آموزش جامع LINQ

آموزش جامع LINQ

نگارش 1.1

نویسنده : علی اقدم

بهار 1390

نکته قابل توجه

این کتاب در حال تکمیل شدن می باشد ، اگر در آن اشکال و یا اشکالاتی و یا اینکه نکته و مطلب جا افتاده ای برای افروده شدن وجود دارد لطفا آنها را از طریق آدرس های زیر به من اطلاع بدهید تا در ویرایش بعدی کتاب اعمال کنم.

www.aliaghdam.ir

info@aliaghdam.ir

info@aliaghdam.com

فهرست مطالب

1 مقدمه ای بر LINQ

امروزه با وجود زبان های شی گرا که قابلیت های زیادی را در اختیار توسعه دهندگان قرار می دهد، روش های مختلفی برای ارتباط با پایگاه داده های رابطه ای وجود دارد. با این وجود فقدان روشی مشخص و آسان برای اتصال به انواع پایگاه داده های رابطه ای و به صورت کلی به هر نوع منبع داده ای که به صورت شئ نیست، احساس می شود، البته باید بگویم می شد.

شاید شما بگویید که ADO.NET می تواند با استفاده از مفهوم DataSet به این آرمان دست یابد اما برای تحقق آن می بایست از یک شی DataAdapter استفاده کرد. یک شی DataAdapter چهار شی Command را در خود پیاده سازی می کند که این اشیاء برای انجام عملیات Select ، Delete ، Update و Insert بر روی پایگاه داده مورد استفاده قرار می گیرند ولی توجه داشته باشید که برای انجام این عمل شما می بایست این اشیاء را با عبارت SQL مناسب خود مقدار دهی کنید و که برای با انجام این عمل تا هنگام اجرای برنامه نمی توانیم از صحت عبارت SQL خود اطمینان کسب کنید. این بدان معناست که عبارات SQL در زبان های دات نت بیگانه هستند و عبارت SQL مقدار دهی شده برای دات نت و برنامه نویس در تاریکی است! و تا اجرا نشدن عبارت SQL از صحت عبارت هیچ اطلاعی نداریم.

LINQ چیست؟

در کنفرانس توسعه دهندگان حرفه ای مایکروسافت[1] در سال 2005، آقای هلسبرگ[2] یک تکنولوژی جدید که بتوان به وسیله آن با هر نوع منبع داده ای به یک روش یکسان اتصال برقرار کرد، به نام LINQ معرفی نمود.

LINQ مخفف عبارت Language-Integrated Query است توجه داشته باشید که آن را لینک (Link) تلفظ کنید.

LINQ یک راه حل یکسان برای اتصال برقرار کردن و بازیابی اطلاعات از هر شی که رابط IEnumerable را پیاده سازی کرده باشد فراهم میکند. بوسیله LINQ می توان با آرایه ها ومجموعه های[3] درون حافظه ، پایگاه داده های رابطه ای و حتی اسناد XML را به عنوان منبع داده در نظر گرفت و با آن کار کرد!.

بوسیله LINQ می توان اطلاعات را از هر منبع داده ای با گرامری مشابه و خوش شکل بازیابی کرد.گرامری که بسیار شبیه به نوشتار نحوی SQL است، توجه داشته باشید که هدف تیم سازنده LINQ، اضافه کردن یک راه جدید برای بازیابی دادها نیست، بلکه فراهم کردن یک مجموعه دستورات محلی و جامع برای بازیابی اطلاعات[4] که از هر نوع منبع داده ای پشتیبانی می کند.

LINQ یک سری مجموعه دستورات توانمند را ارائه می کند که بوسیله آنها می توان پرس و جوهای پیاده سازی کرد که از مواردی چون Join ها، توابع Aggregation، مرتب سازی، فیلتر و... پشتیبانی کند. این دستورات را language-level می نامند و دیگر نیازی به کامپایل برای دیدن نتیجه نیست! بله این مشکلی بود که در تکنولوژی ADO.NET با آن دست و پنجه نرم می کردیم، یعنی برای مشاهده نتیجه کوئری آن را اجرا می کردیم که مشکلاتی از قبیل خطایابی برنامه نویسی را دشوار می کرد و عرصه توسعه را بسیار طولانی تر.

شاید شما فکر کنیدکه LINQ ابزاری است که بتوان با آن کوئری های بر روی آرایه و مجموعه ها، پایگاه داده و یا XML پیاده سازی کرد ولی این تعریف درستی از LINQ نیست بلکه یک تکنولوژی است که بتوان Provider های را پیاده سازی کرد تا بوسیله آن با منابع داده ارتباط برقرار کرد به طور مثال Provider های مانند LINQ to SQL و یا LINQ to XML که توسط تیم توسعه .NET. پیاده سازی شده اند که به آن مدل Provider اطلاق می شود[5].

البته برای اینکه تیم توسعه دهنده لینک بتواند لینک را طراحی کند می بایست یک سری قابلیت ها را در زبان های دات نتی بوجود می آورد تا بتوان پرس و جو ها را در همه ی زبان های دات نتی به یک صورت تولید و استفاده کرد به

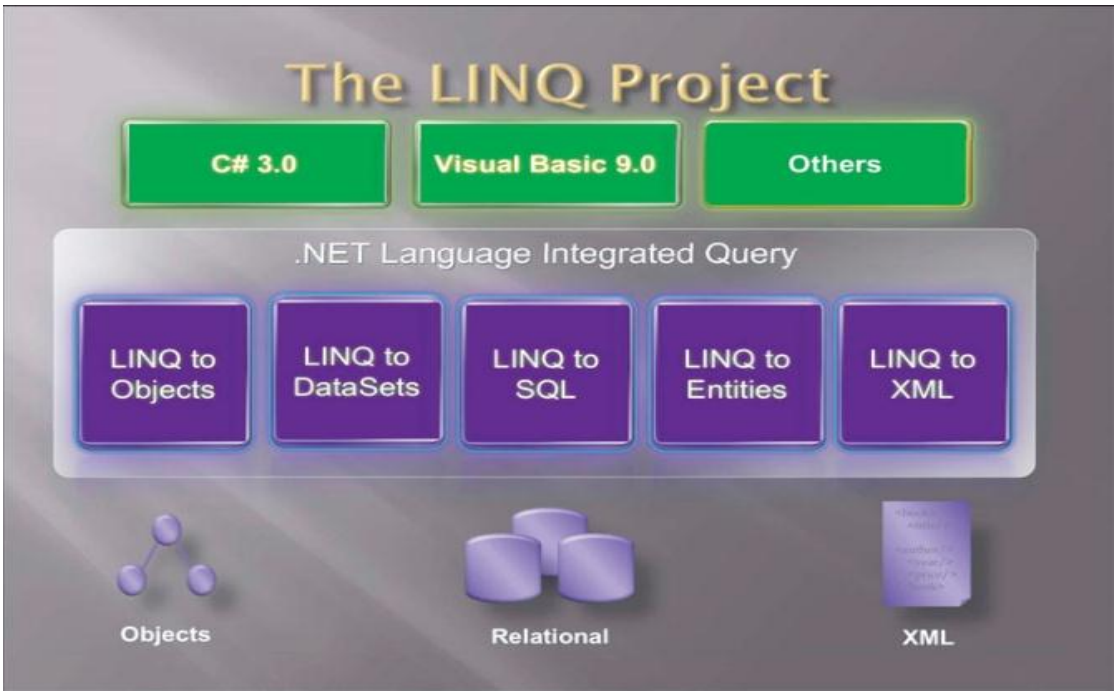
[1] PDC *Professional Developers Conference*
[2] *Anders Hejlsberg*
[3] Collection
[4] Query Expression
[5] LINQ Provider Model

همین دلیل با معرفی لینک تکنولوژی های جدیدی همانند توابع الحاقی که کارکرد اصلی لینک را تحقق می دهد و توابع بی نام ، عبارات لامبدا و چندی دیگر که در فصل بعدی به بررسی هریک از آنها خواهیم پرداخت.

توجه

هدف این کتاب آموزش پیاده سازی Provider نیست ولی برای آشنایی می توانید به مقاله آقای پدرام رضایی مراجعه کنید (http://tinyurl.com/LINQProviders).

نمودار زیر توسط اعضای تیم توسعه دات نت و LINQ تهیه شده است که به خوبی نحوه عملکرد و معماری LINQ را نشان می دهد.



بالا ترین سطح در تصویر نشان دهنده زبان های است که پشتیبانی کاملی از LINQ انجام می دهند.

سطح میانی این نمودار 5 بخش اصلی پروژه لینک را نشان می دهد :

LINQ to Objects: یک API است و متدهای که نشان دهنده عملگر های استاندارد پرس و جو می باشند را فراهم می کند. این متدها برای بازیابی اطلاعات از تمامی اشیائی که رابط IEnumerable را پیاده سازی کرده باشند ، استفاده می شود(آرایه و مجموعه عام و غیر عام درون حافظه).

LINQ to DataSet: این مدل،از عملیات پرس و جو بر روی DataTable ها و DataSet های موجود در ADO.NET پشتیبانی می کند.

LINQ to SQL: نامی است که برای APIمعین شده که به وسیله آن با می توان از بانک های رابطه ای مانند SQL Sever استفاده کرد.به طور خلاصه باعث تسهیل در استفاده از بانک اطلاعاتی را برای استفاده از بانک اطلاعاتی برای پرس و جو ،درج ،حذف و ویرایش می شود. برای استفاده ازLINQ to SQL می بایست یک ارجاع به اسمبلی System.Data.Linq.dll داشته باشید.

LINQ to Entities: یک راه حل ارائه شده توسط Microsoft ORM می باشد و توسعه یافته LINQ to SQL است.LINQ to Entities بین پایگاه داده ی فیزیکی و طراحی منطقی و تجاری قرار می گیرد و اجازه استفاده از آن را به صورت موجودیت[1] ها می دهد.

LINQ to XML: علاوه بر تعمیم عملگرهای استاندارد پرس و جو شامل یک سری خصوصیات ویژه XML برای ایجاد اسناد XML و همچنین پرس و جو بر روی آنها می باشد البته تیم توسعه لینک خصوصیت جدیدی برای استفاده از اسناد XML طراحی نکرده بلکه استاندارد XML DOM را پشتیبانی کرده است یعنی دیگر نیازی به یادگیریXPath ندارید .برای استفاده از LINQ to XML می بایست یک ارجاع به اسمبلی System.Xml.Linq.dll به پروژه اضافه کنید.

البته برنامه نویسان می توانند این Provider ها را توسعه دهند و یا اینکه برای مصارف خاص از Provider های را توسعه داده و از آنها استفاده کنند. در زیر لیستی از Provider های توسعه یافته به همراه لینک مربوطه، موجود است :

- LINQ Extender
- LINQ over C# project
- LINQ to Active Directory
- LINQ to Amazon
- LINQ to CRM
- LINQ to Excel
- LINQ to Expressions
- LINQ to Flickr
- LINQ to Geo
- LINQ to Google
- LINQ to Indexes

[1] Entities

- LINQ to JavaScript
- LINQ to JSON
- LINQ to LDAP
- LINQ to LLBLGen Pro
- LINQ to Lucene
- LINQ to Metaweb
- LINQ to MySQL
- LINQ to NCover
- LINQ to NHibernate
- LINQ to Opf3
- LINQ to Parallel (PLINQ (
- LINQ to RDF Files
- LINQ to Sharepoint
- LINQ to SimpleDB
- LINQ to Streams
- LINQ to WebQueries
- LINQ to WMI

مدل رابطه ای دارای مزایایی است که در نگاه اول متوجه آنها نمی شویم ولی با نگرش دقیق در آن به این مزایا پی می بریم.

1. برنامه نویس می تواند با تصور خود کوئری طراحی کند و آنها را به صورت بصری ویرایش کند.
2. با فراهم شدن گزینه قبل شرایطی بوجود می آید که که برنامه نویس می تواند کوئری خود را به حداکثر کارایی خود برساند چون کوئری را مشاهده می کند.
3. برنامه نویس می تواند Provider ی برای منبع داده خود طراحی کند تا دیگران با آن به منبع داده او دسترسی داشته باشند به طور مثال اگر شما یک web service داشته باشید و بخواهید کاربران تحت یک سیستم به آن دسترسی داشته باشند برای این منظور می توانید یک Provider طراحی کنید.

اسمبلی های مر کزی LINQ:

System.Core.dll: انواعی را تعریف می کند که LINQ API مرکزی را نمایش مـی دهنـد. ایـن یکی از اسمبلی های است که شما باید به آن ارجاع داشته باشید.

System.Data.Linq.dll: کارایی برای استفاده LINQ با پایگاه داده های رابطه ای را مهیا مـی کند. (LINQ to SQL)

System.Xml.Ling.dll: کارایی برای استفاده LINQ با اسناد XML را فراهم می کند.(LINQ to XML)

نوشتن اولین برنامه توسط LINQ

برای نوشتن اولین برنامه لینک خود یک برنامه کنسول ایجاد کنید و سپس کد زیر را در آن بنویسید :

```
using System;
using System.Linq;

string[] myWords = { "hello world", "hello LINQ",
"hello Aghdam" };

var items =
    from item in myWords
    where item.EndsWith("LINQ")
    select item;

foreach (var item in items)
    Console.WriteLine(item);
```

اگر کد بالا را اجرا کنید خروجی برابر با hello LINQ خواهد بود!

همانطور که مشاهده فرمودید عبارت کوئری بالا بسیار شبیه به کوئری های SQL است ،حالا می خواهیم قسمت های این کد را شرح دهیم و اگر با این عملگر ها آشنا نیستید نگران نشوید، در فصول بعدی با آنها آشنا خواهید شد.

در قسمت کوئری یک متغیر از نوع var به نام items تعریف شده است که برای خروجی کوئری مورد استفاده قرار می گیرد ، سپس items توسط یک عبارت پرس و جوی LINQ مقداردهی اولیه شده است. در قسمت اول از عبارت پرس و جو ،from برای تعیین نام منبع داده استفاده می شود . متغیر item در عبارت نشانگر یک عضو در مجموعه items است .

در قسمت where شرط های لازم برای بازیابی اطلاعات از منبع داده تبیین شده است که تابع EndWith از کلاس string فراخوانی شده که در صورتی که قسمت پایانی رشته با "LINQ" به پایان برسد ،این تابع مقدار true برمی گرداند و سرانجام در قسمت select ،قسمت ها / بخش ها / یا فیلد های که می خواهیم نمایش دهیم را انتخاب می کنیم.

2 خصوصیات جدید #C برای LINQ

همانطور که در فصل قبل گفتیم LINQ توانایی خود را بوسیله قابلیت های جدیدی به دست می آورد که برای استفاده از LINQ می بایست از خصوصیات جدید #C استفاده کنیم ،برای اینکه بتوانیم درک بهتری از عبارات LINQ داشته باشیم لازم است تا این خصوصیات جدید را که در C# 2.0 و C# 3.0 به زبـان #C اضـافه گردیده را فرا بگیریم.

این خصوصیات جدید عبارتند از:

- نوع های بی نام - Anonymous types
- مقدار دهنده اولیه به اشیاء - Object Initializers
- نوع بندی ضمنی - Type Inference
- توابع توسعه - Extension Methods
- عبارات لامبدا - Lambda Expressions
- عبارات پرس و جو – Query Expresions

نوع های بی نام – Anonymous types

شما به عنوان یک برنامه نویس OO ، مزایای تعریف کلاس ها برای نمایش جزئیات و کارایی یک موجودیت برنامه نویسی را می دانید .هر وقت شما نیاز به تعریف یک کلاس داشته باشید ، آن را تعریف و پیاده سازی می کنید ولی هنگامی که شما می خواهید می خواهید کلاسی را برای مدلسازی مجموعه ای از داده های کپسوله شده بدون تابع، رویداد و یا کارایی سفارشی دیگری ایجاد کنید و حتی این مدل سازی فقط درون پروژه شما مورد استفاده قرار گرفته باشد و دیگر قصد استفاده از آن را نداشته باشید ،چکار انجام می دهید؟ آیا کلاس جدیدی ایجاد می کنید ؟

اینجاست که نوع های بی نام به کمک شما می آیند و یک میان بر بسیار بزرگ را در جلوی پای شما قرار می دهد. وقتی می خواهید یک نوع بی نام ایجاد کنید این کار را با استفاده از کلمه کلیدی var انجام می دهید. نوع های بی نام این قابلیت را فراهم می کنند که انواع قوی نوع بندی شده را بدون نیاز به ایجاد کلاس ها، ایجاد کنید.

در LINQ از نوع های بی نام استفاه زیادی می شود چون پاسخ پرس و جوها ممکن است هر نوعی باشد و از آنها به عنوان منبع داده موقتی استفاده می شود.

به مثال زیر توجه کنید.

```
      static void Main(string[] args)
      {
         var person = new { ID = 1 ,
                            FName = "Ali",
                            LName = "Aghdam",
                            Job = "Student" };

       Console.WriteLine("The Person Name is {0} {1}.", person.FName,
person.LName);

       Console.ReadLine();
      }
```

در عبارت بالا بعد از کلمه کلیدی new هیچ گونه نوعی تعیین نشده که کامپایلر یک نوع بی نام ایجاد می کند. نوع های بی نام به برنامه نویس اجازه می دهد که از خروجی پرس و جو ها بدون نیاز به ساخت کلاس جدید، استفاده کنند.

مقدار دهنده اولیه به اشیاء – Object Initializers

امروزه در برنامه نویسی برای پیاده سازی موجودیت ها از کلاس ها استفاده می کنیم که در مهندسی نرم افزار به این روش Entity Types اطلاق می شود و به عنوان بسته های اطلاعاتی محسوب می شوند ولی در طی این امر مشکلاتی وجود دارد که یکی از آن ها پیاده سازی سازنده های مختلف است. با قابلیت جدید سی شارپ یعنی مقدار دهنده اولیه به اشیاء می توان تا حد بسیار زیادی از این پیچیدگی جلوگیری کرد و همچنین تا حد زیادی از بار کدنویسی کاست به طوری که می توان در هنگام ایجاد نمونه از کلاس به فیلد های عمومی و Property ها دسترسی پیدا کرده و به صورت سفارشی آنها را مقدار دهی نمود.

