LINQ

Introduzione al funzionamento di LINQ

Progetto di Informatica classe 4^{a}

Ambiente: .NET 4.0/C# 4.0

Anno 2016

Indice generale

1 Introduzione

Questo tutorial introduce i principi di funzionamento di LINQ (**L**anguage **IN**tegrated **Q**uery).

Il linguaggio si fonda su: ***extension methods***, ***lambda expressions*** e ***generics***. Qui analizzo soprattutto il funzionamento degli *extension methods;* per gli altri due "pilastri" del linguaggio vedi i tutorial **Delegate** e **Generics**.

1.1 Cos'è LINQ

LINQ è una API di interrogazione e manipolazione di sorgenti di dati: collezioni, XML, database, etc. È utilizzabile in due forme:

- mediante chiamate "concatenate" a metodi: ***fluent syntax***.
- attraverso una sintassi simile a SQL (LINQ propriamente detto).

In questo tutorial tratterò soltanto la prima forma.

1.2 Su cosa si basa LINQ

LINQ è fondato su *extension methods* e *lambda expressions*, e utilizza dei *delegate* appositamente definiti.

Gli *extension methods* sono metodi statici che possono essere usati come se fossero metodi di istanza; ciò favorisce l'implementazione della cosiddetta *fluent syntax*.

Le *lambda expressions* forniscono una sintassi semplice per passare del codice esecutivo a un metodo.

1.3 Fluent syntax

Prima di procedere oltre è opportuno chiarire il concetto di *fluent syntax*: indica l'esecuzione di più metodi all'interno della stessa istruzione.

Ad esempio, consideriamo il seguente problema: data una stringa, togliere gli spazi iniziali e finali, eliminare i primi tre caratteri e trasformare tutto in maiuscolo.

Ecco una soluzione che esegue un passo per volta:

```
var s = "   ---rossi, verdi, bianchi     ";
var tmp1 = s.Trim();              // -> "---rossi, verdi, bianchi"
var tmp2 = tmp1.Substring(3);     // -> "rossi, vedi, bianchi"
var s2   = tmp2.ToUpper();        // -> "ROSSI, VERDI, BIANCHI"
```

Ma si può ottenere lo stesso risultato con un'unica istruzione:

```
var s2 = s.Trim().Substring(3).ToUpper();
```

Ogni metodo agisce direttamente sul valore restituito dal metodo precedente; il valore restituito dall'ultimo metodo rappresenta il risultato finale.

Ebbene, a questa tecnica di "chiamata concatenata" si da il nome di *fluent syntax*.

1.4 Esempio d'uso di LINQ

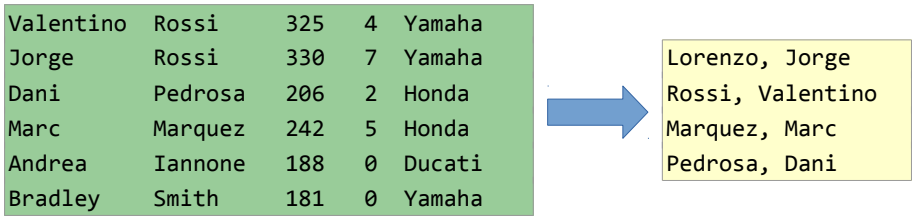
Vediamo subito un esempio di LINQ all'opera. Supponiamo di aver definito la classe `Pilota`:

```
public class Pilota
{
    public string Nome { get; set; }
    public string Cognome { get; set; }
    public string Team { get; set; }
    public int Punti { get; set; }
    public int Vittorie { get; set; }
}
```

E di avere una collezione di piloti, restituita dal metodo `GetPiloti()`:

```
public List<Pilota> GetPiloti()
{
    var piloti = new List<Pilota>();
    piloti.Add(new Pilota { Nome = "Valentino", Cognome = "Rossi",
                            Punti = 325, Vittorie=4, Team = "Yamaha" });
    piloti.Add(new Pilota { Nome = "Jorge", Cognome = "Lorenzo",
                            Punti = 330, Vittorie=7, Team = "Yamaha" });
    ...
    return piloti;
}
```

Dalla lista vogliamo ottenere l'elenco dei nominativi di coloro che hanno vinto almeno una gara, ordinati in base al punteggio.



1.4.1 Soluzione standard

Si realizza in tre passi: filtrare i piloti con almeno una vittoria, ordinarli e selezionare il nominativo concatenando cognome e nome:

```
static string[] GetNominativiPiloti(List<Pilota> piloti)
{
    var vincitori= new List<Pilota>();
    foreach (var pilota in piloti)
    {
        if (pilota.Vittorie > 0)
            vincitori.Add(pilota);
    }

    OrdinaPilotiPerPunteggio(vincitori);

    var nominativi = new string[vincitori.Count];
```

```
    for (int i = 0; i < nominativi.Length; i++)
    {
        nominativi[i] = vincitori[i].Cognome + ", " + vincitori[i].Name;
    }
    return nominativi;
}

private static void OrdinaPilotiPerPunteggio(List<Pilota> piloti)
{
    ...
}
```

Segue la soluzione che usa LINQ:

```
static string[] DemoLinq(List<Pilota> piloti)
{
    return piloti.Where(p => p.Vittorie > 0)
        .OrderByDescending(p => p.Punti)
        .Select(p => p.Cognome + ", " + p.Name)
        .ToArray();
}
```

Per capire il funzionamento della seconda soluzione occorre comprendere *extension methods* e *lambda expressions*. Le seconde sono trattate in **Delegate**; qui introdurrò gli *extension methods* e quindi unirò i due concetti.

