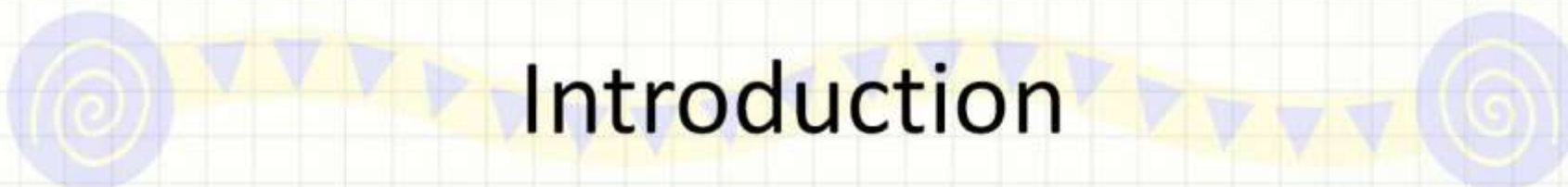


LA MANIPULATION DES DONNÉES

François-Xavier BAILLET





Introduction

- Dans le code
- Fichiers (TXT, XML,...)
- Bases de données
 - Locale (type SQLite, Access)
 - SGDB (Oracle, MySQL, SQL Server)
- Mode de connexion .NET
 - Connecté
 - Déconnecté

Introduction : Base de données

- Locale
 - Avantages :
 - ✓ Rapide
 - ✓ Simple à gérer
 - ✓ Accès de type fichier
 - ✓ Autonome
 - Inconvénients :
 - ✓ Accès concurrents
 - ✓ Fichiers...
 - ✓ Pas de sécurité (accès, données)
 - ✓ Politique de sauvegarde spécifique
- Indication : un Sqlite c'est parfois bien utile...

Introduction : Base de données

- SGBD

- Avantages :

- ✓ Rapide sur des gros volume de données
 - ✓ Administrable
 - ✓ Sécurité des données et des accès
 - ✓ Fournit beaucoup de services
 - ✓ Politique de sauvegardes

- Inconvénients :

- ✓ Installation (et encore...)
 - ✓ Gestion plus complexe

- Indication : un SGDB comme MySQL ou MariaDB est un bon compromis

Mode connexion sources de données

- La plate-forme .NET permet l'exploitation d'une source de données de deux manières différentes :
- mode connecté, l'application
 1. ouvre une connexion avec la source de données
 2. travaille avec la source de données en lecture/écriture
 3. ferme la connexion

Rq : données à jour, gestion simple, mais pb de ressources (∃ pool de connexions)
- mode déconnecté, l'application
 1. ouvre une connexion avec la source de données
 2. obtient une copie mémoire de tout ou partie des données de la source
 3. ferme la connexion
 4. travaille avec la copie mémoire des données en lecture/écriture
 5. lorsque le travail est fini :
 - ouvre une connexion
 - envoie les données modifiées à la source de données pour qu'elle les prenne en compte,
 - Ferme la connexion.

Rq : bcp de connexions possibles, mais données pas à jour, conflits de mise à jour



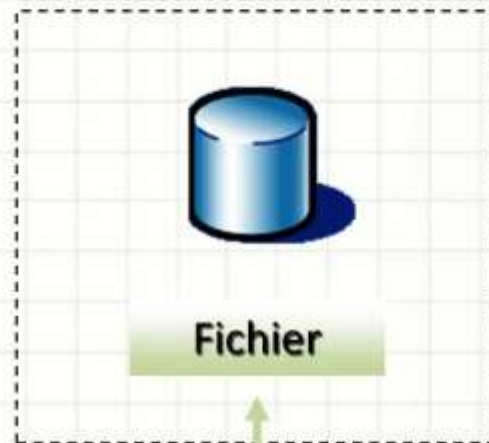
→ Pas de meilleur choix, dépend surtout de l'utilisation, même si le mode connecté...