به طور مثال موجودیت Person را با پیاده سازی زیر در نظر بگیرید..

```
class Person
{
    public int ID
    { get; set; }

    public string FName
    { get; set; }

    public string LName
    { get; set; }
}
```

خوب با توجه به موجودیت بالا که سه شناسه را تعریف کرده ،سازنده به چه شکلی خواهد بود ؟ اگر از من بپرسید می گویم هیچ نیازی به استفاده از سازنده در مورد کلاس بالا نیست! به چه شکل ؟ به شکل زیر :

```
Person person = new Person
{
    ID = 1,
    FName = "Ali",
    Lname = "Aghdam"
};
```

و حتی به صورت زیر:

```
Person person = new Person
{
    ID = 1,
    Lname = "Aghdam"
};
```

نوع بندی ضمنی – Type Inference

کلمه کلیدی var (نوع بندی ضمنی)به کامپایلر اعلام می کند که خودش در مورد نوع متغیر تصمیم گیری می کند و هیچ موقع برنامه نویس نمی تواند به صورت صریح نوع آن را مشخص کند البته این تصمیم گیری برای نوع متغیر در زمان استفاده و مقداردهی شدن انجام می گیرد. نمونه زیر یک مثال ساده از var را نشان می دهد.

```
var i = 1;
i = "Hello LINQ"; // An error generated by this line
```

توضیح: در خط اول با مقدار دهی 1 به متغیر i کامپایلر نوع متغیر i را از نوع System.Int32 در نظر می گیرد، با این اوصاف منطقی است که از خط دوم خطا داشته باشد.

بیشتر بدانیم

در اصل var یک کلمه کلیدی C# نیست ولی می توان از این توکن بدون رخ دادن خطا به عنوان یک نوع داده استفاده کرد اما در هنگام کامپایل شدن کد، کامپایلر آن را از روی قراین به عنوان یک کلمه کلیدی می شناسد.

از این قابلیت می توان برای گاهش تکرار استفاده کرد مثلا کد زیر را در نظر بگیرید:

```
List<int> myNumbers = new List<int>(1, 2, 3);
```

در کد بالا ما نیازی به قید کردن صریح List<int> در تعریف متغیر myNumbers نداشتیم و می توانیم آن کد را به صورت زیر و کوتاه تر بنویسیم.

```
var myNumbers = new List<int>(1, 2, 3);
```

نکته

توجه داشته باشید این عمل را طوری انجام دهید تا در هنگام مراجعه دوباره به کد بتوانید نحوه و نوع متغیر را تشخیص دهیم.

در هنگام استفاده نوع بندی ضمنی محودیت های وجود دارد که در زیر به معرفی آنها می پردازم:

اولین و مهم ترین محدودیت این که نوع بندی ضمنی تنها به متغیر های درون یک تابع یـا خصوصـیت اعمـال مـی شود. بنابراین استفاده از کلمه نوع بندی ضمنی برای تعریف مقادیر بازگشتی، پارامترها و یا داده های اختصاصـی یـک نوع غیر مجاز است.

```
class varTestClass
{
    //Error : var cannot be used as field Data!
    private var myNumber = 1;

    //Error : var cannot be used as return value
    //or parameter type!
    public var myMethod(var x, var y)
    {
    }
}
```

متغیرهای نوع بندی ضمنی می بایست در زمان تعریف مقدار دهی شوند تا نوع آنها مشخص گردد و در صـورت رهـا شدن بدون مقدار دهی(با مقدار null) دستور مذکور با خطا روبه رو خواهد شد (همانند قوانین تعریف یک متغیر از نوع const).

```
//error: must assign value!
var myNumber;

//error: must assign value at exact time of declaration!
var myWord;
myWord= "Hello LINQ";
```

برای تکمیل مثال قبل توجه کنید که امکان تعریف یک متغیر محلی با نوع بندی ضـمنی nullable بـا اسـتفاده از توکن ? وجود ندارد.

```
//can't define nullable implicit variable,
//as implicit variables can never be initially assigned
//null to begin with!

var? myNumber = 1;
var? noValue = null;
```

توابع توسعه – Extension Methods

توابع توسعه امکان به دست آوردن کارایی جدید را بدون نیاز به اصلاح مستقیم نوع مورد توسعه و یا انواع کامپایل شده موجود (کلاس ها ، struct ها و پیاده سازی های اینترفیس) و همچنین انواع در حال کامپایل کنونی را بوجود می آورد (به دلیل در دسترس نبودن کد و یا اجازه ندادن کلاس برای ارث بری). این تکنیک برای تزریق کارایی جدید به انواعی که کد پایه آنها وجود ندارد ،بسیار سودمند خواهد بود و قابلیت اصلی پرس و جو لینک توسط توابع توسعه به دست آمده است.

در تعریف توابع توسعه اولین محدودیتی که با آن روبه رو می شویم این است که آنها باید درون یک کلاس static تعریف شوند، بنابراین هر تابع توسعه می بایست با کلمه کلیدی static تعریف شود دومین محدودیت این است که ما برای اعلام این تابع به عنوان تابع توسعه به کامپایلر می بایست با یک کلمه کلیدی this در اولین (و فقط اولین) پارامتر ورودی تابع استفاده کنیم.

نگاتی که باید در هنگام تعریف توابع توسعه باید به آنها توجه کنید:

- اگر یک تابع توسعه تعریف کرده اید ولی یک توسعه داخلی با الگویی مشابه (نه البته یکسان) وجود داشت، اولویت فراخوانی با توسعه داخلی است.
- خصوصیات، رویدادها و عملگرها قابل توسعه نیستند ولی مطرح شده اند و امید است در نسخه های بعدی C# این قابلیت ها نیز افزوده شوند.

بیشتر بدانیم

توابع توسعه ذاتا توابع ایستای معمولی هستند که می توانند در یک نمونه از نوع توسعه یافته مورد استفاده قرار گیرند که با توجه به قواعد نحوی توابع ایستا نمی توانند به اعضای (فیلد و یا توابع) دیگر نوع توسعه یافته ، دسترسی پیدا کنند که با توجه به این مسعله توسعه دادن با به ارث بردن به کلی تفاوت دارد و شما نمی توانید یک فیلد یک کلاس را توسط تابع توسعه خود مورد استفاده قرار دهید.

تعریف توابع توسعه

همانطور که اشاره شد در تعریف توابع توسعه اولین پارامتر ورودی تابع با کلمه کلیدی this شروع می شود و نام کلاس مورد نظر برای توسعه نیز قید گردد و تابع می بایست از نوع static باشد. مثال زیر نحوه تعریف یک تابع توسعه را نشان می دهد به قسمت های ضخیم دقت کنید.

```
static class MyExtensionMethodes
{
```

```
        ///
        /// Returns a converted null and space to an empty string.
        ///
        public static string  ConvertNullToEmptyString(this string strInput )
        {
          return ( String.IsNullOrWhiteSpace(strInput) ? string.Empty : strInput
    );
        }
      }
```

بررسی نحوه عملکرد مثال بالا به عهده خودتان!

نکته

توابع توسعه تمامی قابلیت های توابع ایستای معمول را دارا می باشند یعنی می توان آنها را بوسیله توابع ایستا و یا نمونه سازی شده فراخوانی نمود.

فراخوانی توابع توسعه در سطح نمونه ای

برای استفاده از توابع توسعه تنها کافی است که تابع توسعه در فضای نام جاری باشد و یا اینکه یک ارجاع به آن فضای نام در فضای نام جاری وجود داشته باشد.

به طور مثال از تابع توسعه مثال بالا در این مثال استفاده می کنیم.

```
string strTest = null;
strTest = strTest.ConvertNullToEmptyString();
```

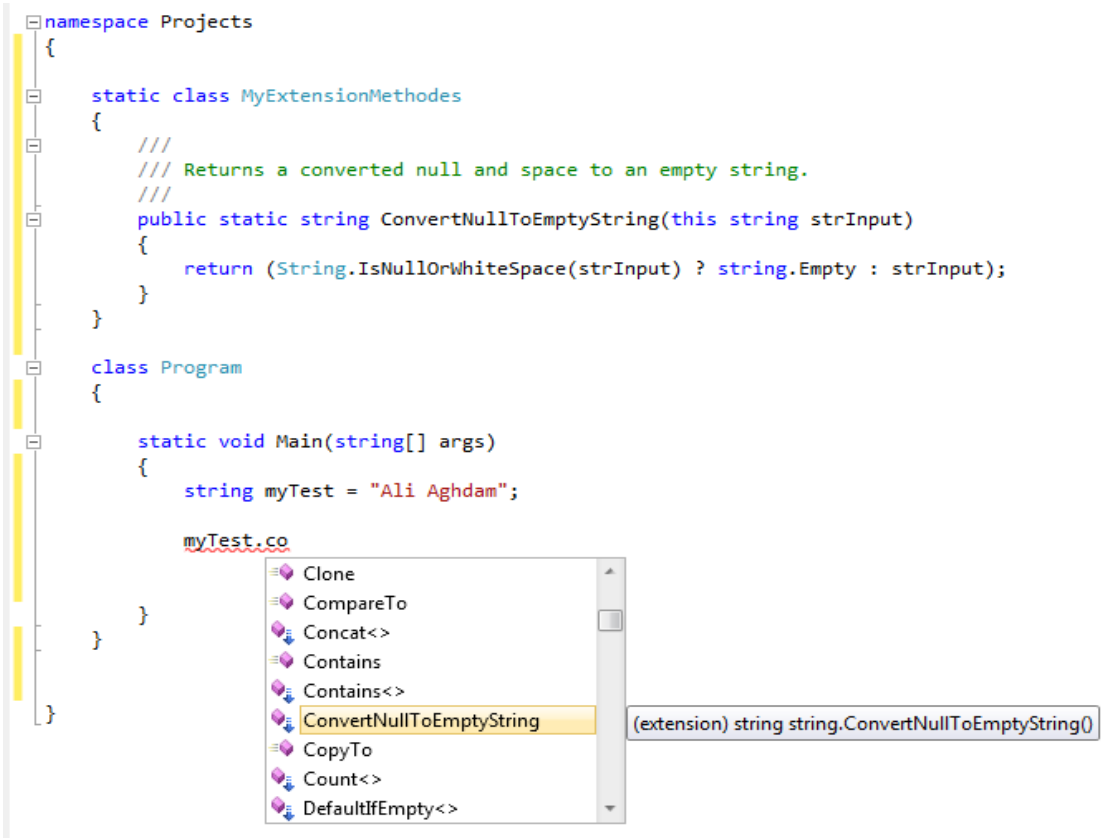
فراخوانی توابع توسعه در سطح ایستا

همانطور که قبلا اشاراتی شد می توان توابع توسعه را به صورت توابع ایستای معمولی مورد استفاده قرار داد که البته بعد از تبدیل شدن کد به کد IL کد قبلی با کد جدید که همان استفاده معمول از تابع ایستا است ، جایگزین می گردد.

```
string strTest = null;
strTest = MyExtensionMethodes.ConvertNullToEmptyString(strTest);
```

استفاده Intelisense از توابع توسعه

زمانی که توابع توسعه ای را ایجاد می کنید و قصد استفاده از آنها را دارید ،مکانیزم Intelisense ویژوال استودیو آنها را تشخیص می دهد و نمایش می دهد تا نیازی به به خاطر سپاری آنها را نداشته باشید. Intelisense توابع توسعه را به وسیله یک شکلک با یک فلش رو به پایین به نمایش می گذارد.



همانطور که مشاهده می کنید توابع توسعه دیگری نیز وجود دارند که اکثریت آنها نیز توابع توسعه LINQ هستند.

توسعه رابط ها بوسیله توابع توسعه

در ابتدای توضیحات بیان نمودم که امکان توسعه رابط[1] ها نیز وجود دارد ولی ماهیت انجام این عمل با توسعه یک کلاس تفاوت دارد. این عمل را به صورت قدم به قدم انجام می دهیم تا مراحل آن را به خوبی درک کنید.

برای شروع یک رابط جدید به نام ILocatable ایجاد کنید و سپس پیاده سازی زیر را برای آن تعریف کنید.

```
// Define a normal CLR interface in C#.
public interface ILocatable
{
   int Longitude { get; set; }
   int Latitude { get; set; }
}
```

[1] Interface

حال این رابط را به یک کلاس اعمال کنید

```
// Implementation of ILocatable.
public class Car : ILocatable
{
   public int Longitude { get; set; }
   public int Latitude { get; set; }
}
```

رابط ILocatable دارای دو متد است که در کلاس Car پیاده سازی شده است. بافرض اینکه ما به کد رابط ILocatable دسترسی نداریم ویا نمی خواهیم در آن تغییری بدهیم و می خواهیم آن را توسعه دهیم که برای این عمل را نمی توان به شکل معمول انجام دهیم .

برای اینکه یک رابط را توسعه دهیم می بایست پیاده سازی آن توابع را نیز مهیا کنید و این را به این صورت فرض کنید که کلاس های که این رابط را پیاده سازی می کنند شامل یک متد با این پیاده سازی هستند. این عمل را در مثال زیر مشاهده کنید.

```
public static class LocatableExtensions
{
   public static void MoveNorth(this ILocatable locatable, int degrees)
   {
       // ...
   }

   public static void MoveWest(this ILocatable locatable, int degrees)
   {
       // ..
   }
}
```

حال زمانی که از کلاسی که رابط ILocatable را پیاده سازی کرده نمونه سازی شود، آن کلاس می تواند به توابع توسعه داده شده دسترسی پیدا کند.

```
Car car = new Car();

car.MoveNorth(23);

car.MoveWest(23);
```

بیشتر بدانیم

برای استفاده از توابع توسعه بهتر است کتابخانه ای از آن در دسته بندی های مختلف تهیه کنید تا از آنها بتوانید در پروژه های مختلف به راحتی استفاده کنید همچنین یک پایگاه اینترنتی وجود دارد که در آن کاربران توابع توسعه خود را به اشتراک می گذارند که می تواند به عنوان یک پایگاه عظیم اطلاعاتی مورد استفاده قرار گیرد.

این پایگاه توابع توسعه را برای سه زبان C#، F# و ویژوال بیسیک و در دسته بندی های بسیار زیاد ارائه می کند.

آدرس این پایگاه : www.ExtensionMethod.net

عبارات لامبدا – Lambda Expressions

عبارات لامبدا توابع ناشناخته ای هستند که می توانند شامل عبارات و قطعات کد باشند که می توان از آنها برای ساخت درخت های عبارت[1] و Delegate ها استفاده نمود. عبارات لامبدا این امکان را فراهم می کنند که توابعی ایجاد نموده و آنها را به عنوان آرگومان به متدها ارسال کرد.

```
   ۱     ۲      ۳            ۴
public int myMethod(int x , int y)
{
 //... ۵
}
```

[1] Expression Tree

عبارات لامبدا یک تابع را عنوان خروجی برگردانده و توانایی تعریف توابع Inline را فراهم می کنند.توجه کنید که نوشتن عبارات Lambda هیچ پیچیدگی ندارد و خواهید فهمید که بسیار هم در LINQ و Delegate ها پر کاربرد هستند.

تعریف عبارات لامبدا

برای تعریف یک تابع معمول می بایست پنج قسمت را به صورت صریح تعریف کرد :

1. نوع دسترسی به تابع
2. نوع خروجی تابع
3. نام تابع
4. لیست پارامتر های ورودی
5. بدنه تابع

ولی برای تعریف عبارت لامبدا فقط دو مرحله از مراحل تعریف تابع را انجام می دهیم.

1. لیست پارامتر ها
2. بدنه تابع

قسمت اول عبارت لامبدا (عبارت داخل پرانتز) به عنوان آرگومان های تابع در نظر گرفته می شوند و اگر عبارت لامبدای مورد نظر شما فاقد آرگومان باشد می توانید از این قسمت صرف نظر کنید. قسمت دوم یعنی=> به کامپایلر اعلام می کند که این عبارت یک عبارت لامبدا است و قسمت سوم بدنه تابع را نشان می دهد که در این عبارت بدنه ساده است ولی اگر شما خواستید بدنه بیشتر را در عبارت خود داشته باشید می بایست دستورات درون brase قرار دهید و آنها را با سمی کولون جدا کنید.

مثال زیر نحوه تعریف یک عبارت لامبدا را نشان می دهد:

```
(int x) => x + 1;
```

عبارت بالا معادل عبارت زیر است :

```
int func(int x)
{
return x + 1;
}
```

بیشتر بدانیم

در هنگام کامپایل شدن عبارت لامبدا ،کامپایلر آن عبارت را به عنوان Delegate در نظر می گیرد و پیاده سازی آن را در کلاس جاری قرار می دهد و نوع دسترسی به تابع مورد نظر خصوصی در نظر گرفته می شود. به این ها توابع بی نام[1] اطلاق می گردد چون نام تابع توسط کامپایل انتخاب می گردد و هیچ قانون مشخصی برای آن وجود ندارد یعنی با چند بار کامپایل یک کد ، چندین نام مختلف به آن نسبت داده می شود.

