Sommario

10

Panoramica generale

1 Introduzione

Se nella versione 2.0 del .NET e del linguaggio C# ci si è voluti concentrare sull'aspetto della programmazione "Generica", con la versione 3.0 del C# si è affrontato il problema dell'accesso a dati provenienti da diverse sorgenti; Oggetti in menoria, database ed xml sono solo alcune delle possibili fonti di dati, ma anche spesso di problemi.

Invece di aggiungere altre classi e metodi che facilitassero la gestione dedei dati, il team di sviluppo di Microsoft ha deciso di fare un passo in avanti, astraendo le parti fondamentali dell'accesso ai dati per queste particolari sorgenti.

Il risultato è stato LINQ (Language INtegrated Query) che fornisce un meccanismo di supporto a livello di linguaggio pr l'interrogazione di dati di "tutti" i tipi; Vettori, Collezioni, Database, documenti XML ed altro ancora.

1.1 C'era una volta ObjectSpaces

Eravamo intorno alla metà del 2001 quando in una PDC (Professional Developer Conference) Mircosoft fece vedere le prime dimostrazioni di una nuova tecnologia chiamata ObjectSpaces che doveva implementare una spece di Object-Maping relazionale e facilitare in questo modo l'accesso dai database da parte delle applicazioni.

Noi vedemmo per la prima volta dal vivo questa tecnologia in una presentazione del suo inventore (Luca Bolognese, italiano al 100%) ad una Windows Professional Conference nell'ottobre 2003.

ObjectSpaces era una API per l'accesso ai dati per ADO.NET che consentiva di trattare gli stessi indipendentemente dal database sul quale fossero fisicamente memorizzati ed introduceva un linguaggio di interrogazione proprietario chiamato OPath.

Nel 2004 ci fu l'annuncio che ObjectSpaces sarebbe diventato parte integrante di WinFS (il nuovo futuribile File System ce avrebbe dovuto far parte di Windows Vista), ma quando si persero le tracce di WINFS non si ebbe più notizie nemmeno di ObjectSpaces

1.2 XQuery per l'XML

Quanto accaduto per ObjectsSpaces avvenne per XQuery, un modello per astrarre l'accesso a documenti XML, con il solito problema di introdurre un proprio linguaggio proprietario per l'esecuzione delle interrogazioni.

1.3 Cω (Il linguaggio C-Omega)

Cω (si pronuncia "see-omega") era un progetto dei laboratori di ricerca Microsoft che estendeva il linguaggio C# in alcune aree come la gestione dei processi asincroni ed estendeva dei tipi di dato da utilizzare nell'accesso a DB ed XML (conosciuti come Xen e X#).

In verità questo "strano" linguaggio faceva un sacco di altre cose, tra cui provare ad integrare l'accesso ai dati nelle interrogazioni utilizzando C# ed SQL oppure C# e XQuery.

2 Arriva LINQ

Tutti questi antenati "scomparvero" fino al 2005, quando, sempre alla solita PDC, Microsoft annunciò il progetto LINQ.

LINQ è stato ideato dal "solito" Anders Hejlsberg ed altri per risolvere la problematica della differenza di accesso ai dati all'interno dei linguaggi come C# e VB.NET.

Con LINQ si può interrogare praticamente "tutto" ed è il motivo per cui sono stati abbandonati gli altri progetti in favore di questa tecnologia

2.1 LINQ per la manipolazione e l'interrogazione dei dati

A causa del suo nome che sta appunto per Language INtegrated Query, si è spesso portati ad immaginare l'utilizzo di LINQ solo per l'esecuzione di interrogazioni, mentre s dovrebbe pensare a questa tecnologia come a un motore per l'interazione tra i dati.

È infatti possibile utilizzare LINQ per convertire strutture dati, crearne di nuove, selezionare sottoinsiemi di collezioni esistenti o unirne due o più e per iniziare a prendere confidenza con questa tecnologia analizzeremo per primo questo aspetto.

Per la maggior parte, però, LINQ ha questo scopo primario sia che si voglia recuperare un singolo valore, un oggetto, oppure un insieme di oggetti, campi di un database oppure elementi di un documento Xml.

In LINQ gli insiemi di oggetti vengono chiamate sequenze e molte di queste sequenza sono del tipo `IEnumerable<T>` dove `T` è il tipo di dato memorizzato nella sequenza e quindi se avessimo una sequenza di interi essa verrebbe memorizzata in una variabile di tipo `IEnumerable<int>`.

2.2 Componenti

Per poter utilizzare LINQ è necessario dotarsi di tutti quegli strumetni chefanno parte dell'ondata di "Orcas" (il nome in codice utilizzato nelle versioni preliminari) che includono Visual Studio 2008, le varie versioni Express (Visual C#, VB, Web Develper) ed ovviamente il .NET Framework versione 3.5

LINQ è costituito da compilatori e liberie, ma non da un runtime, essendo questo rimasto invariato nel passaggio dalla versione 2.0 a quella 3.x del .NET.

LINQ non è un qualcosa di hard-coded all'interno del CLR, ma si compone di parti che possono essere estese implementando determinate interfacce.

Nel .NET 3.5 vengono forniti i componenti che Microsoft ritiene indispensabile e cioè:

- LINQ to Object per l'esecuzione di query su vettori e collezioni in memoria
- LINQ to XML per l'esecuzione di query documenti o frammenti di XML
- LINQ to SQL per l'esecuzione di query sul database SQLServer
- LINQ to DataSet per l'esecuzione di query su dataset in memoria
- LINQ to Entity per l'esecuzione di query su qualsiasi database

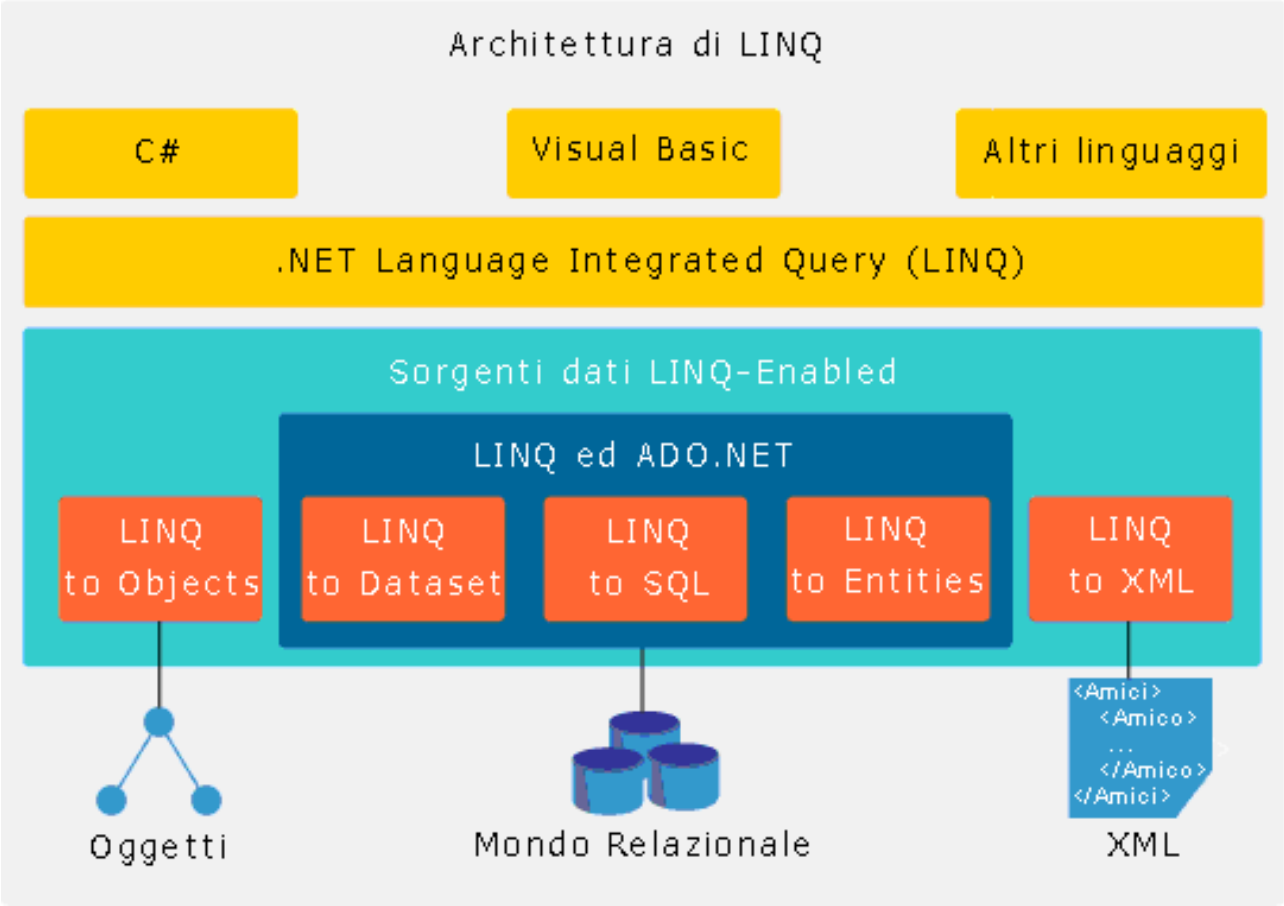

Figura 10-1 Architettura di LINQ

3 Come LINQ estende il .NET

La creazione della tecnologia di LINQ ha portato una serie di benefici, ma anche la necessità di introdurre alcune nuove caratteristiche del .NET necessarie alla riuscita del progetto ed in dettaglio:

- Variabili locli tipizzate implicitamente
- Inizliazzazione di Oggetti e Collezioni
- Tipi Anonimi
- Espressioni Lambda
- Metodi di Estensione

3.1 Variabili locali tipizzate implicitamente

Con questa caratteristica il tipo di una variabile locale viene "dedotto" dall'espressione utilizata per inizializzarla. Per fare ciò si utilizza la parola chiave `var` come nel seguente esempio

```
var num = 50;
var str = "Pietro";
```

```
var ogg = new MiaClasse();
var numeri = new int[] { 1, 2, 3 };
var lista = new List<int>();
```

Il compilatore genererà il giusto codice IL come se avessimo scritto in questa maniera

```
int num = 50;
string str = "stefano";
MiaClasse ogg = new MiaClasse();
int[] numeri = new int[] { 1, 2, 3 };
List<int> lista = new List<int>();
```

E' molto importante notare che la variabile non è senza tipo che verrà poi inizializzato a livello di runtime (un po' come l'object del JavaScript) ma essendo dedotto in fase di compilazione essa sarà una normale variabile tipizzata fortemente.

3.2 Inizializzazione di Oggetti e Collezioni

Per comprendere l'utilità di questa caratteristica ipotizziamo di avere una semplice classe come quella che segue.

```
public class Socio
{
    private string _cognome;
    public string Cognome
    {
        get { return _cognome; }
        set { _cognome = value; }
    }

    private string _nome;
    public string Nome
    {
        get { return _nome; }
        set { _nome = value; }
    }
}
```

Se volessimo dichiarare ed inizializzare un oggetto di questo tipo dovremmo scrivere così:

```
Socio io = new Socio();
io.Cognome = "Bagiacchi";
io.Nome = "Fabrizio";
```

utilizzando l'inizializzazione di oggetti si può avere una forma più compatta:

```
Socio tu = new Socio() { Cognome = "Bagiacchi", Nome = "Fabrizio" };
```

Sfruttando questa opportunità inizializzare anche una collezioni nel momemto della sua dichiarazione, come nell'esempio seguente:

```
List<int> voti = new List<int>() { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
```

come se fosse un vettore.