(D'ora in avanti userò l'acronimo EM per *extension method*.)

2 Extension method

Un EM è un metodo statico che estende il funzionamento di un certo tipo, che qui chiamerò *tipo bersaglio*. Il metodo viene utilizzato come se fosse definito all'interno del *tipo bersaglio*[1], anche se in realtà è implementato altrove.

Il metodo deve definire almeno un parametro, appartenente al *tipo bersaglio*; inoltre deve essere implementato all'interno di una classe statica.

Di seguito è definito lo scheletro di due EM:

```
public static class <classe_statica>
{
    public static <tipo_bersaglio> Metodo(this <tipo_bersaglio> arg)
    {
        ...
    }

    public static void Metodo(this <tipo_bersaglio> arg1, <tipo> arg2, ...)
    {
        ...
    }
}
```

Nota bene: il primo parametro è prefissato da **this**; è l'uso di questa parola chiave che trasforma dei normali metodi statici in EM.

Per un determinato *tipo bersaglio* si possono definire quanti EM si vuole, dentro la stessa classe o suddivisi in più classi. Non esistono altri vincoli: gli EM possono essere utilizzati come qualsiasi altro metodo statico.

2.1 Esempio di *extension method* per il tipo string

Supponiamo di implementare un metodo di utilità per il tipo **string**: il metodo inverte il *case* delle lettere, trasformando maiuscole in minuscole e viceversa:

```
public static class StringHelper
{
    public static string InvertiCaseLettere(string testo)
    {
        var sb = new StringBuilder(testo.Length);
        foreach (var ch in testo)
        {
            if (ch >= 'a' && ch <= 'z')
                sb.Append(char.ToUpper(ch));
            else
                sb.Append(char.ToLower(ch));
        }
        return sb.ToString();
    }
}
```

1 E viene riconosciuto come tale anche dall'*intellisense* di Visual Studio.

Il metodo è utilizzabile nel modo consueto:

```
var s = "Ciao! Come STAI?";
s = StringHelper.InvertiCaseLettere(s);   //-> cIAO! cOME stai?
```

Funziona correttamente, ma la sua sintassi è "spigolosa", soprattutto se viene usato insieme ad altri metodi della classe **string**:

```
var s = "  ---Ciao! Come STAI?";
s = StringHelper.InvertiCaseLettere(s.Trim().Substring(3)); //-> cIAO! cOME stai?
```

L'uso del metodo è reso più naturale trasformandolo in un EM. Basta aggiungere `this` davanti al tipo **string**:

```
public static class StringHelper
{
    public static string InvertiCaseLettere(this string testo)
    {
        ...
    }
}
```

Ora può essere usato sia in modo consueto, che come metodo d'istanza di **string**:

```
var s = "Ciao! Come STAI?";
s = StringHelper.InvertiCaseLettere(s); //-> cIAO! cOME stai?

var s2 = "Ciao! Come STAI?";
s2 = s2.InvertiCaseLettere();           //-> cIAO! cOME stai?
```

Inoltre si integra con gli altri metodi di **string** e consente l'uso della *fluent syntax*:

```
var s = "   ---Ciao! Come STAI?";
s = s.Trim().Substring(3).InvertiCaseLettere();      //-> cIAO! cOME stai?
```

2.2 *Extension methods* che restituiscono il *tipo bersaglio*

Non è il tipo del valore ritornato a rendere un metodo un EM; d'altra parte, un EM che ritorna un valore del *tipo bersaglio* è più facilmente utilizzabile con la *fluent syntax*.

Molti *extension methods* utilizzati in LINQ seguono questo pattern.

2.3 Estendere un tipo mediante *extension methods*

Ha poco senso implementare un EM soltanto per semplicità d'uso. È utile implementare un insieme di EM dalle funzionalità correlate, in modo da aggiungere nuove funzioni a un tipo.

(Nota bene: un EM si aggiunge alla lista dei metodi di istanza mostrati dall'*intellisense* di Visual Studio; ciò significa che troppi EM possono ridurre l'utilità dell'*intellisense*.)

3 LINQ: Extension methods + lambda expressions

Consideriamo il seguente frammento di codice:

```
var piloti = GetPiloti();

var list = piloti.Where(p => p.Vittorie > 0)  //->Rossi.. Lorenzo.. Pedrosa.. Marquez..
    .Select(p => p.Cognome + ", " + p.Name)   //->"Rossi, Valentino", "Lorenzo, Jorge",..
```

Viene eseguita una ***query*** su `piloti`, e cioè una richiesta che produce un nuovo elenco, in questo caso formato dai nominativi dei piloti che hanno vinto almeno una gara.

Dobbiamo capire in che modo i metodi **Where()** e **Select()** producono questo risultato.