نام این تابع با علامت " < " شروع می شود چون شما هیچ موقع نمی توانید ادعا کنید که تابعی نوشته اید که با این علامت را شروع می شود! پس کامپایلر از این راه آنها را تشخیص می دهد همچنین کامپایلر یک خصیصه System.Runtime.CompilerServices.CompilerGeneratedAttribute نیز به تابع اضافه می کند تا دچار اشتباه نشود.به عبارت زیر دقت کنید

```
internal sealed class AClass
{
    public static void CallbackWithoutNewingADelegateObject()
    {
        ThreadPool.QueueUserWorkItem(obj => Console.WriteLine(obj), 5);
    }
}
```

عبارت بالا بعد از کامپایل به کد IL ، به عبارات صخیم شده دقت کنید

```
internal sealed class AClass {

    // This private field is created to cache the
  delegate object.
    // Pro: CallbackWithoutNewingADelegateObject will
  not create
    // a new object each time it is called.
    // Con: The cached object never gets garbage
  collected
    [CompilerGenerated]
    private static WaitCallback
  <>9__CachedAnonymousMethodDelegate1;

    public static void
  CallbackWithoutNewingADelegateObject() {
        if (<>9__CachedAnonymousMethodDelegate1 ==
```

[1] anonymous function

```
          null) {
                // First time called, create the delegate
              object and cache it.
                <>9__CachedAnonymousMethodDelegate1 =
                new
              WaitCallback(<CallbackWithoutNewingADelegate
              Object>b__0);
            }

            ThreadPool.QueueUserWorkItem(<>9__CachedAnonymo
          usMethodDelegate1, 5);
        }

        [CompilerGenerated]
        private                  static                  void
      <CallbackWithoutNewingADelegateObject>b__0(Object
      obj) {
        Console.WriteLine(obj);
    }
  }
```

اگر دقت کرده باشید در استفاده های معمول عبارت لامبدا قصد دارد تا یک سطح برنامه نویسی را کم کنـد و خـود این مسئولیت را به دوش بکشد (البته کامپایلر).

عبارات پرس و جو – Query Expressions

در هنگام تعامل با پایگاه های داده در واقع ما از دو زبان برای این تعامل استفاده می کنیم نخستین زبان ،زبان برنامه نویسی ما است (مثلا C#) و دیگری زبانی که با پایگاه داده ارتباط برقرار می کنیم (مثلا SQL).برای اینکه بتوانیم با پایگاه داده ارتباط برقرار کنیم عبارت SQL را در قالب متن به سیستم میانجی پایگاه داده ارسال می کنیم که تا زمان اجرا نشدن کد نمی توانیم از صحت این عبارت مطلع شویم.

در C# 3.0 تکنولوژی LINQ ما را از وابسته بودن به عبارات SQL متنی رها ساخت و بوسیله عبارات پرس و جو قابلیت نوشتن عباراتی نزدیک به عبارات SQL با قابلیت Language-Level را فراهم می کند.

عبارات پرس و جو با جزء from شروع و با select و یا group به پایان می رسد.جزء from می تواند با جزء های from ، let و where ادامه داشته باشد. جزء from نقش سازنده ،let نقش محاسبه گر مقدار ، select و group نقش شکل دادن به نتایج و where نقش فیلتر را ایفا می کنند. جزء های دیگر نیز وجود دارند که در فصل بعدی به تفضیا آنها را شرح خواهیم داد.

قاعده نوشتار عبارت پرس و جو به صورت زیر است:

```
Query-expression:
     from-clause  query-body

from-clause:
     from typeopt identifier in expression join-clausesopt

join-clauses:
     join-clause
     join-clauses join-clause

join-clause:
     join typeopt identifier in expression on expression equals expression
     join typeopt identifier in expression on expression equals expression
       into identifier

query-body:
     from-let-where-clausesopt orderby-clauseopt select-or-group-clause query-
       continuationopt

from-let-where-clauses:
     from-let-where-clause
     from-let-where-clauses from-let-where-clause

from-let-where-clause:
```

```
      from-clause
      let-clause
      where-clause

let-clause:
      let identifier = expression

where-clause:
      where boolean-expression

orderby-clause:
      orderby orderings

orderings:
      ordering
      orderings , ordering


ordering:
      expression ordering-directionopt

ordering-direction:
      ascending
      descending

select-or-group-clause:
      select-clause
      group-clause

select-clause:
      select expression

group-clause:
      group expression by expression

query-continuation:
      into identifier join-clausesopt query-body
```

در واقع این عبارات در هنگام کامپایل به شکل متدی خود به وسیله توابع توسعه تبدیل می گردند و شما نیز می توانید به صورت مستقیم شکل متدی عبارت پرس و جو را بنویسید. عبارت پرس و جوی زیر را در نظر بگیرید

```
from Person p in Persons
where p.LName == "Aghdam"
select p
```

عبارت بالا ابتدا به عبارت زیر تبدیل می گردد:

```
from Person p in Persons.Cast<Person>()
where p.Lname == "Aghdam"
select p
```

عبارت بالا ابتدا به عبارت زیر تبدیل می گردد:

```
Persons.Cast<Person>().Where( p => p.LName == "Aghdam")
```

مقدمه ای بر نوشتار LINQ

3

بوسیله LINQ می توان بر روی مجموعه ای پرس و جو انجام داد و یا عناصر مجموعه ای را مدیریت کرد. خصوصیت کلیدی LINQ ادغام شدن آن با مجموعه ی زیادی از زبان ها است یعنی می توان از نوشتار LINQ در زبان های مختلف که آن را پشتیبانی می کنید به یک صورت استفاده نمود که این ویژگی برای برنامه نویسانی که از چندین زبان استفاده می کنند بسیار مهم است.

همانطور که در فصل قبلی بیان شد، LINQ زیر ساخت های برای انواع منابع داده را در خود دارد، شامل LINQ to Object ، LINQ to XML ، LINQ to Entities و... . همه ی این زیر ساخت ها توسط یک سری توابع توسعه ایجاد میشوند که یک سری کلمات کلیدی برای نوشتن پرس و جو ها ایجاد می کنند، در این فصل به شرح آنها می پردازیم.

پرس و جو های LINQ

LINQ عملکرد خود را از یک سری عملگر های پرس و جو که بوسیله توابع توسعه پیاده سازی شده انـد، بـه دسـت می آورد. این عملگر های پرس و جو بر روی اشیائی کـه رابـط هـای IEnumerable و IQueryable را پیـاده سازی کرده باشند، اعمال می گردند.

نوشتار پرس و جوها

برای تشریح نوشتار LINQ از یک مثال استفاده می کنیم تا بتوانیم بهتر به واقعیـت Integrity بـودن LINQ پـی ببریم. فرض کنید می خواهید به وسیله LINQ to Object پرس و جوی را بر روی نوع Developer انجام دهید و نمونه های از آن که زبان C# به عنوان زبان اصلی آنهاست را انتخاب نمایید. به کد زیر که به زبان C# نوشته شـده دقت کنید:

```
using System;
using System.Linq;
using System.Collections.Generic;

public class Developer {
    public string Name;
    public string Language;
    public int Age;
}


class App {

    static void Main() {

        Developer[] developers = new Developer[] {
            new Developer {Name = "Ali", Language = "C#"},
            new Developer {Name = "Vahid", Language = "C#"},
            new Developer {Name = "Yaser", Language = "VB.NET"}};

        var developersUsingCSharp =
            from d in developers
            where d.Language == "C#"
            select d.Name;

        foreach (var item in developersUsingCSharp) {
            Console.WriteLine(item);

        }
    }
}
```

وقتی کد بالا را اجرا می کنید در خروجی نام های Ali و Vahid نمایش داده می شوند.

می توان در زبان Visual Basic نیز پرس و جوی بالا را به همان صورت استفاده نمود (قسمت ضخیم مثال قبل)، به کد زیر در زبان Visual Basic دقت کنید:

```
Imports System
Imports System.Linq
Imports System.Collections.Generic

Public Class Developer
    Public Name As String
    Public Language As String
    Public Age As Integer
End Class

Module App
    Sub Main()
        Dim developers As Developer() = New Developer() {
        New Developer With {.Name = "Paolo", .Language = "C#"},
        New Developer With {.Name = "Marco", .Language = "C#"},
        New Developer With {.Name = "Frank", .Language = "VB.NET"}}

        Dim developersUsingCSharp = From d In developers
                                    Where d.Language = "C#"
                                    Select d.Name

        For Each item In developersUsingCSharp
            Console.WriteLine(item)
        Next
    End Sub
End Module
```

هر دو پرس و جوی استفاده شده در مثال های قبل Query Expression نام دارند که در فصل قبلی به آنها پـر داخیم. این پرس و جوها بسیار به عبارات SQL شبیه هستند و تنها تفاوت اندکی در شکل با هم دارند. به عنوان مثـال عبارت پرس و جوی قبلی دارای یک قسمت برای انتخاب بود(عملیات پرتو):

```
    select d.Name;
```

آن قسمت انتخابی بر روی بر روی مجموعه ای از آیتم ها اعمل می شد:

```
    from d in developers
```

همچنین عملیات انتخابی بوسیله یک شرط خاص انجام می شد:

```
    where d.Language == "C#"
```

بیشتر بدانیم

اگر در فهم عبارات پرس و جوی LINQ مشکل دارید اصلا نگران نشوید چون اصولا LINQ را می بایست به صورت عملی یاد گرفت.

برای اینکه بتوانید عبارات SQL خود را به راحتی به عبارات پرس و جوی LINQ تبدیل کنید می توانید به پیوست شماره دو مراجعه کنید که در آن نحوه تبدیل همه ی دستورات SQL به LINQ را توضیح شده است.

4 عملگرهای استاندارد پرس و جو

عملگرهای استاندارد پرس وجو[1] API ای است که امکان انجام پرس و جو را بر روی آرایه ها و مجموعه ها و انواع منابع داده را فراهم می کند.

عملگرهای استاندارد پرس وجو در واقع توابعی هستندکه درکلاس های ایستای موجود در فضای نام *System.Linq* و به عنوان متدهای توسعه با مدل های مختلفی تعریف شده اند. این توابع در اسمبلی *System.Core.dll* قرار دارند ، و از آنها در هر زبان تحت دات نتی که Generic ها و ویژگی های جدید دات نت را پشتیبانی کند، می توان استفاده نمود.

در این در فصل ما LINQ to Object را بررسی می کنیم ولی من مناسب دیدم که این فصل را با LINQ to Object نام گزاری نکنم!

از عملگرهای استاندارد پرس و جو می توان بر روی هر شئ که واسط *IEnumerable<T>* و *IQueryable<T>* را پیاده سازی کند، استفاده کرد توجه داشته باشید که به جای T می توان هر نوع دیگری قرار داد.

[1] Standard Query Operators

نکته بسیار مهم

از این قسمت به بعد برای توضیح مثال های در مورد عملگرهـای اسـتاندارد پـرس و جـو از سـه کـلاس Order، Customer و Product استفاده خواهیم کرد که پیاده سازی آنها در پیوست شماره یک موجود است.

انواع عملگرهای استاندارد پرس وجو

در زیر لیستی از انواع این عملگرها آمده است که ادامه هر کدام را به طور کامل توضیح خواهم داد.

عملیات	عملگر	توضیحات
Aggregate	Aggregate	یک تابع را بر روی یک مجموعه اعمال می کند.
	Average	میانگین عناصر یک مجموعه را محاسبه می کند.
	Sum	مجموع عناصر یک مجموعه را محاسبه می کند.
	Count	تعداد عناصر یک مجموعه را با یک نوع int بر می گرداند.
	Long Count	تعداد عناصر یک مجموعه را با یک نوع Long بر می گرداند.
	Min	کمترین مقدار در یک مجموعه از مقادیر عددی را بر می گرداند.
	Max	بیشترین مقدار در یـک مجموعـه از مقـادیر عـددی را بـر مـی گرداند.
Concatenation	Concat	عناصر دو مجموعه را با هم ادغام می کند.
Conversion	Cast	عناصر یک مجموعه را به یک نوع معین شده ، تبدیل می کند.
	ToArray	از مجموعه معین شده یک آرایه می سازد.
	ToDictionary	از روی مجموعـه مشـخص شـده یـک شـئ از نـوع کـلاس Dictionary<K,E> ایجاد می کند.
	ToList	از روی مجموعـه مشـخص شـده یـک شـئ از نـوع کـلاس List<T> ایجاد می کند.
	ToLookup	از روی مجموعـه مشـخص شـده یـک شـئ از نـوع کـلاس LookUp<K,T> ایجاد می کند.
Element	DefaultIfEmpty	اگر مجموعه مشخص شده تهی باشد ، یک مقدار پـیش فـرض به خروجی ارسال می کند.

عملیات	عملگر	توضیحات
	ElementAt	عنصری از مجموعه را بر اساس ایندکس معین شده بر می گرداند.
	ElementAtOrDefault	عنصری از مجموعه را بر اساس ایندکس معین شده بر می گرداند و در صورتی که ایندکس خارج از محدوده باشد یک مقدار پیش فرض را بر می گرداند.
	First	اولین عنصر یک مجموعه را بر می گرداند.
	FirstOrDefault	اولین عنصر یک مجموعه را بر می گرداند و اگر اولین عنصر در دسترس نباشد یک مقدار پیش فرض بر می گرداند.
	Last	آخرین عنصر یک مجموعه را بر می گرداند.
	LastOrDefault	آخرین عنصر یک مجموعه را بر می گرداند و اگر آخرین عنصر در دسترس نباشد یک مقدار پیش فرض را بر می گرداند.
	Single	یک عنصر از مجموعه که با شرط مطابقت داشته را به عنوان خروجی بر می گرداند.
	SingleOrDefault	یک عنصر از مجموعه که با شرط مطابقت داشته را به عنوان خروجی بر می گرداند و اگر عنصری پیدا نشود یک مقدار پیش فرض بر می گرداند.
Equality	SequenceEqual	دو مجموعه را برای یکسان بودن بررسی می کند و مقدار Boolean برمی گرداند
Generation	Empty	از نوع تعیین شده یک مجموعه تهی ساخته و بر می گرداند.
	Range	یک مجموعه عددی را شامل عناصری از پارامتر اول ورودی تا پارامتر دوم ورودی را ساخته و بر می گرداند.
	Repeat	یک مجموعه را که شامل تکرارهای از عنصر تعیین شده است و بر می گرداند.
Grouping	GroupBy	عناصر یک مجموعه را گروه بندی می کند.
Join	GroupJoin	دو مجموعه را بر اساس کلیدهای همسان گروه بندی می کند.
	Join	اتصال داخلی دو مجموعه را بر اساس کلیدهای همسان انجام می دهد.

عملیات	عملگر	توضیحات
Ordering	OrderBy	عناصر یک مجموعه را بر اساس یک و یا چند کلید معین به صورت صعودی مرتب می کند.
	OrderByDescending	عناصر یک مجموعه را بر اساس یک و یا چند کلید معین به صورت نزولی مرتب می کند.
	ThenBy	عناصر یک مجموعه مرتب را بر اساس یک و یا چند کلید به صورت صعودی مرتب می کند.
	ThenByDescending	عناصر یک مجموعه مرتب را بر اساس یک و یا چند کلید به صورت نزولی مرتب می کند.
	Reverse	تمامی عناصر یک مجموعه را از نظر چیدمان برعکس می کند.
Partitioning	Skip	به تعداد مشخص شده ای از عناصر مجموعه صرف نظر کرده و بقیه را بر می گرداند.
	SkipWhile	به وسیله یک عبارت شرطی ، از تعداد مشخصی از عناصر مجموعه صرف نظر کرده و بقیه را بر می گرداند.
	Take	تعداد مشخصی از عناصر مجموعه را به عنوان خروجی برگردانده و از باقیمانده عناصر صرف نظر می کند.
	TakeWhile	تعداد مشخصی از عناصر مجموعه را بوسیله یک شرط جدا و به عنوان خروجی برگردانده و از باقیمانده عناصر صرف نظر می کند.
Projection	Select	عناصری از مجموعه که می بایست برگردانده شوند را تعیین می کند.
	SelectMany	یک عملیات پرتو یک به چند از روی عناصر مجموعه انجام می دهد.
Quantifier	All	کلیه عناصر مجموعه را بر اساس یک شرط مشخص چک می کند.
	Any	بررسی می کند که آیا عنصری از مجموعه با شرط داده شده مطابقت می کند یا نه.
	Contains	وجود عنصر مورد نظر را در مجموعه بررسی می کند.
Restriction	Where	عناصر یک مجموعه را بر اساس مشخص شده فیلتر می کند.
	OfType	عناصر یک مجموعه را بر اساس یک نوع معین شده ، فیلتر می کند.

عملیات	عملگر	توضیحات
Set	Distinct	عناصر متمایز در یک مجموعه را بر می گرداند.
	Except	یک مجموعه جدید را از عناصر متفاوت دو مجموعه مجزا ایجاد می کند.
	Intersect	یک مجموعه جدید را از عناصر مشابه دو مجموعه مجزا ایجاد میکند.
	Union	از حاصل اجتماع دو مجموعه یک مجموعه جدید ایجاد می کند.
	Zip	عناصر یک مجموعه را با عناصر متناظر در مجموعه دیگر ادغام می کند
	AsEnumerable	این عملگر یک مجموعه را به یک مجموعه از نوع IEnumerable<TSource> تبدیل می کند.

عملگر شرطی - Restriction Operator

عملگر شرطی نتیجه پرس و جو ها را بر اساس یک شرط تعیین شده فیلتر می کند که از پرکاربرد ترین عملگرهای پرس و جو در LINQ است. در ادامه به بررسی این نوع عملگر خواهیم پرداخت.

عملگر Where

این عملگر نتیجه پرس وجو را بر اساس آرگومان ورودی (به عنوان شرط) محدود می کند. فرم کلی نوشتار این عملگر به دو صورت زیر است :

```
public static IEnumerable<TSource> Where<TSource>(
    this IEnumerable<TSource> source,
    Func<TSource, Boolean> predicate);

public static IEnumerable<TSource> Where<TSource>(
    this IEnumerable<TSource> source,
    Func<TSource, Int32, Boolean> predicate);
```

تفاوت این دو فرم از عملگر Where تنها در پارامتر دوم آنها است این پارامتر همان شرطی است که هر عنصر در یک مجموعه با آن مقایسه می گردد. در فرم دوم عملگر Where یک پارامتر از نوع int وجود دارد که نشان دهنده اندیس هر عضو در مجموعه است که صفر شروع می شود.