Unendo le due cose possiamo scrivere il seguente codice

```
List<Socio> soci = new List<Socio>{
                        new Socio { Cognome = "Bagiacchi", Nome = "Fabrizio" },
                        new Socio { Cognome = "Inghirami", Nome = "Luca" }
};
```

3.3 Tipi anonimi

In realtà la classe creata precedentemente è stata realizzata solo "definire" un dato che contenga un cognome ed un nome in maniera strutturata; probabilmente, però, useremo questa classe solo in poche (se non in una sola) occasioni e quindi sarebbe interessante avere un meccanismo che ci consentisse di creare "al volo" tipi di dato per poter organizzare meglio alcune informazioni in certi momenti.

Con l'introduzione dei tipi anonimi, è stata data la possibilità di definire tipi in-linea senza necessariamente definire una classe per questi tipi.

In altre parole tornando all'esempio precedente immaginiamo di voler usare un tipo Socio senza, però, essere costretti a definire una classe Socio (in maniera quindi anonima)

Il codice risultante sarebbe il seguente.

```
var nuovo = new { Cognome = "Gennaioli", Nome = "Gianluca" };
```

All'interno dell'editor avremo pieno supporto all'intellisense con tutti i vantaggi del caso, mentre l'accesso ai dati membro dell'oggetto avverrà nella solita maniera.

```
. . .
nuovo.Cognome = "Sartini";
nuovo.Nome = "Andrea";
Console.WriteLIne(nuovo.Cognome);
. . .
```

3.4 Espressioni Lambda

Le espressioni lambda sono un modo per scrivere blocchi di codice in-linea e sono una evoluzione dei metodi anonimi introdotti dal .NET 2.0 (e diventati quindi obsolti).

Al contrario dei metodi anonimi (terribilmente verbosi a scriversi) le espressioni lambda forniscono una sintassi concisa e funzionale; esse vengono scritte come una lista di parametri (che possono essere definiti implicitamente) seguiti dalla coppia di simboli => seguiti da una espressione o da un blocco di codice come nell'esempio seguente

```
mioIndirizzo => indirizzo.Via == "Abetone"
```

Per dimostrare l'origine e l'evoluzione di questo costrutto ipotizziamo di voler ordinare un vettore separando i numeri pari da quelli dispari.

Prima dell'introduzione del .Net dovevamo implementare l'algoritmo di ordinamento scegliendo tra quelli con cui avevamo più dimestichezza, mentre con la versione del .NET 1.1 era necesssario scrivere un metodo di helper che contenesse la "logica" di ordinamento.

```
Array.Sort(elenco, Confronta);
....
int Confronta(int a, int b)
{
    int ris = (a % 2).CompareTo(b % 2);
    if (ris == 0)
        ris = a.CompareTo(b);
    return ris;
```

```
}
```

Successivamente, co**on** l'introduzione dei metodi anonimi, il codice necessario si poteva ridurre a questo:

```
Array.Sort(elenco, delegate(int a, int b)
{
    int ris = (a % 2).CompareTo(b % 2);
    if (ris == 0)
    ris = a.CompareTo(b);
    return ris;
    }
);
```

Con le funzioni lambda tutto diventa molto più leggibile e compatto.

```
Array.Sort(elenco, ((a, b) =>
    {
        int ris = (a % 2).CompareTo(b % 2);
        if (ris == 0)
            ris = a.CompareTo(b);
        return ris;
    })
);
```

Giusto per fare un altro esempio, ipotizzando di voler ordinare l'elenco dei soci visto in precedenza in base al nome decrescente, la espressione lambda sarà essere la seguente.

```
soci.Sort((s1, s2) => s2.Nome.CompareTo(s1.Nome));
```

3.5 Metodi di estensione

Abbiamo già parlato a lungo dei metodi di estensione e qui ci limitiamo a rinfrescare il fatto che con questa caratteristica si possono estendere tipi di dato senza la necessità di derivarli e questi metodi diventeranno parte integrante del tipo.

Per definire un metodo di estensione lo si deve dichiarare di tipo statico all'interno di una classe statica, e se ipotiziamo di voler aggiungere alla classe string un metodo che verifichi se una stringa è scritta in maiuscolo questo sara il codice necessario.

```
public static class Estensione
{
    public static bool IsMaiuscola(this string s)
    {
        return s == s.ToUpper();
    }
}
```

Da notare la parola chiave **this** prima del primo parametro del metodo; la sua presenza trasforma un comune metodo in un metodo di estensione che potrà essere utilizzato in questo modo;

```
string parola = "CIAO";
bool ok = parola.IsMaiuscola();
```

11

Le basi di LINQ

1 Introduzione

Le espressioni di query sono la caratteristica che ha attratto di più gli sviluppatori fin dalla prime versioni, in quanto fornisce una sintassi SQL-Like per l'accesso ai dati strutturati.

Nelle espressioni di query si ritrovano tutte le innovazioni del .Net analizzate fin qua e sono una estensione al linguaggio C# presente a partire dalla verisione 3.0.

Passiamo allora a fare un esempio pratico che ci permetterà di applicare tutti i concetti fin qua espressi ed iniziare ad ianalizzare la basi di LINQ per incominciare a intravvederne la potenza.

Ipotizziamo di avere la seguente classe che rappresenta l'anagrafica di un socio di un club:

```
public class Socio
{
    private string _cognome;
    public string Cognome
    {
        get { return _cognome; }
        set { _cognome = value; }
    }

    private string _nome;
    public string Nome
    {
        get { return _nome; }
        set { _nome = value; }
    }

    private string _provincia;
    public string Provincia
    {
        get { return _provincia; }
        set { _provincia = value; }
    }
}
```

Utilizziamo l'inizializzazione di oggetti e collezioni per creare un elenco dei soci

```
List<Socio> elenco = new List<Socio>() {
    new Socio {Nome="Massimo", Cognome="Santinelli", Provincia="PG"},
    new Socio {Nome="Marco",  Cognome="Biagini", Provincia="PG"},
    new Socio {Nome="Danilo",  Cognome="Carobini", Provincia="PU"},
```

```
    new Socio {Nome="Marco",  Cognome="Basili", Provincia="PG"},
    new Socio {Nome="Luca",  Cognome="Inghirami", Provincia="AR"}
};
```

ed ora che abbiamo i dati su cui lavorare scriviamo la nostra prima espressione di query.

```
var soci = from dato in elenco
           where dato.Provincia == "PG"
           orderby dato.Cognome
           select new { dato.Cognome, dato.Provincia };
```

Eseguendo il codice otterremo il seguente risultato.

```
{ Cognome = Basili, Provincia = PG }
{ Cognome = Biagini, Provincia = PG }
{ Cognome = Santinelli, Provincia = PG }
```

In realtà per eseguire una query LINQ non è richiesto l'uso delle espressioni di query in quanto esse vengono tradotte in una notazione classica che utilizza classi, metodi ed oggetti e potremmo quindi riscrivere la query di prima in quest'altro modo

```
var soci = elenco
           .Where(dato => dato.Provincia == "PG")
           .OrderBy(dato => dato.Cognome)
           .Select(dato => new { dato.Cognome, dato.Provincia });
```

ottenendo lo stesso risultato.

2 Struttura delle espressioni di Query

Le espressioni di query sono la caratteristica che ha attratto di più gli sviluppatori fin dalla prime versioni, in quanto fornisce una sintassi SQL-Like per l'accesso ai dati strutturati.

Andiamo ad analizzare i singoli frammenti delle due espressioni aiutandoci con la seguente figura

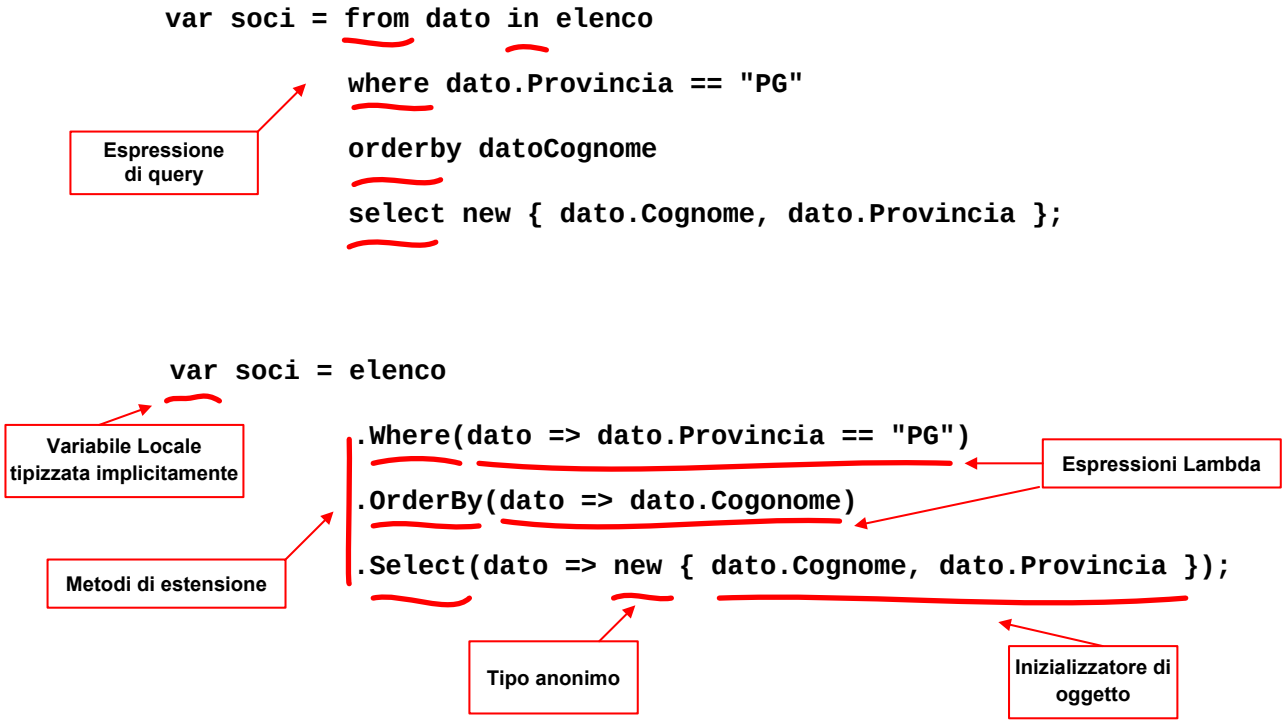

Figura 11-1 Espressioni di query

La prima cosa che salta all'occhio quando si osservano le espressioni di query è il fatto che contrariamente al linguaggio SQL la parola chiave **`select`** è posta in fondo all'espressione che inizia sempre s**on** la parola chiave **`from`**.

Questa "stranezza" ha alimentato numerosi dibattiti nel corso dello sviluppo di LINQ, ma la principale ragione di questo scambio sta nella funzionalità di Intellisense tipica degli ambienti di sviluppo odierni.

Senza l'inversione tra **`from`** e **`select`**, l'intellisense non avrebbe alcuna idea di quale variabile mostrare nell'elenco che di solito appare quando si preme la barra spaziatrice.

Specificando invece da dove arrivano i dati, l'intellisense ha la possibilità di mostrare l'ambito di scelta delle variabili disponibili.

L'ordine di esecuzione dei vari metodi è invece abbastanza evidente, ed infatti partendo dall'alto verso il basso avremo:

a) recupero del contenuto della variabile elenco

b) esecuzione del filtro sull'elenco

c) ordinamento dei risultati

d) selezione del cognome e della provincia

3 Anatomia di una espressione di query

Ci sono molti elementi in gioco in LINQ per far si che tutto funzioni oltre a quelli appena visti ed è giunto il momento di andarli ad analizzare.

3.1 Sequenza

Il primo concetto da analizzare è quello di sequenza e per fare ciò riprendiamo la query precedente.

```
var soci = elenco
           .Where(dato => dato.Provincia == "AR")
           .OrderBy(dato => dato.Cognome)
           .Select(dato => new { dato.Cognome, dato.Provincia });
```

Visto che la variabile `soci` è di tipo `var`, la domanda da porsi è: "Qual è il valore ritornato dall'espressione ?"

