTable table = ds.Tables["customers"];
SqlCommandBuilder bldr = new SqlCommandBuilder(da);
Console.WriteLine("Instruction SQL de updateCommand : {0} ", bldr.GetUpdateCommand().CommandText);
Console.WriteLine("Instruction SQL de insertCommand : {0}", bldr.GetInsertCommand().CommandText);
Console.WriteLine("Instruction SQL de deleteCommand : {0}", bldr.GetDeleteCommand().CommandText);
ctn.Close();
Console.ReadKey();
```

4 . Mettre à jour la base de données :

b. Génération automatique de commandes :

□ Ce code Affiche les informations suivantes :



```
C:\Users\RAMZI\source\repos\fournisseursADO\fournisseursADO\bin\Debug\fournisseursADO.exe

Instruction SQL de updateCommand : UPDATE [customers] SET [customer_id] = @p1, [company_name] = @p2, [contact_name] = @p3, [contact_title] = @p4, [address] = @p5, [city] = @p6, [region] = @p7, [postal_code] = @p8, [country] = @p9, [phone] = @p10, [fax] = @p11 WHERE (([customer_id] = @p12) AND ([company_name] = @p13) AND ((@p14 = 1 AND [contact_name] IS NULL) OR ([contact_name] = @p15)) AND ((@p16 = 1 AND [contact_title] IS NULL) OR ([contact_title] = @p17)) AND ((@p18 = 1 AND [address] IS NULL) OR ([address] = @p19)) AND ((@p20 = 1 AND [city] IS NULL) OR ([city] = @p21)) AND ((@p22 = 1 AND [region] IS NULL) OR ([region] = @p23)) AND ((@p24 = 1 AND [postal_code] IS NULL) OR ([postal_code] = @p25)) AND ((@p26 = 1 AND [country] IS NULL) OR ([country] = @p27)) AND ((@p28 = 1 AND [phone] IS NULL) OR ([phone] = @p29)) AND ((@p30 = 1 AND [fax] IS NULL) OR ([fax] = @p31)))

Instruction SQL de insertCommand : INSERT INTO [customers] ([customer_id], [company_name], [contact_name], [contact_title], [address], [city], [region], [postal_code], [country], [phone], [fax]) VALUES (@p1, @p2, @p3, @p4, @p5, @p6, @p7, @p8, @p9, @p10, @p11)

Instruction SQL de deleteCommand : DELETE FROM [customers] WHERE (([customer_id] = @p1) AND ([company_name] = @p2) AND ((@p3 = 1 AND [contact_name] IS NULL) OR ([contact_name] = @p4)) AND ((@p5 = 1 AND [contact_title] IS NULL) OR ([contact_title] = @p6)) AND ((@p7 = 1 AND [address] IS NULL) OR ([address] = @p8)) AND ((@p9 = 1 AND [city] IS NULL) OR ([city] = @p10)) AND ((@p11 = 1 AND [region] IS NULL) OR ([region] = @p12)) AND ((@p13 = 1 AND [postal_code] IS NULL) OR ([postal_code] = @p14)) AND ((@p15 = 1 AND [country] IS NULL) OR ([country] = @p16)) AND ((@p17 = 1 AND [phone] IS NULL) OR ([phone] = @p18)) AND ((@p19 = 1 AND [fax] IS NULL) OR ([fax] = @p20)))
```

4 . Mettre à jour la base de données :

b. Exemple d'appel à la méthode Update du DataAdapter :

