

# I314A

## .NET Outils et Concepts d'Application d'Entreprise

Semaine 4  
LINQ To Entities

# Entity Framework

Layers	.NET Framework Components
Frontend	Winforms, WPF MVVM, ASP.NET MVC, ConsoleApplication
Backend – Service (UCC)	ASP.NET Web API, WCF
Backend – Business Logic	C# Classes
Backend – DAL - Repository	Pattern Repository
Backend – DAL	LINQ To Entities – Entity Framework
Database	SQL Server

# Entity Framework

- ORM (Object Relational Mapping)

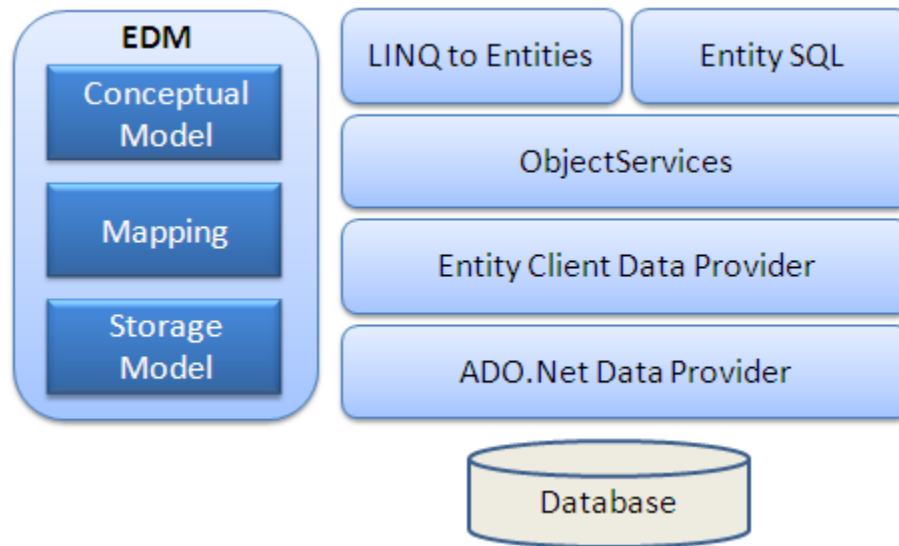
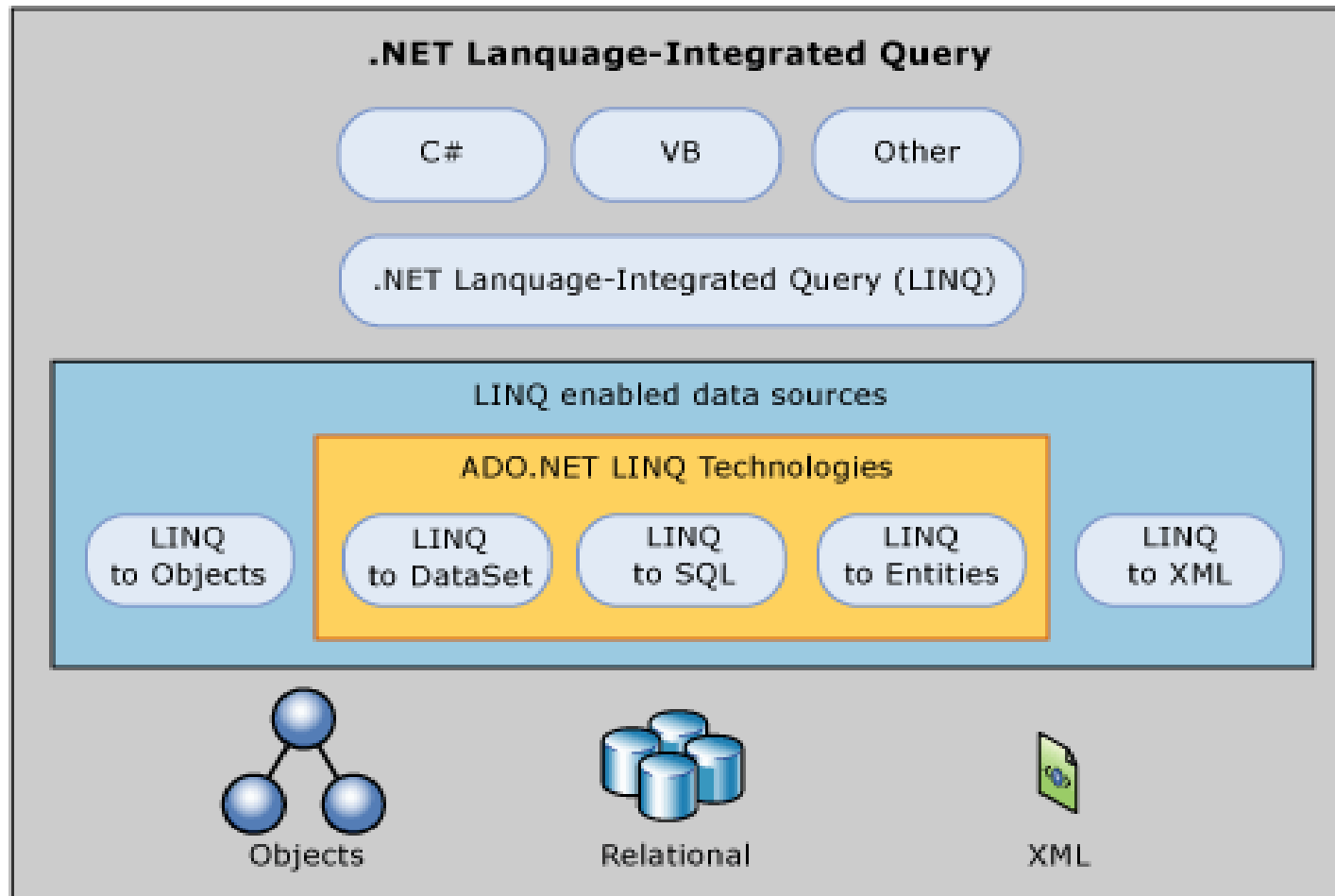