3.1 Implementare un metodo che "filtra" una lista

Consideriamo il seguente problema: implementare un metodo d'estensione a **List<string>** che consenta di filtrare i suoi elementi in base a una condizione. Ecco alcuni esempi di ciò potremmo ottenere da un simile metodo. Data una lista di stringhe:

- ottenere una lista di quelle che cominciano per 'A';
- ottenere una lista di quelle più lunghe di 5 caratteri;
- ottenere una lista di quelle non vuote. Eccetera.

Segue un'implementazione che risponde alla prima richiesta[2]:

```
public static class ListExtension
{
    public static List<string> Filtra(this List<string> list)
    {
        var result = new List<string>();
        foreach (var s in list)
        {
            if (s.StartsWith("A"))
                result.Add(s);
        }
        return result;
    }
}
```

Ecco un esempio d'uso del metodo:

```
var città  = new List<string>() {"Ancona", "Roma", "Firenze", "Ascoli"};
var listaFiltrata = città.Filtra(); // -> "Ancona", "Ascoli"
```

Si tratta di un'implementazione poco efficace, poiché la condizione di filtro è stabilita nel metodo. Perché il metodo sia utile è necessario che la condizione sia stabilita dal codice chiamante e che il metodo si limiti ad applicarla. Serve un *delegate*.

(Vedi "Delegate predefiniti" nel tutorial **Delegate**).

2 Il metodo non tiene conto di eventuali stringhe nulle.

3.1.1 Implementare il filtro mediante una lambda expression

Filtrare un elenco di stringhe significa: per ogni stringa stabilire se debba fare parte del risultato oppure no. Il filtro è espresso dal seguente *delegate*:

```
Func<string, bool> filtro;   // string -> bool  (true: inclusa;  false: esclusa)
```

Ecco dunque una versione di `Filtra()` che usa una condizione stabilita dal chiamante:

```
public static List<string> Filtra(this List<string> list, Func<string, bool> filtro)
{
    var result = new List<string>();
    foreach (var s in list)
    {
        if (filtro(s))
            result.Add(s);
    }
    return result;
}
```

Ora è possibile filtrare l'elenco in base a una condizione qualsiasi:

```
...
var elenco = città.Filtra(s=>s.StartsWith("R"));  //-> "Roma"
elenco = città.Filtra (s => s.Length > 5);        //-> "Ancona", "Firenze", Ascoli"
elenco = città.Filtra (s => !s.EndsWith("a"));    //-> "Firenze", "Ascoli"
```

3.2 Implementare una “proiezione” dei dati: creare nuovi dati

Una facoltà di LINQ è quella di consentire la creazione di nuovi dati a partire da dati esistenti. È ciò che fa il codice presentato a inizio capitolo, che da una lista di piloti produce una lista di stringhe:

```
var piloti = GetPiloti();

var list = piloti.Where(p => p.Vittorie > 0)  //->Rossi.. Lorenzo.. Pedrosa.. Marquez..
    .Select(p => p.Cognome + ", " + p.Name); //->"Rossi, Valentino","Lorenzo, Jorge",..
```

Questa operazione si chiama ***proiezione***: una funzione che, dato un elemento, ne produce un altro, eventualmente di tipo diverso.

3.2.1 Implementare la proiezione sul tipo List<Pilota>

Vogliamo implementare un EM che esegua la proiezione **Pilota → string** su una lista di piloti. Il *delegate* da utilizzare è:

```
Func<Pilota, string> selettore;     // Pilota -> string
```

Segue il metodo che implementa la proiezione:

```
public static List<string> Seleziona(this List<Pilota> list, Func<Pilota, string> sel)
{
    var result = new List<string>();
    foreach (var p in list)
```

```
    {
        string s = sel(p);
        result.Add(s);
    }
    return result;
}
```

Nota bene: `sel`, definito *selettore*, è una funzione che ritorna una stringa; dunque il metodo ritorna una lista di stringhe.

Ora, a partire da un elenco di piloti è possibile produrre una lista di stringhe:

```
var piloti = GetPiloti();

var nomi = piloti.Seleziona(p => p.Nome); //-> "Valentino", "Jorge", "Dani"...
var team = piloti.Seleziona(p => p.Team); //-> "Yamaha", "Yamaha", "Honda"...
var CognomiEPunti = piloti.Seleziona(p => string.Format("{0}:{1}", p.Cognome, p.Punti));
// -> "Rossi:325", "Lorenzo:330"...
```

3.3 Usare filtro e proiezione contemporaneamente

Implementiamo prima un filtro per la lista piloti. È necessario utilizzare il *delegate*:

```
Func<Pilota, bool> filtro // Pilota -> bool
```

Ecco il codice:

```
public static List<Pilota> Filtra(this List<Pilota> list, Func<Pilota, bool> filtro)
{
    var result = new List<Pilota>();
    foreach (var p in list)
    {
        if (filtro(p))
            result.Add(p);
    }
    return result;
}
```

Ed ecco alcuni esempi che combinano filtro e proiezione usando la *fluent syntax*:

```
var vincenti = piloti.Filtra(p => p.Vittorie > 0)
                     .Seleziona(p => p.Cognome); //-> "Rossi", "Lorenzo", ...

var teamVincenti = piloti.Filtra (p => p.Vittorie == 0)
                         .Seleziona(p => p.Team.ToUpper()); //-> "DUCATI"
```

3.4 Reale implementazione di LINQ: uso dei *generics*

Gli esempi predenti introducono i principi sottostanti al funzionamento di LINQ, ma si intuisce facilmente che LINQ ha un'implementazione diversa[3]. Infatti, i metodi proposti:

- estendono collezioni di un determinato tipo (**List<string>**, **List<Pilota>**, ...)
- forniscono soltanto un certo tipo di proiezione: **Pilota→string**.

