

# LINQ



**Alice Colella**

Junior Developer@icubedsrl

[Alice.Colella@icubed.it](mailto:Alice.Colella@icubed.it)

**Roberto Ajolfi**

Senior Developer@icubedsrl

[Roberto.ajolfi@icubed.it](mailto:Roberto.ajolfi@icubed.it)



# Introduzione a Linq



# LINQ

LINQ è l'acronimo per Language-Integrated Query

LINQ consiste in framework per integrare le capacità di **querying** all'interno del linguaggio C#.

# Cosa significa?

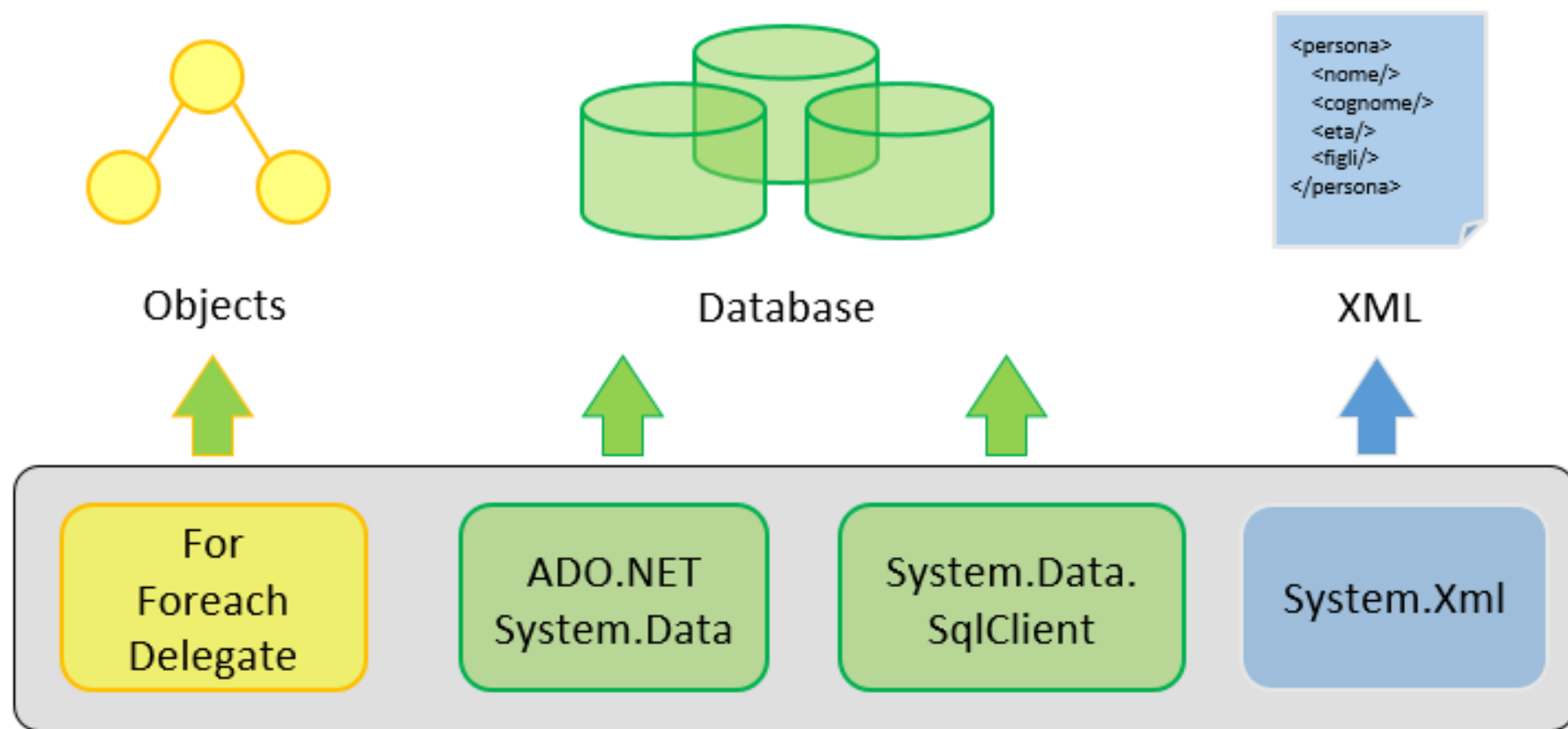
Query : interrogazione di una sorgente per aggiornare o estrarne dati secondo un certo criterio.