Per rispondere a questa domanda bisogna sapere che tutto si basa sulla presenza dell'interfaccia `IEnumerable<T>`.

Infatti l'esecuzione di espressioni di query può essere effettuata solamente su tipi di dato che implementano questa interfaccia e sono quindi in grado di permettere la enumerazione degli elementi.

In LINQ una collezione che implementa questa interfaccia si chiama **sequenza** e quindi se abbiamo una variabile di tipo `IEnumerable<T>` allora possiamo dire di avere una sequenza di oggetti di tipo `T`.

La maggior parte deli operatori di query di LINQ sono metodi di estensione presenti nella classe statica `System.Linq.Enumerable` (presente nell'assembly `System.Core.DLL`) che come primo argomento hanno un oggetto di tipo `IEnumerable<T>`[1] e ritornano oggetti di tipo `IEnumerable<T>`.

Quindi, per rispondere alla domanda che ci eravamo posti prima: "Una espressione di query restituisce (quasi) sempre una sequenza".

3.2 Esecuzione differita

Adesso che sappiamo che cosa viene restituito, ci verrebbe naturale pensare che l'esecuzione dell'istruzione avvenga immediatamente e che la variabile destinataria del risultato sia immediatamente popolata.

Peccato, però, che gli operatori non restituiscano immediatamente l'intera sequenza nel momento in cui essi vengono chiamati e quindi la nostra variabile non conterrà il risultato, ma il "potenziale" risultato.

Gli operatori di espressione, infatti, ritornano un oggetto che solamente quando viene enumerato "rende" (yield[2]) disponibile l'elemento della sequenza.

In definitiva è solamente durante l'enumerazione dell'oggetto che la query viene effettivamente svolta e questo è quello che si chiama esecuzione differita (*deffered query execution* o anche *lazy evaluation*).

Per dimostrare ciò scriviamo il seguente frammento di codice

```
int[] voti = new int[] { 5, 7, 8 };
var esame = from valore in voti
            select valore;

// equivalente a
// var esame = voti.Select(valore => valore);

foreach (var voto in esame)
```

[1] E' bene notare però che la presenza del parametro T esclude dall'utilizzo di LINQ tutte quelle collezioni "pre" .NET 2.0 come ArrayList ed altri, ma di questo ci occuperemo più avanti.

[2] Per un approfondimento sugli iteratori consultare i capitoli precedenti.

```
{
    Console.WriteLine(voto);
}
```

Il risultato sarà il seguente:

```
5
7
8
```

Proviamo adesso ad aggiungere le seguenti righe che modificano il primo elemento e ripetono il ciclo per visualizzare la sequenza

```
voti[0] = 6;
foreach (var voto in esame)
{
    Console.WriteLine(voto);
}
```

Il risultato che otterremo sarà diverso:

```
6
7
8
```

Al contrario di quanto si possa inizialmente credere questo è uno dei concetti più importanti di LINQ in quanto senza di esso le sue prestazioni sarebbero veramente scarse.

Ipotizziamo infatti di avere una espressione che restituisca "potenzialmente" 10000 elementi ma che, dopo aver analizzato il valore del primo, decidiamo che gli altri non ci interessino.

Con l'esecuzione differita solamente il primo valore verrebbe restituito mentre gli altri non verrebbero nemmeno caricati in memoria.

Inoltre questa caratteristica ci consenti di scrivere una query una volta e riutilizzarla quante volte ci pare.

3.3 Esecuzione immediata

Il comportamento precedente, però non ha solo vantaggi ma anche un effetto collaterale.

Se infatti tra una enumerazione e l'altra la sequenza sorgente dalla quale restituire i dati cambia anche l'enumerazione della variabile di query cambierà di conseguenza.

Per ovviare a ciò LINQ mette a disposizione dei metodi di estensione che provocano l'immediata esecuzione della espressione e la memorizzazione della sequenza in una collezione dati, come una lista od un vettore.

3.4 Gli operatori di query standard

Adesso che abbiamo visto che cosa accade dietro le quinte di una espressione di query in LINQ, possiamo concentrare la nostra attenzione sui uno dei protagonisti di questa tecnologia.

Stiamo parlando degli operatori di query standard e la tabella che segue li elenca raggruppandoli per funzionalità:

Tabella 11-1 Gli operatori standard di query raggruppati in famiglie

SCOPO / NOME	ESECUZIONE DIFFERITA	RITORNA SEQUENZA	DESCRIZIONE

Filtro			
OfType	x	x	Seleziona i valori in base alla possibilità di effettuare il cast verso un certo tipo
Where	x	x	Seleziona i valori in base ad un predicato
Proiezione			
Select	x	x	Seleziona i valori in base ad una funzione di selezione
SelectMany	x	x	Seleziona i valori, in base ad una funzione di selezione e combina le sequenze risultatnti in un unica sequenza, eseguendo una proiezione del tipo uno-a-molti effettuando quindi un prodotto cartesiano.
Partizionamento			
Skip	x	x	Salta n elementi da una sequenza
SkipWhile	x	x	Salta gli elementi di una sequenza in base ad un predicato finché un elemento non soddisfa quella condizione
Take	x	x	Restituisce n elementi da una sequenza
TakeWhile	x	x	Restituisce gli elementi di una sequenza in base ad un predicato finché un elemento non soddisfa la condizione
Join			
Join	x	x	Unisce due sequenze in base ad una funzione chiave di selezione ed estrae coppie di valori
GroupJoin	x	x	Unisce due sequenze in base ad una funzione chiave di selezione e raggruppa i risultati corrispondenti per ogni elemento
Concatenazione			
Concat	x	x	Concatena due sequenze per formarne una
Ordinamento			
OrderBy		x	Ordina i valori in ordine crescente
OrderByDescending		x	Ordina i valori in ordine decrescente
ThenBy		x	Esegue un ordinamento secondario in ordine crescente
ThenByDescending		x	Esegue un ordinamento secondario in ordine decerescente
Reverse		x	Inverte gli ordini degli elementi di una sequenza
Raggruppamento			

GroupBy	x	x	Raggruppa gli elementi che hanno un attributo in comune. Ogni gruppo è rappresentato da un oggetto di tipo IGrouping<TKey, TElement>
ToLookup		x	Inserisce gli elementi in una collezione di tipo Lookup<TKey, TElement> in base ad una funzione chiave di selezione
Insiemi			
Distinct	x	x	Rimuove i valori duplicati da una sequenza
Except	x	x	Restituisce i valori di una sequenza che con sono presenti in un'altra sequenza
Intersect	x	x	Restituisce i valori comuni presenti in due sequenze
Union	x	x	Restituisce i valori (univoci) che sono in una sequenza o nell'altra
Conversione			
AsEnumerable	x		Restituisce il tipo in ingresso come IEnumerable<T>
AsQueryable	x		Converte un tipo IEnumerable in IQueryable
Cast	x	x	Effettua il cast degli elementi di una sequenza verso un certo tipo
OfType	x	x	Seleziona i valori in base alla possibilità di effettuare il cast verso un certo tipo
ToArray			Converte una sequenza in un Vettore
ToDictionary			Inserisce gli elementi in un oggetto di tipo Dictionary<TKey, TElement> in base ad una funzione chiave di selezione
ToList			Converte una sequenza in una Lista
ToLookup			Inserisce gli elementi in una collezione di tipo Lookup<TKey, TElement> in base ad una funzione chiave di selezione
Uguaglianza			
SequenceEqual			Restituisce Vero se due sequenze sono uguali
Elemento			
ElementAt			Restituisce l'elemento alla posizione specificata
ElementAtOrDefault			Restituisce l'elemento alla posizione specificata oppure il valore default<T> se l'indice è fuori range
First			Restituisce il primo elemento della sequenza oppure quello che soddisfa una certa condizione
FirstOrDefault			Restituisce il primo elemento della sequenza oppure quello che soddisfa una certa condizione. Se non esistono restituisce default<T>

Last			Restituisce l'ultimo elemento della sequenza oppure quello che soddisfa una certa condizione
LastOrDefault			Restituisce l'ultimo elemento della sequenza oppure quello che soddisfa una certa condizione. Se non esistono restituisce default<T>
Single			Restituisce il solo elemento di una sequenza o quello che soddisfa una condizione. Se la sequenza non contiene quell'elemento solleva una eccezione di tipo InvalidOperationException
SingleOrDefault			Restituisce il solo elemento di una sequenza o quello che soddisfa una condizione. Se la sequenza non contiene quell'elemento restituisce default<T> mentre se ne contiene più di uno solleva una eccezione di tipo InvalidOperationException
Generazione			
DefaultIfEmpty	x	x	Fornisce una valore di default per una sequenza vuota
Empty	x	x	Restituisce una sequenza vuota
Range	x	x	Genera una sequenza di numeri interi in un intervallo specificato
Repeat	x	x	Genera una sequenza con un valore ripetuto
Quantificazione			
All			Restituisce Vero se tutti gli elementi di una sequenza soddisfano una condizione
Any			Restituisce Vero se un qualunque elemento di una sequenza soddisfa una condizione
Contains			Restituisce Vero se una sequenza contiene un certo elemento
Aggregazione			
Aggregate			Esegue una funzione di aggregazione sui valori di una sequenza
Average			Calcola la media dei valori di una sequenza
Count			Conta i valori di una sequenza e può, opzionalmente, contare solo quelli che soddisfano una condizione
LongCount			Conta i valori di una sequenza molto grande e può, opzionalmente, contare solo quelli che soddisfano una condizione
Max			Restituisce il valore massimo di una sequenza
Min			Restituisce il valore minimo di una sequenza
Sum			Calcola la somma dei valori di una sequenza

3.5 Le espressioni di query

L'ultimo concetto chiave di LINQ è formato dalle espressioni di query, che sono una estensione del linguaggio C# presente a partire dalla versione 3.0

Abbiamo già visto che gli operatori di query sono dei metodi statici che consentono di scrivere delle espressioni di query con una sintassi classica composta da metodi, classi ed oggetti.

```
var soci = elenco
           .Where(dato => dato.Provincia == "PG")
           .OrderBy(dato => dato.Cognome)
           .Select(dato => new { dato.Cognome, dato.Provincia });
```

ma abbiamo anche visto che è possibile esprimerla con una sintassi molto più SQL-Like

```
var soci = from dato in elenco
           where dato.Provincia == "PG"
           orderby dato.Cognome
           select new { dato.Cognome, dato.Provincia };
```

Questa è chiamata espressione di query oppure sintassi di query
I due frammenti di codice sono semanticamente identici con il vantaggio che la espressione di query è molto più dichiarativa e facile da scrivere.

La espressione di query fornisce una sintassi integrata nel linguaggio che è molto simile a quella utilizzata da SQL oppure da XQuery. Essa opera su una o più sogenti dati applicando uno o più operatori di query.

Quando si utilizzano queste espressioni, il compilatore traduce "magicamente" la istruzione in chiamate standard agli operatori di query.

Analizziamo ora la sintassi di una espressione di query che suonerà molto familiare a chi conosce SQL.

