

ADO.NET – Disconnected Mode

Il **Disconnected Mode** consente di

- manipolare i dati recuperati dall'origine dati
- **successivamente riconciliarli con l'origine dati**

Le classi disconnesse forniscono un modo comune di lavorare con i dati disconnessi, indipendentemente dall'ambiente di origine data.

ADO.NET – Disconnected Mode

Principali classi utilizzate in Disconnected Mode:

- DataSet
- DataTable
- DataColumn
- DataRow
- Constraint
- DataRelation

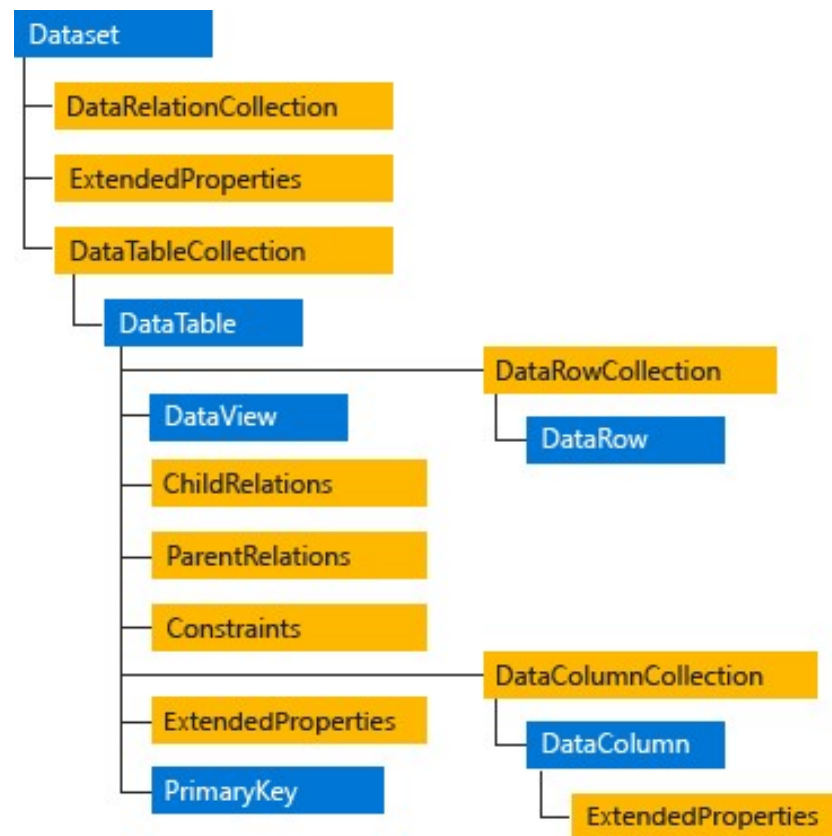
ADO.NET - DataSet

Il **DataSet** è progettato esplicitamente per l'accesso ai dati indipendentemente da qualsiasi origine dati: può essere utilizzato con origini dati multiple e diverse o utilizzato per gestire i dati locali per l'applicazione.

Il **DataSet** contiene una raccolta di uno o più oggetti **DataTable** costituiti da

- righe e colonne di dati
- informazioni relative a chiave primaria, chiavi esterne, vincoli e relazione sui dati