مثال 1: در پرس و جوی زیر اجناسی که قیمت آنها بیشتر از 10 باشد به عنوان خروجی برگردانده می شود.

```
List<Product> products = new List<Product>
        {   new Product() { Name = "product 1", UnitPrice = 10 },
            new Product() { Name = "product 2", UnitPrice = 8  },
            new Product() { Name = "product 3", UnitPrice = 12 }
        };

IEnumerable<Product> retProducts = from p in products
                                   where p.UnitPrice > 10
                                   select p;
foreach (var item in retProducts)
    Console.WriteLine(item.Name);
```

وقتی پرس وجوی قبل کامپایل می گردد به معادل توابع الحاقی تبدیل می شود و در واقع توابع الحاقی هستند ولی برای اینکه بتوان از فرم پرس وجوی شبیه به نوشتار SQL استفاده کرد به این صورت نوشته می شوند که می توان از هر دو فرم برای نوشتن پرس وجوها استفاده کرد. به عنوان مثال عبارت پرس و جو بالا در هنگام کامپایل به عبارت زیر تبدیل می شوند ، به نحوه استفاده از توابع توسعه و عبارات لامبدا توجه کنید .

```
IEnumerable<Product> retProducts = products.Where(p => p.UnitPrice > 10);
```

مثال 2: در برنامه زیر از نوع دوم عملگر استفاده شده است. در این برنامه خریدارانی که کد ایندکس آنها با کد IDشان برابر باشد ، برگردانده می شوند.

```
List<Customer> customers = new List<Customer>()
  {
      new Customer() {Name ="Ali"   , Family = "Aghdam" , CustomerID =0 },
      new Customer() {Name ="Ali"   , Family = "Nasiri" , CustomerID =2 },
      new Customer() {Name ="Arash" , Family = "Novin"  , CustomerID =3 }
  };

var query = customers.Where( ( p , index ) => p.CustomerID == index );

foreach (var item in query )
     Console.WriteLine(item.Name);
```

نکته بسیار مهم

اگر هر یک از آرگومان های عملگر Where با null مقدار دهی شوند یک استثناء ArgumentNullExceptio تولید خواهد شد.

عملگر OfType

عملگر OfType اعضاء یک مجموعه را برحسب یک نوع مشخص فیلتر می کند و فقط عناصری که از آن نوع باشند در نتیجه پرس و جو قرار می گیرند. فرم این عملگر به صورت زیر است :

```
public static IEnumerable<T> OfType<T>(this IEnumerable source){
     foreach (object item in source)
       if (item is T)
         yield return (T)item;
}
```

همانطور که از فرم این عملگر معلوم است اعضای source تک تک بررسی می شوند و در صورتی که با نوع T مطابقت داشته باشند برگردانده می شوند.

مثال: در برنامه زیر می خواهیم از بین انواع مختلف درون لیست فقط عناصری که از نوع Customer هستند را انتخاب کنیم.

```
//using System.Collections

ArrayList complexList = new ArrayList();

complexList.Add("Test String 1");
complexList.Add(new DateTime(2011,1,1));
complexList.Add(10);

complexList.Add(new Customer() {Name = "Ali" , Family ="Aghdam"});


var query = complexList.OfType<Customer>();


foreach (var item in query)
       Console.WriteLine( item.Name + "  " + item.Family );
```

مثال بالا به صورت کاملا دقیق نحوه کارکرد عملگر OfType را نمایش می دهد.

نکته بسیار مهم

اگر هر یک از آرگومان های عملگر OfType با null مقدار دهی شوند یک استثناء *ArgumentNullExceptio* تولید خواهد شد.

عملگرهای پرتو – Projection Oprators

از این عملگرها برای تغییر شکل دادن اعضاء مجموعه و انتقال آن (آنها) به مجموعه دیگر استفاده می شود البته می توان اعضاء مجموعه اول را بدون تغییر در مجموعه دوم قرار داد. در ادامه این عملگرها را بررسی می کنیم.

عملگر Select

این عملگر از روی آرگومان های ورودی ، شئ جدید و قابل شمارشی ایجاد کرده و آن را بر می گرداند. فرم کلی این عملگر به صورت زیر است :

```
public static IEnumerable<S> Select<T, S>(this IEnumerable<T> source,
                                          Func<T, S> selector);

public static IEnumerable<S> Select<T, S>(this IEnumerable<T> source,
                                          Func<T, int, S> selector);
```

این عملگر همانند ماده Select در SQL است. تفاوت دو فرم این عملگر در پارامتر دوم آنها است که نوع دوم یک اندیس (شروع از صفر) که نشان دهنده محل هر عنصر در مجموعه است را دریافت می کند.

مثال 1 : در برنامه زیر عبارت پرس و جو همه عناصر مجموعه را بر می گرداند (معادل SELECT * در SQL)

```
List<Customer> customers = new List<Customer>()
    {
        new Customer() {Name ="Ali"   , Family = "Aghdam" , CustomerID =0 },
        new Customer() {Name ="Ali"   , Family = "Nasiri" , CustomerID =2 },
        new Customer() {Name ="Arash" , Family = "Novin"  , CustomerID =3 }
    };

var query = from c in customers
            select c;

foreach (var item in query)
    Console.WriteLine(item.Name + "  " + item.Family);
```

مثال 2 : در برنامه زیر عبارت پرس و جو همه عناصر مجموعه به همراه اندیس عناصر در مجموعه برگردانده می شوند. (فرم دوم عملگر Select)

```
List<Customer> customers = new List<Customer>()
    {
        new Customer() {Name ="Ali"   , Family = "Aghdam" , CustomerID =0 },
        new Customer() {Name ="Ali"   , Family = "Nasiri" , CustomerID =2 },
```

```
            new Customer() {Name ="Arash" , Family = "Novin"  , CustomerID =3 }
        };

    var query = customers.Select(
                        (p,index) => new{position=index,p.Name,   p.Family });

    foreach (var item in query)
        Console.WriteLine(item.Name +" "+
                          item.Family + ",Position= " +
                          item.position );
```

عملگر SelectMany

عملکرد این عملگر همانند عملگر Select است با این تفاوت که می توان از نتیجه پرس و جوی قبلی استفاده کـرد که در واقع همانند عملکرد ماده join در SQL است. این عملگر دارای دو فرم کلی به صورت زیر است:

```
    public static IEnumerable<S> SelectMany<T, S>(this IEnumerable<T> source,
              Func<T, IEnumerable<S>> selector);

    public static IEnumerable<S> SelectMany<T, S>(this IEnumerable<T> source,
              Func<T, int, IEnumerable<S>> selector);
```

همانطور که در بالا گفته شد این عملگر این قابلیت را فراهم میکند که از نتیجه پرس و جوهای قبلی استفاده کرد که این امکان را بوسیله برگرداندن نتیجه پرس و جو از نوع *IEnumerable<S>* بوسیله پارامتر دوم یعنی Selector بوجود می آید.

تفاوت این دو فرم در پارامتر دوم است که فرم اول یک نوع *IEnumerable<S>* را بر می گرداند و فرم دوم برای برگرداندن نتیجه نیاز به یک اندیس برای مشخص شدن محل عنصر در مجموعه است، نیاز دارد.

مثال: در برنامه زیر سفارش های مشتری به نام Aghdam که بعد از سال 2010 انجام گرفته، برگردانده می شود.

```
    List<Customer> customers = new List<Customer>()
        {
            new Customer() {Name ="Ali"   , Family = "Aghdam" , CustomerID =0 },
            new Customer() {Name ="Ali"   , Family = "Nasiri" , CustomerID =2 },
            new Customer() {Name ="Arash" , Family = "Novin"  , CustomerID =3 }
        };

    List<Order> order1 = new  List<Order>() {
          new Order(){ OrderID = 1 , OrderDate = new DateTime(2010,1,1)},
```

```
            new Order(){ OrderID = 2 , OrderDate = new DateTime(2011,1,1)}
            };



customers[0].Orders = order1;

var query =customers.
                Where(c => c.Family == "Aghdam").
                SelectMany(c =>
                    c.Orders.
                    Where(o => o.OrderDate.Year > 2010).
                        Select(o => new { c.Family , o.OrderID })
                );

foreach (var item in query)
    Console.WriteLine(item.Family + "  " + item.OrderID );
```

معدل عبارت پرس و جوی بالا بوسیله عبارات پرس و جو در C# 3.0 به صورت زیر است:

```
var query = from c in customers
                        where c.Family == "Aghdam"
                        from o in c.Orders
                        where o.OrderDate.Year > 2010
                        select new { c.Name, o.OrderID };
```

نکته بسیار مهم

اگر هر یک از آرگومان های عملگر SelectMany با null مقدار دهی شوند یک استثناء *ArgumentNullExceptio* تولید خواهد شد.

عملگرهای اتصال Join Operators

این نوع عملگرها برای متحد کردن چند مجموعه عناصر که دارای اشتراکاتی هستند، استفاده مـی شـود. عملگرهـای اتصال در LINQ دقیقا همانند ماده های اتصال در SQL عمل می کنند. هر مجموعه عنصـر و یـا منبـع داده ویژگـی های کلیدی را دارا می باشد که بوسیله آنها می توان داده ها را مقایسه و جمع آوری نمود.

عملگر Join

این عملگر همانند INNER Join در پایگاه داده های رابطـه ای عمـل مـی کنـد یعنـی دو مجموعـه را بـر اسـاس کلیدهای که در هر دو مرتبط هستند و به عنوان آرگومان به آن ارسال می گردند، ترکیب می کند. این عملگر بـه فـرم زیر است(بدون سربارگذاری):

```
public static IEnumerable<V> Join<T, U, K, V>(
                                        this IEnumerable<T> outer,
                                        IEnumerable<U> inner,
                                        Func<T, K> outerKeySelector,
                                        Func<U, K> innerKeySelector,
                                        Func<T, U, V> resultSelector);
```

پارامتر Outer نشان دهنده نوع منبع داده خارجی و پارامتر inner نشان دهنده نوع منبع داده داخلی است .

پارامترهای outerKeySelector و innerKeySelector تعیین می کنند که داده ها چطور از منـابع inner و outer استخراج گردند. نوع دوم هردو آنها از نوع K می باشند که تعادل میان این دو، شرط Join را پدید می آورد. تابع resultSelector که به عنوان آخرین پارامتر تعیین شده است برای جفت عناصر داخلی و خارجی (تطابق داده شده) بررسی شده و شئ نتیجه برگردانده می شود.

عملگر Join ترتیب عناصر خارجی را حفظ میکند و همچنین برای هر عنصر خارجی، ترتیب عناصر تطبیق داده شـده داخلی را نیز حفظ میکند.

در پایگاه داده های رابطه ای عملگرهـای Join دیگـری هماننـد left outer joins وجـود دارد ولـی ایـن نـوع اتصالات به صورت صریح در LINQ پیاده سازی نشده ولی در زیر مجموعه قابلیت های عملگـر GroupJoin قـرار دارند.

مثال : در برنامه زیر اشیاء Customer و Order با توجه به مقدار CustomerID به هم دیگر متصل می شوند و در خروجی شئ داریم که شامل اطلاعاتی ترکیب شده از این دو شئ است.

```
    List<Customer> customers = new List<Customer>()
        {
            new Customer() {Name ="Ali"   , Family = "Aghdam" , CustomerID =0 },
            new Customer() {Name ="Ali"   , Family = "Nasiri" , CustomerID =1 },
            new Customer() {Name ="Arash" , Family = "Novin"  , CustomerID =2 }
        };

List<Order> orders = new  List<Order>() {
   new Order(){CustomerID = 0, OrderID = 1,OrderDate = new DateTime(2010,1,1)} ,
   new Order(){CustomerID = 1, OrderID = 2,OrderDate = new DateTime(2011,1,1)}
};

    var query =
         from c in customers
         join o in orders on c.CustomerID equals o.CustomerID
         select new {FullName =  c.Name + " " +
                         c.Family ,
                         c.CustomerID ,
                         o.OrderDate ,
                         TotalOrder = o.Total
                    };


    foreach (var item in query)
        Console.WriteLine(item.FullName +
                          " ,ID= " + item.CustomerID +
                          " ,Order Date= " + item.OrderDate +
                          " ,Total Order="+ item.TotalOrder );
```

معادل عبارت پرس و جوی بالا بوسیله توابع توسعه در C# 3.0 به صورت زیر است:

```
    var query =customers.Join(
                          orders,
                          c => c.CustomerID,
                          o => o.CustomerID,
                          (c, o) => new { FullName = c.Name + "  " + c.Family ,
                                          o.OrderDate,
                                          TotalOrder = o.Total,
                                          c.CustomerID }
                            );
```

عملگر GroupJoin

این عملگر برای انواع خاصی از Join ها مورد استفاده قرار می گیرد، همانند left outer joins . این عملگر دارای دو سربار گذاری به صورت زیر است:

```
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);
```

این عملگر همانند عملگر Join عمل می کند ولی با این تفاوت که نتیجه عملیات join را در قالب یک مجموعه جدید قرار می دهد.

مثال .

```
List<Customer> customers = new List<Customer>()
    {
        new Customer() {Name ="Ali"   , Family = "Aghdam" , CustomerID =0 },
        new Customer() {Name ="Ali"   , Family = "Nasiri" , CustomerID =1 },
        new Customer() {Name ="Arash" , Family = "Novin"  , CustomerID =2 },
        new Customer() {Name ="Arash" , Family = "Novin"  , CustomerID =3 }
    };
List<Order> orders = new  List<Order>() {
                          new Order(){CustomerID = 0,OrderID =0},
                          new Order(){CustomerID = 1,OrderID =1},
                          new Order(){CustomerID = 2,OrderID =2},
                          new Order(){CustomerID = 1,OrderID =3},
                          new Order(){CustomerID = 0,OrderID =4},
     };

var query = from c in customers
            join o in orders on c.CustomerID equals o.CustomerID into q1
            select new { CustomerName = c.Family, orders = q1 };

foreach (var item in query)
{
    Console.WriteLine(item.CustomerName + ", Orders = ");
    foreach (var order in item.orders)
        Console.WriteLine("\t order ID={0}", order.OrderID);
}
```

عملگرهای دسته بندی – Grouping Operators

این نوع عملگر ها برای دسته بندی عناصر بسته به یک کلید درونی مورد استفاده قرار می گیرند.

عملگر Group By

بوسیله این عملگر می توان عناصری از مجموعه نتیجه را بوسیله یک شرط خاص (تابع گزینشی) دسته بندی کرد. عملکرد عملگر دسته بندی Group by در LINQ دقیقا مشابه ماده Group By در SQL است.

عملگر Group By تنها عملگر در این گروه می باشد که شامل 8 نوع سربارگذاری است، در زیر 4 سربارگذاری اول این عملگر آمده است.

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

public static IEnumerable<IGrouping<TKey, TSource>> GroupBy<TSource, TKey>(
    this IEnumerable<TSource> source, Func<TSource, TKey> keySelector,
    IEqualityComparer<TKey> comparer);

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<IGrouping<TKey, TElement>> GroupBy<TSource, TKey,
TElement>(
    this IEnumerable<TSource> source, Func<TSource, TKey> keySelector,
    Func<TSource, TElement> elementSelector,
    IEqualityComparer<TKey> comparer);
```

در بارگذاری های این عملگر، یک نوع از IEnumerable<IGrouping<TKey, TElement >> برای خروجی ارسال می گردد که در خود یک خصوصیت فقط خواندنی Key فراهم کرده است. نحوه پیاده سازی این رابط به صورت زیر است.

```
public interface IGrouping<TKey, TElement> : IEnumerable<TElement> {
TKey Key { get; }
}
```

در زمان اجرای پرس و جو حاوی عملگر Group by پارامتر اول تابع یعنی source بوسیله ارزیابی و شمارش پارامتر های keySelector و elementSelector تعیین می گردد و خروجی در یک نمونه از <<

IEnumerable<IGrouping<TKey, TElement درج می شود که خصوصیت فقط خواندنی Key آن برای نمایش دادن فیلدی جلوی قسمت by در عملگر group by قرار می گیرد، مورد استفاده قرار می گیرد.

مثال1: در این مثال مشتریان بر اساس کشورشان دسته بندی می شوند.

```
List<Customer> customers = new List<Customer>()
{
    new Customer() {Name ="Ali"   , Family = "Aghdam" , Country = "iran" ,
                    CustomerID =0 },
    new Customer() {Name ="Ali"   , Family = "Nasiri" , Country = "england"
                  , CustomerID =1 },
    new Customer() {Name ="Arash" , Family = "Novin"  , Country = "india" ,
                    CustomerID =2 },
    new Customer() {Name ="Arash" , Family = "Novin"  , Country = "iran" ,
                    CustomerID =3 }
};


var query = from c in customers
            group c by c.Country;


foreach (var CountryGroup in query)
{
    Console.WriteLine( CountryGroup.Key);
    foreach (var customerInGroup in CountryGroup)
    {
        Console.WriteLine(customerInGroup);
    }
}
```

مثال 2.

```
    var query = from c in customers
                group c by c.Country into cc
                select new { Country = cc.Key  };


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

در این مثال مشتریان بر اساس کشورشان دسته بندی می شوند و و در نتیجه این دسته بندی در درون متغیر cc قرار می گیرد و سپس گروه کشورها بر اساس خصوصیت key مربوط به cc به عنوان خروجی پرس و جو انتخاب می شوند.

مثال 3: نمایش تعداد مشتریان از هر دسته کشور

```
var query = from c in customers
            group c by c.Country into cc
            select new { Cuntry = cc.Key , Count = cc.Count() };


foreach (var item in query)
{
    Console.WriteLine(item.Count +  " Customer from " + item.Cuntry);
}
```

عملگر های مرتب سازی - Ordering Operators

عملگر های مرتب سازی برای تنظیم جایگاه هر عنصر در مجموعه و نحوه چیدمان آنها مورد استفاده قرار می گیرد.

عملگر OrderBy

این عملگر عناصر یک مجموعه را بر اساس یک کلید به صورت صعودی مرتب می کند که عملکردی مانند ماده Order By در SQL دارد.

فرم کلی این عملگر به دو صورت زیر است.

```
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);
```

در پیاده سازی این عملگر پرامتر keySelector برای تعیین فیلدی که عملیات مرتب سازی بر اساس آن بر روی source صورت می گیرد، استفاده می شود. در فرم (بارگزاری) دوم این عملگر از پارامتر compare می توان برای عملیات مرتب سازی سفارشی استفاده کرد. این عملگر در زمان اجرای برنامه عناصر مجموعه را بر اساس پرامتر keySelector ارزیابی می نماید و یک نمونه از نوع IOrderedEnumerable<TSource> را برمی گرداند.