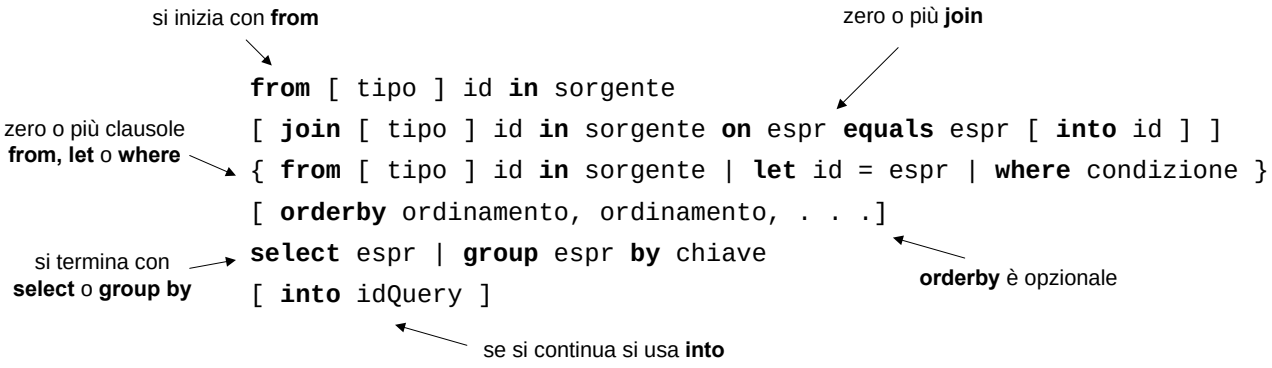

Figura 11-2 Sintassi delle espressioni di query

Una espressione di query inizia sempre con la clausola **`from`** e termina con una clausola **`select`** oppure `group`.

Dopo la clausola **`from`** ci possono essere zero o più clausole **`from`**, `let`, **`where`**, **`join`** oppure `orderby`

- Ogni clausola **`from`** genera una "variabile" contenente gli elementi di una sequenza.
- Ogni clausola `let` introduce una variabile calcolata a partire da variabili precedenti.
- Ogni clausola **`where`** rappresenta un filtro che esclude dei valori dal risultato finale

- Ogni clausola **join** confronta le chiavi specificate nella sequenza sorgente con quelle di un'altra sequenza restituendo le coppie corrispondenti
- Ogni clausola order**by** ordina i valori in base ad un certo criterio
- Le clausole finali **select** o **group** stabiliscono la "forma" che avranno i dati restituiti a partire dalle variabile introdotte nella espressione di query
- Alla fine la clausola into può essere utilizzata per suddividere più query trattando il risultato di una query come generatore di un'altra query.

3.6 Traduzione delle espressioni di query ed operatori di query

La traduzione che effettua il compilatore tra espressioni ed operatori avviene in base a delle regole espresse dalla seguente tabella.

Tabella 11-2 Corrispondenza tra operatori di query e sintassi C#

OPERATORE DI QUERY	SINTASSI C#
Cast	Utilizza un esplicito tipo per la variabile: Esempio **from int** i **in** sequenza
GroupBy	**group** .. by o **group** .. **by** .. into
Join	**join** .. **in** .. **on** .. equals ..
GroupJoin	**join** .. **in** .. **on** .. equals .. into ..
OrderBy	orderby
OrderByDescending	order**by** .. descending
Select	**select**
SelectMany	Più clausole **from**
ThenBy	order**by** .., ..
ThenByDescending	order**by** .., .. **descending**
Where	**where**

Come si può notare non tutti gli operatori hanno delle equivalenti parole chiavi in linguaggio C# (al contrario, per esempio del VB.NET) e questo significa che nelle query più semplici si potranno utilizzare quelle proposte dal linguaggio, mentre per interrogazioni più complesse si dovranno chiamare direttamente gli operatori di query.

Per chiarire meglio il concetto facciamo un esempio riprendendo la query precedente

```
var soci = from dato in elenco
           where dato.Provincia == "PG"
           orderby dato.Cognome
```

```
            select new { dato.Cognome, dato.Provincia };
```

ed ipotizziamo di voler prendere solo i primi due valori con l'operatore Take(); non essendoci una corrispondente parola chiave del linguggio dovremmo scrivere la query seguente

```
var primi2Soci = (from dato in elenco
                  where dato.Provincia == "PG"
                  orderby dato.Cognome
                  select new { dato.Cognome, dato.Provincia })
                  .Take(2);
```

Quello che abbiamo fatto è stato applicare l'operatore di espressione `Take`() al risultato della query scritta il linguaggio C# avendo avuto cura di racchiuderla tra parentesi tonde.

La versione della query fatta integralmente con gli operatori standard risulterebbe la seguente:

```
var primi2Soci = elenco
                  .Where(dato => dato.Provincia == "AR")
                  .OrderBy(dato => dato.Cognome)
                  .Select(dato => new { dato.Cognome, dato.Provincia })
                  .Take(2);
```

Che è sicuramente più leggibile e si presta a meno errori.

Sta solo al programmatore decidere quale delle due forme sia la più conveniente ad usarsi; in certi casi si preferirà utilizzare una combinazione di operatori ed espressioni mentre per rendere le cose più chiare mentre in altri contesti sarà più facile (e perfino più leggibile) utilizzare solo gli operatori.

12

Linq to Object

1 Introduzione

Adesso che abbiamo a disposizione tutti gli strumenti iniziamo a vedere LINQ in azione mediante il suo utilizzo su oggetti e collezioni dati, in quello che è uno dei pilastri di questa tecnologia e cioè Linq-to-Object.

Per comprendere meglio le funzionalità vedremo la sua implementazione utilizzando gli operatori di query standard e, se possibile, anche la versione corrispondente sfruttando le espressioni di query.

E' bene ricordare, inoltre, che questi operatori sono dei metodi di estensione che hanno come primo parametro una sequenza, che altro non è che un oggetto di tipo `IEnumerable<T>`, e quindi potremmo utilizzarli su qualsiasi collezione che implementi questa interfaccia.

Per scoprire la potenza di LINQ, possiamo iniziare a vedere i più semplici operatori a disposizione che ci consentono di risolvere le problematiche più comuni che si possono incontrano nella realizzazione di programmi.

2 Operatori di aggregazione, quantificazione ed elemento

2.1 Aggregazione: Average, Count, Max, Min, Sum

Gli operatori di aggregazione sono quelli di più semplice e diretto utilizzo, almeno nella loro forma più semplice e vediamo il loro utilizzo effettuando una serie di calcoli all'interno di una collezione.

Questi 4 metodi di estensione hanno una serie di overload per tutti i tipi di dato numerici ed un paio di sintassi di utilizzo

```
public static T Sum(
    this IEnumerable<T> source
)

public static T Sum<TSource>(
    this IEnumerable<TSource> source,
    Func<TSource, T> selector
)
```

E' posibile applicare l'operatore ad una intera sequenza oppure solamente a dei valori ottenuti mediante una funzione lambda.

Ipotiziamo, quindi di avere un vettore di interi come quello che segue.

```
int[] voti = { 4, 5, 6, 7, 8 };
```

e scriviamo del codice per utilizzare questi 5 operatori.

```
double media = voti.Average();
int conta = voti.Count();
int max = voti.Max();
int min = voti.Min();
int somma = voti.Sum();
```

visualizzando le 5 variabili otterremo il seguente risultato:

```
Media=6, Conteggio=5, Massimo=8, Minimo=4, Somma=30
```

Ovviamente è possibile applicare questi operatori anche a dati come collezioni costituite da oggetti complessi.

Prendiamo ad esempio la seguente classe che rappresenta le quote associative versate dai membri di un club.

```
public class Quota
{
    private int _id;
    public int Id
    {
        get { return _id; }
        set { _id = value; }
    }

    private int _idSocio;
    public int IdSocio
    {
        get { return _idSocio; }
        set { _idSocio = value; }
    }

    private int _importo;
    public int Importo
    {
        get { return _importo; }
        set { _importo = value; }
    }

    private int _anno;
    public int Anno
    {
        get { return _anno; }
        set { _anno = value; }
    }
}
```

e dichiariamo, creiamo ed inizializziamo una collezione con alcuni dati, sfruttando, in questa maniera, le altre nuove caratteristiche del C# 3.0

```
Quota[] quote = new Quota[] {
```

```
    new Quota {Id = 1, IdSocio = 1, Importo = 100, Anno=2007},
    new Quota {Id = 2, IdSocio = 1, Importo = 200, Anno=2008},
    new Quota {Id = 3, IdSocio = 2, Importo = 100, Anno=2008},
    new Quota {Id = 4, IdSocio = 4, Importo = 100, Anno=2007},
    new Quota {Id = 5, IdSocio = 3, Importo = 400, Anno=2008}
};
```

Calcoliamo, a questo punto il valore massimo versato tra tutti i soci.

```
int maxQuota = quote.Max(quota => quota.Importo);
```

che darà come risultato:

```
Max quota=400
```

La funzione lambda serve come selettore per determinare il membro della classe su cui effettuare il calcolo; se avessimo scritto in quest'altra maniera

```
int ultimoAnno = quote.Max(quota => quota.Anno);
```

avremmo ricavato l'ultimo anno per cui è stata pagata una quota ottenendo:

```
Ultimo anno=2008
```

2.2 Quantificazione: All, Any, Contains

Passiamo ad altri 3 operatori che servono, invece a verificare la presenza o meno di alcuni elementi all'interno di una collezione.

Il metodo `All` ha solo questo prototipo

```
public static bool All<TSource>(
    this IEnumerable<TSource> source,
    Func<TSource, bool> predicate
)
```

nel quele è possibile specificare una funzione per testare ogni elemento per una certa condizione

Il metodo `Any` ha due overload

```
public static bool Any<TSource>(
    this IEnumerable<TSource> source
)

public static bool Any<TSource>(
    this IEnumerable<TSource> source,
    Func<TSource, bool> predicate
)
```

Il primo testa la presenza di un qualsiasi elemento, mentre nel secondo è possibile fornire una funzione per testare ogni elemento per una certa condizione.

Il metodo `Contains` ha anch'esso due overload

```
public static bool Contains<TSource>(
    this IEnumerable<TSource> source,
    TSource value
)
public static bool Contains<TSource>(
```

```
    this IEnumerable<TSource> source,
    TSource value,
    IEqualityComparer<TSource> comparer
)
```

Riprendiamo il vettore contenente i voti dell'esempio precedente e scriviamo il seguente frammento di codice.

```
int[] voti = { 4, 5, 6, 7, 8 };

bool promosso = voti.All(voto => voto >= 6);
bool sospeso = voti.Any(voto => voto < 6);
bool grave = voti.Contains(4);
```

Il risultato ottenuto sarà il seguente.

```
Promosso:False
Sospeso:True
Grave:True
```

Questo perché non **tutti** i voti sono >= 6 (operatore `All`) c'è **almeno** un voto < 6 (operatore `Any`) ed è presente **almeno** un 4 (operatore `Contains`).

2.3 Elemento: ElementAt, ElementAtOrDefault, First, FirstOrDefault, Last, LastOrDefault, Single, SingleOrDefault

I metodi `ElementAt` ed `ElementAtOrDefault` hanno un solo prototipo:

```
public static TSource ElementAtOrDefault<TSource>(
    this IEnumerable<TSource> source,
    int index
)
```

Nel quale è possibile specificare l'indice dell'elemento della sequenza da restituire.

I metodi `First`, `Last`, `Single` (e le loro versioni `xxxOrDefault`) hanno due overload

```
public static TSource First<TSource>(
    this IEnumerable<TSource> source
)
public static TSource First<TSource>(
    this IEnumerable<TSource> source,
    Func<TSource, bool> predicate
)
```

Qui vediamo quelli del metodo `Single`; il primo overload fa recuperare il prmo elemento di una sequenza, mentre il secondo permette di restituire ilprimo elemento di una sequenza che soddisfa una certa condizione

Per capire il funzionamento di questi operatori cambiamo i dati di esempio utilizzando una collezione di stringhe contenente i nomi di alcuni amici.

```
string[] amici = {"Andrea", "Barbara", "Marco", "Marco",
                  "Massimo", "Nicola", "Roberta", "Roberta" };
```

e proviamo ad eseguire queste linee di codice:

```
string secondo = amici.ElementAt(1);
```

```
string primoM = amici.First(nome => nome.StartsWith("M"));
string primoS = amici.FirstOrDefault(nome => nome.StartsWith("S"));
string ultimoM = amici.Last(nome => nome.StartsWith("M"));
string unicoA = amici.Single(nome => nome == "Andrea");
```

e visualizzando le variabili otterremo in uscita il seguente risultato:

```
Barbara
Marco

Massimo
Andrea
```

E' bene porre un po' di attenzione all'operatore `First` in quanto resituisce il primo elemento di una sequenza che soddisfa una certa condizione, ma nel caso non ce ne fosse nessuno verrebbe sollevata una eccezione.