LINQ permette di usare un unico linguaggio per accedere a diverse tipologie di sorgente dati

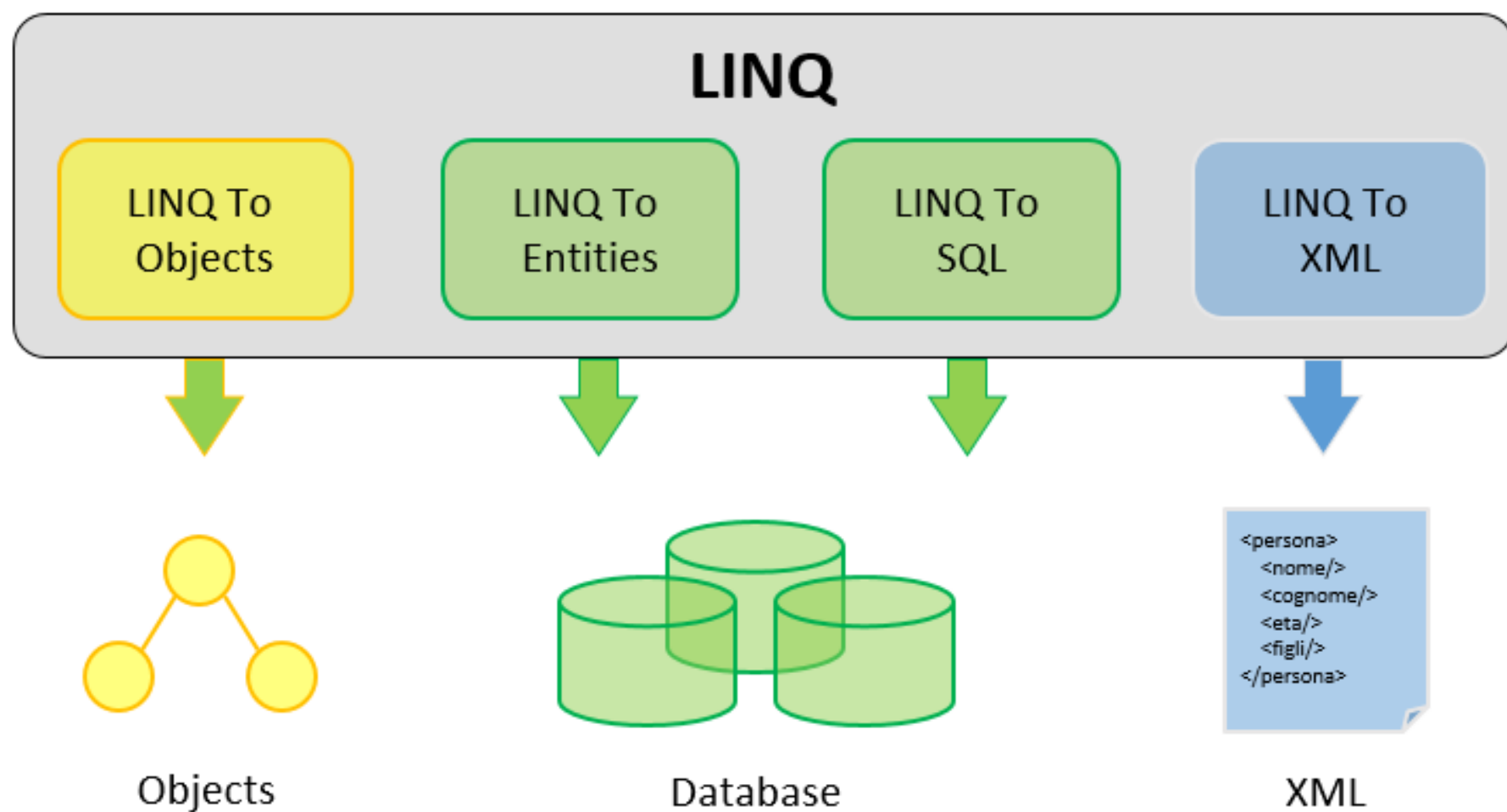
- documenti XML
- ADO Dataset
- SQL database
- Collections che supportano IEnumerable o IEnumerable<T>