Base Locale-Serveur



Protocole

Client



Accès fichier (partage)

Driver



Les API (*Application Programming Interface*) Microsoft

APIs

API POUR VB

REMOTE

COM

.NET

DAO

RDO

ADO

ADO.NET

temps



SqlLib

ODBC

OLEDB

ACCES NATIF
SQL SERVER

ABSTRACTIO
N SGBD

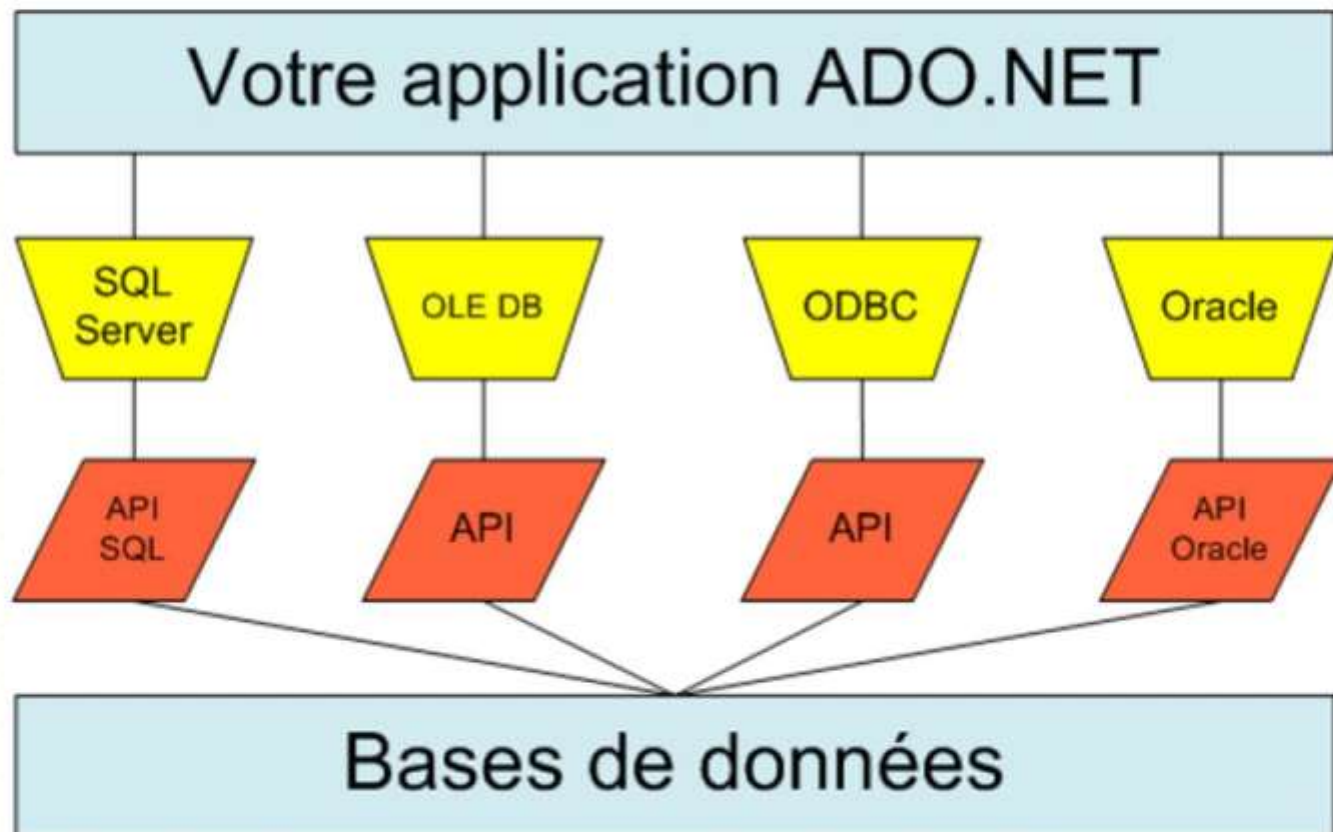
SOURCES NON
RELATIONNELLES

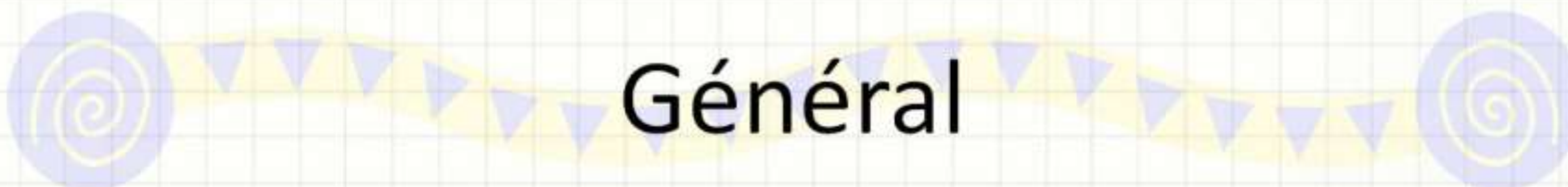
Drivers

ADO.NET

- Librairie (**Fournisseur Managé**) d'accès au données
 - Access, forcément
 - SQL Server, bien sûr
 - Bien d'autres SGBD (Oracle, Mysql...)
 - Mais aussi :
 - Données XML
 - Fichiers Excel (sans le traitement des données)
 - Interface de classes entre le programme et la base de données
- ADO (ActiveX Data Object) \neq ADO.Net

ADO.NET





Général

- Chaque fournisseur a son API, mais on retrouvera toujours ...
- ...Un descriptif de la connexion via
 - Objet
 - Chaine de connexion
 - ➔ Dans le code ou fichier de configuration (app.config)
- Des objets de manipulations des données

Accès BdD : classe de base

Classe	Description
Command	Stocke les informations sur la commande et permet son exécution sur le serveur de base de données.
CommandBuilder	Permet de générer automatiquement des commandes ainsi que des paramètres pour un <i>DataAdapter</i>
Connection	Permet d'établir une connexion à une source de données spécifiée
DataAdapter	Permet le transfert de données de la base de données vers l'application et inversement (par exemple pour une mise à jour, suppression ou modification de données). Il est utilisé en mode déconnecté
DataReader	Permet un accès en lecture seule à une source de données
Transaction	Représente une transaction dans le serveur de la base de données

Remarque : La classe *Connection* n'apparaît pas dans le Framework 3.5. En effet, les classes des fournisseurs managés ont leur propre classe tel que *SqlConnection*.

Accès BdD : Interface portable et ...

- Classe "Fournisseur" => code spécifique
- Solution 1:
 - Utilisation classes spécifiques pour connexion