Per evitare ciò si deve utilizzare `FirstOrDefault` che in questo caso restituisce invece un valore di default[3] (null se si tratta di stringhe, la riga vuota nell'esepio)

3 Operatori di partizionamento, concatenazione ed insieme

3.1 Partizionamento: Skip, SkipWhile, Take, TakeWhile

In questa categoria abbiamo gli operatori che restituiscono solamente una parte di una sequenza.

I metodi `Skip` e `Take` hanno un solo prototipo:

```
public static IEnumerable<TSource> Skip<TSource>(
    this IEnumerable<TSource> source,
    int count
)
```

Nel quale è possible specificare quati elemente saltare (Skip) o prendere (Take)

I metodi `xxxWhile` hanno invece due prototipi

```
public static IEnumerable<TSource> SkipWhile<TSource>(
    this IEnumerable<TSource> source,
    Func<TSource, bool> predicate
)

public static IEnumerable<TSource> SkipWhile<TSource>(
    this IEnumerable<TSource> source,
    Func<TSource, int, bool> predicate
)
```

Utilizzando il primo è possibile specificare una funzione di selezione, mentre nel secondo si può anche utilizzare l'indice dell'elemento all'interno di questa funzione.

Riutiliziamo il vettore con i nomi degli amici e mettiamoli alla prova.

```
string[] amici = {"Andrea", "Barbara", "Marco", "Marco",
                  "Massimo", "Nicola", "Roberta", "Roberta" };
```

```
var dalSestoAmico = amici.Skip(6);
var amiciDallaN = amici.SkipWhile(nome => !nome.StartsWith("N"));
var primi5Amici = nomi.Take(2);
var amiciFinoAllaM = amici.TakeWhile(nome => !nome.StartsWith("N"));
```

Avremo questi risultati:

"Roberta", "Roberta" **(saltati i primi 6 elementi)**

"Nicola", "Roberta", "Roberta" **(salta gli elementi finche** non **ne trovi uno che inizia** con **"N")**

"Andrea", "Barbara " **(primi 2 elementi)**

"Andrea", "Barbara", "Marco", "Marco", Massimo" **(prendi gli elementi finche** non **ne arriva uno che inizia** con **la "N")**

Questi operatori restituiscono una sequenza e l'utilizzo dell'inferenza del tipo mediante l'uso di variabili di tipo var non è obbligatoria ma utile, in quanto, essendo una sequenza di tipo `IEnumerable<T>` avremmo dovuto scrivere qualcosa del tipo:

```
IEnumerable<string> dalSestoAmico = amici.Skip(5);
```

certamente non molto chiara da comprendere.

Se avessimo voluto assegnare il risultato dell'espressione ad un vettore di stringhe saremmo stati costretti ad effettuare un cast, cosa di cui ci occuperemo, però, più avanti.

Le versione `xxxWhile` degli operatori, recuperano gli elementi della sequenza mentre è suddisfatta la condizione espressa dalla funzione lambda, ed in questo caso

```
var amiciDallaN = amici.SkipWhile(nome => !nome.StartsWith("N"));
```

sta ad indicare: "salta gli elementi mentre l'elemento (nome) non inizia con la lettera "N"

3.2 Concatenazione: Concat

Questo operatore, molto semplicemente, concatena, restituendole, due sequenze tra loro appendendo la seconda alla prima.

E' presente un solo prototipo:

```
public static IEnumerable<TSource> Concat<TSource>(
    this IEnumerable<TSource> first,
    IEnumerable<TSource> second
)
```

Se utilizziamo come esempio, oltre al vettore di amici precedente, anche quest'altro vettore di stringhe contenete altri nomi:

```
string[] nomi = { "Angelo", "Elena", "Essandra","Martina", "Marco",
                  "Massimo", "Riccardo", "Roberta" };
```

L'esecuzione del seguente frammento di codice:

[3] Stesso discorso per gli operatori ElementAt, Last, Single e le loro versioni xxxOrDefault

```
var appende = amici.Concat(nomi);
```

provocherà la creazione di una sequenza con questi dati:

```
"Andrea","Barbara","Marco","Marco",Massimo","Nicola","Roberta","Roberta",
"Angelo","Elena","Essandra","Martina","Marco","Massimo","Riccardo","Roberta"
```

3.3 Insieme: Distinct, Except, Intersect, Union

Per l'elaborazione di funzione di insiemistica, si utiilzzano questi 4 operatori; facciamo degli esempi utilizzando le due sequenze di nomi di esempio viste in precedenza.

```
var amiciDistinti = amici.Distinct();
var amiciUnione = amici.Union(nomi);
var amiciIntersezione = amici.Intersect(nomi);
var amiciEccetto = amici.Except(nomi);
```

le sequenze risultanti saranno le seguenti:

```
"Andrea","Barbara","Marco",Massimo","Nicola","Roberta"
```

```
"Andrea","Barbara","Marco",Massimo","Nicola","Roberta","Angelo","Elena",
"Essandra","Martina","Riccardo"
```

```
"Marco",Massimo","Roberta"
```

```
"Andrea","Barbara","Nicola"
```

4 Operatori di conversione

4.1 Conversione: Cast, OfType, ToArray, ToList

In precedenza abbiamo detto che gli operatori di LINQ operano solamente su sequenze sia come elementi da elaborare sia da restituire ed è presente un solo prototipo per ognuno dei metodi

```
public static IEnumerable<TResult> Cast<TResult>(
    this IEnumerable source
)
public static IEnumerable<TResult> OfType<TResult>(
    this IEnumerable source
)
public static TSource[] ToArray<TSource>(
    this IEnumerable<TSource> source
)
public static List<TSource> ToList<TSource>(
    this IEnumerable<TSource> source
)
```

Questi operatori consentono di trasformare alcuni tipi di collezioni del .Net in maniera da essere utilizzate con LINQ e viceversa.

Se infatti avessimo la seguente lista di voti memorizzata in un ArrayList

```
ArrayList listaVoti = new ArrayList() { 5, 6, 6, 5, 8 };
```

e volessimo applicare, ad esempio, l'operatore Max() in questo modo:

```
int mas = listaVoti.Max();
```

otterremmo un errore in fase di compilazione, in quanto la classe `ArrayList` non implementa l'interfaccia `IEnumerabla<T>` ma solo la `IEnumerable`

Il problema viene risolto effettuando un conversione del tipo da `ArrayList` a `IEnumerable<T>` mediante l'applicazionen dell'operatore `Cast`().

```
int mas = listaVoti.Cast<int>().Max();
```

La tipizzazione debole di certe collezioni come l'ArrayList potrebbe portare ad inserire, volontariamente o meno elementi di tipo non omogeneo.

```
ArrayList listaVotiMezzi = new ArrayList() { 5, 6, "6,5", 5, 8 };
```

In questo caso l'esecuzione dell'istruzione precedente

```
int masMezzi = listaVotiMezzi.Cast<int>().Max();
```

provoca il sollevamento di una eccezione, perché l'operatore di cast "pretende" che tutti gli elementi siano di un certo tipo.

Per ovviare a questo inconveniente si deve utilizzare l'operatore `OfType`() che effettua il cast solo degli elementi di un certo tipo igorando gli altri. Questa istruzione

```
int masMezzi = listaVotiMezzi.OfType<int>().Max();
```

non provocherà più eccezioni.

Gli atri due operatori della famiglia di conversione, effettuano una conversione della sequenza risultato in collezioni di un certo tipo.

Abbiamo visto in un esempio precedente

```
string[] amici = {"Andrea", "Barbara", "Marco", "Marco",
                  Massimo", "Nicola", "Roberta", "Roberta" };
            var dalSestoAmico = amici.Skip(6);
```

che la variabile risultato non era un vettore di stringhe.

Per poterlo ottenere dobbiamo applicare l'operatore di conversione appropriato e scrivere:

```
string[] dalSestoAmicoVet = amici.Skip(6).ToArray<string>();
List<string> dalSestoAmicoList = amici.Skip(6).ToList<string>();
```

5 Operatori di Proiezione

5.1 Select, SelectMany

Dopo aver visto gli operatori più semplici è giunto il momento di approfondire l'analisi di LINQ parlando degli operatori che hanno anche una corriposndente traduzione nel linguaggio C#.

Select

L'operatore `Select`() consente di creare una sequenza di uscita di un certo tipo di elementi, a partire da una sequenza di ingresso che non è necessariamente dello stesso tipo.

Sono presenti due overload

```
public static IEnumerable<TResult> Select<TSource, TResult>(
    this IEnumerable<TSource> source,
    unc<TSource, TResult> selector
)
public static IEnumerable<TResult> Select<TSource, TResult>(
    his IEnumerable<TSource> source,
    unc<TSource, int, TResult> selector
)
```

Nel primo è possibile definire una funzione di proiezione in unnuovo elemento, mentre nel secondo si può anche utilizzare l'indice dell'elemento nella sequenza.

Quando si invoca il metodo Select, questo prende un oggetto di tipo T e si restituisce un oggetto di tipo S chiamando la funzione posta all'interno ed iterando su tutti gli elementi della sequenza di ingresso.

Nella sua forma più semplice l'operatore si può scrivere in questo modo.

```
var altriTutti = amici
                .Select(nome => nome);
```

a cui corrisponde la seguente espressione di query

```
var tutti = from nome in amici
            select nome;
```

in questo caso, molto banalmente, tutti gli elementi della sequenza di input vanno a finire in quella di output senza nessun tipo di trasformazione.

Abbiamo detto, però che è posibile effettuare delle trasformazioni dai dati di input in quelli di output

Se volessimo restituire la lista di nomi in caratteri maiuscoli potremmo infatti scrivere questo codice.

```
var altriTuttiMaiusc = amici
                      .Select(nome => nome.ToUpper());
```

a cui corrisponde la seguente espressione di query

```
var tuttiMaiusc = from nome in amici
                  select nome.ToUpper();
```

ottenendo il seguente risultato.

```
ANDREA,BARBARA,MARCO,MARCO,MASSIMO,NICOLA,ROBERTA,ROBERTA
```

Utilizziamo adesso una collezione di oggetti che rappresentano i soci di un club per poter fare selezioni un po' più complesse.

```
public class Socio
{
    private int _id;
    public int Id
    {
        get { return _id; }
        set { _id = value; }
```

```
    }

    private string _cognome;
    public string Cognome
    {
        get { return _cognome; }
        set { _cognome = value; }
    }

    private string _nome;
    public string Nome
    {
        get { return _nome; }
        set { _nome = value; }
    }

    private string _provincia;
    public string Provincia
    {
        get { return _provincia; }
        set { _provincia = value; }
    }
}

Socio[] soci = new Socio[] {
    new Socio {Id = 1, Cognome="Santinelli", Nome="Massimo", Provincia="PG"},
    new Socio {Id = 2, Cognome="Biagini", Nome="Marco", Provincia="PG"},
    new Socio {Id = 3, Cognome="Carobini", Nome="Danilo", Provincia="PU"},
    new Socio {Id = 4, Cognome="Basili", Nome="Marco", Provincia="PG"},
    new Socio {Id = 5, Cognome="Inghirami", Nome="Luca", Provincia="AR"}
};
```

e scriviamo del codice per poter selezionare solo il cognome ed il nome del socio.

```
var opCognomeENome = soci
              .Select(anag => new { Cognome = anag.Cognome, Nome = anag.Nome });
```

a cui corrisponde la seguente espressione di query

```
var cognomeENome = from anag in soci
                   select new { Cognome = anag.Cognome, Nome = anag.Nome };
```

in questo caso si è utilizzata la capacità di creare un tipo anonimo presente nel C# 3.0.

```
new { Cognome = anag.Cognome, Nome = anag.Nome }
```

Il selettore mediante l'utilizzo della parola chiave `New`, crea "al volo" un nuovo tipo di dato senza nome (in realtà un nome gli viene assegnato e probabilmente sarà qualcosa del tipo `AnonymousType#1`).