# Accesso ai dati senza LINQ



# Accesso ai dati con LINQ



# LINQ – Evoluzione

**Obiettivo:** integrare query all'interno di C#

```
Sequence<Employee> andrew =  
employees.where(Name=="Andrew");
```

## LINQ in C# 2.0

```
IEnumerable<Employee> andrew =  
EnumerableExtensions.Where(employees,  
delegate (Employee e)  
{  
    return e.Name == "Andrew";  
});
```

## PROBLEMI

Il codice non sembra una query

Classi statiche

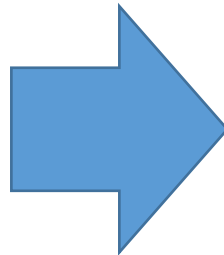
Metodi anonimi

E' necessario definire i tipi

# LINQ – Evoluzione

## LINQ IN C# 2.0

```
IEnumerable<Employee> andrew =  
EnumerableExtensions.Where(employees,  
delegate (Employee e)  
{  
    return e.Name == "Andrew";  
});
```



## LINQ oggi

```
var andrew =  
    from e in employees  
    where e.Name == "Andrew"  
    select e;
```

## PRO

Simile a SQL

Sintassi più comprensibile

Integrato nel linguaggio



# Variabili di tipo implicito

C# è un linguaggio tipizzato: ogni variabile ha un tipo (es. int, string, boolean ...)

La dichiarazione del tipo della variabile può essere



**“Esplicito”**

```
string name = "Marco";
```



**Implicito**

```
var lastName = "Rossi";
```

# Come funziona il tipo implicito?

- Il compilatore avvia un processo di “type inference” a partire dal valore della variabile in cui determina e assegna il tipo più appropriato.
- Non è un tipo “generico” che cambia a seconda dell’uso!
- È *quasi* sempre opzionale (tranne per gli anonymous types)

# Modi d'uso della keyword “var”

- Variabili locali

```
var lastName = "Rossi";
```

- Nell'inizializzazione del for

```
for (var i= 0; i<10; i++) {}
```

- Nell'inizializzazione del foreach

```
foreach(var number in list)
```

- Nello statement using

```
using( var file = new StreamReader(""))
```

# Anonymous Types

Gli anonymous types forniscono una maniera per incapsulare delle **proprietà read only** in un tipo non dichiarato in precedenza.

Sono “anonimi” poichè lo sviluppatore non dichiara il nome del tipo.

Il compilatore genera il nome del tipo, il quale non è accessibile dallo sviluppatore.

# Implementazione Anonymous Types

Per implementare un anonymous type, la keyword “var” è necessaria!

```
var person = new { firstName = "Marco", lastName = "Rossi" };
```

L'operatore new crea la nuova istanza del tipo e viene seguito dall'inizializzazione di un Object.

In questo caso l'Object ha due proprietà: firstName e lastName.

# Operatori

- Reference: **System.Linq**
- Estende le funzionalità di **IEnumerable<T>** e **IQueryable<T>**  
->sono degli Extension Methods!

```
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);
}
```

# Extension Methods

- **Obiettivo:** Estendere le funzionalità ai tipi esistenti (Classi ed Interfacce)

```
public static class StringExtensions
{
    static public double ConvertToDouble(this string value)
    {
        double result = double.Parse(value);
        return result;
    }
}
```

```
string text = "43.45"
double result = text.ConvertToDouble();
```

- Utilizzo della parola chiave **this**
- Il metodo viene “**agganciato**” al tipo dopo la parola chiave **this**

# Extension Methods

- Possono essere definiti per **classi non generiche e statiche**
- Devono essere **metodi statici**



# Operatori

Necessitano di un delegato

I delegate possibili sono:

1. Funzioni
2. Func/Action
3. Lambda Expression

Delegato:

- Tipo di dato
- Puntatore a funzione, quindi dichiara la firma di una funzione

```
public delegate int Delegato(string value);
```