LINQ, dal canto suo, è in grado di processare qualsiasi collezione e proiettare qualsiasi tipo di dato. LINQ raggiunge questo risultato utilizzando i *generics*: interfacce generiche come *tipi bersaglio* e *delegate* generici come parametri in *extension methods* generici.

L'argomento non viene qui approfondito (Vedi il tutorial **Generics**). A titolo di esempio fornisco un'implementazione generica del metodo `Filtra()` che accetta qualsiasi collezione e usa qualsiasi tipo di filtro:

```
public static IEnumerable<T> Filtra<T>(this IEnumerable<T> list, Func<T, bool> filtro)
{
    var result = new List<T>();
    foreach (var item in list)
    {
        if (filtro(item))
            result.Add(item);
    }
    return result;
}
```

Segue un'implementazione generica di `Seleziona()`:

```
public static IEnumerable<TR> Seleziona<T,TR>(this IEnumerable<T> list, Func<T,TR> sel)
{
    var result = new List<TR>();
    foreach (var item in list)
    {
        result.Add(sel(item));
    }
    return result;
}
```

3.4.1 Concetto di sequenza: IEnumerable<>

Il *tipo bersaglio* di entrambi i metodi è **IEnumerable<>**, un'interfaccia che rappresenta una generica *sequenza di elementi*. Vettori, liste e tutti tipi di collezione sono delle *sequenze* (implementano **IEnumerable<>**), ma non è vero il contrario.

L'unica operazione ammessa su una sequenza è quella di accedere ai suoi elementi uno per volta, dal primo all'ultimo.

(In realtà `Filtra()` e `Seleziona()` hanno implementazioni inefficienti: i metodi usati da LINQ usano degli **iteratori** per restituire un elemento per volta della sequenza, invece di costruire una nuova collezione contenente gli elementi da restituire. Vedi tutorial **Iteratori**.)

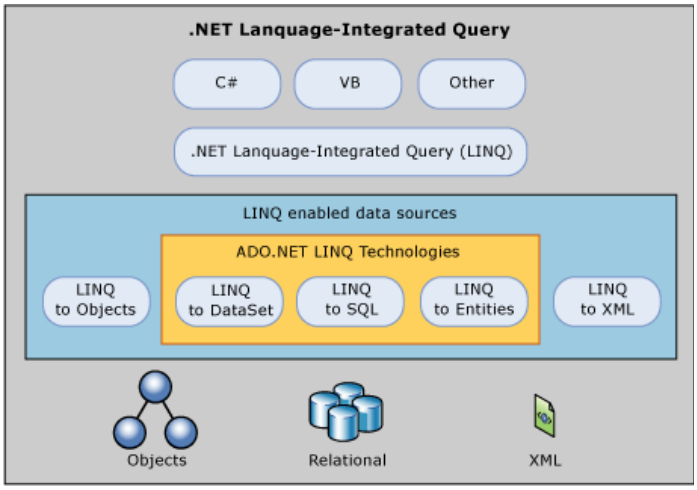
3 Oltre al fatto che LINQ è in grado di fare ben altro rispetto ai semplici esempi proposti.

4 LINQ Query

Dopo aver esaminato i meccanismi sui quali si fonda LINQ, prenderemo in considerazione le operazioni più comuni del linguaggio.

Prima di procedere, però, è importante comprendere che LINQ non riguarda le liste in sé, ma è utilizzabile su qualsiasi sorgente di dati che fornisca un cosiddetto *LINQ Provider*.

La figura sottostante mostra le varie "declinazioni" di LINQ: **LINQ to Objects**, **LINQ to XML**, etc, in base alla sorgente dei dati:



Qui utilizzerò *LINQ to Objects*, applicabile a sequenze di oggetti in memoria (vettori, liste, etc).

4.1 Cos'è una "LINQ Query"

Una *LINQ Query* (interrogazione) è ciò che abbiamo visto finora: l'invocazione di uno o più metodi LINQ su una sequenza di elementi. La *query* produce un risultato, spesso rappresentato da una sequenza di elementi dello stesso tipo.

4.1.1 Operatori LINQ

In LINQ gli *extension methods* sono convenzionalmente chiamati ***operatori***. Di seguito prenderò in considerazione gli operatori più comuni, categorizzandoli e in base alla natura del risultato che producono.

4.2 Filtro / ordinamento / partizionamento / insiemi / concatenazione

Questi operatori producono una nuova sequenza dello stesso tipo di quella processata. Gli elementi non sono modificati; semplicemente: viene cambiata la loro posizione nella sequenza e/o alcuni di essi vengono esclusi dal risultato.

4.2.1 Where

Filtra gli elementi della sequenza.