ADO.NET - DataSet



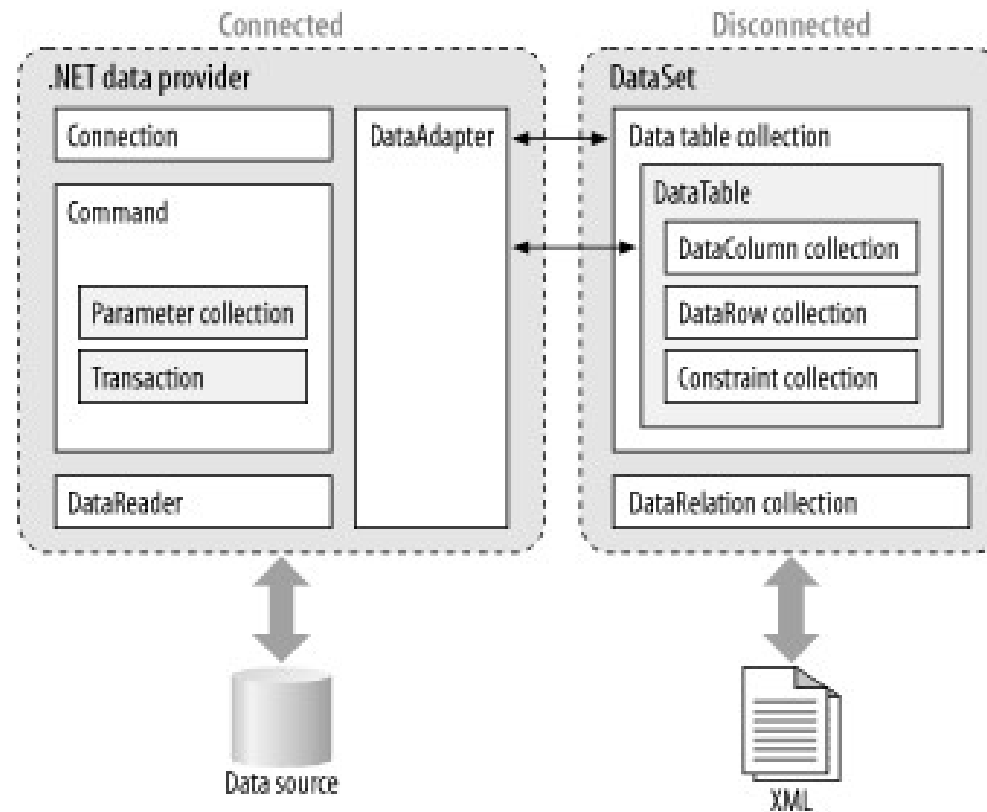
ADO.NET - Adapter

Il **DataAdapter** fornisce il ponte tra l'oggetto **DataSet** e l'origine dati.