 - Interfaces implémentées pour manipuler les données
- Solution 2 :
 - Utilisation classes spécifiques
 - Regrouper et encapsuler toutes les requêtes dans une classe spécifique

Accès BdD : Interface "Portable"

Interface	Description
IDataAdapter	Permet de remplir et actualiser un objet DataSet et de mettre à jour une source de données.
IDataReader	Permet de lire un ou plusieurs flux de données en lecture seule à la suite de l'exécution d'une commande.
IDataParameter	Permet d'implémenter un paramètre pour une commande.
SqlCommand	Permet de donner une commande qui s'exécutera au moment de la connexion à une source de données.
SqlConnection	Représente une connexion unique avec une source de données.
SqlDataAdapter	Représente un jeu de méthodes qui permet d'exécuter des opérations sur des bases de données relationnelles (insertion, sélection, ...).
SqlTransaction	Représente une transaction à exécuter au niveau d'une source de données.

Inconvénient : Beaucoup moins de possibilités que les fournisseurs managés

Accès BdD : solution 2...

- Exemple avec utilisation de SQLiteManager

```
//*****  
// Prototype : private static Variant GetParameter ( string Name, Variant Default )  
// Description : Obtiens la valeur du paramètre  
//*****  
/// <summary>  
/// Obtiens la valeur du paramètre.  
/// </summary>  
/// <param name="Name">Nom du paramètre.</param>  
/// <param name="Default">Valeur si paramètre non trouvé.</param>  
/// <returns>Objet <b>Variant</b> contenant la valeur du paramètre.</returns>  
//  
private static Variant GetParameter ( string Name, Variant Default )  
{  
    //-----  
    #region // Implémentation de la Procédure  
    //-----  
  
    Variant Result = Variant.Empty;  
  
    try  
    {  
        if ( Instance.Sql != null )  
            Result = Instance.Sql.ExecuteScalarPrepared ( "Get_Parameter", Name );  
    }  
    catch {}  
  
    return ( Result.IsEmpty ) ? Default : Result;  
  
    //-----  
    #endregion  
    //-----  
}
```