```
string chaine = "Data Source=.;Initial Catalog=northwind;Integrated Security=True";
SqlConnection ctn = new SqlConnection(chaine);
SqlCommand cmd = new SqlCommand("Select * from customers where country = 'France'", ctn);
ctn.Open();
DataSet ds = new DataSet();
SqlDataAdapter da = new SqlDataAdapter(cmd);
da.Fill(ds, "customers");
DataTable table = ds.Tables["customers"];
// Effacer Marie Bertrand
table.Rows[7].Delete();
//Changer l'adresse de carine schmitt
table.Rows[4]["address"] = "9 rue Benjamin Franklin";
//Ajout d'un nouveau client
DataRow ligne = table.NewRow();
ligne["customer_id"] = "ENIEC";
ligne["company_name"] = "Eni Ecole Informatique";
ligne["contact_name"] = "Marcel Dupond";
ligne["contact_title"] = "Directeur";
ligne["address"] = "24 rue Crébilon";
ligne["country"] = "France";
ligne["city"] = "Nantes";
table.Rows.Add(ligne);
SqlCommandBuilder bldr = new SqlCommandBuilder(da);
da.Update(table);
```

4 . Mettre à jour la base de données :

c. Utilisation de commandes personnalisées :

L'utilisation de **commandes personnalisées** permet de **choisir le type d'action effectué** lors de la mise à jour de la base de données. Par exemple, l'effacement d'une ligne peut se traduire par l'affectation d'une valeur particulière à un champ de l'enregistrement. Dans ce cas, l'instruction **SQL exécutée dans la DeleteCommand** sera une instruction **UPDATE** plutôt qu'une instruction **DELETE**.

Exemple : le code suivant crée une commande personnalisée pour la suppression :

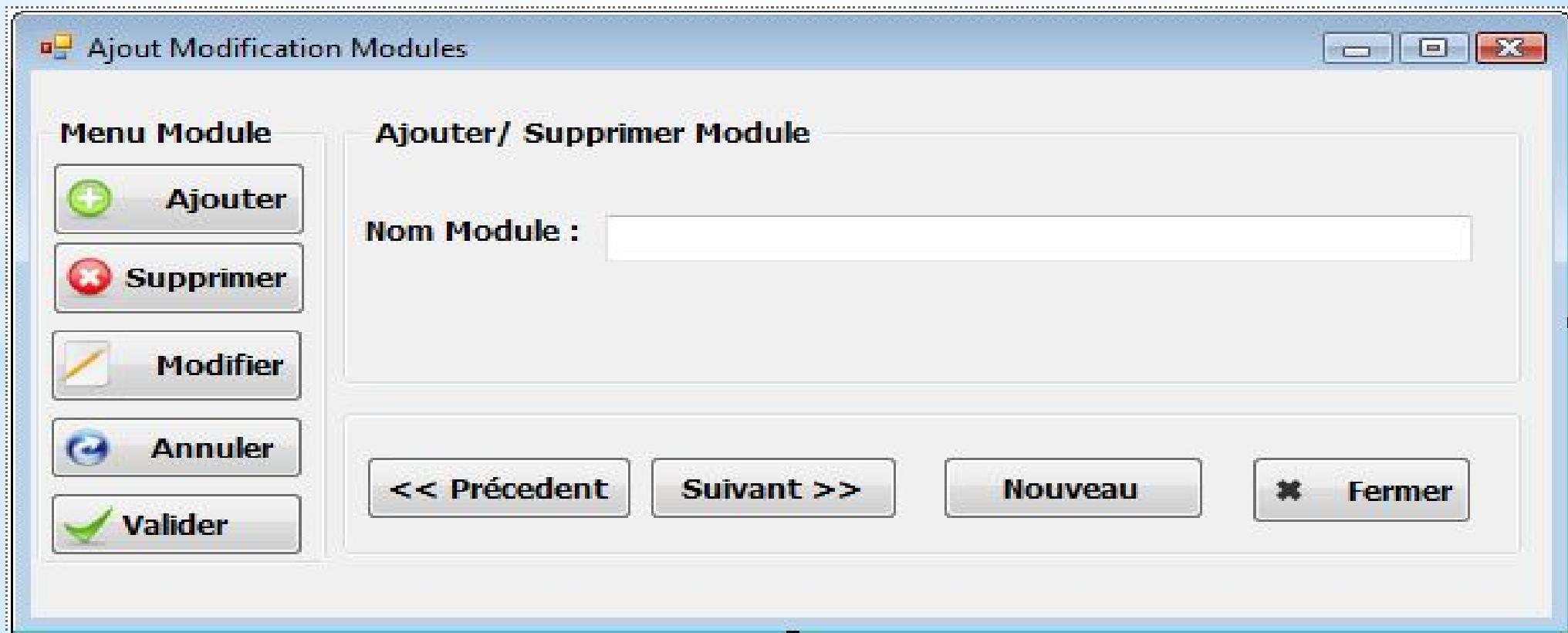
```
delCmd = New SqlCommand  
delCmd.Connection = ctn  
delCmd.CommandText = " UPDATE Customers set Archive=1 where CustomerID=@num"  
codeClient = New SqlParameter("@num", SqlDbType.NChar, 5, ParameterDirection.  
delCmd.Parameters.Add(codeClient)  
da.DeleteCommand = delCmd  
bldr = New SqlCommandBuilder(da)
```

RQ : Les commandes personnalisées sont **compatibles avec les commandes générées automatiquement** par le **SqlCommandBuilder** puisque celui-ci ne génère une commande que si la propriété **InsertCommand**, **DeleteCommand** ou **UpdateCommand** est égale à **nothing** dans le **DataAdapter**. Si une commande existe, elle n'est pas remplacée par le **SqlCommandBuilder**.

Atelier 2

TP3 : Génération automatique de commandes et mise à jour de la base de données

1 . On vous demande de programmer la validation des différentes opérations d'ajout, modification et suppression des données dans un premier temps au niveau de votre dataSet, avec les boutons Valider et Annuler. Gérer le déplacement entre les différentes lignes de votre DataTable en cas d'annulation.