Il tipo di dato avrà due campi pubblici chiamati `Cognome` e `Nome`, inizializzati rispettivamente dal valore della proprietà `Cognome` e `Nome` dell'oggetto `anag` che altri non è che il valore che rappresenta ogni singolo elemento della sequenza.

Se visualizziamo il contenuto della sequenza di uscita in questo modo:

```
foreach (var item in cognomeENome)
{
    Console.WriteLine(item);
}
```

Otterremo il seguente risultato.

```
{ Cognome = Santinelli, Nome = Massimo }
{ Cognome = Biagini, Nome = Marco }
{ Cognome = Carobini, Nome = Danilo }
{ Cognome = Basili, Nome = Marco }
{ Cognome = Inghirami, Nome = Luca }
```

Quando si creano dei tipi anonimi è buona norma assegnare ai campi sempre dei nomi significativi per migliorare la leggibilità del codice e nel caso non lo facessimo verrebbero utilizzati quelli di origine dei campi.

```
var anagEAnno = from anag in soci
    select new { Anagrafica = anag.Cognome + " " + anag.Nome, anag.Provincia };
```

con il seguente risultato.

```
{ Anagrafica = Santinelli Massimo, Provincia = PG }
{ Anagrafica = Biagini Marco, Provincia = PG }
{ Anagrafica = Carobini Danilo, Provincia = PU }
{ Anagrafica = Basili Marco, Provincia = PG }
{ Anagrafica = Inghirami Luca, Provincia = AR }
```

SelectMany

L'operatore `SelectMany`() effettua il prodotto cartesiano tra due sequenze mappando ciascun elemento della prima sequenza con un elemento della seconda.

Questo metodo ha 4 overload;

```
public static IEnumerable<TResult> SelectMany<TSource, TResult>(
    this IEnumerable<TSource> source,
    Func<TSource, IEnumerable<TResult>> selector
)
public static IEnumerable<TResult> SelectMany<TSource, TResult>(
    this IEnumerable<TSource> source,
    Func<TSource, int, IEnumerable<TResult>> selector
)
public static IEnumerable<TResult> SelectMany<TSource, TCollection, TResult>(
    this IEnumerable<TSource> source,
    Func<TSource, IEnumerable<TCollection>> collectionSelector,
    Func<TSource, TCollection, TResult> resultSelector
)
public static IEnumerable<TResult> SelectMany<TSource, TCollection, TResult>(
    this IEnumerable<TSource> source,
    Func<TSource, int, IEnumerable<TCollection>> collectionSelector,
    Func<TSource, TCollection, TResult> resultSelector
)
```

Per illustrare quello che accade, mettiamo assieme la collezione che contiene l'elenco dei soci con quella che rappresenta le quote annuali e scriviamo questa espressione:

```
var opNomiEQuote = soci
                  .SelectMany(socio => quote
```

```
                    .Select(quota => new { socio.Cognome, quota.Importo })
                );
```

a cui corrisponde la seguente espressione di query

```
var nomiEQuote = from socio in soci
                 from quota in quote
                 select new { socio.Cognome, quota.Importo };
```

ognuno dei soci presenti nella collezione viene accoppiato con un tutti gli elementi presenti nella collezione delle quote e la sequenza risultante quindi sarà la seguente.

```
{ Cognome = Santinelli, Importo = 100 }
{ Cognome = Santinelli, Importo = 400 }
{ Cognome = Biagini, Importo = 100 }
{ Cognome = Biagini, Importo = 200 }
{ Cognome = Biagini, Importo = 100 }
{ Cognome = Biagini, Importo = 100 }
{ Cognome = Biagini, Importo = 400 }
{ Cognome = Carobini, Importo = 100 }
{ Cognome = Carobini, Importo = 200 }
{ Cognome = Carobini, Importo = 100 }
{ Cognome = Carobini, Importo = 100 }
{ Cognome = Carobini, Importo = 400 }
{ Cognome = Basili, Importo = 100 }
{ Cognome = Basili, Importo = 200 }
{ Cognome = Basili, Importo = 100 }
{ Cognome = Basili, Importo = 100 }
{ Cognome = Basili, Importo = 400 }
{ Cognome = Inghirami, Importo = 100 }
{ Cognome = Inghirami, Importo = 200 }
{ Cognome = Inghirami, Importo = 100 }
{ Cognome = Inghirami, Importo = 100 }
{ Cognome = Inghirami, Importo = 400 }
```

E’ chiaro che una sequnza di questo tipo non ha un grande significato in quanto non si dovrebbe tenere conto di tutte le coppie generate, ma solamente di quelle in cui c’è corrispondenza tra gli identificativi dei soci.

La cosa avrebbe più senso se volessimo generare una sequenza in grado di consentire l’inseriento delle quote associative per gli anni futuri 2009 e 2010.

```
var quoteFut = from socio in soci
               from anno in anni
               select new { socio.Cognome, Anno = anno, Importo = 0 };
```

ed ottenere questa sequenza.

```
{ Cognome = Santinelli, Anno = 2009, Importo = 0 }
{ Cognome = Santinelli, Anno = 2010, Importo = 0 }
{ Cognome = Biagini, Anno = 2009, Importo = 0 }
{ Cognome = Biagini, Anno = 2010, Importo = 0 }
{ Cognome = Carobini, Anno = 2009, Importo = 0 }
{ Cognome = Carobini, Anno = 2010, Importo = 0 }
{ Cognome = Basili, Anno = 2009, Importo = 0 }
{ Cognome = Basili, Anno = 2010, Importo = 0 }
{ Cognome = Inghirami, Anno = 2009, Importo = 0 }
{ Cognome = Inghirami, Anno = 2010, Importo = 0 }
```

Selezione dell’indice

Sia `Select`() che `SelectMany`() offrono la possibilità di recuperare l'indice dell'elemento che stanno elaborando.

```
var opIndiceSoci = soci
       .Select((anag, i) => new { Indice = i, Cognome = anag.Cognome}         );
```

a cui corrisponde la seguente espressione di query

```
int i = 0;
var indiceSoci = from anag in soci
                 select new { Indice = i++, Cognome = anag.Cognome};
```

ottenendo questa sequenza

```
{ Indice = 0, Cognome = Santinelli }
{ Indice = 1, Cognome = Biagini }
{ Indice = 2, Cognome = Carobini }
{ Indice = 3, Cognome = Basili }
{ Indice = 4, Cognome = Inghirami }
```

6 Operatore di filtro

6.1 Where

Spesso vi è la necessità, come nell'esempio precedente di filtrare una parte dei risultati in uscita in base ad un certo criteri.

L'operatore `Where`() svolge proprio questo compito, filtrando i valori della sequenza in ingresso prima ancora di svolgere le successive operazioni e sono presenti due overload:

```
public static IEnumerable<TSource> Where<TSource>(
    this IEnumerable<TSource> source,
    Func<TSource, bool> predicate
)
public static IEnumerable<TSource> Where<TSource>(
    this IEnumerable<TSource> source,
    Func<TSource, int, bool> predicate
)
```

Utilizzando il primo metodo si definisce una funzione si selezione, che nel secondo ha anche a disposizione l'indice dell'elemento della sequenza.

Ipotiziamo di voler disporre solamente degli amici che iniziano con la lettera "M"; la query da scrivere sarà al seguente.

```
var opAmiciM = nomi
               .Where(amico => amico.StartsWith("M"))
               .Select(amico => amico);
```

a cui corrisponde la seguente espressione di query

```
var amiciM = from amico in nomi
             where amico.StartsWith("M")
             select amico;
```

ed otterremo il risultato voluto.

```
Martina
Marco
Massimo
```

Possiamo utilizzare l'operatore di filtro anche per emulare il comportamento di altri operatori

In precedenza abbiamo visto come poter "saltare" n elementi una sequenz utilizzando l'operatore `Skip`(). Ebbene la stessa cosa possiamo riprodurla con l'operatore `While`() .

```
var opPrimi5Amici = amici
                    .Where((nome, indice) => indice < 5).
                    .Select(nome => nome);
```

a cui corrisponde la seguente espressione di query

```
int indice = 0;
var primi5Amici = from nome in amici
                  where (indice++ < 5)
                  select nome;
```

7 Operatori di Ordinamento

7.1 Orderby, OrderByDescending

L'operatore di ordinamento `OrderBy`() è un altro di quelli molto semplici da utilizzare e consente di effettuare ordinamenti in senso crescente o decrescente ed hanno entrambi due prototipi

```
public static IOrderedEnumerable<TSource> OrderBy<TSource, TKey>(
    this IEnumerable<TSource> source,
    Func<TSource, TKey> keySelector
)
public static IOrderedEnumerable<TSource> OrderBy<TSource, TKey>(
    this IEnumerable<TSource> source,
    Func<TSource, TKey> keySelector,
    IComparer<TKey> comparer
)
```

Nel primo è possibile specificare la funzione di selezione della chiave di ordinamento, mentre nella seconda un eventuale metodo alternativo di comparazione.

Ipotizzando di avere una collezione come questa:

```
string[] colleghi = { "Urbano", "Claudio", "Pino", "Luca", "Gianluca";
```

l'esecuzione del seguente frammento di codice

```
var opColleghiOrd = colleghi
                    .OrderBy(nome => nome);
```

a cui corrisponde la seguente espressione di query

```
var colleghiOrd = from nome in colleghi
                  orderby nome
                  select nome;
```

ci farà ottenere questo risultato

```
Claudio
Gianluca
Luca
Pino
Urbano
```

È ovviamente possibile specificare anche un selettore differente su cui svolgere l'ordinamento come in questo caso:

```
var opColleghiOrdLun = colleghi
                       .OrderBy(nome => nome.Length)
                       .Select(nome => new { nome, nome.Length });
```

a cui corrisponde la seguente espressione di query

```
var colleghiOrdLun = from nome in colleghi
                     orderby nome.Length
                     select new { nome, nome.Length };
```

Ed ottenere il risultato voluto

```
{ nome = Pino, Length = 4 }
{ nome = Luca, Length = 4 }
{ nome = Urbano, Length = 6 }
{ nome = Claudio, Length = 7 }
{ nome = Gianluca, Length = 8 }
```

L'operatore `OrderByDescending()` si comporta allo stesso modo dell'operatore `OrderBy()` con la differenza che ordina i dati in senso decrescente.

```
var opColleghiOrdDesc = colleghi
                          .OrderByDescending(nome => nome);
```

a cui corrisponde la seguente espressione di query

```
var colleghiOrdDesc = from nome in colleghi
                      orderby nome descending
                      select nome;
```

ci darà questo risultato

```
Urbano
Pino
Luca
Gianluca
Claudio
```

7.2 Thenby, ThenByDescending

Questi altri due operatori consentono di effettuare un secondo ordinamento all'interno di un primo, sono disponibili due overload simili a quelli dei metodi `OrderBy()` con la differenza che si possono applicare solo a collezioni già ordinate.

```
public static IOrderedEnumerable<TSource> ThenBy<TSource, TKey>(
    this IOrderedEnumerable<TSource> source,
    Func<TSource, TKey> keySelector
)
public static IOrderedEnumerable<TSource> ThenBy<TSource, TKey>(
    this IOrderedEnumerable<TSource> source,
    Func<TSource, TKey> keySelector,
    IComparer<TKey> comparer
```

```
)
```

Un tipico caso potremmo averlo se volessimo ordinare i nomi dei nostri colleghi in base alla lunghezza ed in caso di lunghezza uguale ordinare in senso crescente in base al nome per ottenere questo risultato:

```
{ nome = Luca, Length = 4 }
{ nome = Pino, Length = 4 }
{ nome = Urbano, Length = 6 }
{ nome = Claudio, Length = 7 }
{ nome = Gianluca, Length = 8 }
```

Sarà sufficiente utilizzare prima l'operatore `OrderBy` per stabilire l'ordinamento principale, e poi l'operatore ThenBy per quello secondario:

```
var opColleghiOrdLunAlf = colleghi
                          .OrderBy(nome => nome.Length)
                          .ThenBy(nome => nome)
                          .Select(nome => new { nome, nome.Length });
```

a cui corrisponde la seguente espressione di query

```
var colleghiOrdLunAlf = from nome in colleghi
                        orderby nome.Length, nome
                        select new { nome, nome.Length };
```

E' interessante notare come utilizzando l'espressione di query è sufficiente separare con la virgola i nomi di selettori in base a cui effettuare l'ordinamento, mentre utilizzando gli operatori si deve necessariamente utilizzare `ThenBy()`.