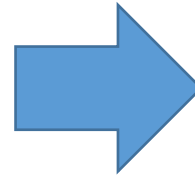
```
...public delegate TResult Func<in T, out TResult>(T arg);
```

```
...public delegate void Action();
```

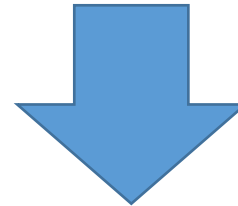
# Lambda Expression

## Metodo Anonimo

```
IEnumerable<string> filteredList =  
cities.Where(StartsWithL);  
public bool StartsWithL(string name)  
{  
    return name.StartsWith("L");  
}
```



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



## Lambda Expression

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

# Lambda Expression

- Utilizzano la type inference
- Possono essere passate come parametro in un metodo se esso lo consente

```
(input-parameters) => { <sequence-of-statements> }
```

```
var result = list.Where(n => n > 2);
```

# Lambda Expression

- Rappresentazione sintetica
- Sintassi definita tramite funzioni Anonime
  - Non richiede la parola chiave ***delegate***
  - Non richiede la parola chiave ***return***
- Utilizzo dell'operatore **=>**
  - **A sinistra:** firma della funzione
  - **A destra:** statement della funzioni

# 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");  
    }  
);
```

# Lambda Expression

Lambda Expression usano particolari **delegate**:

- **Action<T>**
  - Non ritornano un valore
- **Func<T>** e **Expression<T>**
  - Ritornano un valore

```
Func<int, int> square = x => x * x;  
Func<int, int, int> mult = (x, y) => x * y;  
Action<int> print = x => Console.WriteLine(x);  
print(square(mult(3, 5)));
```

# Lambda Expression

## Func<T>

- Delegato generico
- Non accetta parametri
- Restituisce un tipo T

## Expression <Func<T>>

- Rappresentano una lambda expression tipizzata
- Expression come struttura dati

```
Expression<Func<int, int, int>> Multiply = (x, y) => x * y;  
Func<int, int, int> mult = Multiply.Compile();  
int result = mult(2,3);
```

# Expressions Trees

Compilazione a Runtime  
`squareExpression.Compile();`

## Expression<T>

- Il compilatore genera un **expression tree**