→ fonction utilisée dans le code

→ Appel à la requête SQL prédéfinie

Accès BdD : solution 2...

```
*****
// Prototype : private static bool OpenDataSource ( bool CtrlVersion )
// Description : Ouvre la base et vérifie sa validité
// *****
// <summary>
// Ouvre la base et vérifie sa validité
// </summary>
// <returns>
// <b>True</b> indique que la base de données est présente et exploitable, sinon
// <b>False</b>.
// </returns>
//
private static bool OpenDataSource ( bool CtrlVersion )
{
    // -----
    // #region // Implémentation de la Procédure
    // -----
    // -----
    Instance.FSql = new SQLiteManager ( Instance.SQLiteFile, "", 10000, false );
    try
    {
        // -----
        Instance.FSql.Open ();
        // -----
        if ( Instance.FSql.Status == ConnectionState.Open )
        {
            // -----
            Instance.FSql.PrepareQuery ( "Get_Parameter", "Select `Valeur` From `t_parametre` Where `Nom` = ?", 1 ); ➔ Définition de la requête SQL
            // -----
            return true;
            // -----
        }
        // -----
    }
    // -----
    catch { try { Instance.FSql.Close (); } catch { } }
    // -----
    Instance.FSql = null;
    return false;
    // -----
    // #endregion
    // -----
}
// *****
```

Les chaînes de connexion

- Contient l'ensemble des propriétés pour établir la connexion.
- => Texte en clair et format spécifique
- Déclaration dans le code ou un fichier de configuration (App.config)
- Ressources en ligne :
 - <http://www.connectionstrings.com>
- [Exemple de chaine de connexion](#)

Les chaînes de connexion : propriétés

➔ Dépendant de la base choisie...

Exemple avec MySQL vs SQLServer

connexionStringMYSQL = @"Server=myServerAddress;Port=portnumber;
Database=myDataBase;Uid=myUsername;Pwd=myPassword;";

connexionStringSQLSERVER = @"Data Source=myServerAddress, portnumber;Initial
Catalog=myDataBase;UserId=myUsername; Password=myPassword;";

Dans fichier App.config

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <connectionStrings>
    <add name="dbArticlesMySQL" connectionString="Server=localhost; Database=dbarticles; Uid=root; Pwd=root;"/>
  </connectionStrings>
</configuration>
```

Utilisation du style :

```
// exploitation du fichier de configuration [App.config]
String connectionString = null;
try {
    connectionString = ConfigurationManager.ConnectionStrings["dbArticlesMySQL"].ConnectionString;
} catch (Exception e) {
    ...
}
```

- On retrouvera toujours à minima :
 - La localisation de la base
 - Le nom de la base de données
 - Des identifiants de connexion


- Propriétés pour vérifier état de la connexion, par exemple : Open,...

Le mode connecté : explorons

Permet de manipuler les données en étant connecté à la base


- Type de commande :
 - Type texte
 - Type procédure stockée
- Type d'exécution :
 - Insert, Update, Delete (ExecuteNonQuery)
 - Select Count... (ExecuteScalar) (résultat unique)
 - Select (ExecuteReader)

Le mode connecté : commandes

➔ Dépendant de la base choisie...forcément 

Objet *Command*

Nom	Type de Source
SqlCommand	SQL Server
OleDbCommand	OLE DB
OdbcCommand	ODBC
OracleCommand	Oracle
MySqlCommand	MySQL

Mais il existe des méthodes et propriétés communes 

Le mode connecté : propriétés communes

Nom	Description
CommandText	Permet de définir l'instruction de requêtes SQL ou de procédures stockées à exécuter. Lié à la propriété CommandType.
CommandTimeout	Permet d'indiquer le temps en secondes avant de mettre fin à l'exécution de la commande.
CommandType	Permet d'indiquer ou de spécifier la manière dont la propriété CommandText doit être exécutée.
Connection	Permet d'établir une connexion.
Parameters	C'est la collection des paramètres de commandes. Lors de l'exécution de requêtes paramétrées ou de procédures stockées, vous devez ajouter les paramètres objet dans la collection.
Transaction	Permet de définir la SqlTransaction dans laquelle la SqlCommand s'exécute

Le mode connecté : méthodes communes, évènements

Nom	Description
Cancel	Permet de tenter l'annulation de l'exécution d'une commande.
ExecuteNonQuery	Permet d'exécuter des requêtes ou des procédures stockées qui ne retournent pas de valeurs.
ExecuteReader	Permet d'exécuter des commandes et les retourne sous forme de tableau de données (ou des lignes).
ExecuteScalar	Permet d'exécuter les requêtes ou les procédures stockées en retournant une valeur unique.
ExecuteXMLReader	Permet de retourner les données sous le format XML.

Evènements	Description
Disposed	Permet d'appeler la dernière méthode avant que l'objet ne soit détruit.
StatementCompleted (uniquement pour SqlCommand)	Se produit lorsque l'exécution d'une instruction se termine.

Utiliser des commandes

3 méthodes (exemple avec SQLServer) avec une connexion établie

1. Directement, nécessite l'utilisation de 2 propriétés :