Richiede come argomento un delegate che, dato un elemento della sequenza, produce come risultato **true** (l'elemento viene incluso nel risultato) o **false** (l'elemento viene scartato)

```
int[] data = { -1, 2, 3, -4, -5, 6 };
var result = data.Where(n => n % 2 == 0);    //-> -2, 4, 6
result     = data.Where(n => n < 0);         //-> -1, -4, -5
```

4.2.2 OrderBy (e OrderByDescending)

Restituisce una sequenza ordinata in base alla chiave specificata.

Richiede un delegate, **keySelector**, che stabilisca la chiave di ordinamento. Per i tipi primitivi (**int**, **double**, **string**, etc) è l'elemento stesso della sequenza.

```
int[] data = { 6, 2, -4, 3, -5, -1 };
var result = data.OrderBy(n => n);     //-> -5, -4, -1, 2, 3, 6
```

Se il tipo della sequenza è strutturato, è necessario che **keySelector** specifichi il campo da usare per l'ordinamento:

```
class Pet
{
    public string Name { get; set; }
    public int Age { get; set; }
}
...
Pet[] pets = { new Pet { Name="Black", Age=10 },
               new Pet { Name="Lampo", Age=4 },
               new Pet { Name="Rosco", Age=6 } };

var orderedPets = pets.OrderBy(p => p.Age);                 //-> Lampo, Rosco, Black
orderedPets     = pets.OrderByDescending(p => p.Name);      //-> Rosco, Lampo, Black
```

4.2.3 ThenBy e ThenByDescending

Entrambi sono usati in congiunzione con ***OrderBy*** e ***OrderByDescending*** per eseguire un doppio ordinamento: principale e secondario[4].

Ad esempio, si potrebbe ordinare i piloti in base al team e, a parità di team, in base al punteggio in ordine decrescente:

```
var lista = piloti.OrderBy(p => p.Team) // ordina per team
   .ThenByDescending(p => p.Punti);     // ordina per punteggio a parità di team
//->Ducati   Iannone   188
//->Honda    Marquez   242
//->Honda    Pedrosa   206
...
```

4.2.4 Reverse

Inverte l'ordine degli elementi.

```
int[] data = { 6, 2, -4, 3, -5, -1 };
var result = data.Reverse();  //-> -1, -5, 3, -4, 2, 6
```

4 In realtà è possibile implementare anche tre o più livelli di ordinamento; è sufficiente chiamare più volte l'operatore **ThenBy**.

4.2.5 Skip

Salta i primi "n" elementi e restituisce il resto della sequenza.

```
int[] data = { 6, 2, -4, 3, -5, -1 };
var result = data.Skip(2);  //-> -4, 3, -5, -1  (salta 6 e 2)
```

4.2.6 Take

Restituisce i primi "n" elementi e scarta gli altri.

```
int[] data = { 6, 2, -4, 3, -5, -1 };
var result = data.Take(2);  //-> 6, 2
```

4.2.7 Distinct / Except / Intersect / Union

Eseguono le tipiche operazioni sugli insiemi: eliminazione duplicati, differenza, intersezione e unione.

Distinct elimina gli elementi duplicati.

```
int[] data = { 6, 2, -4, 2, -5, 2 };
var result = data.Distinct();  //-> 6, 2, -4, -5
```

Except (differenza tra insiemi) esclude dal risultato gli elementi che sono presenti nella seconda sequenza e rimuove i duplicati:

```
int[] data = { 1, 2, 3, 2, 5, 2 };
int[] escludeData = { 1, 7, 5, 4 };
var result = data.Except(escludeData);  //-> 2, 3
```

Intersect (intersezione tra insiemi) restituisce soltanto gli elementi che sono presenti *anche* nella seconda sequenza e rimuove i duplicati.

```
int[] data = { 1, 2, 3, 2, 5, 2 };
int[] includeData = { 1, 7, 5, 4, 2 };
var result = data.Intersect(includeData);  //-> 1, 2, 5
```

Union (unione tra insiemi) restituisce gli elementi di entrambe le sequenze e rimuove i duplicati.

```
int[] data = { 1, 2, 5, 2 };
int[] moreData = { 1, 7, 5, 2, -4 };
var result = data.Union(moreData);  //-> 1, 2, 5, 7, -4
```

4.2.8 Concat

Unisce due sequenze senza rimuovere i duplicati.

```
int[] data = { 1, 2, 5, 2 };
int[] moreData= { 1, 7, 5, 2, -4 };
var result = data.Concat(moreData);  //-> 1, 2, 5, 2, 1, 7, 5, 2, -4
```

4.3 Proiezione

La proiezione è un'operazione che trasforma gli elementi, eventualmente modificando anche il tipo. Esistono due operatori: *Select* e *SelectMany*; qui vedremo soltanto il primo.

4.3.1 Select

Per ogni elemento della sequenza crea un nuovo elemento.

Select usa un ***selector***, e cioè un delegate che, dato un elemento della sequenza, restituisce un nuovo elemento, eventualmente di tipo diverso.