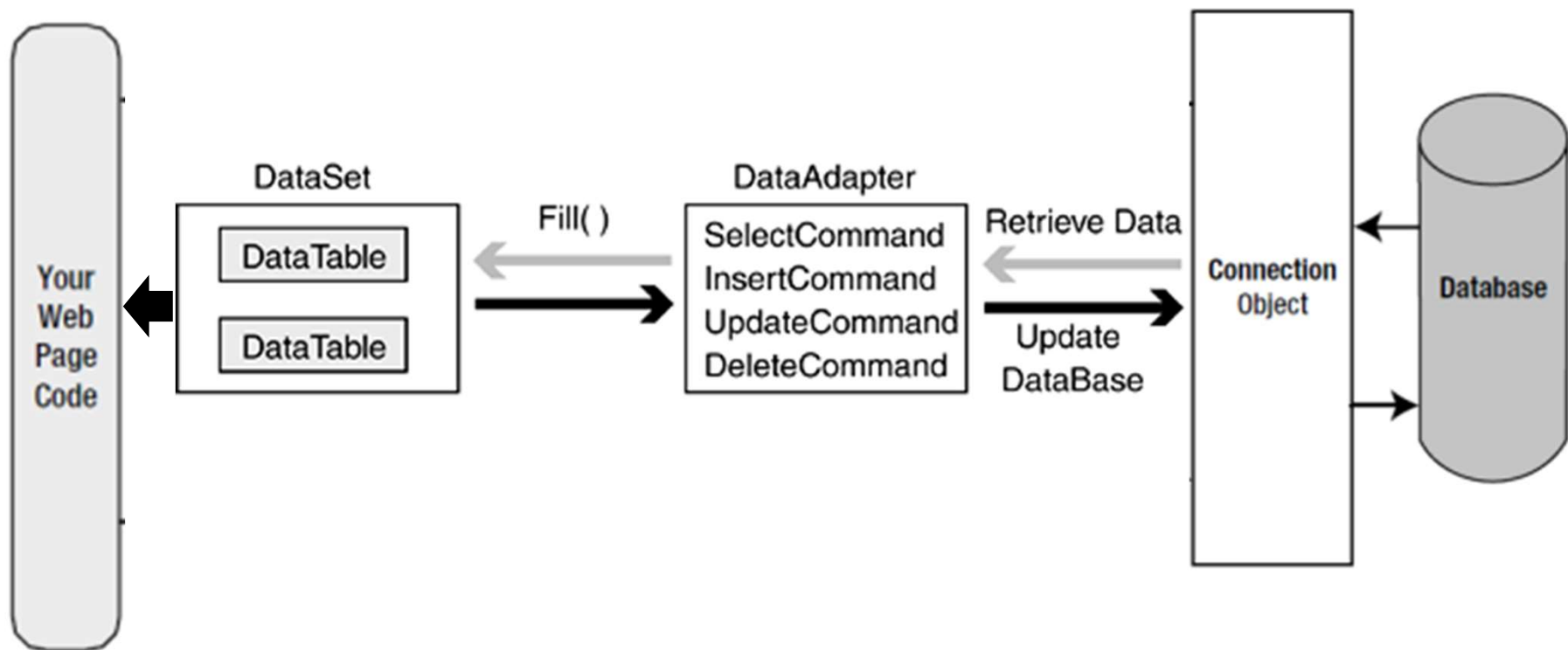
DataAdapter utilizza oggetti *Command* per eseguire comandi SQL sull'origine dati per caricare sia *DataSet* con dati sia per riconciliare le modifiche apportate ai dati nel *DataSet* con l'origine dati

La classe **DataAdapter** permette di collegare l'origine dati e le classi disconnesse tramite le classi connesse.

ADO.NET - Adapter



ADO.NET – Disconnected Mode



Demo

Disconnected Mode



Gestione della concorrenza

In un ambiente con più utenti sono disponibili due modelli per l'aggiornamento di dati in un database: la concorrenza ottimistica e la concorrenza pessimistica.

La **concorrenza pessimistica** implica il blocco di righe nell'origine dati per impedire agli altri utenti di apportare modifiche ai dati che possano influire sugli utenti correnti.

La **concorrenza ottimistica** considera la presenza di una violazione nel caso in cui, dopo che un utente riceve un valore dal database, tale valore viene modificato da un altro utente prima che il primo utente abbia effettuato un tentativo di modifica.

Concorrenza ottimistica

Alle ore 13.00 l'Utente1 legge una riga dal database contenente i seguenti valori:

Situazione database

CustomerID	Nome	Cognome
101	Maria	Smith

Situazione visualizzata

CustomerID	Nome	Cognome
101	Maria	Smith

Alle ore 13.01 l'Utente2 legge la stessa colonna.

Alle ore 13.03 l'Utente2 modifica **FirstName** da "Maria" a "Maria Teresa" e aggiorna il database.

Situazione database

CustomerID	Nome	Cognome
101	Maria Teresa	Smith

Situazione visualizzata

CustomerID	Nome	Cognome
101	Maria	Smith

Concorrenza ottimistica

Alle ore 13.05 l'Utente1 cambia il nome di "Maria" in "Teresa" e cerca di aggiornare la riga.

Situazione database

CustomerID	Nome	Cognome
101	Maria Teresa	Smith

Situazione visualizzata

CustomerID	Nome	Cognome
101	Maria	Smith

Cosa succede a questo punto?

