

Introduzione al funzionamento di LINQ

Ambiente: .NET 4.0/C# 4.0

Anno 2018

Indice generale

1	Introduzione.....	4
1.1	Cos'è LINQ.....	4
1.2	Su cosa si basa LINQ.....	4
1.3	Fluent syntax.....	4
1.4	Esempio d'uso di LINQ.....	5
1.4.1	Soluzione “standard”	5
2	Extension method.....	7
2.1	Esempio di <i>extension method</i> per il tipo string.....	7
2.2	<i>Extension method</i> che restituiscono il <i>tipo bersaglio</i>	8
3	LINQ: Extension method + lambda expression.....	9
3.1	Implementare un metodo che “filtra” una lista.....	9
3.1.1	Implementare il filtro mediante una lambda expression.....	10
3.2	Implementare una “proiezione” dei dati: creare nuovi dati.....	10
3.2.1	Implementare la proiezione sul tipo List<Pilota>.....	10
3.3	Usare filtro e proiezione contemporaneamente.....	11
3.4	Reale implementazione di LINQ: uso dei <i>generics</i>	12
3.4.1	Concetto di sequenza: IEnumerable<>.....	12
4	LINQ Query.....	13
4.1	Cos'è una “LINQ Query”	13
4.1.1	Operatori LINQ.....	13
4.2	Filtro / ordinamento / partizionamento / insiemi / concatenazione.....	13
4.2.1	Where.....	13
4.2.2	OrderBy (e OrderByDescending).....	13
4.2.3	ThenBy e ThenByDescending.....	14
4.2.4	Reverse.....	14
4.2.5	Skip.....	14
4.2.6	Take.....	14
4.2.7	Distinct / Except / Intersect / Union.....	14
4.2.8	Concat.....	15
4.3	Proiezione.....	15
4.3.1	Select.....	15
4.4	Elemento.....	16
4.4.1	First / FirstOrDefault / Last / LastOrDefault.....	16
4.4.2	Single / SingleOrDefault.....	16
4.5	Aggregazione.....	17

4.5.1	Count / Sum / Average / Max / Min.....	17
4.6	Quantificazione.....	17
4.6.1	Any / All.....	17
4.6.2	Contains.....	17
4.7	Conversione.....	17
4.7.1	ToArray / ToList.....	17
4.7.2	OfType.....	18
4.8	Generazione.....	18
4.8.1	Range.....	18
4.8.2	Repeat.....	18
5	Esecuzione immediata o differita degli operatori.....	19
5.1.1	Esecuzione differita di una query.....	19
6	Uso di “oggetti anonimi”	20
6.1	Ottenere nuovi dati mediante oggetti anonimi.....	20
6.1.1	Creare oggetti anonimi.....	20
6.1.2	Proiettare dati mediante oggetti anonimi.....	21
7	Esempio d'uso di LINQ.....	22
7.1	Caricamento file CSV con LINQ.....	22
7.1.1	Uso di una lambda expression in Select.....	23
7.1.2	Uso di un oggetto anonimo.....	23

1 Introduzione

Questo tutorial introduce i principi di funzionamento di LINQ (**L**anguage **I**Ntegrated **Q**uery). Il linguaggio si fonda su: **extension method**, **lambda expression** e **generics**. Qui analizzo il funzionamento degli *extension method*; per gli altri due “pilastri” del linguaggio vedi i tutorial **Delegate** e **Generics**.

1.1 Cos'è LINQ

LINQ è linguaggio/API di interrogazione. Consente di elaborare sequenze di oggetti allo scopo di ottenere determinate informazioni. Esiste in due forme:

- **API**: insieme di metodi eseguibili in modo “concatenato”, attraverso la cosiddetta **fluent syntax**.
- **Linguaggio**: 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 quattro “pilastri”:

- **Generics**: è indipendente dal tipo degli oggetti (stringhe, interi, etc).
- **Delegate**: mediante l'uso di *lambda expression* consente di stabilire le operazioni da eseguire sui dati.
- **Extension method**: le operazioni sono eseguite mediante metodi statici utilizzabili come se fossero metodi d'istanza. (Rendono possibile la *fluent syntax*.)
- **Tipi astratti**: le operazioni sono applicabili a qualsiasi tipo che rappresenti una sequenza di elementi (il tipo deve implementare l'interfaccia `IEnumerable<>`).