مثال1. مرتب سازی به صورت صعودی بر اساس نام و نام فامیلی مشتری.

```
List<Customer> customers = new List<Customer>()
    {
        new Customer() {Name ="Ali"    , Family = "Aghdam" , Country = "iran" ,
                        CustomerID =0  },
```

```
            new Customer() {Name ="Ali"    , Family = "Nasiri" , Country = "england"
                            , CustomerID =1 },
            new Customer() {Name ="Arash" , Family = "Novin"  , Country = "india" ,
                            CustomerID =2 },
            new Customer() {Name ="Ali" , Family = "Novin"  , Country = "iran" ,
                            CustomerID =3 }
        };

    var query = from c in customers
        orderby  c.Family
                    select c;

    foreach (var item in query)
        Console.WriteLine(item);
```

عملگر OrderBy descending

این عملگر عناصر یک مجموعه را بر اساس یک کلید به صورت نزولی مرتب می کند که عملکردی مانند ماده Order By دارد.

```
public static IOrderedEnumerable<TSource> OrderByDescending<TSource, TKey>(
    this IEnumerable<TSource> source,
    Func<TSource, TKey> keySelector);

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

در پیاده سازی این عملگر پرامتر keySelector برای تعیین فیلدی که عملیات مرتب سازی بر اساس آن بر روی source صورت می گیرد، استفاده می شود. در فرم (بارگزاری) دوم این عملگر از پارامتر compare می توان برای عملیات مرتب سازی سفارشی استفاده کرد. این عملگر همانند عملگر OrderBy در زمان اجرای برنامه عناصر مجموعه را بر اساس پرامتر keySelector ارزیابی می نماید و یک نمونه از نوع IOrderedEnumerable<TSource> را برمی گرداند.

مثال. عبارت پرس و جو مثال1 در عملگر OrderBy به صورت نزولی.

```
var query = from c in customers
    orderby  c.Family descending
                select c;
```

عملگر Thenby

عملگر OrderBy این عملیات مرتب سازی را بر اساس یک کلید را اماکن پذیر می نمود ولی برای اینکه بتوان از عملیات مرتب سازی به صورت صعودی را بر اساس چند کلید انجام داد، می بایست از عملگر ThenBy استفاده نمود.

این عملگر دارای دو سربارگذاری به صورت زیر است.

```
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);
```

این عملگر همانند عملگر OrderBy معمول است تنها با این تفاوت که این عملگر می تواند تنها بر روی نوع IOrderedEnumerable<TSource> عملیات مرتب سازی را انجام داد که بدین وسیله می بایست این عملگر را بعد از عملگر OrderBy و یا Orderby Descending و یا Thenby Descending استفاده نمود.

البته در زمان نوشتن پرس و جوها به صورت عبارت های پرس و جوی LINQ از عملگر OrderBy استفاده می شود و کلیدها بوسیه کاما (،) از یکدیگر جدا کامپایل شدن پرس و جو ها کامپایلر خود این مورد را متوجه شده و عبارت متدی آن را تنظیم می کند.

به طور مثال عبارت پرس و جوی زیر را در نظر بگیرید.

```
var query = from c in customers
    orderby  c.Name , c.Family
                select c;
```

این پرس و جو در هنگام کامپایل به صورت متدی زیر تبدیل می گردد.

```
var query = customers.OrderBy(c => c.Name)
                     .ThenBy(c => c.Family);
```

همانطور که در عبارت بالا مشاهده می کنید عملگر ThenBy در صورت متدی خود نوشته می شود و در نوع "عبارت پرس و جو" از همان عملگر OrderBy استفاده می شود و کلیدها با کاما از یکدیگر جدا می شوند. برای کامپاسل شدن عبارات می توان ترتیب زیر را نوشت که در هر صورت می بایست یک عملگر OrderBy در ابتدای مرتب سازی وجود داشته باشد تا یک نوع IOrderedEnumerable<TSource> تهیه شود تا عملگر های دیگر بر روی آن عمل کنند.

Source . OrderBy . ThenBy . ThenBy ...

عملگر ThenByDecending

این عملگر کارکردی دقیقا همانند عملگر Thenby دارد و تنها تفاوت آنها در این است که این تابع مجموعه را به صورت نزولی مرتب می کند. این تابع دارای دو سربازگذاری به صورت زیر است.

```
public static IOrderedEnumerable<TSource> ThenByDescending<TSource, TKey>(
    this IOrderedEnumerable<TSource> source,
    Func<TSource, TKey> keySelector);

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

این عملگر همانند عملگر ThenBy بر روی مجموعه ای از نوع IOrderedEnumerable<TSource> عمل کند که به همین دلیل می بایست بعد از عملگرهای OrderBy ، OrderByDescending و یا ThenBy استفاده شود و در نوشتار "عبارت پرس و جو" می بایست کلمه Descending را مقابل کلید نوشت و در نوع متدی می بایست به صورت مستقیم از متد ThenByDescending استفاده نمود.

مثال.

```
var query = from c in customers
            orderby c.Name, c.Family descending
            select c;
```

معادل با صورت متدی زیر :

```
var query = customers.OrderBy(c => c.Name)
                     .ThenByDescending(c => c.Family);
```

عملگر Reverse

این عملگر همانند نام خود برای وارون کردن عناصر مجموعه به کار می رود یعنی عناصر چینش عناصر نتیجه را برعکس می کند و فقط دارای یک بارگزاری به صورت زیر است.

```
public static IEnumerable<TSource> Reverse<TSource>(
    this IEnumerable<TSource> source);
```

کارکرد این عملگر بسیار ساده است و در زمان اجرا عناصر مجموعه را در یک نمونه جدید از IEnumerable<T> اما به صورت عکس قرار می دهد.

مثال1. در این مثال مرتب سازی به صورت صعودی بر اساس نام مشتری صورت می گیرد و سپس بوسیله عملگر Reverse چیدمان عناصر برعکس می شود (یعنی به صورت نزولی).

```
List<Customer> customers = new List<Customer>()
{
 new Customer() {Name ="Ali",Family="Aghdam",Country="iran",CustomerID =0  },
 new Customer() {Name ="Ali",Family="Nasiri",Country="england",CustomerID =1 },
 new Customer() {Name ="Arash",Family="Novin",Country="india",CustomerID =2 },
 new Customer() {Name ="Ali",Family="Novin",Country= "iran" , CustomerID =3 }
};

var query = from c in customers
    orderby  c.Family
                select c;

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

عملگرهای تجمعی – Agreagate Operators

این عناصر برای محاسبات بر روی عناصر مجموعه ها استفاده می شوند به عنوان مثال جمع تمام عناصر در یک مجموعه و یا میانگین عناصر یک مجموعه.

عملگر Count

از عملگر Count می توان برای محاسبه تعداد عناصر یک مجموعه استفاده نمود.

```
public static int Count<TSource>(
    this IEnumerable<TSource> source);

public static int Count<TSource>(
    this IEnumerable<TSource> source,
    Func<TSource, Boolean> predicate);
```

فرم اول این عملگر زمانی مورد استفاده قرار می گیرد که source رابط IEnumerable<TSource> را پیاده سازی کرده باشد، در این صورت از متد Count() پیاده سازی شده در درون این واسط برای محاسبه تعداد عناصر استفاده می شود ولی زمانی که source رابط IEnumerable<TSource> را پیاده سازی نکرده باشد بوسیله تابع پارامتر predicate عناصر مجموعه را شمارش می کند و سپس با افزودن 1 به به آن تعداد عناصر را برمی گرداند.

مثال.

```
int query = customers.Count();
```

نکته بسیار مهم

توجه داشته باشید که نوع خروجی این عملگر از نوع int می باشد و در صورتی که تعداد عناصر مجموعه از محدوده int تجاوز کند، استثنای OverflowException رخ خواهد داد، در این صورت می توانید از عملگر LongCount استفاده کنید

برای به دست آوردن محدوده Int می توانید از متد های int.MaxValue و int.MinValue استفاده کنید.

عملگر LongCount

از این عملگر نیز همانند عملگر Count برای محاسبه تعداد عناصر مجموعه استفاده می شود و تنها تفاوت آنها این است که عملگرLongCount خروجی را از نوع Long بر می گرداند که بوسیله آن می توان تعداد عناصر بیشتر را مورد شمارش قرار داد.

```
public static long LongCount<TSource>(
    this IEnumerable<TSource> source);

public static long LongCount<TSource>(
    this IEnumerable<TSource> source,
    Func<TSource, Boolean> predicate);
```

فرم اول این عملگر زمانی مورد استفاده قرار می گیرد که source رابط IEnumerable<TSource> را پیاده سازی کرده باشد، در این صورت از متد Count() پیاده سازی شده در درون این واسط برای محاسبه تعداد عناصر استفاده می شود ولی زمانی که source رابط IEnumerable<TSource> را پیاده سازی نکرده باشد بوسیله تابع پارامتر predicate عناصر مجموعه را شمارش می کند و سپس با افزودن 1 به تعداد آن عناصر را برمی گرداند.

مثال.

```
long query = customers.LongCount();
```

عملگر Sum

از این عملگر برای مجموع عناصر عددی در یک مجموعه استفاده می شود و دارای دو سربارگزاری به صورت زیر است.

```
public static Numeric Sum(this IEnumerable<Numeric> source);

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

در دو فرم این عملگر به نوع برگشتی Numeric دقت کنید. این نوع می تواند یکیاز انواع زیر باشد.

Int , Nullable<int> , long , Nullable<long> , double , Nullable<double> , decimal , Nullable<decimal>

در فرم نخست این عملگر مجموع تمامی عناصر موجود در یک مجموعه برگشت داده می شود ولی در فرم دوم این مجموع عناصری از مجموعه محاسبه می شود که توسط تابع selector تعیین شده باشد.

مثال.

```
int[] integers = { 5, 3, 8, 9, 1, 7 };
int sum = integers.Sum();
Console.WriteLine("Total of all Numbers : {0}", sum.ToString());
```

عملگر Min و Max

عملگرهای Min و Max به ترتیب کوچکترین و بزرگترین عنصر در یک مجموعه را بر می گردانند.

```
public static Numeric Min/Max(
    this IEnumerable<Numeric> source);

public static TSource Min<TSource>/Max<TSource>(
    this IEnumerable<TSource> source);

public static Numeric Min<TSource>/Max<TSource>(
    this IEnumerable<TSource> source,
    Func<TSource, Numeric> selector);

public static TResult Min<TSource, TResult>/Max<TSource, TResult>(
    this IEnumerable<TSource> source,
    Func<TSource, TResult> selector);
```

این دو عملگر مجموعه source را بررسی می کنند و سپس برای پیدا کردن بزرگترین و یا کوچکترین مقدار، تابع selector را فراخوانی می کنند، در صورتی که پارامتر selector تعیین نشده باشد، کوچکترین و یا بزرگترین عنصر بر اساس مقدارشان انتخاب و برگردانده می شوند.

مثال1. استفاده از فرم اول این عملگرها برای پیدا کردن کوچکترین و بزرگترین اعداد در بین مجموعه اعداد مثال قبلی.

```
int[] integers = { 5, 3, 8, 9, 1, 7 };
int max = integers.Max();
int min = integers.Min();
Console.WriteLine("Min ={0} and Max = {1}", max , min);
```

مثال2. استفاده از فرم دوم این عملگر برای پیدا کردن بیشترین سفارش، مشتری با کد 1.

```
List<Customer> customers = new List<Customer>()
{
   new Customer() {
                    Name ="Ali",
                    Family = "Aghdam" ,
                    Country = "iran" ,
                    CustomerID =0  },
};

List<Order> orders = new List<Order>()
{
    new Order() { CustomerID = 0 , Total = 1000},
    new Order() { CustomerID = 0 , Total = 1200},
    new Order() { CustomerID = 0 , Total = 200},
    new Order() { CustomerID = 0 , Total = 500}
};


var query = from c in customers
            join o in orders
            on c.CustomerID equals o.CustomerID
            select new { c.CustomerID, c.Family, o.Total };



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

// Max Order
Console.WriteLine( query.Max(c => c.Total) );

//Min Order
Console.WriteLine( query.Min(c => c.Total) );
```

عملگر Average

با استفاده از این عملگر می توان میانگین عناصر موجود در یک مجموعه را محاسبه نمود.

```
public static Result Average(
    this IEnumerable<Numeric> source);

public static Result Average<TSource>(
    this IEnumerable<TSource> source,
    Func<TSource, Numeric> selector);
```

در فرم نخست این عملگر میانگین عناصر موجود در مجموعه source محاسبه می شود و در فرم دوم میانگین عناصر بر اساس تابع selector محاسبه می شوند. نوع خروجی در این عملگر با عملگر های قبلی تفاوت دارد و به nullabilityبودن نوع های عددی بستگی دارد در صورتی که نوع های عددی Int32 و Int64 خروجی از نوع Double خواهد بود و اگر نوع های عددی Nullable<Int32> و Nullable<Int64> باشد خروجی از نوع Nullable<Double> خواهد بود.

مثال1. استفاده از فرم اول

```
int[] integers = { 5, 3, 8, 9, 1, 7 };

double average = integers.Average();

Console.WriteLine("Average = {0}", average);
```

مثال2. استفاده از فرم دوم

```
List<Customer> customers = new List<Customer>()
{
    new Customer() {Name ="Ali"     , Family = "Aghdam" , Country = "iran" ,
CustomerID =0  },
};

List<Order> orders = new List<Order>()
{
    new Order() { CustomerID = 0 , Total = 1000},
    new Order() { CustomerID = 0 , Total = 1200},
    new Order() { CustomerID = 0 , Total = 200},
    new Order() { CustomerID = 0 , Total = 500}
};

var query = from c in customers
            join o in orders
            on c.CustomerID equals o.CustomerID
            select new { c.CustomerID, c.Family, o.Total };
```

```
Console.WriteLine( query.Average(c => c.Total) );
```

عملگر Aggregate

این تابع این اماکن را فراهم می کند که یک تابع را بر روی هریک از اعضاء یک مجموعه اجرا نماییم.

```
public static T Aggregate<TSource>(
        this IEnumerable<TSource> source,
        Func<TSource, TSource, TSource> func);

public static TAccumulate Aggregate<TSource, TAccumulate>(
        this IEnumerable<TSource> source,
        TAccumulate seed,
        Func<TAccumulate, TSource, TAccumulate> func);

public static TResult Aggregate<TSource, TAccumulate, TResult>(
        this IEnumerable< TSource > source,
        TAccumulate seed,
        Func<TAccumulate, TSource, TAccumulate> func,
        Func<TAccumulate, TResult> resultSelector);
```

اولین عنصر در source به عنوان اولین مقدار عملگر Aggregate (یعنی نقطه شروع)در نظر گرفته می شود. در فرم اول این عملگر نقطه شروع از اولین عنصر در مجموعه source در نظر گرفته می شود در فرم دوم پارامتر seed از نوع TAccumulate به عنوان نقطه شروع برای تابع Aggregate در نظر گرفته می شود و در فرم سوم این عملگر تابع resultSelector به عنوان نوعی شرط برای پایان کار aggregate در نظر گرفته می شود.

مثال1. شبیه سازی Sum به وسیله عملگر Aggregate

```
List<int> SampleList = new List<int>() { 1, 1, 2, 3, 5, 8, 13 };

int AggCount = SampleList.Aggregate((counter, listItem) =>
                                                    counter += listItem);

Console.WriteLine(string.Format("Aggregate count is: {0}", AggCount));
```

مثال2. شبیه سازی نوعی Reverse برای رشته بوسله عملگر Aggregate

```
string sentence = "the quick brown fox jumps over the lazy dog";
// Split the string into individual words.

string[] words = sentence.Split(' ');
// Join each word to the beginning of the new sentence to reverse the word oder.
```

```
string reversed = words.Aggregate((workingSentence, next) =>
                                                    next + " " + workingSentence);

Console.WriteLine(reversed);
/*
This code produces the following output:

dog lazy the over jumps fox brown quick the
*/
```

عملگرهای قسمت بندی – Partitioning Operators

این عملگرها برای تقسیم بندی مجموعه ها بدون مرتب سازی، به دو یا چند قسمت مورد استفاده قرار می گیرند. همچنین این عملگرها یک قسمت را برمی گردانند و از باقی عناصر مجموعه صرف نظر می کنند. توجه داشته باشید که برای پیاده سازی مکانیزم صفحه بندی استفاده زیادی از این عملگرها خواهید نمود.

عملگر Take

این عملگر تعدادی از عناصر مجموعه را بر اساس مقدار ارسالی به آن، بر می گرداند. فرم کلی این عملگر به صورت زیر است.

```
public static IEnumerable<TSource> Take<TSource>(
    this IEnumerable<TSource> source,
    Int32 count);
```

دراین عملگر از مجموعه source به تعداد count تا از عناصر به خروجی ارسال خواهند شد.

مثال1. 5 مقدار اول از مجموعه SampleList

```
List<int> SampleList = new List<int>() { 1, 1, 2, 3, 5, 8, 13 };

var query = SampleList.Take(5);

foreach (var item in query)
{
    Console.Write(item + ",  ");
}
```

عملگر Skip

این عملگر به تعدادی مشخص از عناصر مجموعه صرف نظر کرده و سپس عناصر باقیمانده در مجموعه را به خروجی ارسال می کند.

```
public static IEnumerable<TSource> Skip<TSource>(
    this IEnumerable<TSource> source,
    Int32 count);
```

دراین عملگر در ابتدا مجموعه source به تعداد count تا شمارش شده و سپس عناصر باقیمانده در مجموعه source خروجی ارسال خواهند شد.