2. Compléter le code des boutons Valider et Supprimer pour faire la mise à jour de la base de données (méthode Update de votre da DataAdapter) avec la génération automatiquement des différentes commandes SQL à travers un objet SqlCommandBuilder (dans remplirDataSet).

5. Filtrer et trier des données dans un DataSet

a. Introduction :

Il est fréquent d'avoir besoin de limiter la quantité de données visibles dans une **DataTable** ou encore de modifier l'ordre des lignes.

La première solution qui vient à l'esprit est de **recréer une requête SQL** avec une restriction ou une **clause ORDERBY**. C'est oublier que nous sommes dans un mode de fonctionnement déconnecté et qu'il est souhaitable de limiter les accès à la base, voire pire, que la base n'est pas disponible. Nous devons donc n'utiliser que les données disponibles, en faisant attention à ne pas en perdre.

La classe **DataView** va nous être très utile pour solutionner nos problèmes. Cette classe va nous servir à modifier la vision des données dans la **DataTable** sans risque pour les données elles mêmes.

Il peut y avoir plusieurs **DataView** pour une même **DataTable**, elles correspondent à des points de vue différents de la **DataTable**.

Pratiquement toutes les opérations réalisables sur une **DataTable** le sont aussi par l'intermédiaire d'une **DataView**. Deux solutions sont disponibles pour obtenir une **DataView** :

- **Créer une instance par l'un des constructeurs.**
- **Utiliser l'instance, par défaut, fournie par la propriété DefaultView.**

5. Filtrer et trier des données dans un DataSet

a. Introduction :

■ Le premier constructeur utilisable attend simplement comme paramètre la **DataTable** à partir de laquelle est générée la **DataView**.

Dans ce cas, il n'y a aucun filtre ni aucun tri effectué sur les données visibles par la **DataView**.

■ Le deuxième constructeur permet de spécifier un filtre, un ordre de tri et la version des lignes concernées. Pour être visibles dans la **DataView**, les lignes devront correspondre à tous ces critères. Les différents critères peuvent aussi être modifiés par trois propriétés :

o **RowFilter** : Cette propriété accepte une chaîne de caractères représentant la condition devant être remplie pour qu'une ligne soit visible. Cette condition a une syntaxe tout à fait similaire aux conditions d'une clause WHERE. Les opérateurs And et Or peuvent également être utilisés pour associer plusieurs conditions.

b. **Filtrer et trier des données** : L'exemple suivant affiche le nom des clients commerciaux ou directeurs de vente en France :