```
SqlCommand commande= new SqlCommand();  
commande.Connection = connexion;  
commande.CommandText = "SELECT * FROM Employe";
```
2. Constructeur surchargé :

```
commande = new SqlCommand("SELECT * FROM Employe", connexion);
```
3. Méthode CreateCommand :

```
SqlCommand commande = connexion.CreateCommand();  
commande.CommandText = "SELECT * FROM Employe";
```

➔ Permet aussi la mise en place de procédure stockée (*Command type : StoredProcedure*), plus performante car précompilée dans le cache du serveur

Utiliser des commandes

Procédure stockée

Utilisation :

propriété *CommandText* avec le nom de la procédure stockée dans la base
propriété *CommandType* doit avoir la valeur *StoredProcedure* (au lieu de *Text*)

Exemple :

```
SqlConnection connexion = new SqlConnection(@"Data Source=. \SQLServer; Initial Catalog=Essai; Integrated Security=True");
SqlCommand command = connexion.CreateCommand();

connexion.Open();

string nom;
string requete = "RecupInformation";

command.CommandText = requete;
command.CommandType = CommandType.StoredProcedure;

Console.WriteLine("Quel nom ?");
nom = Console.ReadLine();

SqlParameter nom_param = new SqlParameter("@NOM", nom);

command.Parameters.Add(nom_param);

IDataReader lecture = command.ExecuteReader();

while (lecture.Read()){
    Console.WriteLine("Identifiant : {0} Nom : {1} Prenom : {2} Age : {3}",
        lecture.GetString(1), lecture["Nom"], lecture.GetString(2), lecture.GetInt32(3));
}

connexion.Close();
connexion.Dispose();
```


Illustrations par l'exemple

```
// exécution d'une requête de mise à jour
static void ExecuteUpdate ( string connectionString, string requête ) {
    // on gère les éventuelles exceptions
    try {
        using ( SqlConnection connexion = new SqlConnection ( connectionString ) ) {
            // ouverture connexion
            connexion.Open();
            // exécute sqlCommand avec requête de mise à jour
            SqlCommand sqlCommand = new SqlCommand ( requête, connexion );
            int nbLignes = sqlCommand.ExecuteNonQuery();
            // affichage résultat
            Console.WriteLine ( "Il y a eu {0} ligne(s) modifiée(s)", nbLignes );
        }
    } catch ( Exception ex ) {
        ....
    }
}