مثال1. صرف نظر کردن از 5 عنصر اول در لیست SampleList و انتخاب عناصر باقیمانده

```
List<int> SampleList = new List<int>() { 1, 1, 2, 3, 5, 8, 13 };

var query = SampleList.Skip(5);

foreach (var item in query)
{
    Console.Write(item + ",  ");
}
```

عملگر TakeWhile

این عملگر عناصر یک مجموعه را تا زمان برقرار بودن یک شرط معین به خروجی ارسال می کند.

```
public static IEnumerable<TSource> TakeWhile<TSource>(
    this IEnumerable<TSource> source,
    Func<TSource, Boolean> predicate);

public static IEnumerable<TSource> TakeWhile<TSource>(
    this IEnumerable<TSource> source,
    Func<TSource, Int32, Boolean> predicate);
```

در این عملگر مجموعه source تا زمانی که شرط مربوط به پارامتر predicate نقض گردد، شمارش شده و در یک نمونه از نوع IEnumerable<TSource> قرار می گیرد. در فرم دوم تابع predicate یک پارامتر از نوع int را دارا می باشد که محل قرار گیری عناصر بر روی مجموعه را نشان می دهد.

مثال1.استفاده از فرم اول عملگر TakeWhile

```
List<int> SampleList = new List<int>() { 1, 1, 2, 3, 5, 8, 13 };

var query = SampleList.TakeWhile(s => s <= 5 );

foreach (var item in query)
{
    Console.Write(item + ",  ");
}
```

مثال2. استفاده از فر دوم عملگر TakeWhile

```
List<int> SampleList = new List<int>() { 1, 1, 2, 3, 5, 8, 13 };

var query = SampleList.TakeWhile( (s,index) => (s > index) );

foreach (var item in query)
{
    Console.Write(item + ",  ");
}
```

عملگر SkipWhile

این عملگر از عناصر یک مجموعه تا زمان نقض یک شرط معین صرف نظر می کند و باقیمانده عناصر مجموعه را به خروجی ارسال می کند.

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

public static IEnumerable<TSource> SkipWhile<TSource>(
    this IEnumerable<TSource> source,
    Func<TSource, Int32, Boolean> predicate);
```

در این عملگراز عناصر مجموعه source تا زمانی که شرط مربوط به پارامتر predicate نقض گردد، صرف نظر می شود و باقیمانده عناصر مجموعه در قالب یک نمونه از نوع IEnumerable<TSource> به خروجی ارسال می گردند. در فرم دوم تابع predicate یک پارامتر از نوع int را دارا می باشد که محل قرار گیری عناصر بر روی مجموعه را نشان می دهد.

مثال1. استفاده از فرم اول عملگر SkipWhile

```
List<int> SampleList = new List<int>() { 1, 1, 2, 3, 5, 8, 13 };

var query = SampleList.SkipWhile( s => s < 5 );

foreach (var item in query)
{
    Console.Write(item + ",  ");
}
```

مثال2. استفاده از فرم دوم عملگر SkipWhile

```
List<int> SampleList = new List<int>() { 1, 3, 2, 3, 5, 8, 13 };

var query = SampleList.SkipWhile((s, index) => (s > index));

foreach (var item in query)
{
    Console.Write(item + ",  ");
}
```

عملگر الحاقی – Concatation Operator

دو عمل ملحق نمودن دو مجموعه به یکدیگر الحاق سازی و یا Concatenation گفته می شود و تنها عملگر این خانواده، عملگر Concat است.

عملگر Concat

این عملگر دو مجموعه را به هم متصل می کند.

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

خروجی این عملگر از نوع IEnumerable<TSource> است و تنها شرط در استفاده از این عملگر یکسان بودن دو پارامتر first و second از یک نوع است.

مثال1. متصل کردن دو مجموعه SampleList1 و SampleList2 به یکدیگر.

```
List<int> SampleList1 = new List<int>() { 1, 3, 2, 3, 5, 8, 13 };
List<int> SampleList2 = new List<int>() { 16, 20, 25 };

var query = SampleList1.Concat(SampleList2);

foreach (var item in query)
    Console.Write(item + ",  ");
```

عملگر های عنصری – Element Operators

این دسته از عملگر ها یک عنصر خاص و یا یک عنصر از مجموعه را بر می گردانند.

عملگر First

این عملگر اولین عنصر مجموعه را بر می گرداند. فرم کلی این عملگر به دو صورت زیر است.

```
public static TSource First<TSource>(
    this IEnumerable<TSource> source);

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

در فرم نخست این عملگر اولین عنصر مجموعه به عنوان خروجی برگردانده می شود ولی در فرم دوم، عناصر مجموعه از ابتدا بوسیله شرط که همان تابع predicate است، بررسی می شوند و اولین عنصری که صحت شرط را برقرار کند به عنوان اولین عنصر به خروجی ارسال می شود.

مثال1. استفاده از فرم اول این عملگر.

```
List<int> SampleList = new List<int>() { 1, 3, 2, 3, 5, 8, 13 };
var query = SampleList.First();
Console.Write(query);
```

مثال2. استفاده از فرم دوم این عملگر.

```
List<int> SampleList = new List<int>() { 1, 3, 2, 3, 5, 8, 13 };
var query = SampleList.First( c => c>1 );
Console.Write(query);
```

عملگر FirstOrDefault

این عملگر اولین عنصر مجموعه را بر می گرداند و در صورتی که عنصری در مجموعه وجود نداشته باشد و یا شرط را برقرار نکند، مقدار پیش فرضی به خروجی ارسال می شود. فرم کلی این عملگر به دو صورت زیر است.

```
public static TSource FirstOrDefault<TSource>(
    this IEnumerable<TSource> source);

public static TSource FirstOrDefault<TSource>(
    this IEnumerable<TSource> source,
    Func<TSource, Boolean> predicate);
```

در فرم نخست این عملگر اولین عنصر مجموعه به عنوان خروجی برگردانده می شود ولی در فرم دوم عناصر مجموعه از ابتدا بوسیله شرط که همان تابع predicate است، بررسی می شوند و اولین عنصری که صحت شرط را برقرار کند به عنوان اولین عنصر به خروجی ارسال می شود. در صورتی که در هر یک از دو فرم این عملگر عنصری موجود نداشته باشد، مقدار پیش فرضی از آن نوع به خروجی ارسال می شود.

مثال1. استفاده از فرم اول عملگر FirstOrDefault .

```
List<int> SampleList = new List<int>() { 1, 3, 2, 3, 5, 8, 13 };

var query = SampleList.FirstOrDefault();

Console.Write(query);
```

مثال2. استفاده از فرم دوم عملگر FirstOrDefault .

```
List<int> SampleList = new List<int>() { 1, 3, 2, 3, 5, 8, 13 };

var query = SampleList.FirstOrDefault( c => c > 3 );

Console.Write(query);
```

عملگر Last

این عملگر آخرین عنصر مجموعه را بر می گرداند. فرم کلی این عملگر به دو صورت زیر است.

```
public static TSource Last<TSource>(
    this IEnumerable<TSource> source);

public static TSource Last<TSource>(
    this IEnumerable<TSource> source,
    Func<TSource, Boolean> predicate);
```

در فرم نخست این عملگر آخرین عنصر مجموعه به عنوان خروجی برگردانده می شود ولی در فرم دوم، عناصر مجموعه از انتها بوسیله شرط که همان تابع predicate است، بررسی می شوند و اولین عنصری (از انتهای مجموعه) که صحت شرط را برقرار کند به عنوان آخرین عنصر به خروجی ارسال می شود.

مثال1. استفاده از فرم اول این عملگر.

```
List<int> SampleList = new List<int>() { 1, 3, 2, 3, 5, 8, 13 };

var query = SampleList.Last();

Console.Write(query);
```

مثال2. استفاده از فرم دوم این عملگر.

```
List<int> SampleList = new List<int>() { 1, 3, 2, 3, 5, 8, 13 };

var query = SampleList.Last ( c => c>1 );

Console.Write(query);
```

عملگر LastOrDefault

این عملگر آخرین عنصر مجموعه را بر می گرداند و در صورتی که عنصری در مجموعه وجود نداشته باشد و یا شرط را برقرار نکند، مقدار پیش فرضی به خروجی ارسال می شود. فرم کلی این عملگر به دو صورت زیر است.

```
public static TSource LastOrDefault<TSource>(
    this IEnumerable<TSource> source);

public static TSource LastOrDefault<TSource>(
    this IEnumerable<TSource> source,
    Func<TSource, Boolean> predicate);
```

در فرم نخست این عملگر آخرین عنصر مجموعه به عنوان خروجی برگردانده می شود ولی در فرم دوم عناصر مجموعه از انتها بوسیله شرط که همان تابع predicate است، بررسی می شوند و اولین عنصری (از انتها) که صحت شرط را برقرار کند به عنوان آخرین عنصر به خروجی ارسال می شود. در صورتی که در هر یک از دو فرم این عملگر عنصری موجود نداشته باشد، مقدار پیش فرضی از آن نوع به خروجی ارسال می شود.

مثال1. استفاده از فرم اول عملگر FirstOrDefault .

```
List<int> SampleList = new List<int>() { 1, 3, 2, 3, 5, 8, 13 };

var query = SampleList.LastOrDefault();

Console.Write(query);
```

مثال2. استفاده از فرم دوم عملگر FirstOrDefault .

```
List<int> SampleList = new List<int>() { 1, 3, 2, 3, 5, 8, 13 };

var query = SampleList.LastOrDefault( c => c > 3 );
```

```
Console.Write(query);
```

عملگر Single

از این عملگر برای انتخاب یک عنصر خاص و یکتا از یک مجموعه استفاده می شود.فرم کلی این عملگر به دو صورت زیر است.

```
public static TSource Single<TSource>(
    this IEnumerable<TSource> source);

public static TSource Single<TSource>(
    this IEnumerable<TSource> source,
    Func<TSource, Boolean> predicate);
```

فرم اول این عملگر زمانی که تنها یک عملگر در مجموعه source موجود باشد، آن عملگر به عنوان خروجی ارسال می شود ولی در صورتی که مجموعه source تهی و یا بیشتر از یک عنصر داشته باشد، استثنا InvalidOperationException رخ می دهد.

در صورتی که از فرم دوم استفاده شود یعنی اینکه یک شرط با پارامتر predicate ارسال شود و در بین عناصر مجموعه تنها یک عنصر شرط را برقرار کند، آن عنصر به خروجی ارسال می شود ولی درصورتی که هیچ عنصری شرط را برقرار نکند و یا چندین عنصر شرط predicateرا برقرار کنند، استثناء InvalidOperationExceptionرخ می دهد.

مثال1. استفاده از فرم اول عملگر Single

```
List<int> SampleList = new List<int>() { 1, 3, 2, 3, 5, 8, 13 };

var query = SampleList.Single();
//throw a InvalidOperationException Exception

Console.Write(query);
```

مثال2. استفاده از فرم دوم عملگرSingle

```
List<int> SampleList = new List<int>() { 1, 3, 2, 3, 5, 8, 13 };


var query = SampleList.Single( c => c == 3);
```

```
//throw a InvalidOperationException Exception

Console.Write(query);
```

عملگر SingleOrDefault

از این عملگر برای انتخاب یک عنصر خاص و یکتا از یک مجموعه استفاده می شود.فرم کلی این عملگر به دو صورت زیر است.

```
public static TSource SingleOrDefault<TSource>(
    this IEnumerable<TSource> source);

public static TSource SingleOrDefault<TSource>(
    this IEnumerable<TSource> source,
    Func<TSource, Boolean> predicate);
```

فرم اول این عملگر زمانی که تنها یک عملگر در مجموعه source موجود باشد، آن عملگر به عنوان خروجی ارسال می شود ولی در صورتی که مجموعه source تهی و یا بیشتر از یک عنصر داشته باشد، یک نمونه پیش فرض از نوع source به خروجی ارسال می گردد.

در صورتی که از فرم دوم استفاده شود یعنی اینکه یک شرط با پارامتر predicate ارسال شود و در بین عناصر مجموعه تنها یک عنصر شرط را برقرار کند، آن عنصر به خروجی ارسال می شود ولی درصورتی که هیچ عنصری شرط را برقرار نکند و یا چندین عنصر شرط predicateرا برقرار کنند، یک نمونه پیش فرض از نوع source به خروجی ارسال می گردد.

توجه داشته باشید که در هر دو فرم این عملگر، اگر چندین عنصر شرط predicateرا برقرار کنند، استثناء InvalidOperationExceptionرخ می دهد.

مثال1. استفاده از فرم اول عملگر SingleOrDefault .

```
List<int> SampleList = new List<int>() { 1, 3, 2, 3, 5, 8, 13 };


var query = SampleList.SingleOrDefault();
//throw a InvalidOperationException Exception

Console.Write(query);
```

مثال2. استفاده از فرم دوم عملگرSingleOrDefaul .

```
List<int> SampleList = new List<int>() { 1, 3, 2, 3, 5, 8, 13 };

var query = SampleList. SingleOrDefault ( c => c == 4);
//throw a InvalidOperationException Exception

Console.Write(query);
```

عملگر ElementAt

این عملگر عنصری را که اندیس آن به عنوان پارامتر ارسال می شود، برمی گرداند. (اندیس از صفر شروع می شـود) فرم کلی این عملگر به دو صورت زیر است.

```
public static TSource ElementAt<TSource>(
    this IEnumerable<TSource> source,
    Int32 index);
```

در این عملگر بوسیله شمارش عناصر مجموعه source ، عنصری که دارای اندیس index است به خروجی ارسال می شود و در صورتی که اندیس وارد شده معتبر نباشد یک استثنا از نـوع ArgumentNullException رخ مـی دهد. نکته قابل توجه در این عملگر این است که اگر مجموعه source رابط IList<T> را پیاده سازی کـرده باشـد عناصر به صورت مستقیم شمارش نمی شوند بلکه بوسیله توابع داخلی ای رابط عنصر دارای اندیس index برگردانـده می شود.

مثال.

```
List<int> SampleList = new List<int>() { 1, 3, 2, 3, 5, 8, 13 };

var query = SampleList.ElementAt(4);

Console.Write(query);
```

عملگر ElementAtOrDefault

این عملگر عنصری را که اندیس آن به عنوان پارامتر ارسال می شود، برمی گرداند. (اندیس از صفر شروع می شـود) فرم کلی این عملگر به دو صورت زیر است.

```
public static TSource ElementAt<TSource>(
    this IEnumerable<TSource> source,
    Int32 index);
```

در این عملگر بوسیله شمارش عناصر مجموعه source ، عنصری که دارای اندیس index است به خروجی ارسال می شود و در صورتی که اندیس مورد نظر معتبر نباشد، یک نمونه پیش فرض از نوع source به خروجی ارسال مـی شود. نکته قابل توجه در این عملگر این است که اگر مجموعه source رابط IList<T> را پیاده سازی کـرده باشـد عناصر به صورت مستقیم شمارش نمی شوند بلکه بوسیله توابع داخلی ای رابط عنصر دارای اندیس index برگردانـده می شود.

مثال.

```
List<int> SampleList = new List<int>() { 1, 3, 2, 3, 5, 8, 13 };

var query = SampleList.ElementAt(4);

Console.Write(query);
```

عملگر DefaultEmpty

این عملگر برای جایگزین کردن یک عنصر در مجموعه با عنصر پیش فرض تعویض می کند. فرک کلی این عملگر به دو صورت زیر است:

```
public static IEnumerable<TSource> DefaultIfEmpty<TSource>(
    this IEnumerable<TSource> source);

public static IEnumerable<TSource> DefaultIfEmpty<TSource>(
    this IEnumerable<TSource> source,
    TSource defaultValue);
```

به صورت پیش فرض این عملگر یک آرایه از نوع source را برمی رگرداند ولی مجموعه source تهی باشد یک نمونه پیش فرض از نوع source را برمی گرداند. توجه داشته باشید که اگر از فرم نخست استفاده شود یعنی نوعی برای پیش فرض معین نشده باشد، از null استفاده می شود.

مثال1. استفاده از حالت اول عملگر DefaultEmpty .

```
    var expr = customers.DefaultIfEmpty();  // Null
```

مثال1. استفاده از حالت دوم عملگر DefaultEmpty .

```
 List<Customer> customers = new List<Customer>()
 {
 new Customer() {Name ="Ali",Family = "Aghdam",Country = "iran" , CustomerID =0  },
 new  Customer()  {Name  ="Majid",Family="Shah  Mohammadi",Country="iran",CustomerID  =1
}
 };
      var query = from c in customers
                  select new { c.CustomerID, c.Family } ;

      foreach (var item in query.DefaultIfEmpty() )
      {
          Console.WriteLine("Family : " + item.Family);
      }

      Customer defaultCustomer = new Customer() {
                          Family = "Default Name" , CustomerID=2 };
      List<Customer> emptyCustomer = new List<Customer>();

      foreach (var item in emptyCustomer.DefaultIfEmpty(defaultCustomer) )
      {
          Console.WriteLine("Family : " + item.Family);
      }
```

عملگرهای تولیدی – Generation Operators

این دسته از عملگرها برای تولید مجموعه ای از عناصر مورد استفاده قرار می گیرد. در ادامه این نوع عملگرها را بررسی خواهیم کرد.

عملگر Repeat

این عملگر یک مجموعه جدید که حاصل تکرار یک عنصر به تعداد معینی است، ایجاد می کند و به خروجی ارسال می کند. فرم کلی این عملگر به صورت زیر است:

```
public static IEnumerable<TResult> Repeat<TResult>(
    TResult element,
    int count);
```

این عملگر در زمان اجرا از نوع عملگر element به تعداد count تکرار نموده و مجموعه حاصل که از نوع IEnumerable<TResult> است، برمی گرداند.