Questo esempio

```
var opColleghiOrdLunAlfDesc = colleghi
                              .OrderBy(nome => nome.Length).
                              .ThenByDescending(nome => nome).
                              .Select(nome => new { nome, nome.Length });
```

a cui corrisponde la seguente espressione di query

```
var colleghiOrdLunAlfDesc = from nome in colleghi
                            orderby nome.Length, nome descending
                            select new { nome, nome.Length };
```

imposterà invece un ordinamento secondario in senso decrescente.

8 Sub Query

Abbiamo visto come poter utilizzare gli operatori di selezione per recuperare gli elementi da due sequenze, ma abbiamo anche visto che l'utilizzo del solo operatore `SelectMany` non è sufficiente per implementare una relazione uno-a-molti.

Quello che serve è infatti utilizzare gli operatori di selezione applicando un filtro per poter ottenere il risultato desiderato.

```
var opNomiEQuoteSel = soci
                      .SelectMany(socio => quote.Where(q => q.IdSocio == socio.Id)
```

```
                .Select(quota => new { socio.Cognome, quota.Importo })
                );
```

a cui corrisponde la seguente espressione di query

```
var nomiEQuoteSel = from socio in soci
                    from quota in quote
                    where socio.Id == quota.IdSocio
                    select new { socio.Cognome, quota.Importo };
```

Questa volta la selezione sarà corretta

```
{ Cognome = Santinelli, Importo = 100 }
{ Cognome = Santinelli, Importo = 200 }
{ Cognome = Biagini, Importo = 100 }
{ Cognome = Carobini, Importo = 400 }
{ Cognome = Basili, Importo = 100 }
```

Potremmo però anche fare in un altro modo, scrivendo una espressione di query all'interno di un'altra espressione di query, implementando in questo modo una sub-query.

```
var opQuoteVersate = soci
                    .Select(socio => new
                    {
                        socio.Cognome,
                        Versamenti = quote
                        .Where(quota => quota.IdSocio == socio.Id)
                        .Select(quota => new { quota.Importo, quota.Anno})
                    });
```

a cui corrisponde la seguente espressione di query

```
var quoteVersate = from socio in soci
                   orderby socio.Cognome
                   select new
                   {
                       socio.Cognome,
                       Versamenti = from quota in quote
                              where quota.IdSocio == socio.Id
                              select new { quota.Importo, quota.Anno }
                   };
```

Ora avremmo veramente una relazione uno-a-molti tra la collezione dei Soci e quella delle Quote, e potremmo quindi scrivere il seguente codice consumer per visuliazzare la sequenza di uscita:

```
foreach (var s in quoteVersate)
{
    Console.WriteLine(s.Cognome);
    foreach (var q in s.Versamenti)
    {
        Console.WriteLine("  {0}", q.Importo);
    }
}
```

Come si può notare ogni elemento della sequenza che rappresenta un socio ha al suo interno un campo che contiene la sequenza di tutti i versamenti effettuati da quel socio.

```
Basili
  100 - 2007
Biagini
  100 - 2008
Carobini
  400 - 2008
Inghirami
Santinelli
  100 - 2007
  200 – 2008
```

Avremo in questo modo ottenuto un comportamento simile alla `LeftJoin` tipica del linguaggio SQL.

9 Operatori di collegamento

9.1 Join

Esiste per un modo migliore e più ufficente per implementare una relazione tra due collezioni mediante l'operatore di collegamento `Join`() che ha due overload

```
public static IEnumerable<TResult> Join<TOuter, TInner, TKey, TResult>(
    this IEnumerable<TOuter> outer,
    IEnumerable<TInner> inner,
    Func<TOuter, TKey> outerKeySelector,
    Func<TInner, TKey> innerKeySelector,
    Func<TOuter, TInner, TResult> resultSelector
)
public static IEnumerable<TResult> Join<TOuter, TInner, TKey, TResult>(
    this IEnumerable<TOuter> outer,
    IEnumerable<TInner> inner,
    Func<TOuter, TKey> outerKeySelector,
    Func<TInner, TKey> innerKeySelector,
    Func<TOuter, TInner, TResult> resultSelector,
    IEqualityComparer<TKey> comparer
)
```

Nel primo prototipo si specificano:

- la prima sequenza
- la seconda sequenza da unire alla prima
- una funzione per estrarre la chiave di collegamento per ogni elemento della prima sequenza
- una funzione per estrarre la chiave di collegamento per ogni elemento della seconda sequenza
- una funzione per creare l'elemento risultante dall'unione di due elementi corrispondenti

Nel secondo prototipo si può anche definire una funzione si comprazione alternativa

Vediamo l'utilizzo dell'operatore `Join`() per poter effettuare una qualcosa di simile alla Inner Join tipica dell'SQL, prelevando, cioè, tutti i valori della prima sequenza che abbiano una corrispondenza anche nella seconda sequenza.

```
var opSociQuote = soci
                  .Join(quote,
                          s => s.Id,
                          q => q.IdSocio,
                          (s, q) => new { s.Cognome, q.Importo }
                        );
```

a cui corrisponde la seguente espressione di query

```
var sociQuote = from s in soci
                join q in quote
                on s.Id equals q.IdSocio
                select new { s.Cognome, q.Importo };
```

l'esecuzione produrrà una sequenza, uguale a quella che avremmo ottenuto con l'utilizzo degli operatori di selezione e filtro

```
{ Cognome = Santinelli, Importo = 100 }
{ Cognome = Santinelli, Importo = 200 }
{ Cognome = Biagini, Importo = 100 }
{ Cognome = Carobini, Importo = 400 }
{ Cognome = Basili, Importo = 100 }
```

E' ovvio che possiamo applicare gli operatori di ordinamento alla sequenza risultato come nell'esempio che segue che ordina in base al cognome.

```
var opSociQuoteOrd = soci
                     .Join(quote,
                             s => s.Id,
                             q => q.IdSocio,
                             (s, q) => new { s.Cognome, q.Importo }
                             )
                     .OrderBy(s => s.Cognome)
                     .Select(v => new { v.Cognome, v.Importo });
```

a cui corrisponde la seguente espressione di query

```
var sociQuoteOrd = from s in soci
                   join q in quote
                   on s.Id equals q.IdSocio
                   orderby s.Cognome
                   select new { s.Cognome, q.Importo };
```

ed ottenere questo risultato

```
{ Cognome = Basili, Importo = 100 }
{ Cognome = Biagini, Importo = 100 }
{ Cognome = Carobini, Importo = 400 }
{ Cognome = Santinelli, Importo = 100 }
{ Cognome = Santinelli, Importo = 200 }
```

Fino ad ora abbiamo visto come poter effettuare operazioni che somiglino alla `LeftJoin` del linguaggio SQL prelevando cioè tutti i valori della prima sequenza e mettendoli in relazione con quelli della seconda sequenza.

9.2 GroupJoin

Se si vogliono raggruppare i dati tra di loro per effettuare somme oppure conteggi, si deve utilizzare l'operatore di `GroupJoin`(), che ha anch'esso due overload

```
public static IEnumerable<TResult> GroupJoin<TOuter, TInner, TKey, TResult>(
    this IEnumerable<TOuter> outer,
    IEnumerable<TInner> inner,
    Func<TOuter, TKey> outerKeySelector,
    Func<TInner, TKey> innerKeySelector,
    Func<TOuter, IEnumerable<TInner>, TResult> resultSelector
)
public static IEnumerable<TResult> GroupJoin<TOuter, TInner, TKey, TResult>(
    this IEnumerable<TOuter> outer,
    IEnumerable<TInner> inner,
    Func<TOuter, TKey> outerKeySelector,
    Func<TInner, TKey> innerKeySelector,
    Func<TOuter, IEnumerable<TInner>, TResult> resultSelector,
    IEqualityComparer<TKey> comparer
)
```

Nel primo prototipo si specificano:

- la prima sequenza
- la sequenza da unire alla prima
- una funzione per estrarre la chiave di collegamento per ogni elemento della prima sequenza
- una funzione per estrarre la chiave di collegamento per ogni elemento della seconda sequenza
- una funzione per creare un elemento composto da un elemento della prima sequnza e da una collezioni di elementi corrispondenti presenti nella seconda sequenza

Nel secondo prototipo si può anche definire una funzione si comparazione alternativa

Quindi, a differenza dell'operatore di `Join`(), questo si comporta come se si stesse eseguendo una SubQuery mettendo a disposizione quindi una nuova sequenza "figlio" su cui poter effettuare i calcoli.

```
var opQuoteVersateG = soci
                      .GroupJoin(quote,
                             s => s.Id,
                             q => q.IdSocio,
                             (s, versato) => new
                                 {
                                     Cognome = s.Cognome,
                                     Versamenti = versato
                                 }
                             );
```

a cui corrisponde la seguente espressione di query

```
var quoteVersateG = from s in soci
                    join q in quote
                    on s.Id equals q.IdSocio into versato
```

```
                    select new
                    {
                        Cognome = s.Cognome,
                        Versamenti = versato
                    };
```

ed otterremo gli stessi risultati della subquery equivalenti ad una Left Join in SQL.

```
Santinelli
  100 - 2007
  200 - 2008
Biagini
  100 - 2008
Carobini
  400 - 2008
Basili
  100 - 2007
Inghirami
```

Sftruttando questa cosa potremmo quindi anche effettuare dei calcoli come somme e conteggi nella sequenza figlio.

```
var opTotVersato = soci
                  .GroupJoin(quote,
                             s => s.Id,
                             q => q.IdSocio,
                             (s, versato) => new
                                 {
                                     Cognome = s.Cognome,
                                     Totale = versato.Sum(q => q.Importo),
                                     NumQuote = versato.Count()
                                 }
                             );
```

a cui corrisponde la seguente espressione di query

```
var totVersato = from s in soci
                 join q in quote
                 on s.Id equals q.IdSocio into versato
                 select new
                 {
                     Cognome = s.Cognome,
                     Totale = versato.Sum(q => q.Importo),
                     NumQuote = versato.Count()
                 };
```

e la sequenza di uscita risulterà la seguente:

```
{ Cognome = Santinelli, Totale = 300, NumQuote = 2 }
{ Cognome = Biagini, Totale = 100, NumQuote = 1 }
{ Cognome = Carobini, Totale = 400, NumQuote = 1 }
{ Cognome = Basili, Totale = 100, NumQuote = 1 }
{ Cognome = Inghirami, Totale = 0, NumQuote = 0 }
```

Per concludere questo paragrafo scriviamo una query che sfrutti un po' tutto quello visto in precedenza raggruppando le quote versate dai soci sommando gli importi, contando le quote versate

ed ordinandole in senso decrescente in base al totale degli importi e, in caso di eguaglianza in base al cognome in senso crescente.

```
var opSommaSpeseContaOrd = soci
                          .GroupJoin(quote,
                                 s => s.Id,
                                 q => q.IdSocio,
                                 (s, versamenti) => new
                                     {
                                           Cognome = s.Cognome,
                                      Totale = versamenti.Sum(q => q.Importo),
                                           NumQuote = versamenti.Count()
                                     }
                                 )
                          .OrderByDescending(q => q.Totale)
                          .ThenBy(q => q.Cognome);
```

a cui corrisponde la seguente espressione di query

```
var sommaSpeseContaOrd = from versamenti in
                         (from s in soci
                          join q in quote
                          on s.Id equals q.IdSocio into versamenti
                          select new
                             {
                                 Cognome = s.Cognome,
                                 Totale = versamenti.Sum(q => q.Importo),
                                 NumQuote = versamenti.Count()
                             }
                         )
                orderby versamenti.Totale descending, versamenti.Cognome ascending
                select versamenti;
```

10 Operatore di raggruppamento

10.1 GroupBy

L'ultimo operatore rimasto è quello che ci consente di raggruppare gli elementi di una sequenza tra di loro in base ad una certa chiave.