```
Expression<Func<int, int>> squareExpression = x => x * x;
```



```
ParameterExpression x;  
Expression<Func<int, int>> squareExpression =  
Expression.Lambda<Func<int, int>>(  
Expression.Multiply(x = Expression.Parameter(typeof(int), "x" ),x),  
new ParameterExpression[] { x });
```



# Demo

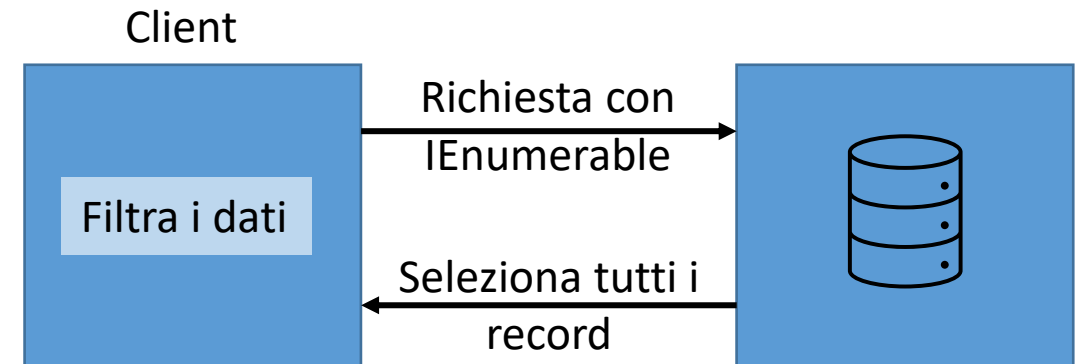


Implicit and Anonymous Types  
Delegati e Espressioni Lambda in LINQ  
Set-up Applicazione  
Extension Method



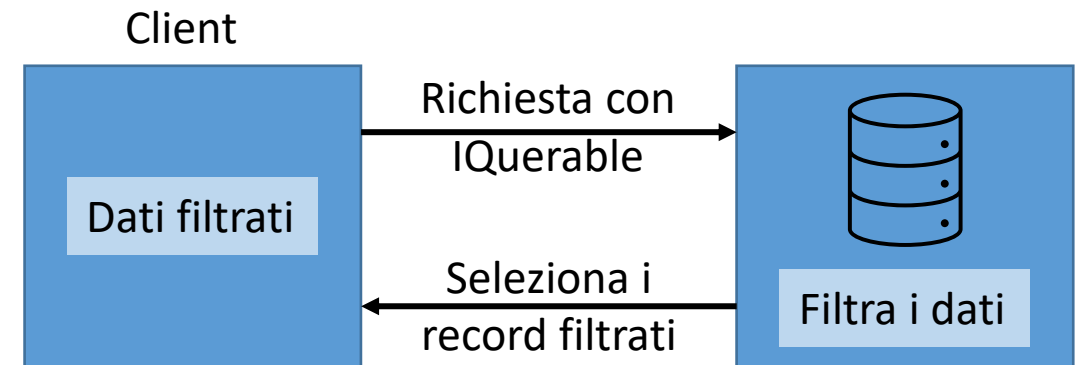
# IEnumerable

- Interfaccia del namespace *System.Collections*
- Esegue la query caricando i dati in memoria e in seguito filtrandoli
- Adatto per query su documenti XML e Objects



# IQueryable


- Interfaccia del namespace *System.Linq*
- Esegue la query filtrando i record direttamente sul server. Tiene in memoria i dati già filtrati.
- Adatto per query più “estese” (es. SQL Server)
- Eredita dall'interfaccia *IEnumerable*



# Esecuzione di Query

## IEnumerable<T>


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



LINQ TO OBJECT

## IQueryable<T>

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



LINQ TO SQL

# LINQ to Objects



# Struttura di una Query in LINQ

1. Inizializzazione sorgente dati
2. Creazione della query
3. Esecuzione della query

# Creazione della Query

## Method Syntax:

- Extension Methods
- Lambda expressions
  - Delegati
  - Expression Trees



```
IEnumerable<string> filterCities =cities
    .Where( city =>city.StartsWith("L") &&
           city.Lenght <15)
    .Select(city);
```

## Query Syntax:

- Inizia sempre con from
- Finisce con select o group




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

# 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)  
{  
    ...  
}
```



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();
```



# Esecuzione Differita

- L'esecuzione di una query non avviene al momento della creazione della query in generale
- La query vengono eseguite quando bisogna vederne il risultato (es: vederne i risultati con foreach, o per un conteggio .Count() )
- Si può forzare l'esecuzione con alcuni operatori (esempio: .ToList())

# Operatori Standard

- Extension Methods – Namespace ***System.Linq***
  - Enumerable e Queryable
- **IEnumerable<T>** e **IQueryable<T>**
- Due categorie di operatori
  - **Esecuzione differita (deferred)**
  - **Esecuzione immediata**

# Demo



Esecuzione Differita  
Query e Method Syntax



# 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 – Selezione, Partizionamento

## Selezione

Metodi	Descrizione
Select	Proiezione in una sequenza basata su una funzione di trasformazione
SelectMany	Proiezione di sequenze multiple

```
string[] famousQuotes =  
{  
    "Advertising is legalized lying" ,  
    "Advertising is the greatest art form  
    at the twentieth century"  
};
```

```
var query = (from sentence in famousQuotes  
             from word in sentence  
             .Split(' ') select word).Distinct();
```

# LINQ – Selezione, Partizionamento

## Partizionamento

Metodi	Descrizione
Skip / SkipWhile	Salta gli elementi fino alla condizione o un predicata
Take / TakeWhile	Seleziona gli elementi fino ad una condizione o un predicate

```
var query = numbers.Skip(2).Take(2);
```

```
var query = numbers.SkipWhile(n => n < 5)  
                    .TakeWhile(n => n < 10);
```

# LINQ – Operatori Standard

## Filtro

Metodi	Descrizione
Where	Filtra dati attraverso Predicate
OfType	Filtra dati attraverso il loro tipo