Concorrenza ottimistica

Soluzioni:

- specifica di una colonna timestamp in cui memorizzare l'ultima modifica apportata

CustomerID	Nome	Cognome	Timestamp
101	Maria	Smith	'2021-06-14 13:05:20'

- Specifica di una colonna che contiene il valore attualmente salvato su DB

CustomerID	Nome	Cognome	CurrentValue
101	Maria	Smith	Maria Teresa

Concorrenza ottimistica in ADO.NET

In ADO.NET viene definito l'evento **RowUpdated** che si verifica dopo ogni tentativo di aggiornamento di una riga **Modified** da un **DataSet**.

L'evento restituisce una proprietà **RecordsAffected**, che include il numero di righe modificate da un comando di aggiornamento per una riga modificata in una tabella:

la proprietà **RecordsAffected** restituisce come risultato:

- 0 in caso di violazione della concorrenza ottimistica e di conseguenza non verrà eseguito l'aggiornamento di alcun record. In questo caso viene generata un'eccezione.
- se il valore restituito è superiore a 0 la colonna è stata correttamente modificata

Demo

Gestione della concorrenza



Esercitazione n.2

Realizzare una Console app che acceda al database Ticketing utilizzando il DisconnectedMode di ADO.NET e che implementi le stesse operazioni dell'Esercitazione1:

- Legga la tabella dei Ticket e popoli un DataSet
- Esegua le seguenti operazioni sul DataSet (dopo aver chiuso la connessione col db)
- Stampi la lista dei Ticket
- Permetta l'inserimento di nuovi Ticket (i dati devono essere inseriti dall'utente)
- Permetta la cancellazione di un Ticket (utilizzare l'ID univoco per identificarlo)
- Aggiorni il database