In esso sono presenti 4 overload, più altri 4 che permettono di specificare una funzione di comparazione personalizzata; i primi 4 sono:

```
public static IEnumerable<IGrouping<TKey, TSource>> GroupBy<TSource, TKey>(
    this IEnumerable<TSource> source,
    Func<TSource, TKey> keySelector
)
public static IEnumerable<IGrouping<TKey, TElement>> GroupBy<TSource, TKey,
TElement>(
    this IEnumerable<TSource> source,
```

```
    Func<TSource, TKey> keySelector,
    Func<TSource, TElement> elementSelector
)
public static IEnumerable<TResult> GroupBy<TSource, TKey, TResult>(
    this IEnumerable<TSource> source,
    Func<TSource, TKey> keySelector,
    Func<TKey, IEnumerable<TSource>, TResult> resultSelector
)
public static IEnumerable<TResult> GroupBy<TSource, TKey, TElement, TResult>(
    this IEnumerable<TSource> source,
    Func<TSource, TKey> keySelector,
    Func<TSource, TElement> elementSelector,
    Func<TKey, IEnumerable<TElement>, TResult> resultSelector
)
```

Facciamo un semplice esempio:

```
var opGruppoQuote = quote
                    .GroupBy(q => q.IdSocio);
```

a cui corrisponde la seguente espressione di query

```
var gruppoQuote = from q in quote
                  group q by q.IdSocio;
```

La sequenza che otterremo è composta da elementi costituiti principalmente dalla chiave (esposta dall'attributo `Key`) per cui abbiamo raggruppato gli elementi con al loro interno una sequenza con gli elementi corrispondenti alla chiave.

Per poter esplorare il risultato dell'espressione dobbiamo scrivere un codice consumer simile a questo:

```
foreach (var gruppo in gruppoQuote)
{
    Console.WriteLine("Chiave = {0}", gruppo.Key);
    foreach (var item in gruppo)
    {
        Console.WriteLine("    Importo:{0}, Anno:{1}", item.Importo, item.Anno);
    }
}
```

Ed avremo questo risultato

```
Chiave = 1
    Importo:100, Anno:2007
    Importo:200, Anno:2008
Chiave = 2
    Importo:100, Anno:2008
Chiave = 4
    Importo:100, Anno:2007
Chiave = 3
    Importo:400, Anno:2008
```

È possibile anche selezionare solo alcuni campi del gruppo come in questo caso.

```
var opGruppoQuoteSel = quote
                    .GroupBy(q => q.IdSocio, q => new { q.Anno, q.Importo });
```

con lo stesso risultato della query precedente.

Possiamo ovviamente applicare altri operatori alla nostra espressione per contare o sommare gli importi come in questi esempi:

```
var opGruppoQuoteConta = quote
                         .GroupBy(q => q.IdSocio)
                         .Select(versamenti => new
                                {
                                    Chiave = versamenti.Key,
                                    Quote = versamenti,
                                    NumQuote = versamenti.Count()
                                }
                             );


var opGruppoQuoteContaSel = quote
                         .GroupBy(q => q.IdSocio,
                             (idSocio, versamenti) => new
                                {
                                    Chiave = idSocio,
                                    Quote = versamenti,
                                    NumQuote = versamenti.Count()
                                }
                             );
```

a cui corrisponde la seguente espressione di query

```
var gruppoQuoteConta = from q in quote
                       group q by q.IdSocio into versamenti
                       select new
                       {
                           Chiave = versamenti.Key,
                           Quote = versamenti,
                           NumQuote = versamenti.Count()
                       };
```

Ed utilizzare del codice simile a questo per scorrere la sequenza.

```
foreach (var gruppo in opGruppoQuoteContaSel)
{
    Console.WriteLine("Chiave IDSocio = {0}", gruppo.Chiave);
    Console.WriteLine("  Numero Quote = {0}", gruppo.NumQuote);
    foreach (var item in gruppo.Quote)
    {
        Console.WriteLine("    Importo:{0}, Anno:{1}", item.Importo, item.Anno);
    }
}
```

per ottenere questo risultato:

```
Chiave IDSocio = 1
  Numero Quote = 2
    Importo:100, Anno:2007
```

```
    Importo:200, Anno:2008
Chiave IDSocio = 2
  Numero Quote = 1
    Importo:100, Anno:2008
Chiave IDSocio = 4
  Numero Quote = 1
    Importo:100, Anno:2007
Chiave IDSocio = 3
  Numero Quote = 1
    Importo:400, Anno:2008
```

11 Oggetti Funzione

11.1 I Functor

Come abbiamo potuto vedere, un gran parte degli operatori di query hanno la possibilità di specificareun delegato di tipo `Func` come argomento dell'operatore.

Fino ad ora abbbiamo utilizzato questa caratteristica solamente per selezionare i dati su cui poter lavorare, ma è giunto il momento di approfondirla un po' per dimostrare tutte le sue potenzialità.

Ipotiziamo di voler aumentare di 100 euro tutti gli importi delle quote versate dai soci nell'anno 2008.

Con il sistema tradizionale dovremmo fare un ciclo foreach simile a questo

```
foreach (var item in quote)
{
    item.Importo += 100;
}
```

Utilizzando LINQ possiamo scrivere

```
var aumento = from quota in quote
              select quota.Importo += 100;
```

Abbiamo però il problema che l'operatore `Select` è di tipo differito e quindi non viene eseguito finchè non viene valutata l'espressione

Per forzarne l'esecuzione basta invocare l'operatore `ToList`() sulla variabile aumento in qusto modo

```
aumento.ToList();
```

Visualizzando la collezione delle quota avremo il risultato che gli importi delle quote saranno state aumentate di 100 euro

```
Id:1, IdSocio:1, Importo:300, Anno:2007
Id:2, IdSocio:1, Importo:400, Anno:2008
Id:3, IdSocio:2, Importo:300, Anno:2008
Id:4, IdSocio:4, Importo:300, Anno:2007
Id:5, IdSocio:3, Importo:600, Anno:2008
```

Ipotiziamo, a questo punto di fare una cosa più complessa; fino ad ora abbiamo avuto una lista di soci ed una di quote, ma sarebbe molto più corretto avere la lista di quote come attributo e proprietà di ogni socio effettuando quindi una composizione tra la classe Socio e la classe Quota.

Per fare ciò estendiamo la nostra classe Socio aggiungendo una collezione di oggetti di tipo Quota usando un attributo ed una proprietà per esporrlo

```
private List<Quota> _quote;
public List<Quota> Quote
{
    get { return _quote; }
}
```

Ora scriviamo il codice per aggiungere alla lista delle quote mantenuta all'interno di ogni socio, quelle che provengono dalla collezione fin qua usata.

Mixando un po' di programmazione tradizionale con LINQ possiamo scrivere questo codice

```
foreach (var socio in soci)
{
    socio.Quote.AddRange(from q in quote
                         where q.IdSocio == socio.Id
                         select q);
}
```

Oppure, utilizzando gli operatori di query qualcosa di molto più compatto

```
foreach (var socio in soci)
{
    socio.Quote.AddRange(quote.Where( q => q.IdSocio == socio.Id));
}
```

Il ciclo `foreach`, iterando su ogni elemento della collezione dei soci aggiunge le quote come risultato della query nella quale viene effettuata una unione utilizzando l'`Id` come chiave.

Volendo complicarci un po' la vita e volendo utilizzare solo LINQ ci verrebbe in mente di utilizzare una query di questo tipo.

```
var sErrore = from s in soci
              select (s.Quote.AddRange(from q in quote
                                       where q.IdSocio == s.Id
                                       select q));
```

Il codice di cui soprà, però produrrà un errore di compilazione in quanto la clausola **`select`** deve per forza restituire un dato; potremmo quindi correggere l'espressione creando un tipo anonimo in questo modo:

```
var sErrore = from s in soci
              select new { x = s.Quote.AddRange(from q in quote
                                                where q.IdSocio == s.Id
                                                select q)};
```

Ma il compilatore ci darebbe nuovamente errore questa volta a causa del metodo `AddRange`, che restituisce `void`, un valore non ammissibile per una proprietà di un tipo anonimo.

Anche se sembra un problema senza soluzione vengono in nostro aiuto i Functor che altro non sono che degli oggetti che possono essere utilizzati come una specie di puntatori a funzione tipici del C e del C++.

La dichiarazione del delegato `Func` è la seguente

```
public delegate TR Func<TR>();
public delegate TR Func<T0, TR>(T0 ao);
```

```
public delegate TR Func<T0, T1, TR>(T0 ao, T1 a1);
public delegate TR Func<T0, T1, T2, TR>(T0 ao, T1 a1, T2 a2);
public delegate TR Func<T0, T1, T2, T3, TR>(T0 ao, T1 a1, T2 a2, T3 a3);
```

In ogni dichiarazione `TR` rappresenta il tipo di dato restituito, mentre gli altri `T0`, `T1`, `T2`, e `T3`, sono i tipi di dato dei parametri passati in ingresso.

Per capire meglio come funziona questo meccanismo ipotizziamo di voler aumentare di 1 tutti i voti di questa collezione

```
int[] voti = { 4, 5, 6, 7, 8 };
```

Per prima cosa dovremo scrivere il nostro oggetto funzione

```
Func<int, int> Aumenta = i => i+1;
```

per poi applicarlo, mediante l'operatore `Select` a tutti gli elementi della collezione.

```
var votiAumentati = voti.Select(Aumenta).ToArray();
```

Capito come funziona il meccanismo possiamo anche tornare indietro e provare a risolvere in questo modo il problema del "mix" tra le due collezioni.

Per prima cosa scriviamo il `Functor` che ricevuto un oggetto di tipo `Socio` ed una collezione di quote, valorizzi l'attributo corrispondente dell'oggetto `Socio` e successivamente restituisca l'oggetto `Socio` modificato

```
Func<Socio, IEnumerable<Quota>, Socio > AggQuote = ((socio, qt) =>
{
    socio.Quote.AddRange(from q in qt
                         where q.IdSocio == socio.Id
                         select q);
    return socio;
});
```

Dopodichè possiamo scrivere la query che per ogni elemento della lista dei soci applichi questo oggetto funzione.

```
var sociMergeQuote = (from s in soci
                      select AggQuote(s, quote)).ToList();
```

Da notare in fondo alla query l'applicazione dell'operatore `ToList` per forzare l'esecuzione della query.