مثال.

```
Customer majidShahm = new Customer() { Name = "Majid", Family = "Shah Mohammadi", Country = "iran", CustomerID = 1 };

IEnumerable<Customer> customers = Enumerable.Repeat(majidShahm, 4);

foreach (var item in customers)
{
    Console.WriteLine("Family : " + item.Family);
}
```

نکته بسیار مهم

توجه داشته باشید در هنگام استفاده از این عملگر، نوع مورد نظر از انواع ارجاعی[1] باشد، کپی های که از نوع مورد نظر ایجاد می شوند به همان نوع قبل اشاره میکنند، یعنی اگر برنامه ای به صورت زیر داشته باشیم با تغییر newC خصوصیات همه ی کپی ها از شی تغییر خواهند کرد که امری کاملا منطقی است.

```
Customer majidShahm = new Customer() { Name = "Majid", Family = "Shah Mohammadi", Country = "iran", CustomerID = 1 };

IEnumerable<Customer> customers = Enumerable.Repeat(majidShahm, 4);

foreach (var item in customers)
{
```

[1] Reference type

```
        Console.WriteLine("Family : " + item.Family);
    }

    Customer newC = customers.First();

    newC.Family = "Aghdam";

    foreach (var item in customers)
    {
        Console.WriteLine("Family : " + item.Family);
    }


    //Out:
    //Family : Shah Mohammadi
    //Family : Shah Mohammadi
    //Family : Shah Mohammadi
    //Family : Shah Mohammadi
    //Family : Aghdam
    //Family : Aghdam
    //Family : Aghdam
    //Family : Aghdam
```

عملگر Range

این عملگر مجموعه ای مشخص از اعداد پشت سر هم را در طیف مشخصی ایجاد می کند. فرم کلی این عملگـر بـه صورت زیر است:

```
public static IEnumerable<Int32> Range(
    Int32 start,
    Int32 count);
```

این عملگر مجموعه ای از اعداد را از مقدار start به تعداد count تولید می کند.

مثال1.

```
var query = Enumerable.Range(1, 5);

foreach (var item in query)
    Console.WriteLine(item);
```

مثال2. شبیه سازی تابع فاکتوریل بوسیله عملگر Range و Aggregate .

```
static int Factorial(int number)
{
    return (Enumerable.Range(0, number + 1)
    .Aggregate(0, (s, t) => t == 0 ? 1 : s *= t));
```

```
}
```

عملگر Empty

عملگر Empty یک مجموعه تهی از یک نوع مشخص را ایجاد می کند. فرم کلی این عملگر به صورت زیر است:

```
public static IEnumerable<TResult> Empty<TResult>();
```

این عملگر مجموعه ای تهی از نوع *IEnumerable<TResult>* ایجاد کرده و به خروجی ارسال می کند.

مثال.

```
IEnumerable<Order> emptyOrder = Enumerable.Empty<Order>();
```

عملگرهای تنظیم کننده – Set Operators

این دسته از عملگرها برای انجام اعمالی نظیر اجتماع ، اشتراک و... مورد استفاده قرار می گیرد که در این دسته 4 عملگر Distinct، Intersect، Union و Except قرار دارد که در ادامه آنها را بررسی خواهیم کرد.

عملگر Distinct

عملگر Distinct عناصر تکراری موجود در یک مجموعه را حذف می کند که معدل ماده DISTICT در SQL است که در Join از آن استفاده زیادی می شود. این عملگر دارای دو سربارگزاری به صورت زیر است:

```
public static IEnumerable<TSource> Distinct<TSource>(
    this IEnumerable<TSource> source);

public static IEnumerable<TSource> Distinct<TSource>(
    this IEnumerable<TSource> source,
    IEqualityComparer<TSource> comparer);
```

این عملگر برای انجام عملیات در ابتدا یک مجموعه جدید IEnumerable<TSource> ایجاد می کند سپس عناصر مجموعه source را شمارش کرده و عناصری که در مجموعه جدید وجود نداشته باشند به آن اضافه می گردند که در بررسی عناصر برای تکراری نبودن از متدهای GetHashCode و Equal استفاده می شود که می توان برای بررسی عناصر مجموعه از یک مقایسه کننده سفارشی استفاده نمود. مقایسه سفارشی می بایست رابط IEqulityComparer راپیاده سازی نموده باشد.

مثال1.

```
List<int> SampleList = new List<int>() { 1, 3, 2, 3, 1, 8, 13 };

var query = SampleList.Distinct();

foreach (var item in query)
    Console.WriteLine(item);
```

مثال2. استفاده از این عملگر در عملیات Join .

```
var expr =
    (from c in customers
     from o in c.Orders
     join p in products
     on o.IdProduct equals p.IdProduct
     select p
    ).Distinct();
```

عملگر Intersect

این عملگر از دو مجموعه فقط عناصری که در هر دو موجود باشند را برمی گرداند. این عملگر دارای دو سربارگزاری به صورت زیر است:

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

public static IEnumerable<TSource> Intersect<TSource>(
    this IEnumerable<TSource> first,
    IEnumerable<TSource> second,
    IEqualityComparer<TSource> comparer);
```

این عملگر در ابتدا یک مجموعه از نوع IEnumerable<TSource> را ایجاد می کند و سپس عناصر مجموعه first را تک به تک خوانده و با عناصر مجموعه second مقایسه می کند، در صورتی که عنصری در هر دو مجموعه مشترک باشد به مجموعه جدید اضافه می گردد. در فرم نخست این عملگر بررسی مشترک بودن عناصر به وسیله متدهای GetHashCode و Equal انجام می شود ولی می توان از یک مقایسه کننده سفارشی که رابط IEqualityComparer را پیاده سازی کرده باشد، برای مقایسه عناصر استفاده نمود.

```
List<int> SampleList1 = new List<int>() { 1, 3, 2, 3, 1, 8, 13 };
List<int> SampleList2 = new List<int>() { 1, 5, 7, 3, 1, 6};

var query = SampleList1.Intersect(SampleList2);

foreach (var item in query)
    Console.WriteLine(item);
```

عملگر Union

این عملگر عناصر دو مجوعه متفاوت را به یکدیگر متصل می کند و دارای دو سربارگزاری به صورت زیر است:

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

public static IEnumerable<TSource> Union<TSource>(
    this IEnumerable<TSource> first,
    IEnumerable<TSource> second,
    IEqualityComparer<TSource> comparer);
```

این عملگر نیز همانند عملگرهای Distinct و Intersect در ابتدا یک مجموعه از نوع IEnumerable<TSource> را ایجاد می کند و سپس عناصر مجموعه first را تک به تک خوانده و عناصر غیر تکراری را در مجموعه جدید قرار می دهد سپس عناصر مجموعه second را تک به تک خوانده و با عناصر مجموعه جدید مقایسه می کند، و در صورتی که عنصری در مجموعه جدید وجود نداشته باشد به مجموعه جدید اضافه می گردد. در فرم نخست این عملگر بررسی مشترک بودن عناصر به وسیله متدهای GetHashCode و Equal انجام می شود ولی می توان از یک مقایسه کننده سفارشی که رابط IEqualityComparer را پیاده سازی کرده باشد، برای مقایسه عناصر استفاده نمود.

مثال.

```
List<int> SampleList1 = new List<int>() { 1, 3, 2, 3, 1, 8, 13 };
List<int> SampleList2 = new List<int>() { 1, 5, 7, 3, 1, 6};

var query = SampleList1.Union(SampleList2);

foreach (var item in query)
    Console.WriteLine(item);
```

عملگر Except

این عملگر عناصر یک مجوعه را که در مجموعه دیگر وجود نداشته باشد را به خروجی ارسال می کند و دارای دو سربارگزاری به صورت زیر است:

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

public static IEnumerable<TSource> Except<TSource>(
    this IEnumerable<TSource> first,
    IEnumerable<TSource> second,
    IEqualityComparer<TSource> comparer);
```

این عملگر در ابتدا یک مجموعه از نوع IEnumerable<TSource> را ایجاد می کند و سپس عناصر مجموعه first را تک به تک خوانده و عناصر غیر تکراری را در مجموعه جدید قرار می دهد سپس عناصر مجموعه second را تک به تک خوانده و با عناصر مجموعه جدید مقایسه می کند، و در صورتی که عنصری در مجموعه second با عنصری در مجموعه جدید مشترک باشد، آن عنصر از مجموعه جدید حذف می گردد. در فرم نخست این عملگر

بررسی مشترک بودن عناصر به وسیله متدهای GetHashCode و Equal انجام می شود ولی می توان از یک مقایسه کننده سفارشی که رابط IEqualityComparer را پیاده سازی کرده باشد، برای مقایسه عناصر استفاده نمود.

توجه داشته باشید که مجموعه خروجی در این عملگر مجموعه ای شامل تمام عناصر غیر مشترک مجموعه first با second است.

مثال.

```
List<int> SampleList1 = new List<int>() { 1, 3, 2, 3, 1, 8, 13 };
List<int> SampleList2 = new List<int>() { 1, 5, 7, 3, 1, 6};

var query = SampleList1.Except(SampleList2);

foreach (var item in query)
    Console.WriteLine(item);

//output
//2
//8
//13
```

نکته بسیار مهم

در استفاده از عملگرهای الحاقی Distinct ، Intersect ، Union و Except به این نکته توجه کنید که فرم اول این عملگرها برای مقایسه عناصر از متد های GetHashCode و Equal استفاده می کند، در صورتی که عناصری وجود داشته باشند که دارای Reference های متفاوتی باشند ولی از نظر منتطقی با یدیگر برابر باشند، توسط این متد ها عناصر متفاوت نتیجه گیری می شوند. برای گریز از این مشکل می توان از فرم دوم این عملگر ها استفاده نمود و یا اینکه این متدها در شی مورد نظر را بازنویسی نمود.

به عنوان مثال در دو مجموعه زیر یکی از مشتریان از نظر معنایی در هر دو مجموعه وجود دارد ولی بوسیله این توابع به اشتباه انتخاب می گردد:

```
Customer[] customerSetOne = {
    new Customer {CustomerID = 46, Name = "Ali" , Family = "Aghdam"},
    new Customer {CustomerID = 27, Name = "Vali" , Family = "piriZadeh" },
    new Customer {CustomerID = 14, Name = "Majid" , Family = "Shah Mohammadi"}};

Customer[] customerSetTwo = {
    new Customer {CustomerID = 23, Name = "Mohammad" , Family = "Ajhdari"},
    new Customer {CustomerID = 22, Name = "Hossein" , Family = "Aghdam" },
    new Customer {CustomerID = 46, Name = "Ali" , Family = "Aghdam"}};

var customerUnion = customerSetOne.Union(customerSetTwo);

foreach (var item in customerUnion)
```

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

//output
//Name = Ali , Family = Aghdam , CustomerID = 46
//Name = Vali , Family = piriZadeh , CustomerID = 27
//Name = Majid , Family = Shah Mohammadi , CustomerID = 14
//Name = Mohammad , Family = Ajhdari , CustomerID = 23
//Name = Hossein , Family = Aghdam , CustomerID = 22
//Name = Ali , Family = Aghdam , CustomerID = 46
```

برای اجتناب از مشکل می توان یا از فرم دوم استفاده کرد و یا متد های GetHashCode را بازنویسی نمـود. در زیر این توابع را بازنویسی نموده ایم:

```
public class Customer
{
    public int CustomerID;
    public string Name;
    public string Family;

    public override string ToString()
    {
        return String.Format("Name : {0} - Family: {1} , CustomerID : {2}",
        this.Name, this.Family,this.CustomerID);
    }
    public override bool Equals(object obj)
    {
        if (!(obj is Customer))
            return false;
        else
        {
            Customer p = (Customer)obj;
            return p.CustomerID == this.CustomerID ;
        }
    }
    public override int GetHashCode()
    {
        return String.Format("{0}", this.CustomerID)
        .GetHashCode();
    }


}
```

حالا با انجام مثال قبلی خروج به صورت زیر خواهد بود:

```
//output
//Name = Ali , Family = Aghdam , CustomerID = 46
//Name = Vali , Family = piriZadeh , CustomerID = 27
//Name = Majid , Family = Shah Mohammadi , CustomerID = 14
//Name = Mohammad , Family = Ajhdari , CustomerID = 23
//Name = Hossein , Family = Aghdam , CustomerID = 22
```

عملگر Zip

این عملگر در NET 4. به خانواده عملگرهای تنظیم کننده اضافه شده است. فرم کلی این عملگر به صورت زیر است:

```
public static IEnumerable<TResult> Zip<TFirst, TSecond, TResult>(
    this IEnumerable<TFirst> first,
    IEnumerable<TSecond> second,
    Func<TFirst, TSecond, TResult> resultSelector);
```

این عملگر عناصر مجموعه first را با عناصر متناظر متناظر بوسیله اندیس، در مجموعه secondرا با یکدیگر ادغام می کند. دلیل نام گزاری این عملگر به Zip به خاطر شباهت آن به زیپ بوده است، به طوری که دو مجموعه را به یک مجموعه، بدون تغییر موقعیت ، تبدیل می کند.

مثال.

```
Int32[] numbers = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
DayOfWeek[] weekDays = {
                        DayOfWeek.Sunday,
                        DayOfWeek.Monday,
                        DayOfWeek.Tuesday,
                        DayOfWeek.Wednesday,
                        DayOfWeek.Thursday,
                        DayOfWeek.Friday,
                        DayOfWeek.Saturday};

var weekDaysNumbers = numbers.Zip(weekDays,
                                  (first, second) => first + " - " +
second);

foreach (var item in weekDaysNumbers)
    Console.WriteLine(item);

//output
//1 - Sunday
//2 - Monday
//3 - Tuesday
//4 - Wednseday
//5 - Thursday
//6 - Friday
//7 - Saturday
```

عملگرهای کمیت سنج – Quantifier Operators

این دسته عملگرها برای چک کردن مجموعه ها برای برقرار بودن شرط ها استفاده می شود. این دسته شامل عملگر های All ، Any و Contains است که در ادامه آنها را بررسی می کنیم.

عملگر All

این عملگر تمامی عناصر مجموعه را با یک شرط بررسی می کند و در صورتی تمامی عناصر مجموعه شرط را برقرار کنند، این عملگر مقدار true بر می گرداند. فرم کلی این عملگر به صورت زیر است:

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

این عملگر در زمان اجرا کلیه عناصر مجموعه source را با شرط predicate بررسی می کند و در صورتی که همه ی عناصر شرط را برقرار کنند، این عملگر مقدار true را بر می گرداند و در غیر اینصورت مقدار false برگردانده می شود.

مثال.

```
List<int> SampleList1 = new List<int>() { 1, 3, 2, 3, 1, 8, 13 };

bool query = SampleList1.All( c => ((c % 2) == 0));


Console.WriteLine(query);

//output
//False
```

نکته بسیار مهم

اگر مجموعه source تهی باشد، عملگر All همیشه مقدار true را برمی گرداند، این به خاطر این است که متد predicate هیچ موقع فراخوانی نمی شود و عملگر All مقدار true را برمی گرداند.

عملگر Any

این عملگر عناصر مجموعه را با یک شرط بررسی می کند و در صورتی عنصری شرط را برقرار کنند، این عملگر مقدار true بر می گرداند. فرم کلی این عملگر به صورت زیر است:

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

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

این عملگر در زمان اجرا عناصر مجموعه source را با شرط predicate بررسی می کند و در صورتی عنصری شرط را برقرار کنند، این عملگر مقدار true را بر می گرداند و در غیر اینصورت مقدار false برگردانده می شود.

مثال.

```
List<int> SampleList1 = new List<int>() { 1, 3, 2, 3, 1, 8, 13 };

bool query = SampleList1.Any( c => ((c % 2) == 0));

Console.WriteLine(query);

//output
//True
```

نکته بسیار مهم

اگر مجموعه source تهی باشد، عملگر Any همیشه مقدار true را برمی گرداند، این به خاطر این است که متد predicate هیچ موقع فراخوانی نمی شود و عملگر Any مقدار true را برمی گرداند.

عملگر Contains

این عملگر در یک مجموعه به دنبال یک عنصر مشخص می گردد و در صورت وجود، مقدار true را برمی گرداند.

فرم کلی این عملگر به صورت زیر است:

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

public static Boolean Contains<TSource>(
    this IEnumerable<TSource> source,
    TSource value,
    IEqualityComparer<TSource> comparer);
```

این عملگر درهنگام اجرا ابتدا بررسی می کند که مجموعه source رابط ICollection<T> را پیاده سازی کرده باشد در اینصورت از متد Contain مربوط به این رابط استفاده می کند در غیر اینصورت تمامی اعضاء مجموعه را شمارش می کند و در صورت وجود عنصر در مجموعه source مقدار true را برمی گرداند. البته برای چک کردن عناصر از توابع GetHashCode و Equal استفاده می کند (با وجود همان مشکل که در قسمت عملگرهای تنظیم کننده توضیح داده شد) ولی می توان توسط پارامتر compare می توان مقایسه کننده سفارشی را تنظیم نمود.

```
List<int> SampleList1 = new List<int>() { 1, 3, 2, 3, 1, 8, 13 };

bool query = SampleList1.Contains(2);


Console.WriteLine(query);

//output
//True
```

عملگرهای تبدیل – Conversion Operators

این دسته از عملگرها برای تبدیل مجموعه ورودی به انواع دیگر مورد استفاده قرار می گیرد و در این دسته عملگرهای Cast و ToArray ، ToDictionary، ToList ، ToLookup ، AsEnumerable و Asquerable وجود دارند که در ادامه توضیح خواهیم داد.

عملگر Cast

این عملگر عناصر مجموعه را به یک نوع معین تبدیل می کند. فرم کلی این عملگر به صورت زیر است:

```
public static IEnumerable<TResult> Cast<TResult>(
this IEnumerable source);
```

این عملگر عناصر مجموعه source را خوانده و آنها را به نوع تعیین شده TResult تبدیل کرده و در یک نمونه جدید از نوع IEnumerable<TResult> قرار می دهد.

مثال.

```
ArrayList list = new ArrayList { 1, 3, 2, 3, 1, 8, 13 };

IEnumerable<int> query = list.Cast<int>();

foreach (int i in query)
    Console.WriteLine(i);
//output
//1
//3
//2
//3
//3
//1
//8
//13
```

عملگر ToArray

این عملگر عناصر یک مجموعه از نوع IEnumerable<T> را به یک آرایه T[] تبدیل می کند. فرم کلی این عملگر به صورت زیر است:

```
public static TSource[] ToArray<TSource>(
    this IEnumerable<TSource> source);
```

این عملگر در زمان اجرا تمامی عناصر مجموعه source را تک تک خوانده و به نوع TSource تبدیل می کند و سپس در آرایه جدیدی از نوع []TSource قرار می دهد.

مثال.