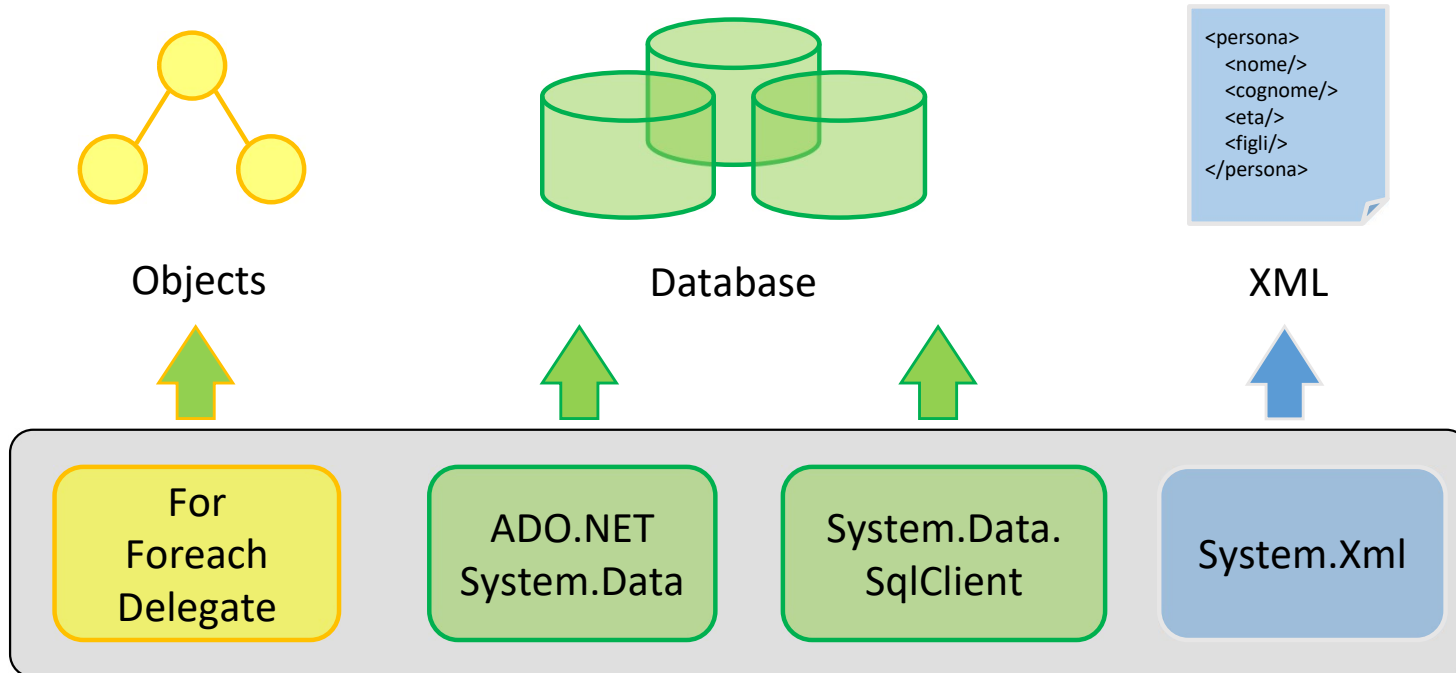
Cosa è LINQ



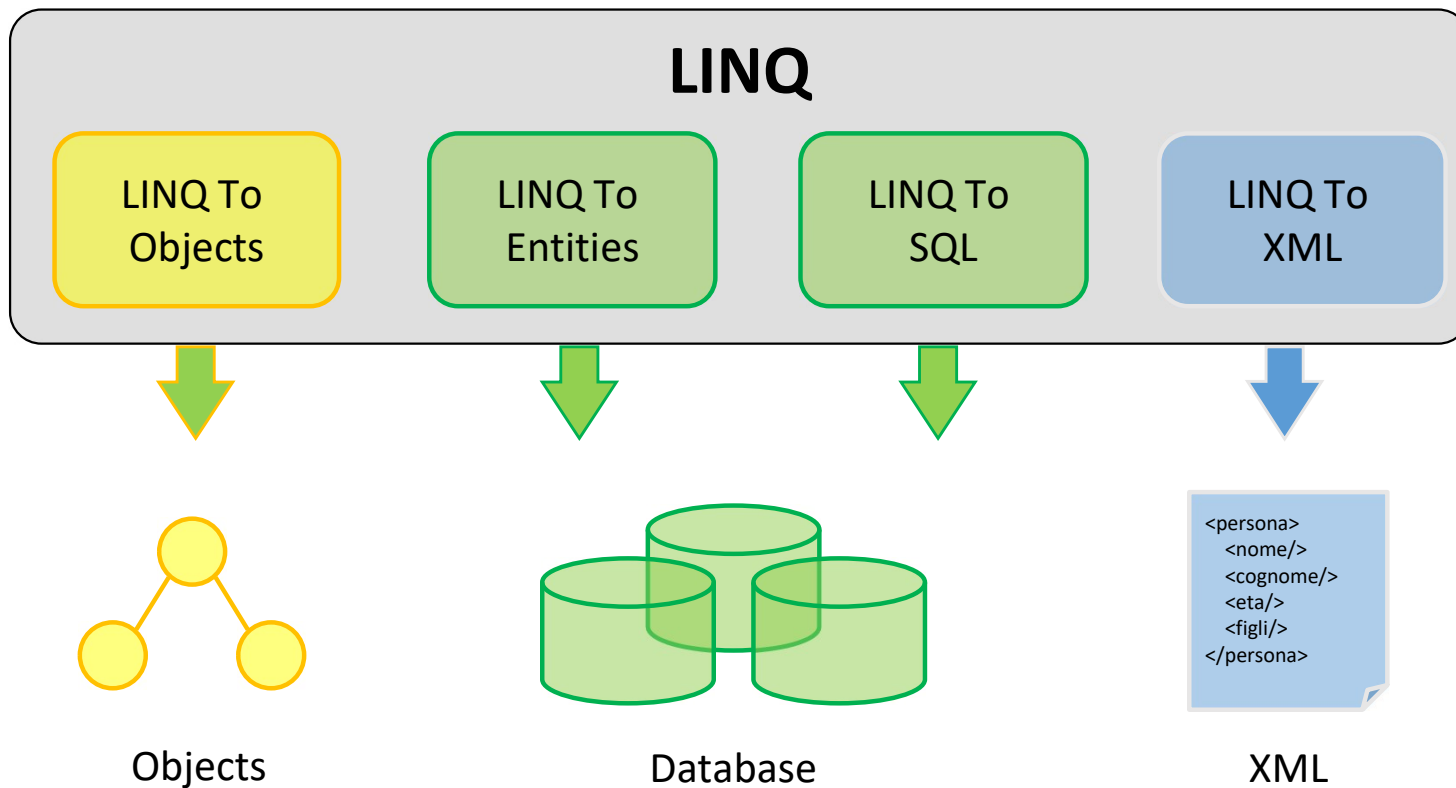
LINQ sta per **L**anguage **I**Ntegrated **Q**uery