Il prossimo esempio mostra come eseguire un'operazione su tutti gli elementi senza cambiare il loro tipo:

```
int[] data = { 1, 2, 5, 2 };
var result = data.Select(e => e * 2);  //-> 2, 4, 10, 4
```

Il secondo esempio mostra come ottenere una sequenza di stringhe a partire da una di `Pet`:

```
Pet[] pets = { new Pet { Name="Black", Age=10 },
               new Pet { Name="Lampo", Age=4 },
               new Pet { Name="Rosco", Age=6 } };

var petsName = pets.Select(p => p.Name);           //-> "Black", "Lampo", "Rosco"
```

4.4 Elemento

Questi operatori restituiscono un singolo elemento della sequenza.

4.4.1 First / FirstOrDefault / Last / LastOrDefault

Restituiscono il primo/ultimo elemento della sequenza.

FirstOrDefault e ***LastOrDefault*** restituiscono un valore predefinito se la sequenza è vota.

```
int[] data = { -4, 2, 5, 7 };

int result = data.First();       //-> -4
result = data.FirstOrDefault();  //-> -4
result = data.Last();            //-> 7
result = data.LastOrDefault();   //-> 7

int[] moreData = { };               // sequenza vuota
result = moreData.FirstOrDefault(); //-> 0 (0 è default per tipo int);
result = moreData.First();          //-> InvalidOperationException
result = moreData.Last();           //-> InvalidOperationException
result = moreData.LastOrDefault();  //-> 0
```

Nota bene:

- Il valore restituito è sempre dello stesso tipo degli elementi della sequenza.
- ***First*** e ***Last*** possono sollevare un'eccezione.

4.4.2 Single / SingleOrDefault

Restituiscono un elemento della sequenza. Richiedono che *uno e un solo elemento soddisfi la condizione*. In caso contrario:

- ***Single***: solleva l'eccezione **InvalidOperationException**.
- ***SingleOrDefault***: ritorna il valore di default se la sequenza è vuota; solleva l'eccezione se la sequenza contiene più di un elemento.

Entrambi esistono in due versioni, con e senza condizione. Nel primo caso l'operatore viene applicato alla sequenza dopo che è stata filtrata dalla condizione.

```
int[] data = { 2 };
int result = data.Single();                       //-> 2
result = data.SingleOrDefault();                  //-> 2

int[] moreData = { 2, 3, 4 };
result = moreData.Single();                      //-> InvalidOperationException
result = moreData.SingleOrDefault();             //-> InvalidOperationException

result = moreData.Single(e => e == 2);           //-> 2
result = moreData.Single(e => e == 6);           //-> InvalidOperationException
result = moreData.SingleOrDefault(e => e == 6); //-> 0
```

4.5 Aggregazione

Gli operatori di aggregazione restituiscono una valore sintetico basato sugli elementi della sequenza (tipicamente un valore numerico).

4.5.1 Count / Sum / Average / Max / Min

Restituiscono conteggio, somma, media, massimo e minimo della sequenza.

```
int[] data = { 1, 2, 3, 7};

int count  = data.Count();    //-> 4
int sum    = data.Sum();      //-> 13
double avg = data.Average();  //-> 2,5
int max    = data.Max();      //-> 7
int min    = data.Min();      //-> 1
```

Il valore restituito è dello stesso tipo degli elementi della sequenza. (***Average*** restituisce **double** se la sequenza è di tipo **int**)

4.6 Quantificazione

Restituiscono **true** se la sequenza soddisfa un certo criterio, **false** altrimenti.

4.6.1 Any / All

Any esiste in due versioni. La prima restituisce **true** se la sequenza contiene almeno un elemento. La seconda restituisce **true** se almeno un elemento soddisfa la condizione specificata.

All restituisce **true** se tutti gli elementi della sequenza soddisfano la condizione specificata.

```
int[] data = { 1, 2, 3, 4};
bool result = data.Any();       //-> true
result = data.Any(e => e < 0);  //-> false (nessun valore negativo)
result = data.All(e => e > 0);  //-> true (tutti maggiori di 0)
result = data.All(e => e < 4);  //-> false (4 non è minore di 4))
```

4.6.2 Contains

Restituisce **true** se la sequenza contiene l'elemento specificato.

```
int[] data = { 1, 2, 3, 4};
bool result = data.Contains(3); // -> true;
result = data.Contains(0);      // -> false;
```

4.7 Conversione

Questi operatori convertono la sequenza in un tipo specificato, oppure eseguono la conversione degli elementi.

4.7.1 ToArray / ToList

Convertono la sequenza in un vettore o un **List<>**.

Ad esempio: dato un vettore di interi, ottenere un secondo vettore contenente quelli positivi:

```
int[] data = { -1, 2, -3, 4};
int[] result = data.Where(e => e >= 0).ToArray(); //-> vettore contenente 2,4
List<int> result2 = data.Where(e => e >= 0).ToList(); //-> lista contenente 2,4
```

4.7.2 OfType

Restituisce gli elementi della sequenza che hanno il tipo specificato. L'operatore è utile soltanto con sequenze polimorfiche.