```
List<int> numbers = new List<int> { 1, 3, 2, 3, 3, 1, 8, 13 };

int[] query = numbers.ToArray();

foreach (int i in query)
    Console.WriteLine(i);

//output
//1
//3
//2
//3
//3
//1
//8
//13
```

عملگر ToList

این عملگر عناصر یک مجموعه از نوع IEnumerable<T> را به یک مجموعه List<T> تبدیل می کند. فرم کلی این عملگر به صورت زیر است:

```
public static List<TSource> ToList<TSource>(
    this IEnumerable<TSource> source);
```

این عملگر در زمان اجرا تمامی عناصر مجموعه source را تک تک خوانده و در به نوع TSource تبدیل می کند و سپس در مجموعه جدیدی از نوع List<TSource> قرار می دهد.

مثال.

```
int[] numbers = { 1, 3, 2, 3, 3, 1, 8, 13 };

List<int> query = numbers.ToList();

foreach (int i in query)
    Console.WriteLine(i);

//output
//1
//3
```

```
//2
//3
//3
//1
//8
//13
```

عملگر ToDictionary

این عملگر یک عناصر مجموعه را به نوع Dictionary<TKey, TSource> تبدیل می کند. فرم کلی این عملگر به صورت زیر است:

```
public static Dictionary<TKey, TSource> ToDictionary<TSource, TKey>(
    this IEnumerable<TSource> source,
    Func<TSource, TKey> keySelector);

public static Dictionary<TKey, TSource> ToDictionary<TSource, TKey>(
    this IEnumerable<TSource> source,
    Func<TSource, TKey> keySelector,
    IEqualityComparer<TKey> comparer);

public static Dictionary<TKey, TElement> ToDictionary<TSource, TKey, TElement>(
    this IEnumerable<TSource> source,
    Func<TSource, TKey> keySelector,
    Func<TSource, TElement> elementSelector);

public static Dictionary<TKey, TElement> ToDictionary<TSource, TKey, TElement>(
    this IEnumerable<TSource> source,
    Func<TSource, TKey> keySelector,
    Func<TSource, TElement> elementSelector,
    IEqualityComparer<TKey> comparer);
```

درفرم نخست این عملگر مجموعه source را تک تک شمارش کرده و یک نوع جدید از Dictionary<TKey , TSource> ایجاد می کند و سپس عناصر مجموعه source را بوسیله تابع keySelector ارزیابی می شوند و مقدار Key برای مجموعه Dictionary را تولید می کند سپس مقدار آن عنصر به عنوان TSource و یا همان Value در نظر گرفته می شود و به همین ترتیب... .

در فرم دوم پارامتر comparer امکان تعیین یک تابع مقایسه کننده سفارشی را تعیین می کند.

در فرم سوم از پارامتر elementSelector می توان برای تعیین نوع TSource و یا همان Value استفاده کرد.

فرم آخر این عملگر مجموعی از فرم های دیگر است یعنی در آن بوسیله elementSelector می توان نوع TSource را تعیین کرد و بوسیله پارامتر comparer می توان تابع مقایسه کننده سفارشی را تعیین کرد.

مثال.

```
var customersDictionary =
        customers
        .ToDictionary(c => c.Name,
        c => new { c.Name, c.City });
```

عملگر ToLookup

این عملگر برای تبدیل یک مجموعه یا لیست به نوع *Lookup<K, T>* مورد استفاده قرار می گیرد. پیاده سازی کلی Lookup<K,T> به صورت زیر است:

```
public class Lookup<K, T> : IEnumerable<IGrouping<K, T>>
{
    public int Count { get; }
    public IEnumerable<T> this[K key] { get; }
    public bool Contains(K key);
    public IEnumerator<IGrouping<K, T>> GetEnumerator();
}
```

این رابط این امکان را فراهم می کند یک کلید می تواند به چندین T اشاره کند که این امکان فراهم ساختن ساختار one-to-many را فراهم می کند.

عملگر ToLookup داری چهار فرم کلی به صورت زیر است:

```
public static Lookup<TKey, TSource> ToLookup<TSource, TKey>(
    this IEnumerable<TSource> source,
    Func<TSource, TKey> keySelector);

public static Lookup<TKey, TSource> ToLookup<TSource, TKey>(
    this IEnumerable<TSource> source,
    Func<TSource, TKey> keySelector,
    IEqualityComparer<TKey> comparer);

public static Lookup<TKey, TElement> ToLookup<TSource, TKey, TElement>(
    this IEnumerable<TSource> source,
    Func<TSource, TKey> keySelector,
    Func<TSource, TElement> elementSelector);

public static Lookup<TKey, TElement> ToLookup<TSource, TKey, TElement>(
    this IEnumerable<TSource> source,
    Func<TSource, TKey> keySelector,
    Func<TSource, TElement> elementSelector,
    IEqualityComparer<TKey> comparer);
```

درفرم نخست این عملگر مجموعه source را تک تک شمارش کرده و یک نوع جدید از Lookup<TKey , TSource> ایجاد می کند و سپس عناصر مجموعه source را بوسیله تابع keySelector ارزیابی می شوند و مقدار Key برای مجموعه Lookup را تولید می کند سپس مقدار آن عنصر به عنوان TSource و یا همان Value در نظر گرفته می شود و به همین ترتیب... .

در فرم دوم پارامتر comparer امکان تعیین یک تابع مقایسه کننده سفارشی را تعیین می کند.

در فرم سوم از پارامتر elementSelector می توان برای تعیین نوع TSource و یا همان Value استفاده کرد.

فرم آخر این عملگر مجموعی از فرم های دیگر است یعنی در آن بوسیله elementSelector می توان نوع TSource را تعیین کرد و بوسیله پارامتر comparer می توان تابع مقایسه کننده سفارشی را تعیین کرد.

مثال.

```
var ordersByProduct =
       ( from c in customers
         from o in c.Orders
         select o )
        .ToLookup(o => o.OrderID);

Console.WriteLine("\n\nNumber of orders for Product 1: {0}\n",
                                    ordersByProduct[1].Count());
```

عملگر AsEnumerable

این عملگر یک مجموعه را به یک مجموعه از نوع IEnumerable<TSource> تبدیل می کند. فرم کلی این عملگر به این صورت است:

```
public static IEnumerable<TSource> AsEnumerable<TSource>(
    this IEnumerable<TSource> source);
```

این عملگر به سادگی مجموعه source را به نوع IEnumerable<TSource> تبدیل می کند.

این دسته از عملگر ها به اصطلاح "conversion on the fly" خوانده می شوند که این امکان را فراهم می کنند که بتوان توابع توسعه عام را بر روی نوع های که توابعی با همان نام دارند، اجرا نمود. به عنوان مثال کلاس Customersرا با پیاده سازی زیر که یک تابع توسعه Where برای آن تعریف شده، در نظر بگیرید.

```
public class Customers : List<Customer>
{
    public Customers(IEnumerable<Customer> items)
        : base(items)
    {
    }
}

public static class CustomersExtension
{
    public static Customers Where(this Customers source,
    Func<Customer, Boolean> predicate)
    {
        Customers result = new Customers(source);
        Console.WriteLine("Custom Where extension method");
        foreach (var item in source)
        {
            if (predicate(item))
                result.Add(item);
        }
        return result;
    }
}
```

حالا اگر بخواهیم پر سو جوی بر روی یک نمونه از شیء Customers انجام دهیم ،متد نتوسعه جدید اجرا خواهد شد و از متد Where نمی توان استفاده نمود. به مثال زیر دقت کنید:

```
List<Customer> customers = new List<Customer>()
{
 new Customer(){Name="Ali",Family="Aghdam",Country="iran",CustomerID =0},
 new Customer(){Name="Majid",Family="Shah Mohammadi",Country="iran",CustomerID=1}
};

  Customers customersList = new Customers(customers);

  var expr =
            from c in customersList
            where c.Country == "iran"
            select c;

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

  //output
  //Custom Where extension method
  //Name = Ali , Family = Aghdam , CustomerID = 0
  //Name = Vali , Family = piriZadeh , CustomerID =1
```

حال اگر بخواهید که متد عام Where مربوط به LINQ را برروی Customers اجرا کنید، نیاز به استفاده از عملگر AsEnumerable خواهید کرد. در پرس و جوی زیر از متد عام Where بر روی Customers استفاده شده است.

```
      List<Customer> customers = new List<Customer>()
      {
            new Customer(){Name="Ali",Family="Aghdam",Country="iran",CustomerID =0},
            new                                    Customer(){Name="Majid",Family="Shah
Mohammadi",Country="iran",CustomerID=1}
      };

      Customers customersList = new Customers(customers);

      var expr =
                from c in customersList.AsEnumerable()
                where c.Country == "iran"
                select c;

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

      //output
      //Name = Ali , Family = Aghdam , CustomerID = 0
      //Name = Vali , Family = piriZadeh , CustomerID =1
```

پیوست 1

در این پیوست پیاده سازی کلاس های به کار رفته برای تشریح عملگر های استاندارد پـرس و جـو در فصـل چهـار آمده است.

کلاس Customer

```
public class Customer
{
    public int CustomerID;
    public string Name;
    public string Family;
    public string Address;
    public string City;
    public string Region;
    public string PostalCode;
    public string Country;
    public string Phone;
    public List<Order> Orders;
}
```

کلاس Order

```
public class Order
{
    public int OrderID;
    public int CustomerID;
    public Customer Customer;
    public DateTime OrderDate;
    public decimal Total;
}
```

کلاس Product

```
public class Product
{
    public int ProductID;
    public string Name;
    public string Category;
    public decimal UnitPrice;
    public int UnitsInStock;
}
```

پیوست 2

نحوه تبدیل عبارات SQL به پرس و جوهای LINQ

دستور Select

عبارت پرس و جوی زیر در SQL برای برگرداندن تمامی رکورد های جدول Person استفاده می شود:

```
SELECT * FROM Person
```

عبارت پرس و جوی معادل با دستور بالا در LINQ به صورت زیر است:

```
from p in context.Persons
                    select p;
```

دستور Select چند ستونی

برای انتخاب چند ستونی در SQL ستون ها را با ویرگول از هم جدا می کردیم:

```
SELECT FName, LName FROM Person
```

در LINQ برای انتخاب چند ستونی از نوع های بی نام استفاده می کنیم:

```
from p in context.Persons
select new
              {
                 p.FName ,
                 p.LName
              }
```

نکته: در LINQ می توان دستور Select چند سطحی نیز نوشت که بیشتر برای کار join ازش استفاده میشه.

دستور Where

از این دستور در SQL برای انتخاب نتایج (ResultSet) بر اساس شرط و یا شروطی استفاده می شود:

```
SELECT * FROM Person
WHERE ID == 1
```

از ماده Where در LINQ برای همین مورد استفاده می شود. عبارت بالا معادل عبارت زیر در LINQ است:

```
from p in context.Persons
where p.ID == 1
select c
```

در عبارت های SQL از ماده ی LIKE برای شرط ها استفاده می شود که از متد های Contain ، StartWith و EndWith می توان برای انجام عملیات همانند ماده LIKE استفاده نمود. به عنوان مثال عبارات SQL زیر را در نظر بگیرید:

```
// 1.
SELECT * FROM Person
WHERE LName LIKE 'A%'

// 2.
SELECT * FROM Person
WHERE LName LIKE '%ghdam'

// 3.
SELECT * FROM Person
WHERE LName LIKE '%Aghdam%'
```

عبارت معادل آن در LINQ بوسیله استفاده از متد Contain و StartWith و EndWith به صورت زیر است:

```
// 1.
from p in context.Persons
where p.LName.StartWith("A")

// 2.
from p in context.Persons
where p.LName.EndWith("ghdam")

// 3.
from p in context.Persons
where p.LName.Contain("Aghdam")
```

همچنین برای موارد پیشرفته تر می توانید از کاراکتر های کنترلی مورد استفاده در SQL (% ، _ ، [] و |) نیز استفاده کنید. اطمینان داشته باشید که این عملگرها به عبارات معادل SQL تبدیل خواهند شد.

نکته: برای اینکه چندین شرط را بررسی کنید می تواند از کاراکتر های &&برای Andو از || برای Or استفاده کنید .

دستور IN وNot IN

از کلمه کلیدی IN در SQL برای تعیین مقدار یک فیلد به صورت دقیق استفاده می شود (می توان بیش از یک مقدار را برای فیلد ها تعیین کرد) و از کلمه کلیدی Not IN برای عملیات مخالف آن.!

```
//IN
SELECT * FROM Customer
WHERE ID IN(1,2,3,4,5)

//Not IN
SELECT * FROM Customer
WHERE ID NOT IN(1,2,3,4,5)
```

برای شبیه سازی این کلمه های کلیدی در LINQ از متد Contains استفاده می کنیم، به صورت زیر:

```
//IN
from f in context.Customer
where f.ID.Contains(1,2,3,4,5)
select f

//Not IN
from f in context.Customer
where ! f.ID.Contains(1,2,3,4,5)
select f
```

دستور Union

از دستور Union در SQL برای تلفیق دو ResultSet در قالب یک ResultSet استفاده می کردیم:

```
SELECT FName, LName FROM Person
UNION
SELECT FName, LName FROM Customer
```

عبارت معادلLINQ ، عبارت بالا را می توان بوسیله متد Union شبیه سازی کرد:

```
var q1 = from p in context.Person
             select new { p.FName , p.LName};

var q1 = from c in context.Customer
             select new { c.FName , c.LName};

var qUnion = q1.Union(q2);
```

دستورUnion All

در SQL دستور Union به صورت خودکار رکوردهای تکراری را حذف می کـرد و در LINQ متـد Union نیـز همین کار را انجام می دهد. در SQL برای جلوگیری از این موضوع از Union All استفاده می کردیم و در LINQ می توان از متد Concat برای این منظوز استفاده نمود.

عبارت LINQ قبلی را می توان به صورت زیر برای شبیه سازی دستور Union All در LINQ نوشت:

```
var q1 = from p in context.Person
             select new { p.FName , p.LName};

var q1 = from c in context.Customer
             select new { c.FName , c.LName};

var qUnion = q1.Concat(q2);
```

نکته :باید توجه داشته باشید که متدهای Union و Concat تنها می توانند بر روی انـواعی عمـل کننـد کـه خصوصیت های با نوع های یکسان داشته باشد.

دستور Group By

ماده Group By سبب می شود سطرهایی که در ستون یا ستون های مشخص شده مقادیر یکسان دارند در یک سطر ترکیب شوند. این دستور در LINQ به وسیله عملگر GroupBy پیاده سازی شده است

```
SELECT Country , Count(*) FROM Customer
GROUP BY Country
```

معادل عبارت SQL بالا در LINQ به صورت زیر است:

```
from c in Context.Customer
group c by c.Country into cc
select new { Cuntry = cc.Key , Count = cc.Count() };
```

نکته: برای Group By چند ستونی به اینجا مراجعه کنید.

دستور Order By :

از این ماده برای مرتب سازی اطلاعات استفاده می شود. در LINQ از عملگر OrderBy و OrderByDescending استفاده می شود.

1. Order By صعودی در SQL

```
SELECT FName , LName , Country FROM  customers
ORDER BY country
```

2. Order By صعودی درLINQ

```
from c in context.Customers
order by c.Country
select new {
       c.FName ,
       c.LName ,
       c.Country
}
```

3. Order By نزولی درSQL

```
SELECT FName , LName , Country FROM  customers
ORDER BY country Desc
```

4. Order By نزولی درLINQ

```
from c in context.Customers
order by c.Country descending
select new {
       c.FName ,
       c.LName ,
       c.Country
}
```

عملیاتJoin

این نوع عملگرها برای متحد کردن چند مجموعه عناصر که دارای اشتراکاتی هستند، استفاده می شود .عملگرهای اتصال در LINQ دقیقا همانند ماده های اتصال در SQL عمل می کنند. هر مجموعه عنصر و یا منبع داده ویژگی های کلیدی را دارا می باشد که بوسیله آنها می توان داده ها را مقایسه و جمع آوری نمود. به طور کلی سه روش Join وجود دارد:

1. **Inner Join**: در این نوع Join ارتباط دو جدول از طریق رابطه منطقی برقرار شده و فقط رکورد هایی در نتیجه شرکت می کنند که مقدار متناظر در هر دو جدول وجود داشته باشد: نمونه این Join درLINQ :

```
from c in customers
join o in orders on c.CustomerID equals o.CustomerID
select new {FullName =  c.Name + " " +
                c.Family ,
                c.CustomerID ,
                o.OrderDate ,
                TotalOrder = o.Total
               };
```

2. **Outer Join**: در Outer Join ارتباط منطقی بین دو جدول عنوان شده ولی بر خلاف Inner Join رکورد هایی که در جدول دیگر دارا مقدار متناظر نیستند، نیز می توانند در نتیجه ظاهر شوند . Outer Joinبه سه صورت Left ، Rightو Full انجام می شود.

نمونه Left Outer Join درLINQ

```
from p in context.Person
join pa in context.PersonAddress on p.Id equals pa.PersonId into tempAddresses
from addresses in tempAddresses.DefaultIfEmpty()
select new { p.FirstName, p.LastName, addresses.State });
```

نمونه Right Outer Join درLINQ

```
from p in context.Person
join pa in context.PersonAddress
on p.Id equals pa.PersonId into tempAddresses
from pa in tempAddresses.DefaultIfEmpty()
select new
{
  PersonName = addresses != null ? addresses.Name : null,
  DepartmentName = pa.State
}
```

نمونه Full Outer Join درLINQ

```
from p in context.Person
join pa in context.PersonAddress on p.Id equals pa.PersonId into tempAddresses
from addresses in tempAddresses.DefaultIfEmpty()
select new { p.FirstName, p.LastName, addresses.State }).Union(
        from pa2 in context.PersonAddress
        join p2 in context.Person on pa2.PersonId equals p2.Id into tempPersons
        from persons in tempPersons.DefaultIfEmpty()
        select new { persons.FirstName, persons.LastName, pa2.State })
```

3. **Cross Join:** در این نوع از Join حاصلضرب دکارتی دو جدول به عنوان نتیجه محاسبه می گردد و نیازی به رابطه منطقی نیست.

```
from p in context.Person
from pa in context.PersonAddress
select new
{
   p.FName,
   p.LName,
   pa.State
}
```