```
DataTable table;
SqlConnection ctn = new SqlConnection();
ctn.ConnectionString = "Data Source=localhost;Initial Catalog - Northwind; Integrated Security = true";
ctn.Open();
SqlCommand cmd = new SqlCommand();
cmd.Connection = ctn;
cmd.CommandText = " SELECT * from Customers";
DataSet ds = new DataSet();
SqlDataAdapter da = new SqlDataAdapter(cmd);
da.Fill(ds, "Clients");
table = ds.Tables["Clients"];
table.DefaultView.RowFilter = "Country='France' and (contact_Title='Sales Agent' or contact_Title='Sales Manager')";
foreach (DataRowView ligne in table.DefaultView)
{
    Console.WriteLine("nom : {0}",ligne["ContactName"]);
}
```

5. Filtrer et trier des données dans un DataSet

b. Filtrer et trier des données :

o **Sort** : Cette propriété accepte, elle aussi, une chaîne de caractères représentant le ou les critères utilisés pour le tri. La syntaxe est équivalente à celle de la clause ORDER BY.

Exemple : L'exemple suivant affiche les clients triés par pays puis par nom, pour un même pays :

```
//on annule le filtre précédent
table.DefaultView.RowFilter = "";
//toutes les lignes sont maintenant visibles
//on ajoute un critère de tri
table.DefaultView.Sort = "Country ASC,Contact_Name ASC";
foreach (DataRowView ligne in table.DefaultView)
{
    Console.WriteLine("Pays : {0} \t nom:{1}",ligne["Country"],ligne["Contac_tName"]);
}
```

o **RowStateFilter** : Cette propriété détermine l'état des lignes et quelle version de la ligne est visible dans la DataView. Huit possibilités sont disponibles :

5. Filtrer et trier des données dans un DataSet

b. Filtrer et trier des données :

o Huit possibilités sont disponibles :

CurrentRows : Présente la **version Current** de toutes les lignes ajoutées, modifiées ou inchangées.

- **Added** : Présente la **version Current** de toutes les lignes ajoutées.
- **Deleted** : Présente la **version Original** de toutes les lignes **effacées**.
- **ModifiedCurrent** : Présente la **version Current** de toutes les lignes modifiées.
- **ModifiedOriginal** : Présente la **version Original** de toutes les lignes modifiées.
- **None** : Aucune ligne.
- **OriginalRows** : Présente la **version Original** de toutes les lignes modifiées, supprimées ou inchangées.
- **Unchanged** : Présente la **version Current** de toutes les lignes inchangées.

5. Filtrer et trier des données dans un DataSet**b. Filtrer et trier des données :**

Exemple : L'exemple suivant supprime deux lignes et les affiche par l'intermédiaire d'un filtre :

```
// on supprime deux lignes
table.Rows[2].Delete();
table.Rows[5].Delete();
// on annule le filtre
table.DefaultView.RowFilter = "";
// on affiche la version originale des lignes supprimées
table.DefaultView.RowStateFilter = DataViewRowState.Deleted;
foreach (DataRowView ligne in table.DefaultView)
{
    Console.WriteLine("Pays : {0} \t nom: {1}", ligne["Country"],ligne["Contact_Name"]);
}
```

5. Filtrer et trier des données dans un DataSet

c. Rechercher des données :

La recherche peut s'effectuer avec les deux méthodes **Find** et **FindRows**. Pour que ces deux méthodes fonctionnent, il est impératif d'avoir au préalable trié les données avec la propriété **Sort**.

o **Find** : Cette méthode retourne l'**index** de la première ligne correspondant au critère de recherche. Si aucune ligne n'est trouvée, cette méthode **retourne -1**.

Elle attend comme paramètre la valeur recherchée. Cette valeur est recherchée dans le champ utilisé comme **critère de tri**. Si le critère de tri est composé de plusieurs champs, il faut passer à la méthode **Find** un tableau d'objets contenant les valeurs recherchées pour chaque champ du critère de tri dans l'ordre d'apparition dans la propriété **Sort**.

Cette méthode est souvent utilisée pour rechercher une ligne à partir de la clé primaire

5. Filtrer et trier des données dans un DataSet

c. Rechercher des données :

Exemple :

```
SqlCommand cmd;
SqlConnection ctn;
DataSet ds;
SqlDataAdapter da;
DataTable table;
string codeClient;
int index;
ctn = new SqlConnection();
ctn.ConnectionString = "Data Source=localhost;Initial Catalog - Northwind;
Integrated Security=true";
ctn.Open();
cmd = new SqlCommand();
cmd.Connection = ctn;
cmd.CommandText = " SELECT * from Customers";
ds = new DataSet();
da = new SqlDataAdapter(cmd);
```