```
ArrayList list = new ArrayList();  
list.Add("value1");  
list.add(new object());  
list.add("value2");  
list.add(new object());  
  
var query =  
from name in list.OfType<string>()  
select name;
```

# LINQ – Operatori Standard

## Ordinamento

Metodi	Descrizione
OrderBy, OrderbyDescending	Ordinamento crescente Ordinamento decrescente
ThenBy, ThenByDescending	Secondo operatore di ordinamento
Reverse	Reverse dei dati

```
string[] names = { "Bob", "Alice", "Alex", "Carol" };
```

```
var query = names.OrderBy(s => s)  
    .ThenBy(s => s.Length);
```

```
query =  
    from name in names  
    orderby name, name.Length  
    select name;
```



# LINQ – Operatori di Confronto

## Confronto

- Utilizzano IEqualityComparer
- I tipi anonimi generati dal compilatore:
  - Override di Equals e GetHashCode
  - Usano tutte le proprietà pubbliche per la comparazione

```
var employees = new List<Employee> {  
    new Employee() { ID=1, Name= "Scott"},  
    new Employee() { ID=2, Name= "Poonam"},  
    new Employee() { ID=1, Name= "Scott"}  
};
```

```
var employees = // 3 employees  
(from employee in employees  
 select employee).Distinct();
```

```
var query = // 2 employees  
(from employee in employees  
 select new { employee.ID, employee.Name  
 }).Distinct();
```

# LINQ – Quantificatori

## Quantificatori

Metodi	Descrizione
All	Tutti gli elemendi soddisfano la condizione
Any	Solo alcuni elementi soddisfano la condizione
Contains	Test se la sequenza contiene elementi specifici

```
int[] twos = { 2, 4, 6, 8, 10 };  
bool areAllEvenNumbers = twos.All(i => i % 2 == 0);  
bool containsMultipleOfThree = twos.Any(i => i % 3 == 0);  
bool hasSeven = twos.Contains(7);
```

# LINQ – Element

## Element

Metodo	Descrizione
ElementAt / ElementAtOrDefault	Ritorna gli elementi ad un indice specifico
First / FirstOrDefault	Ritorna il primo element di una collezione
Last / LastOrDefault	Ritorna l'ultimo elemento di una collezione
Single / SingleOrDefault	Ritorna un singolo elemento

```
string[] empty = { };  
string[] notEmpty { "Hello", "World"};  
var result = empty.FirstOrDefault(); //null  
result = empty.First(); //InvalidOperationException  
  
result = notEmpty(s => s.StartsWith("s"));
```

# Demo



Operatori standard



# LINQ - Raggruppamento

## Raggruppamento

Metodi	Descrizione
GroupBy	Raggruppa gli elementi di una sequenza
ToLookup	Inserisce gli elementi in un dictionary uno a molti

```
int[] numbers { 1,2,3,4,5,6,7,8,9}  
var query = numbers.GroupBy(i => i % 2);
```

# LINQ - Raggruppamento

## Raggruppamento

Metodi	Descrizione
Join	Join tra due sequenze e ritorna una sequenza
GroupJoin	Join tra due sequenze e ritorna un gruppo di sequenze

```
var query = employees.Join(
    departments,
    e => e.DepartmentID,
    d => d.ID,
    (e, d) => new {
        EmployeeName = e.Name,
        DepartmentName = d.Name
    });
```

//sequenza  
//outer key  
//inner key  
//risultato

# LINQ - Aggregazione

## Aggregazione

Metodo	Descrizione
Average	Calcola la media dei valori di una sequenza
Count / LongCount	Conta il numero di elementi di una sequenza
Max	Ritorna il Massimo valore di una sequenza
Min	Ritorna il minimo valore di una sequenza
Sum	Calcola la somma dei valori di una sequenza

# LINQ – Conversione di Tipo

## Conversioni di Tipo

Metodo	Descrizione
AsEnumerable	Ritorna un IEnumerable<T>
AsQueryable	Converte un IEnumerable<T> in IQueryable<T>
Cast	Converte tutti gli elementi in un tipo
OfType	Filtra i valori in base ad un tipo
ToArray	Converte la sequenza in un array (immediatamente)
ToDictionary	Converte la sequenza in Dictionary<K,V>
ToList	Converte la sequenza in List<T>
ToLookup	Raggruppa gli elementi in IGrouping<K,V>



# LINQ – Let e Into

## Keyword Let

- Consente di rendere più leggibili le query