LINQ è un framework per eseguire interrogazioni su sorgenti dati all'interno del linguaggio.

Accesso ai dati senza LINQ



Accesso ai dati con LINQ



LINQ – Query Expression



Query standard per accedere a:

- Oggetti
- Dati relazionali
- Dati XML

Più di 50 operatori predefiniti

- Aggregazione, Proiezione, Join, Partizionamento, Ordinamento

Sintassi e operatori **simile a SQL**

LINQ – Anatomia di una Query



- Due modelli di sintassi
 - Query
 - Lambda Expression
- Possibilità di utilizzare combinate
- Non modificano la sequenza originale

Query Lambda

- Più controllo e flessibilità
- Gli operatori sono applicati in sequenza
- **Select** può essere opzionale

LINQ – Operatori



- Utilizzo di **operatori Standard**
- Libreria di riferimento **System.Linq**
- Utilizzo con tipi **IEnumerable<T>**
- Pieno supporto ed integrazione con Intellisense

Operatori



Tipologia	Operatore
Projection	Select, SelectMany, (From)
Ricerca	Where
Ordinamento	OrderBy, OrderByDescending, Reverse, ThenBy, ThenByDescending
Raggruppamento	GroupBy
Aggregazione	Count, LongCount, Sum, Min, Max, Average, Aggregate,
Paginazione	Take, TakeWhile, Skip, SkipWhile
Insiemistica	Distinct, Union, Intersect, Except
Generazione	Range, Repeat, Empty
Condizionali	Any, All, Contains
Altri	Last, LastOrDefault, ElementAt, ElementAtOrDefault, First, FirstOrDefault, Single, SingleOrDefault, SequenceEqual, DefaultIfEmpty

LINQ - Operatori



- Reference: **System.Linq**
- Estende le funzionalità di **IEnumerable<T>** e **IQueryable<T>**

```
public static class Enumerable
{
    static public IEnumerable<Tsource> Where(this IEnumerable<TSource> source, Func<TSource, bool> predicate)
    ...
    ...
    ...
}
```

```
..public static class Enumerable
{
    ...public static TSource Aggregate<TSource>(this IEnumerable<TSource> source, Func<TSource, TSource> func);
    ...public static TAccumulate Aggregate<TSource, TAccumulate>(this IEnumerable<TSource> source, TAccumulate seed, Func<TSource, TAccumulate, TAccumulate> func);
    ...public static TResult Aggregate<TSource, TAccumulate, TResult>(this IEnumerable<TSource> source, TAccumulate seed, Func<TSource, TAccumulate, TResult> func);
    ...public static bool All<TSource>(this IEnumerable<TSource> source, Func<TSource, bool> predicate);
    ...public static bool Any<TSource>(this IEnumerable<TSource> source);
    ...public static bool Any<TSource>(this IEnumerable<TSource> source, Func<TSource, bool> predicate);
    ...public static IEnumerable<TSource> AsEnumerable<TSource>(this IEnumerable<TSource> source);
    ...public static decimal? Average(this IEnumerable<decimal> source);
    ...public static decimal? Average(this IEnumerable<decimal> source);
    ...public static double? Average(this IEnumerable<double> source);
    ...public static double? Average(this IEnumerable<double> source);
    ...public static float? Average(this IEnumerable<float> source);
    ...public static float? Average(this IEnumerable<float> source);
    ...public static double? Average(this IEnumerable<int> source);
    ...public static double? Average(this IEnumerable<int> source);
    ...public static double? Average(this IEnumerable<long> source);
    ...public static double? Average(this IEnumerable<long> source);
}
```