```
da.Fill(ds, "Clients");
table = ds.Tables["Clients"];
Console.WriteLine("saisir le code client: ");
codeClient = Console.ReadLine();
table.DefaultView.Sort = "Customer_id ASC";
index = table.DefaultView.Find(codeClient);
if (index == -1)
{
    Console.WriteLine("il n'y a pas de client avec ce code");
}
else
{
    Console.WriteLine("le code {0} correspond au client {1}", codeClient,
table.DefaultView[index]["ContactName"]);
}
Console.ReadLine();
```

5. Filtrer et trier des données dans un DataSet

c. Rechercher des données :

FindRows : Cette méthode recherche toutes les lignes correspondant au critère de recherche et retourne ces lignes sous forme d'un **tableau de DataRowView**.

```
SqlCommand cmd;
SqlConnection ctn;
DataSet ds;
SqlDataAdapter da;
DataTable table;
string pays;
string ville;
DataRowView[] lignesTrouvees;
object[] criteres;
ctn = new SqlConnection();
ctn.ConnectionString = "Data Source=localhost;Initial Catalog - Northwind; Integrated Security = true";
ctn.Open();
cmd = new SqlCommand();
cmd.Connection = ctn;
cmd.CommandText = " SELECT * from Customers";
ds = new DataSet();
da = new SqlDataAdapter(cmd);
```

```
da.Fill(ds, "Clients");
table = ds.Tables["Clients"];
Console.Write("saisir le Pays : ");
pays = Console.ReadLine();
Console.Write("saisir la ville: ");
ville = Console.ReadLine();
table.DefaultView.Sort = "Country ASC,City ASC";
criteres = (new object[] {pays, ville});
lignesTrouvees = table.DefaultView.FindRows(criteres);
if (lignesTrouvees.Length == 0)
{
    Console.WriteLine("il n'y a pas de client correspondant");
}
else
{
    Console.WriteLine("les clients suivants correspondent ");
    foreach (DataRowView ligne in lignesTrouvees)
    {
        Console.WriteLine("Nom :(0)", ligne["ContactName"]);
    }
}
Console.ReadKey();
```

6. Les transactions

Les transactions permettent de regrouper dans une entité, un ensemble de commandes SQL. Ce regroupement va garantir que, si l'une des instructions contenues dans la transaction échoue, la base de données pourra retrouver son état initial. L'exemple classique est le virement d'un compte bancaire à un autre. Imaginez que vous ayez, dans votre base de données, une table pour les comptes des particuliers et une table pour les comptes des entreprises. Le transfert d'un compte entreprise vers un compte particulier (le paiement de votre salaire) pourrait s'effectuer avec les instructions suivantes :

```
SqlCommand cmdPart;
SqlCommand cmdEnt;
SqlConnection ctn;
SqlParameter numParticulier;
SqlParameter numEntreprise;
SqlParameter montantPart;
SqlParameter montantEnt;
ctn = new SqlConnection();
ctn.ConnectionString = "Data Source=localhost;Initial Catalog = Northwind;
Integrated Security = true";
ctn.Open();
```

6. Les transactions

```
cmdEnt = new SqlCommand();
cmdEnt.Connection = ctn;
cmdEnt.CommandText = "Update ComptesEntreprise set solde=solde-@montant where
numCompte = @numCompte";
numEntreprise = new SqlParameter("@numCompte", SqlDbType.Int);
numEntreprise.Value = 1234;
cmdEnt.Parameters.Add(numEntreprise);
montantEnt = new SqlParameter("@montant", SqlDbType.Decimal);
montantEnt.Value = 3000;
cmdEnt.Parameters.Add(montantEnt);
cmdEnt.ExecuteNonQuery();
```

6. Les transactions