// exécution d'une requête Select
static void ExecuteSelect(string connectionString, string requête) {
    // on gère les éventuelles exceptions
    try {
        using (SqlConnection connexion = new SqlConnection( connectionString ) ) {
            // ouverture connexion
            connexion.Open();
            // exécute sqlCommand avec requête select
            SqlCommand sqlCommand = new SqlCommand(requête, connexion);
            SqlDataReader reader = sqlCommand.ExecuteReader();
            // permet l'accès en lecture aux enregistrements, avec plusieurs méthodes (GetBytes, GetChars, GetString,....)
            // exploitation des résultats
            ...
        }
    } catch(Exception ex) {
        ....
    }
}
```

- L'objet SqlDataReader
 - Fournit un accès très rapide en lecture seule et séquentielle
 - Créé par un objet Command, contenant une requête, renvoi un curseur de BDD
 - Le SqlDataReader doit être fermé pour libérer les ressources
 - Un seul objet SqlDataReader peut être ouvert par connexion

Illustrations par l'exemple

```
SqlConnection connexion = new SqlConnection(@"Data Source=.\SQLServeur; Initial Catalog=Airbus; Integrated Security=True");

SqlCommand command = connexion.CreateCommand();

string requete = "SELECT e.ID 'ID', e.Nom, e.Prenom, r.Nom FROM Employe e, Role r WHERE ( e.Role = r.ID ) ";

command.CommandText = requete;

connexion.Open();

SqlDataReader lire = command.ExecuteReader();

// Lit les informations de la base de données
Console.WriteLine("Lecture du DataReader \n\n");

while (lire.Read()){
    Console.WriteLine("Id : {0} Nom : {1} Prenom : {2} Role : {3}", lire["ID"], lire.GetString(1),
        lire.GetString(2), lire.GetString(3));
}

connexion.Close(); => fermé, mais connexion toujours disponible, conserve les ressources en mémoire

connexion.Dispose(); => effacement des ressources.
```

[Exemple BaseDonnées dans VS](#)

Le mode déconnecté : explorons

- Réception des données en bloc (Fill → DataSet)
- Stockage d'un morceau de la BDD en mémoire
- ATTENTION ! Select *
- Traitement à posteriori des données sur le poste client.
- Demande explicite de mise à jour de la base (Update)
- Les objets utilisés :
 - DbConnection
 - DbDataAdapter
 - DataSet



Le mode déconnecté : DataSet

- Cache de données en mémoire
- *DataSet* contient la collection d'objets *DataTable* liés avec les objets *DataRelation*

=> Import la partie voulue de la BdD

=> Avec objets (select, connexion,...) et *DataAdapter* permet de relier "Binder" le *DataSet* sur la BdD

=> Parcoure le *DataSet* avec *foreach* ou ...
requête Linq

Le mode déconnecté : DataSet

- Il existe deux types de *DataSet* :
 - Typed DataSet : Cela permet de créer une instance d'une classe DataSet déjà typée dans votre projet.
 - Untyped DataSet : Cela permet de créer une instance d'une classe DataSet non-typé de la classe System.Data.DataSet.
- Voici les éléments d'un *DataSet* :

Objet	Description
DataTable	Correspond à une table. Contient une collection de DataColumn et de DataRow
DataColumn	Représente une colonne de la table.
DataRow	Correspond à un enregistrement de la table.

- Le DataTable peut être créé manuellement (avec des expressions, des auto-incréments, des contraintes à l'image d'un SGDB)

Le mode déconnecté : DataSet

- Création :

```
DataSet mon_DS = new DataSet ();
```

- Ajout des *DataTable* :

```
DataTable table_1 = new DataTable();
```

```
DataTable table_2 = new DataTable();
```

```
mon_DS.Tables.Add(table_1);
```

```
mon_DS.Tables.Add(table_2);
```

- Lien entre plusieurs tables : *DataRelation*

```
DataRelation relation_1 = new DataRelation
```

```
("Rel_T1_T2", table_1.ParentColumns["champ_1"],  
table_2.ParentColumns["champ_1"]);
```

```
mon_DS.Relations.Add(relation_1);
```


Le mode déconnecté : DataSet

- On peut naviguer entre les colonnes des tables grâce à cette relation et avec les méthodes :

Méthode	Description
GetParentRows	Permet d'obtenir les lignes parentes d'un DataRow attaché à un DataRelation.
GetChildRows	Permet d'obtenir les lignes enfants d'un DataRow attaché à un DataRelation.

- On peut aussi fusionner 2 *DataSet* (*Merge*)
- Copier des *DataSet* (*.copy()*)

Le mode déconnecté : DataAdapter

- Tiens, change avec le fournisseur...

- Création :