Ad esempio, ipotizziamo la gerarchia di classi: **Figura** → **TriangoloRettangolo**, **Rettangolo**, **Quadrato** e supponiamo di avere una lista di oggetti appartenenti alle classi suddette:

```
var figure = new List<Figura>();
figure.Add(new Quadrato(10));       //-> lato: 10
figure.Add(new Rettangolo(10, 2));  //-> base: 10, altezza: 2
figure.Add(new Quadrato(20));       //-> lato: 20
figure.Add(new TriangoloRettangolo(3, 4, 5));  //-> cateto1: 3, cateto2: 4, ipotenusa: 5
```

È possibile ottenere l'elenco dei quadrati con la seguente istruzione:

```
var list = figure.OfType<Quadrato>(); //-> primo e terzo elemento
```

Lo stesso risultato si può ottenere, anche se in modo meno conciso:

```
var list = figure.Where(f => f is Quadrato); //-> primo e terzo elemento
```

4.8 Generazione

Consentono di generare una sequenza di elementi. Sono normali metodi statici e dunque devono essere prefissati dalla classe statica di appartenenza: **Enumerable**.

4.8.1 Range

Genera una sequenza progressiva di numeri interi appartenenti a un certo intervallo.

```
var result = Enumerable.Range(1, 5); //-> 1,2,3,4,5
```

4.8.2 Repeat

Genera una sequenza di elementi ripetuti. Il tipo dell'elemento deve essere specificato:

```
var result1 = Enumerable.Repeat<int>(3, 5);          //->3, 3, 3, 3, 3
var result2 = Enumerable.Repeat<string>("A", 4);     //-> "A", "A", "A", "A"
```

4.9 Esecuzione immediata o differita degli operatori

Gli operatori non si differenziano solo per la natura del risultato (sequenza, valore singolo, lista o array), ma anche per la modalità di esecuzione: *immediata* o *differita*.

La modalità di esecuzione *immediata* non necessita di spiegazioni: l'operazione viene eseguita e il risultato immediatamente restituito. La modalità differita richiede un approfondimento.

4.9.1 Esecuzione differita di una query

L'accesso agli elementi di una sequenza (**IEnumerable<>**) avviene mediante un oggetto, chiamato ***iteratore***, che rende disponibili agli elementi uno per volta. L'*iteratore* "promette" soltanto questo: l'accesso a un elemento per volta, *man mano che gli elementi vengono richiesti*.

Questo meccanismo può avere delle conseguenze notevoli. Si consideri il seguente codice:

```
int[] dati = { -1, 2, -3, 4};
var result = dati.Where(e => e > 0);  //-> 2, 4 (?)

dati[0] = 1;                //attenzione: i dati originali sono cambiati!

foreach (var e in result)  //soltanto ora si accede ai dati!
{
    Console.WriteLine(e);  //-> 1, 2, 4
}
```

La variabile `result` non memorizza la sequenza 2, 4, ma un *iteratore* che consente di accedere agli elementi di `dati` maggiori di zero[5]. Se si modifica i dati prima di "consumarli", ecco che l'accesso a `result` riflette le modifiche effettuate.

Si parla dunque di *esecuzione differita*, poiché in realtà la *query* non produce alcun accesso ai dati, ma lo "differisce" a quando i dati sono effettivamente richiesti.

Nella maggioranza dei casi la distinzione tra esecuzione immediata e differita è irrilevante, poiché i dati sono utilizzati immediatamente dopo l'esecuzione della *query*.

5 Non è propriamente esatto, ma comunque corretto nell'ambito del discorso che stiamo facendo.

5 Uso di “oggetti anonimi”

C# fornisce una funzionalità che consente di creare “al volo” degli oggetti senza dover specificare il tipo di appartenenza. Si tratta dei cosiddetti **oggetti anonimi**, che sono ampiamente utilizzati in LINQ per estrarre informazioni da una sequenza.

5.1 Ottenere nuovi dati mediante oggetti anonimi

Consideriamo nuovamente il problema di estrarre i nomi dei piloti memorizzati in una lista:

```
var piloti = GetPiloti();

var list = piloti.Where(p => p.Vittorie > 0)  //->Rossi.. Lorenzo.. Pedrosa.. Marquez..
    .Select(p => p.Cognome + ", " + p.Name); //->"Rossi, Valentino","Lorenzo, Jorge",..
```

La *query* produce una sequenza di stringhe; ma se invece dei soli nominativi volessimo ottenere l'elenco dei nominativi e dei rispettivi team? Saremmo costretti a definire un nuovo tipo di dato per l'occasione:

```
public class PilotaInfo
{
    public string Nominativo { get; set; }
    public string Team { get; set; }
}
```

E scrivere:

```
var list = piloti.Where(p => p.Vittorie > 0)
 .Select(p => new PilotaInfo { Nominativo = p.Cognome + ", " + p.Nome, Team = p.Team });
```

Di fatto abbiamo definito un nuovo tipo solo per estrarre alcune informazioni da una sequenza. L'uso di oggetti anonimi risolve il problema.

5.1.1 Creare oggetti anonimi

Un oggetto anonimo è semplicemente un oggetto senza tipo. Ad esempio:

```
var pilota = new { Nominativo = "Rossi, Valentino", Team = "Yamaha" };
Console.WriteLine("{0} {1}", pilota.Nominativo, pilota.Team);
```

`pilota` è un oggetto privo di tipo, ma che definisce comunque delle proprietà di un tipo ben definito (`Nominativo` e `Team` sono entrambe **string**). Dietro le quinte, C# definisce automaticamente una classe che dichiara le proprietà utilizzate.