Image issue de : [entityframeworktutorial.net](http://entityframeworktutorial.net)

# LINQ Providers



# Considérations

- POCO, DTO, BO
  - POCO : Plain Old CLR Object
    - Objets ignorant la persistance mappées à un modèle de donnée
  - DTO : Data Transfert Object
    - Objets de transfert entre couches (get/set only)
  - BO : Business Object
    - Objets du domaine avec comportement

# Considérations

- POCO, DTO, BO
  - Entity framework utilise/génère des POCO
  - POCO peuvent être des BO
    - Active Record Pattern
    - Petite application
    - lien fort entre la couche métier et Entity framework
  - POCO peuvent être des DTO
    - Architecture N-tier
    - Application Entreprise
    - Séparation des responsabilités

# Différentes Approches

- Database First
  - Génération des POJO et du modèle à partir de la DB
- Model First
  - Créer UML
- Code First
  - Annotations [Key], [Foreign Key], ...

# Fichier EDMX

- Diagrammes
- Entités (POCO)
- Types complexes
- Fonctions et procédures stockées
- Modèle de persistance



# DbContext

- DbContext = Proxy vers la DB
- Créé via import de la DB
  - Contient les tables

```
// create theObjectContext
```

```
NorthwindEntities context = new NorthwindEntities();
```

```
context.
```

```
// retri
```

```
Customer
```

```
// Updat
```

```
cust.Con
```



CreateProxyTypes

CreateQuery<>

Current\_Product\_Lists

Customer\_and\_Suppliers\_by\_Cities

CustomerDemographics

Customers

DatabaseExists

DefaultContainerName

DeleteDatabase

```
Customers  
= "LAZYK"  
comer>();
```

ObjectSet<Customer> NorthwindEntities.Customers  
No Metadata Documentation available.

# DbContext

- DbContext = Proxy vers la DB
  - Opérations (save, delete, refresh, ...)
  - Via les collections générées (DbSet)
    - context.Customers.Remove(cust)
    - context.Customers.Add(cust)
    - cust.name = "mise à jour du nom"
  - Persistance
    - context.SaveChanges()

# Example

```
// create theObjectContext
NorthwindEntities context = new NorthwindEntities();

// retrieve customer LAZY K
Customer cust = (from c in context.Customers
                  where c.CustomerID == "LAZYK"
                  select c).Single<Customer>();

// Update the contact name
cust.ContactName = "Ned Plimpton";

// save the changes
try {
    context.SaveChanges();
} catch (OptimisticConcurrencyException) {
    context.Refresh(RefreshMode.ClientWins,
        context.Customers);
    context.SaveChanges();
}
```