```
cmdPart = new SqlCommand();
cmdPart.Connection = ctn;
cmdPart.CommandText = "Update ComptesParticulier set solde = solde + @montant
where numCompte = @numCompte";
numParticulier = new SqlParameter("@numCompte", SqlDbType.Int);
numParticulier.Value = 5678;
cmdPart.Parameters.Add(numParticulier);
montantPart = new SqlParameter("@montant", SqlDbType.Decimal);
montantPart.Value = 3000;
cmdPart.Parameters.Add(montantPart);
cmdPart.ExecuteNonQuery();
ctn.Close();
Console.ReadKey();
```

6. Les transactions

Que se passe-t-il, si pendant l'exécution de ce code le serveur de base de données devient indisponible à cause d'un problème réseau par exemple ?

L'opération de débit peut avoir été effectuée alors que l'opération de crédit n'a pas pu s'exécuter correctement. Il risque donc d'y avoir un gros problème dans le fonctionnement de l'application (et pour le paiement de votre salaire). Les transactions vont permettre de résoudre ce problème, en regroupant l'exécution d'instructions SQL pour garantir qu'elles seront toutes exécutées ou qu'aucune ne sera exécutée.

Les transactions sont gérées au niveau de la connexion, c'est donc elle qui va nous permettre de démarrer une transaction. La méthode **BeginTransaction** nous retourne une instance de la classe **SqlTransaction**. Pour chaque exécution de commande, nous pouvons alors indiquer si l'exécution doit se passer dans le contexte de la transaction ou à l'extérieur. À la fin du traitement, nous pouvons valider toutes les instructions confiées à la transaction ou au contraire les annuler toutes. La méthode **Commit** valide la transaction alors que la méthode **RollBack** l'annule.

Pour sécuriser le code précédent, nous pourrions utiliser la version suivante :

```
SqlCommand cmdPart;
SqlCommand cmdEnt;
SqlConnection ctn;
SqlParameter numParticulier;
SqlParameter numEntreprise;
SqlParameter montantPart;
SqlParameter montantEnt;
SqlTransaction trans;
ctn = new SqlConnection();
ctn.ConnectionString = "Data Source=.;Initial Catalog=Northwind;Integrated Security = true";
ctn.Open();
trans = ctn.BeginTransaction();
try
{
    cmdEnt = new SqlCommand();
    cmdEnt.Connection = ctn;
    cmdEnt.CommandText = "Update ComptesEntreprise set solde=solde-@montant where numCompte =
    @numCompte";
    numEntreprise = new SqlParameter("@numCompte", SqlDbType.Int);
    numEntreprise.Value = 1234;
    cmdEnt.Parameters.Add(numEntreprise);
    montantEnt = new SqlParameter("@montant", SqlDbType.Decimal);
    montantEnt.Value = 3000;
    cmdEnt.Parameters.Add(montantEnt);
```

```
// place l'execution de la commande dans la transaction
cmdEnt.Transaction = trans;
cmdEnt.ExecuteNonQuery();
cmdPart = new SqlCommand();
cmdPart.Connection = ctn;
cmdPart.CommandText = "Update ComptesParticuliers set solde = solde + @montant where numCompte =
@numCompte";
numParticulier = new SqlParameter("@numCompte", SqlDbType.Int);
numParticulier.Value = 5678;
cmdPart.Parameters.Add(numParticulier);
montantPart = new SqlParameter("@montant", SqlDbType.Decimal);
montantPart.Value = 3000;
cmdPart.Parameters.Add(montantPart);
cmdPart.Transaction = trans;
cmdPart.ExecuteNonQuery();
trans.Commit();
}
catch (Exception ex)
{
    trans.Rollback();
    Console.WriteLine("toutes les operations ont ete annulees");
}
ctn.Close();
```

6. Les transactions

la connexion est interrompue, l'instruction **RollBack** ou **commit** ne pourra pas être acheminée vers le serveur. Dans ce cas, le serveur prend l'initiative d'exécuter un **RollBack** sur toutes les transactions en cours, si la connexion avec le client est perdue.

6. Liaison de quelques Contrôles aux données :

a. Le contrôle DataGridView :

Le **DataGridView** est un élément graphique qui va nous permettre d'afficher des données récupérées sur une Base de Données. Son utilisation est très simple et le rendu final est agréable.



The screenshot shows a Windows application window titled "DataGridView". Inside, there is a "DataGrid" control containing a table with the following data:

	ID	Titre	Artiste	Album	Classement
▶	1	Don't Cry	Guns N' Roses	Album inconnu	
	2	Nothing Else Mat...	Metallica	Metallica	9
	3	Jeune Et Con	Saez	Jours Etranges	
	4	Un Jour En France	Noir Désir	Album inconnu	
	5	Hotel California	Eagles	Album inconnu	
*					

Le DataGrid va avoir la même structure qu'une **DataTable** ou un **DataView**, c'est à dire qu'il possède des attributs de type "**Column**", "**Row**" et "**Item**".

Le contrôle **DataGridView** fournit une table personnalisable pour l'affichage des données. La classe **DataGridView** permet la personnalisation des cellules, des lignes, des colonnes et des bordures grâce à l'utilisation de propriétés telles que :

DefaultCellStyle, **ColumnHeadersDefaultCellStyle**, **CellBorderStyle** et **GridColor**.

6. Liaison de quelques Contrôles aux données :

a. Le contrôle DataGridView :

Vous pouvez utiliser un contrôle **DataGridView** pour afficher des données avec ou sans source de données sous-jacente. Sans spécifier une source de données, vous pouvez créer des colonnes et des lignes qui contiennent des données et les ajouter directement au **DataGridView**.