LINQ

Sostituzione di **foreach** con query Linq

Query Deferred

- Query expression come se dati
- Composizione di query

Definizione

```
IEnumerable<Employee> employee =  
    from p in employees  
    where p.Name == "Scott"  
    select p.Name;
```

Esecuzione

```
foreach (var emp in employee)  
{  
    ...  
    ...  
}
```


LINQ – Lambda Expression



```
IEnumerable<string> filteredList = cities.Where(StartsWithL);

public bool StartsWithL(string name)
{
    return name.StartsWith("L");
}
```



```
IEnumerable<string> filteredList = cities.Where(name => name.StartsWith("L"));
```

LINQ – Lambda Expression



- Rappresentazione sintetica
- (input-parameters) => expression
- Utilizzo dell'operatore =>
 - **A sinistra:** firma della funzione
 - **A destra:** statement della funzioni



LINQ – Lambda Expression

Parametri ed i tipi opzionali

- Non sono richieste parametri, quando sono impliciti

Logica negli statement

- Utilizzo di variabili locali
- Attenzione: le lambda expression dovrebbero essere tenute più semplici possibile

```
IEnumerable<string> filteredList =  
cities.Where((string s) =>  
{  
    string temp = s.ToLower();  
    return temp.StartsWith("L");  
}  
);
```

LINQ – Query Expression



- Lambda expressions
- **Query Expression**

```
IEnumerable<string> filterCities =  
    from city in cities  
    where city.StartsWith("L") && city.Length < 15  
    orderby city  
    select city;
```

Esecuzione Immediata VS Esecuzione Differita



Deferred/Lazy Operators	Immediate/Greedy Operators
La query non viene eseguita nel punto stesso in cui è dichiarata	La query non viene eseguita nel punto stesso in cui è dichiarata
Operatori di Proiezione (Es. Select, Select Many ecc)	Operatori di aggregazione (Es. Count, Average, Min Max, Sum ecc.)
Operatori di restrizione (Es. Take, Skip, Where)	Operatori su elementi (Es. First, Last, Single, ToList ecc.)


LINQ – Esecuzione differita



```
var allAuthors =  
    from a in authorTable  
    where a.Id == 1  
    orderby a.Id  
    select a;
```

allAuthors: è un'espressione!

```
foreach (var author in allAuthors)  
{  
    ...  
}
```

A blue arrow pointing upwards from the 'allAuthors' variable in the 'foreach' loop to the 'allAuthors' variable in the previous code block.

Eseguita una query **ogni volta che si accede** alla variabile

```
var queryAuthors =  
    from a in authorTable  
    where a.Id == 1  
    orderby a.Id  
    select a;  
  
queryAuthors.ToList();
```

Demo

LinQ – Lambda Expression VS Query Expression

LinQ – Esecuzione immediata e differita



Esercitazione n.3

Utilizzando la libreria Linq recuperare:

- Selezionare i Ticket con stato 'Resolving'

- Selezionare i Ticket degli utenti il cui nome comincia per A

- Selezionare i Ticket con data risalente ad un mese fa (n. giorni trascorsi = 30)

- Ordinare i Ticket per data di apertura (dalla più recente alla più lontana)

- Ordinare tutti i Ticket per descrizione in ordine alfabetico

- Raggruppare i ticket per tipologia