```
var allAuthors =  
    from author in authorTable  
    let aname = author.Name.ToLower()  
    where aname == "andrew"  
    orderby a.Id  
    select aname;
```

# LINQ – Let e Into

## Keyword Into

- Permette di continuare una query dopo una proiezione
- Quando si eseguono dei raggruppamenti

```
var allAuthors =  
    from author in authorTable  
    where author.name == "andrew"  
    select author  
        into pAuthor  
    Where pAuthor.Name.Length < 5  
    Select pAuthor
```

# Demo



Join

Operatori di raggruppamento



# LINQ to SQL



# Struttura di una Query in LINQ

1. Inizializzazione sorgente dati
2. Creazione della query
3. Esecuzione della query

# Data Context

- Classe di System.Data.Linq
- Serve per connettersi al database, recuperare “objects” e eventualmente mandare modifiche al db
- Include le funzionalità al SqlConnection di ADO.Net

# Creazione Data Context

Configurazione tramite “Linq to Sql Tools”

- Si crea una Data Connection sul database
- Tramite O/R Designer si crea *il modello* che mappa le classi delle tabelle in un database e il *DataContext*
- Il modello viene salvato in un file *dbml*

# Creazione Query

- Valgono gli stessi argomenti di LINQ to Object
- Unica differenza è l'uso di IQueryable



```
Northwnd db = new Northwnd(@"c:\northwnd.mdf");

// Query for customers in London.
IQueryable<Customer> custQuery =
    from cust in db.Customers
    where cust.City == "London"
    select cust;
```



# Manipolazione Dati

Inserire, aggiornare o eliminare righe dal database:

1. Manipolazione Dato (insertOnSubmit(obj), deleteOnSubmit(obj))
2. Submit al database (SubmitChanges())

# Concurrency

- Try and Catch (ChangeConflictException)
- Si può specificare quando mandare l'eccezione:
  - Manda eccezione al primo fallimento (FailOnFirstConflict)
  - Accumola le eccezioni (ContinueOnConflict)

```
db.SubmitChanges(ConflictMode.FailOnFirstConflict);  
// or  
db.SubmitChanges(ConflictMode.ContinueOnConflict);
```

# Riconciliare Valori nel db

- 1) Preservare i valori del database
- 2) Sovrascrivere i valori del database
- 3) Unire i valori con il database (Merging)

# Preservare i valori del database

Metodo: **OverwriteCurrentValues**

I valori correnti nell'object model vengono sovrascritti.

Database iniziale:

FirstName	LastName	Cachet
Leo	Nimoy	60 000

User 1: Leo ->Leonard

User 2 : 60 000->80 000

Database finale:

FirstName	LastName	Cachet
Leonard	Nimoy	60 000

# Sovrascrivere i valori del database

Metodo: **KeepCurrentValues**

Sovrascrivono i dati del database

Database iniziale:

FirstName	LastName	Cachet
Leo	Nimoy	60 000

User 1: Leo ->Leonard

User 2 : 60 000->80 000

Database finale:

FirstName	LastName	Cachet
Leo	Nimoy	80 000

# Unire i valori del database

Metodo: **KeepChanges**

Unisce i dati correnti del database e del object model

Database iniziale:

FirstName	LastName	Cachet
Leo	Nimoy	60 000

User 1: Leo ->Leonard

User 2 : 60 000->80 000

Database finale:

FirstName	LastName	Cachet
Leonard	Nimoy	80 000

# Demo



DataContext

Manipolazione dati in LINQ to SQL

Gestione Eccezioni



# Grazie!



Ricordate il feedback!





# © 2019 iCubed Srl

La diffusione di questo materiale per scopi differenti da quelli per cui se ne è venuti in possesso è vietata.

iCubed s.r.l.

Piazza Durante, 8 20131 MILANO

Phone: +39 02 57501057

P.IVA 07284390965