```
SqlDataAdapter mon_DA = new SqlDataAdapter("select * from ma_table",  
                                           connexion);
```

- Créer des commandes :

```
SqlCommand modifier = new SqlCommand("update ma_table set  
                                     champ_1 = 'alpha' where champ_2='numerique'");  
mon_DA.Modifier = modifier;
```

- *MissingMappingAction* : propriété pour détecter les conflits (manque de colonnes ou tables)
- *MissingSchemaAction* : propriété pour régler les conflits (ajout table ou colonnes)

Le mode déconnecté : DataTable

- Ajout de données :

```
mon_DS.Tables["table_1"].Rows.Add(5,"bonjour","matin");
```

- Gérer les données :

```
ma_ligne("champ_1")='TOTO'; ma_ligne.Delete;
```

- Obtenir le statut d'une ligne :

RowState (Added, Deleted, Modified, Unchanged,...)

- Gérer des évènements :

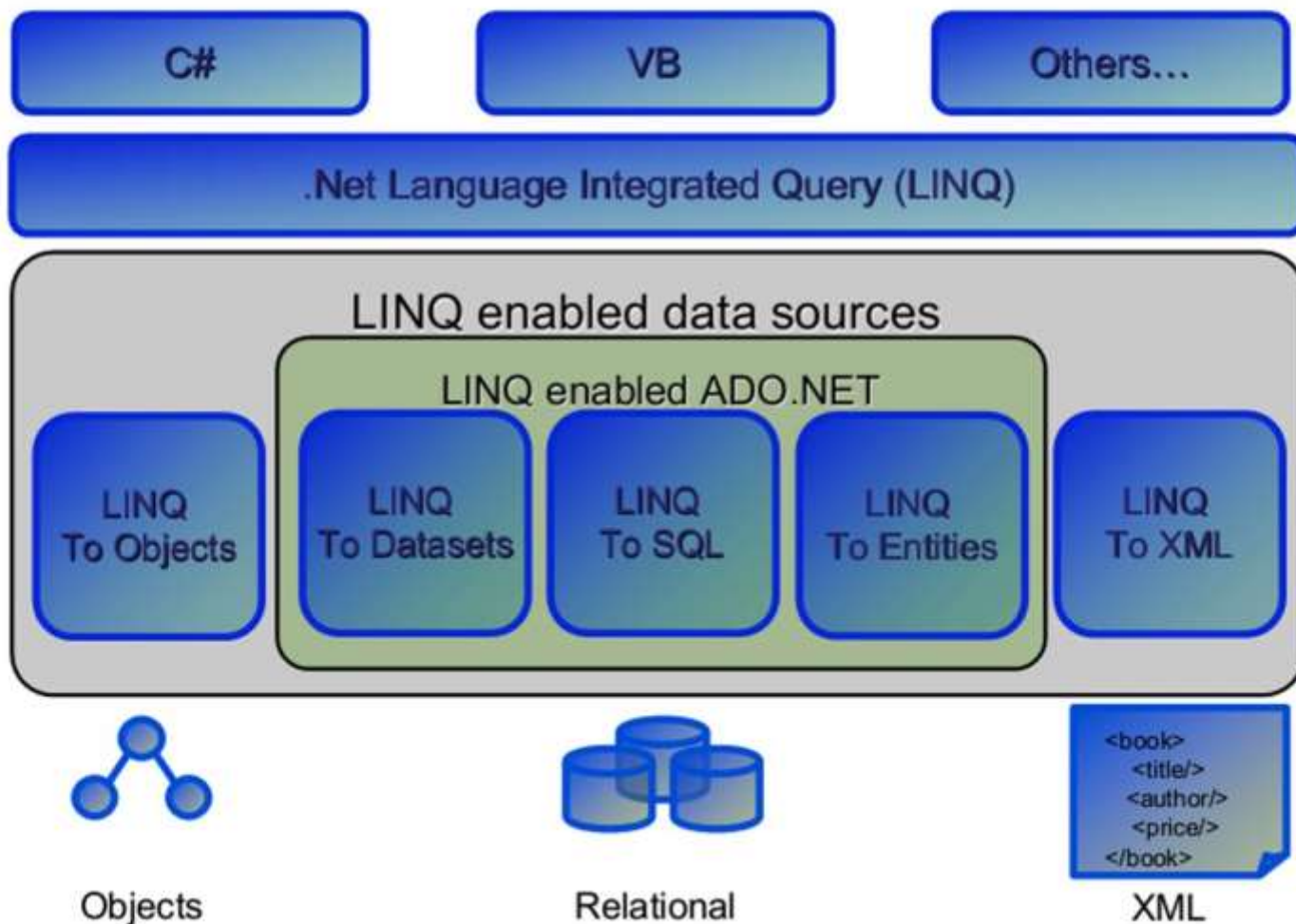
ColumnChanged, RowChanged, RowChanging, TableCleared,....

LINQ : accès simplifié ?

- Les langages objets sont plus ou moins matures.
- Pb : Complexité d'accès aux données non objets (XML et SGBD relationnel)
- Jungle dans les technos actuelles (sql, mapping objet-relationnel, DOM, XPath, XQuery, etc...)
- But :
 - Faire des requêtes sur les données en faisant abstraction de leur type
 - Requête exécutée lors de l'accès aux données

LINQ : accès simplifié ?

- Pas forcément très mûr, apparu en .NET 3.5



LINQ : requête

- Syntaxe :
 - Var result = from item in collection
select item;
- Opérations classiques SQL- like (mais de loin)
 - Sélection,
 - GroupBy, OrderBy
 - Join
- Extensions (expressions Lambda) : Accès à toutes les fonctionnalités du langage SQL couplé aux fonctions C#
- Exemple :

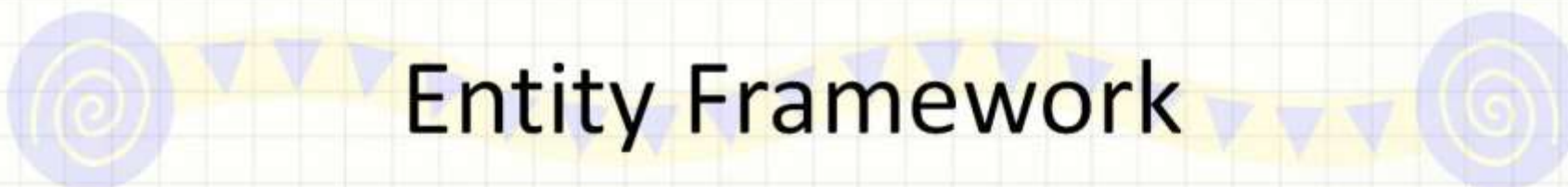
```
List < int > liste =  
    new List < i nt > {4,6,1,9,5,15,8,3};
```