Di seguito mostrerò come l'integrazione di questi quattro elementi produca il funzionamento di LINQ.

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.

Non si tratta di un concetto nuovo o strano. Considera il seguente problema: data una stringa, togliere gli spazi iniziali e finali, eliminare i primi tre caratteri e trasformare tutto in maiuscolo. Segue una soluzione:

```
var s = "  ---rossi, verdi, bianchi  ";
var tmp1 = s.Trim();                // -> "---rossi, verdi, bianchi"(spazi eliminati)
var tmp2 = tmp1.Substring(3);       // -> "rossi, verdi, bianchi" (trattini eliminati)
var s2 = tmp2.ToUpper();            // -> "ROSSI, VERDI, BIANCHI" (conversione in maiuscolo)
```

Ma si può ottenere lo stesso risultato con una sola 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 si dà il nome di *fluent syntax*; è utilizzabile ai metodi che restituiscono un oggetto dello stesso tipo del parametro al quale vengono applicati.

1.4 Esempio d'uso di LINQ

Vediamo subito un esempio di LINQ all'opera. Supponi 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 i nominativi di coloro che hanno vinto almeno una gara, ordinati in base al punteggio.

Valentino	Rossi	325	4	Yamaha
Jorge	Rossi	330	7	Yamaha
Dani	Pedrosa	206	2	Honda
Marc	Marquez	242	5	Honda
Andrea	Iannone	188	0	Ducati
Bradley	Smith	181	0	Yamaha



Lorenzo, Jorge
Rossi, Valentino
Marquez, Marc
Pedrosa, Dani

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) // filtra
    {
```

```

        if (pilota.Vittorie > 0)
            vincitori.Add(pilota);
    }

    OrdinaPilotiPerPunteggio(vincitori); // -> ordina

    var nominativi = new string[vincitori.Count];

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

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

```

Ecco la soluzione che usa LINQ:

```

static string[] DemoLinq(List<Pilota> piloti)
{
    return piloti.Where(p => p.Vittorie > 0)           // filtra
                  .OrderByDescending(p => p.Punti)     // ordina
                  .Select(p => p.Cognome + ", " + p.Nome) // seleziona i nominativi
                  .ToArray();                          // restituisce i nominativi in un array
}

```

Per capire il funzionamento di questa soluzione occorre comprendere *extension method* e *lambda expression*. Le seconde sono trattate in **Delegate**; qui introdurrò gli *extension method* 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 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 un *modulo*.

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, ...)
    {
        ...
    }
}
```

In entrambi, il primo parametro è prefissato da `this`; è ciò che trasforma dei normali metodi in EM.

Per un determinato *tipo bersaglio* si possono definire quanti EM si vuole, nella stessa classe o 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

Supponi di voler implementare un metodo che, data una stringa, inverta 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();
    }
}
```

Segue un esempio d'uso del metodo:

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

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

L'istruzione di chiamata è piuttosto "spigolosa", soprattutto se il metodo viene usato insieme ad altri metodi della classe `string`:

```
var s = " ---Ciao! Come STAI?"; // elimina spazi, i primi 3 caratteri e inverte case
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:

```
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 method* 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* rende possibile la *fluent syntax*. Molti *extension method* utilizzati in LINQ seguono questo pattern.

3 LINQ: Extension method + lambda expression

Considera 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.Nome) //->"Rossi, Valentino", "Lorenzo, Jorge",..
```

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

È necessario comprendere in che modo i metodi `Where()` e `Select()` producono questo risultato.

3.1 Implementare un metodo che “filtra” una lista

Considera il seguente problema: implementare un metodo d'estensione a `List<string>` che consenta di filtrare i suoi elementi in base a una condizione. Ecco ciò che potremmo ottenere da un simile metodo:

- 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 richiesta²:

```
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. Ma perché quest'ultimo sia realmente utile è necessario che la condizione sia stabilita nel codice chiamante. 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. Il filtro viene espresso dal seguente *delegate*:

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

Segue una versione di `Filtro()` che usa una condizione stabilita dal chiamante:

```
public static List<string> Filtro(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.Filtro(s => s.StartsWith("R")); //-> "Roma"
elenco = città.Filtro(s => s.Length > 5); //-> "Ancona", "Firenze", "Ascoli"
elenco = città.Filtro(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.Nome); //-> "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 `Seleziona()` 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 => $"{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;
}

```

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 forniti introducono i principi basilari, ma si intuisce facilmente che LINQ ha un'implementazione diversa³. Infatti, i metodi finora 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 ed eseguire qualsiasi tipo di proiezione. LINQ raggiunge questo risultato utilizzando interfacce generiche come *tipi bersaglio* e *delegate* generici come parametri in *extension method* generici. (Vedi il tutorial **Generics**).

Di seguito 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 i 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, tipicamente mediante un `foreach`.

(`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 LINQ è in grado di fare ben altro rispetto ai semplici esempi proposti.

4 LINQ Query

Di seguito prendo in considerazione le operazioni più comuni del linguaggio. Prima di procedere, però, è importante ricordare che LINQ non riguarda le collezioni, ma è utilizzabile su qualsiasi sorgente di dati per la quale esista un *LINQ Provider*: **LINQ to Objects**, **LINQ to XML**, **Linq To Entities**, etc. 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 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 method* 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 un delegate che, dato un elemento della sequenza, produce `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 secondario⁴.

Ad esempio, si possono 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
```

4.2.5 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 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) esclude dal risultato gli elementi che sono presenti nella seconda sequenza e rimuove i duplicati:

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

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

Intersect (intersezione) 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) 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 mostro soltanto il primo.

4.3.1 Select

Per ogni elemento della sequenza, produce 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 vuota.

```
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 dello stesso tipo degli elementi della sequenza.
- **First** e **Last** sollevano un'eccezione se la sequenza è vuota.

4.4.2 *Single / SingleOrDefault*

Restituiscono un singolo elemento della sequenza. È necessario che *uno e un solo elemento della sequenza 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 risultante contiene più di un elemento.

Entrambi esistono in due versioni, con e senza 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 un 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 l'intera 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<>`.

Di seguito, da un vettore di interi, vengono creati un secondo vettore e una lista, contenenti 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 appartengono al tipo specificato. L'operatore è utile soltanto con sequenze *polimorfiche*, le quali, cioè, contengono elementi di tipo diverso derivanti da un tipo comune.

Data gerarchia di classi: `Figura` → `TriangoloRettangolo`, `Rettangolo`, `Quadrato`, supponi 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
```

C'è una differenza: nel primo caso `list` è di tipo `IEnumerable<Quadrato>`; nel secondo caso di tipo `IEnumerable<Figura>`.

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"
```

5 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.

5.1.1 Esecuzione differita di una query

L'accesso agli elementi di una sequenza (`IEnumerable<>`) avviene mediante un oggetto chiamato **iteratore**, il quale rende disponibili gli 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. Considera il seguente codice:

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

Il risultato prodotto è a prima vista sconcertante, poiché l'esecuzione di `Where()` avviene prima che il vettore sia modificato. Ma ciò acquista un senso se si considera che `result` non memorizza affatto la sequenza dei numeri positivi di `dati`, ma soltanto un *iteratore* che consente di accedere agli elementi di `dati` maggiori di zero⁵. Se si modifica il vettore prima di utilizzare `result`, ecco che questo 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.

I seguenti operatori sono eseguiti immediatamente:

- Risultati riassuntivi: `Count()`, `Min()`, `Max()`, etc.
- Creazione di vettore o lista: `ToArray()`, `ToList()`.

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

6 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**.

6.1 Ottenere nuovi dati mediante oggetti anonimi

Considera 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.Nome); //->"Rossi, Valentino","Lorenzo, Jorge",..
```

La *query* produce una sequenza di stringhe; ma se invece volessimo 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.

6.1.1 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. L'uso di oggetti anonimi impone dei requisiti, tra i quali:

- 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 definito⁶

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\" }
```

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

6.1.2 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);
}
```

7 Esempio d'uso di LINQ

Segue un semplice esercizio che mostra LINQ in azione. Supponi di avere il seguente file:

```
#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; }
}
```

7.1 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 *lambda expression* nell'operatore *Where* filtra le righe vuote e i commenti⁷.

7.1.1 Uso di una lambda expression in Select

Nell'operatore *Select* viene chiamato il metodo `LineToFilm()`; ciò 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();
}
```

7.1.2 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.