```
DataGridView test_dataGridView=new DataGridView();
this.Controls.Add(test_dataGridView);
//Ajout de 5 colonnes
test_dataGridView.ColumnCount = 5;
//paramétrage des entêtes
test_dataGridView.ColumnHeadersDefaultCellStyle.BackColor = Color.Navy;
test_dataGridView.ColumnHeadersDefaultCellStyle.ForeColor = Color.White;
test_dataGridView.ColumnHeadersDefaultCellStyle.Font = new Font(test_dataGridView.Font,
FontStyle.Bold);
//on positionne la grille
test_dataGridView.Name = "test_dataGridView";
test_dataGridView.Location = new Point(8, 8);
test_dataGridView.Size = new Size(100, 60);
```

6. Liaison de quelques Contrôles aux données :

a. Le contrôle DataGridView :

```
//paramétrage du comportement du DataGridView
test_dataGridView.AutoSizeRowsMode = DataGridViewAutoSizeRowsMode.DisplayedCellsExceptHeaders;
test_dataGridView.ColumnHeadersBorderStyle = DataGridViewHeaderBorderStyle.Single;
test_dataGridView.CellBorderStyle = DataGridViewCellBorderStyle.Single;
test_dataGridView.GridColor = Color.Black;
test_dataGridView.RowHeadersVisible = true;
//on donne le nom des colonnes
test_dataGridView.Columns[0].Name = "IdCLI";
test_dataGridView.Columns[1].Name = "Nom";
test_dataGridView.Columns[2].Name = "AdrCLI";
test_dataGridView.Columns[3].Name = "TelCLI";
test_dataGridView.Columns[4].Name = "CredCLI";
test_dataGridView.SelectionMode = DataGridViewSelectionMode.FullRowSelect;
test_dataGridView.MultiSelect = false;
test_dataGridView.Dock = DockStyle.Fill;
```

6. Liaison de quelques Contrôles aux données :

a. Le contrôle DataGridView :