```
foreach ( int i in liste){  
    if (i > 5) {  
        Console.WriteLine(i);  
    }  
}
```

```
using System.Linq;  
List <int> liste = new List <int> {4,6,1,9,5,15,8,3};
```

```
IEnumerable < int > requeteFiltree = from i in  
    liste where i > 5 select i;
```

```
foreach ( int i in requeteFiltree){  
    Console.WriteLine(i);  
}
```

Entity Framework

- EF : Ensemble de technologies d'ADO.Net pour les développement orientés données
- ORM : système d'abstraction de la BDD en objets et propriétés ("mapping objet<->relationnel")
- Objectif : Ne plus se soucier des tables et des colonnes de la BDD
- Transformation du modèle logique et physique de la base en classes et méthodes
- Mappage entre modèle de stockage et modèle conceptuel
- Natif pour SQLServer c'est prévu pour...

Entity Framework

- Création du modèles dans l'espace de dev
- Le langage CSDL (Conceptual Schema Definition Language) : modèle conceptuel
 - Entity Data Model
- Le langage SSDL (Store Schema Definition) : modèle logique
 - Schéma de BDD
- Le langage MSL (Mapping Specification Language) : Mappage entre le modèle de stockage et conceptuel
 - Lien entre votre BDD et votre modèle objet

Entity Framework : Architecture