L'uso di oggetti anonimi impone dei requisiti da rispettare. Tra questi:

- La variabile deve essere locale e, ovviamente, prefissata da **var**, poiché non esiste un tipo per dichiararla.
- L'oggetto non può essere utilizzato fuori dal metodo nel quale viene definito[6]

6 Non è esatto, poiché è possibile passare l'oggetto a un metodo generico o assegnarlo a una variabile **dynamic**

Un oggetto anonimo non definisce metodi, eccetto **ToString()**, che produce una stringa identica all'espressione di inizializzazione.

Ad esempio:

```
var pilota = new { Nominativo = "Rossi, Valentino", Team = "Yamaha" };
string s = pilota.ToString(); //-> "{ Nominativo = "Rossi, Valentino", Team = "Yamaha" }
```

5.1.2 Proiettare dati mediante oggetti anonimi

Nell'operatore *Select* si tratta di creare un oggetto anonimo che definisca le proprietà desiderate. Il codice precedente può dunque essere riscritto così:

```
var list = piloti.Where(p => p.Vittorie > 0)
 .Select(p => new { Nominativo = p.Cognome + ", " + p.Nome, Team = p.Team });
```

Naturalmente, il risultato deve utilizzato immediatamente; ad esempio:

```
var list = piloti.Where(p => p.Vittorie > 0)
    .Select(p => new { Nominativo = p.Cognome + ", " + p.Nome, Team = p.Team });

foreach (var p in list)
{
    Console.WriteLine("{0} {1}", p.Nominativo, p.Team);
}
```

6 Esempio d'uso di LINQ

Segue un semplice esempio che mostra LINQ in azione. Supponiamo di avere il seguente file, in formato CSV:

```
#Catalogo video

Matrix | Monica Bellucci, Hug Weaving, Laurence Fishburne | 2008 | MATRIX.JPG
Iception | Leonardo Di Caprio, Pete Postlethwaite, Michael Caine | 2012 | INCEPTION.JPG
Real steel | Hope Davis, Hugh Jackman, Anthony Mackie | 2012 | REALSTEEL.JPG
Star Wars | Hope Harrison Ford, Mark Hamill, Carrie Fisher | 2011 | STARWARS.JPG
Blade runner|Harrison Ford, Daryl Hannah, Rutger Hauer, Sean Young|2011|BLADERUNNER.JPG
```

Eccetto il commento iniziale e la linea vuota, ogni riga è composta da quattro campi separati dal carattere `|`. Si desidera caricarlo in memoria, memorizzando i dati in una lista di oggetti di tipo `Film`:

```
public class Film
{
    public string Titolo { get; set; }
    public string Cast { get; set; }
    public int Anno { get; set; }
    public string FileImmagine { get; set; }
}
```

6.1 Caricamento file CSV con LINQ

Segue il codice:

```
static void Main(string[] args)
{
    var elencoFilm = File.ReadLines("ElencoFilm.txt")
        .Where(line => !line.Trim().StartsWith("#") && line.Trim() != "")
        .Select(line => LineToFilm(line));

    foreach (var film in elencoFilm)
    {
        Console.WriteLine(film.Titolo);
    }
    Console.ReadKey();
}

static Film LineToFilm(string line)
{
    string[] items = line.Split('|');
    return new Film
    {
        Titolo = items[0].Trim(),
        Cast = items[1].Trim(),
        Anno = int.Parse(items[2].Trim()),
        FileImmagine = items[3].Trim()
    };
}
```

Nota bene: la *lamba expression* nell'operatore *Where* filtra le righe vuote e i commenti[7].

6.1.1 Uso di una lambda expression in Select

Nell'operatore *Select* viene chiamato il metodo `LineToFilm()`; ciò è perfettamente legittimo e semplifica la *query*, ma è possibile eseguire l'intera operazione nella *lambda expression*:

```
static void Main(string[] args)
{
    var elencoFilm = File.ReadLines("ElencoFilm.txt")
        .Where(line => !line.Trim().StartsWith("#") && line.Trim() != "")
        .Select(line =>
        {
            string[] items = line.Split('|');
            return new Film
            {
                Titolo = items[0].Trim(),
                Cast = items[1].Trim(),
                Anno = int.Parse(items[2].Trim()),
                FileImmagine = items[3].Trim()
            };
        });

    foreach (var film in elencoFilm)
    {
        Console.WriteLine(film.Titolo);
    }
    Console.ReadKey();
}
```

6.1.2 Uso di un oggetto anonimo

Se tutto ciò che serve è visualizzare i dati del file, non conviene definire il tipo `Film`, ma utilizzare un oggetto anonimo con identiche proprietà:

```
var elencoFilm = File.ReadLines("ElencoFilm.txt")
    .Where(line => !line.Trim().StartsWith("#") && line.Trim() != "")
    .Select(line =>
    {
        string[] items = line.Split('|');
        return new
        {
            Titolo = items[0].Trim(),
            Cast = items[1].Trim(),
            Anno = int.Parse(items[2].Trim()),
            FileImmagine = items[3].Trim()
        };
    });
...
```

7 La soluzione è poco efficiente, poiché esegue **Trim()** due volte. Si può evitare, ma ciò richiede un uso di LINQ che qui non è stato trattato.