# Classes Entities: associations

- Les clefs étrangères créent des associations
- Les clefs étrangères créent des propriétés de navigation
- Gérées dans les Entities
- Cfr EJB Relations
- Query linq avec join -> mieux vaut utiliser les propriétés de navigation
  - Performance et clarté

# Classes Entities:

## Propriétés de navigation

```
from p in ctx.Persons
join c in ctx.Cities
on p.BornIn equals c.CityID
select new
{
    p.FirstName,
    c.Name
};
```

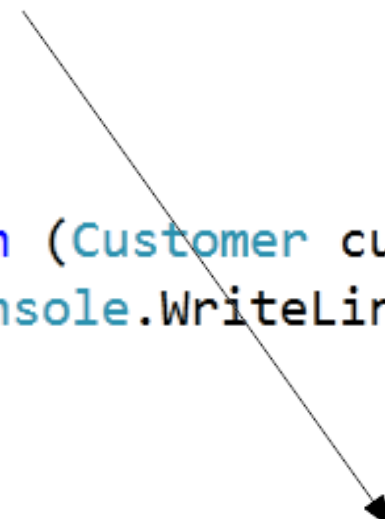
```
from p in ctx.Persons
select new
{
    p.FirstName,
    p.BornInCity.Name
};
```

# IEnumerable vs IQueryable

```
// create theObjectContext
NorthwindEntities context = new NorthwindEntities();

IQueryable<Customer> custs = from c in context.Customers
                              where c.City == "London"
                              select c;

foreach (Customer cust in custs) {
    Console.WriteLine("Customer: {0}", cust.CompanyName);
}
```



Étend IEnumerable → avantage  
performance filtre effectué côté base de  
données

# Lazy Loading : par défaut

```
// create theObjectContext
NorthwindEntities context = new NorthwindEntities();

IQueryable<Customer> custs = from c in context.Customers
                             where c.Country == "UK" &&
                                c.City == "London"
                             orderby c.CustomerID
                             select c;

foreach (Customer cust in custs) {
    Console.WriteLine("{0} - {1}", cust.CompanyName, cust.ContactName);
    Order firstOrder = cust.Orders.First();
    Console.WriteLine("    {0}", firstOrder.OrderID);
}
```

On va chercher les *Orders* à ce moment là via un query.  
On a un query par tour de boucle!

# Eager Loading

```
IQueryable<Customer> custs = from c in context.Customers
                             .Include("Orders")
                             where c.Country == "UK" &&
                                c.City == "London"
                             orderby c.CustomerID
                             select c;

foreach (Customer cust in custs) {
    Console.WriteLine("{0} - {1}", cust.CompanyName, cust.ContactName);
    Order firstOrder = cust.Orders.First();
    Console.WriteLine("    {0}", firstOrder.OrderID);
}
```

Pas de query à chaque  
tour de boucle

Les *orders* sont tout de suite  
chargés en mémoire



# Pattern Repository

Layers	.NET Framework Components
Frontend	Winforms, WPF MVVM, ASP.NET MVC, ConsoleApplication
Backend – Service (UCC)	ASP.NET Web API, WCF
Backend – Business Logic	C# Classes
Backend – DAL - Repository	Pattern Repository
Backend – DAL	LINQ To Entities – Entity Framework
Database	SQL Server

# Pattern Repository

- Séparation de la couche DATA et BUSINESS
  - Facilite les tests (« mock db »)
- Eviter le code redondant

```
public interface IRepository<T>
{
    void Insert(T entity);
    void Delete(T entity);
    IQueryable<T> SearchFor(Expression<Func<T, bool>>
predicate);
    // insertOrUpdate
    bool Save(T entity, Expression<Func<T, bool>> predicate);
    IQueryable<T> GetAll();
    T GetById(int id);
}
```

# Pattern Repository

- BaseRepository

```
public class BaseRepository<TEntity> : IRepository<TEntity> where TEntity :  
class  
{  
    private readonly DbContext _dbContext;  
  
    public BaseRepository(DbContext dbContext)  
    {  
        _dbContext = dbContext;  
    }  
  
    public void Insert(TEntity entity) {  
        .....  
    }
```

# Pattern Repository

- Généricité -> quelques changements

```
_dbContext.CourseSet.Add(course);
```

→

```
_dbContext.Set<TEntity>().Add(entity);
```