```
//Création d'un tableau de 5 strings pour chaque ligne
string[] row0 = {"1", "Ali", "Route de la révolution Bizerte", "0000000" };
string[] row1 = { "2", "Mohamed", "Rue 14 Tubis", "0000000" };
string[] row2 = { "3", "Salah", "32 Cité Ettadhamon tunis 1001", "0000001" };
string[] row3 = { "4", "Jamal", "Mannouba", "665544666" };
string[] row4 = { "5", "Sonia", "13 avenue 7 novembre korba", "65432654" };
string[] row5 = { "6", "Wiam", "32 cité elkawnia nabeul 8000", "87654321" };
string[] row6 = { "7", "Lina", "douar hicher", "09876543" };
//Ajout de ligne
test_dataGridView.Rows.Add(row0);
test_dataGridView.Rows.Add(row1);
test_dataGridView.Rows.Add(row2);
test_dataGridView.Rows.Add(row3);
test_dataGridView.Rows.Add(row4);
test_dataGridView.Rows.Add(row5);
test_dataGridView.Rows.Add(row6);
```

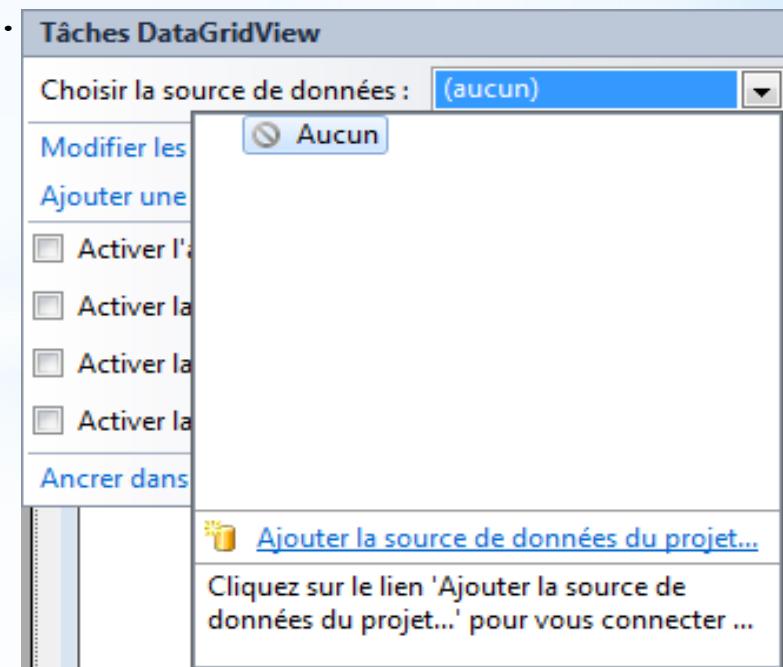
6. Liaison de quelques Contrôles aux données :

a. Le contrôle DataGridView :

Vous pouvez également lier des données (à partir d'une source de données) avec notre DataGridView. Il existe deux grandes voies :

■ En se servant du code C#.NET pour lier une DataTable ou un DataView au DataGridView.

■ Utiliser l'assistant de liaison d'un DataGridView à une BDD. Il faut définir les propriétés **DataSource** et **DataMember** pour lier le DataGridView à une source de données et le remplir automatiquement avec des données en utilisant le smart tag en mode assistant liaison.



Au cours de l'exécution, la liaison peut être faite avec le code suivant :

```
dataGridView.DataSource = ds.Tables("inscrits")
```

6. Liaison de quelques Contrôles aux données :

b. Contrôles ComboBox et ListBox:

Pour lier ces contrôles à une source de données En mode création utilisez les propriétés suivantes :

- **DataSource** : indique la source qui sera utilisée pour ajouter des objets au contrôle.
- **DisplayMember** : spécifie le champ qui sera affiché dans le ComboBox.
- **ValueMember** : Nom de champ qui sera utilisé comme valeur retenue après le choix de l'utilisateur.
- **SelectedValue du DataBinding** : Champ de la table liée au contrôle.
- **Text du DataBinding** : Champ qui sera affiché dans la zone de texte.

(DataBindings)	
(Avancées)	
SelectedItem	(aucun)
SelectedValue	(aucun)
Tag	(aucun)
Text	(aucun)
DataSource	(aucun)
DisplayMember	Particulier
Items	(Collection)
Tag	
ValueMember	

6. Liaison de quelques Contrôles aux données :

b. Contrôles ComboBox et ListBox:

Au cours de l'exécution, la liaison peut être faite avec le code suivant :

```
ComboBox.DataSource = ds.Tables("module")
```

```
ComboBox.DisplayMember = "lib_module"
```

```
ComboBox.ValueMember = "id_module"
```

Atelier 2 Mode Déconnecté ADO.NET

TP4 Manipulation des données dans une DataSet
(chercher/Filtrer/Trier)

C#.NET partie 6: ADO.NET

Utilisation des contrôles liés aux données
(ComboBox/DataGridView)