آموزش تخصصی LINQ
مقدماتی تا پیشرفته

مقدمه

به کتاب آموزشی LINQ خوش آمدید. در طول این کتاب قصد دارم تا شما را با Language Integrated Query یا LINQ و کاربردهای آن در زبان C# آشنا کنم. این ویژگی که به شما قابلیت اجرای کوئری های مختلف از داخل زبان C# را می دهد با ارئه نسخه ۳ از زبان C# معرفی و تکمیل شد.

سر فصل های این کتاب به طور خلاصه به شرح زیر می باشد:

۱. مقدمه ای بر LINQ و ویژگی های آن
۲. ویژگی های جدید C# ۳.۰
۳. ساختار کوئری های LINQ و نحوه اجرای کوئری ها
۴. آشنایی با عملگرهای استاندارد در کوئری های LINQ
   ۴.۱. عملگر های استاندارد (بخش اول)
   ۴.۲. عملگر های استاندارد (بخش دوم)
   ۴.۳. عمبگر های استاندارد (بخش سوم)
   ۴.۴. عملگر های استاندارد (بخش چهارم)
۵. LINQ to XML
۶. LINQ to ADO.NET
   ۶.۱. آشنایی با مفاهیم Object Relational Mapping و LINQ to SQL
   ۶.۲. LINQ to Dataset
   ۶.۳. ADO.NET Entity Framework و LINQ to Entities
۷. LINQ to Provider
۸. پروژه عملی

تمامی سرفصل هایی که در بالا ذکر شد در طول این کتاب مورد بحث قرار می گیرند، اما به طور خلاصه در هر کدام از این بخش ها چه مطالبی مطرح خواهد شد؟

۱. در این بخش با ایجاد یک پروژه ساده مروری کوتاه بر یک کوئری ساده LINQ خواهیم داشت. همچنین با نحوه نوشتن کوئری ها قبل از ارائه LINQ آشنا خواهیم شد. در انتهای این بخش نیز با انواع Provider های موجود برای LINQ به طور خیلی خلاصه آشنا خواهیم شد.
۲. بدلیل آنکه بسیاری از ویژگی های جدید معرفی شده در C# نسخه ۳، جهت ایجاد کوئری های LINQ است، در این بخش با این ویژگی ها بیشتر آشنا خواهیم شد تا بعد ها بتوانیم از آنها در نوشتن کوئری های پیچیده تر استفاده کنیم.
۳. در این بخش با ساختار کلی کوئری های LINQ و همچنین نحوه اجرای کوئری ها آشنا خواهیم شد.
۴. در این بخش اپراتورهای استاندارد LINQ را مورد بررسی قرار خواهیم داد، برای مثال اپراتور های مربوطه برای فیل تر کردن اطلاعات، مرتب سازی، گروه بندی و ... .
۵. در این بخش ابتدا مقدمه کوتاهی بر نحوه استفاده از XML در C# خواهیم داشت و سپس با کوئری های LINQ to XML که قابلیت گرفتن کوئری از مستندات XML را به ما می دهد آشنا خواهیم شد.
۶. می توان گفت که این بخش اصلی ترین بخش مورد بحث ما خواهد بود و بخش زیادی از مقاله به این بخش اختصاص خواهد داشت. در این بخش با قابلیت اجرای کوئری های LINQ بر روی بانک های SQL Server آشنا خواهیم شد.
   ۶.۱. در این بخش ابتدا با مفاهیم ORM آشنا خواهیم شد و سپس به مبحث LINQ to SQL خواهیم پرداخت که قابلیت اجرای کوئری ها بر روی بانک های SQL را به ما می دهد.

۶.۲. این قسمت به مفاهیم LINQ to Dataset اختصاص دارد. Dataset ها نمایشی از جداول بانک اطلاعاتی در حافظه هستند که می از کوئری های LINQ برای استخراج داده ها از Dataset ها هم استفاده کرد.
۶.۳. این بخش به ADO.NET Entity Framework اختصاص دارد، LINQ to Entities چیزی شبیه به LINQ to SQL است، با این تفاوت که به جای اینکه با لایه ای فیزیکی از داده سر و کار داشته باشد، از لایه ای مفهومی استفاده میکند که در این بخش به طور مفصل به این مورد خواهیم پرداخت.

۷. بخش بعدی این مقاله به مبحث ایجاد Provider دلخواه برای کوئری های LINQ اختصاص دارد که یکی از مباحث پیچیده و پیشرفته LINQ می باشد. برای مثال در LINQ to SQL، کلمه SQL نشان دهنده Provider مربوطه برای کوئری های لینک است که از این Provider برای بانک های SQL Server استفاده می شود . در این بخش با نحوه ایجاد یک Provider دلخواه برای کوئری های LINQ آشنا خواهیم شد.
۸. در بخش هشتم این مقاله به ایجاد یک پروژه عملی خواهیم پرداخت تا با ویژگی های کاربردی LINQ بهتر آشنا شویم. این پروژه بر اساس ساختار سه لایه پیاده سازی خواهد شد.

در طول این کتاب آشنایی با موارد زیر برای درک مطالب ضروری می باشد:

۱. مقدمات زبان C#
۲. برنامه نویسی شئ-گرا و مفاهیم آن
۳. جنریک ها و کلاس های مجموعه ای
۴. اینترفیس ها
۵. طراحی و پیاده سازی بانک های اطلاعاتی در SQL Server

تمام کدهای ارائه شده در این کتاب با زبان C# می باشند.

فصل اول :: مقدمه ای بر LINQ و ویژگی های آن

بخش اول سری مقالات آموزش LINQ رو شروع میکنیم . در این بخش مقدمه ای بر LINQ خواهیم داشت و همچنین با قابلیت ها و توانایی های آن به صورت مختصر آشنا خواهیم شد. شاید اولین سوالی که در ذهن هر یک از شما پیش بیاد این باشه که چه تعریفی می توان برای LINQ ارائه داد؟ چه کارهایی می توان با LINQ انجام داد؟ LINQ چه قابلیت هایی را در اختیار یک برنامه نویس قرار می دهد؟ به این سوالات می شود به این صورت جواب داد که LINQ شامل یکسری عملگرهای استاندارد است که به شما امکان می دهد بر روی انواع منابع داده در داخل زبان های سازگار با .NET. مانند VB.NET یا C# کوئری هایی را نوشته و اجرا کنید . نام Language Integrated Query نیز به این دلیل انتخاب شده است که این کوئری ها داخل یک زبان برنامه نویسی مانند C# نوشته و اجرا می شوند . اما منابع داده ای که LINQ می تواند از آنها استفاده کند می تواند یک شئ ایجاد شده، یک فایل مستندات XML، یک بانک SQL Server و یا یک منبع دلخواه باشد. بنابراین LINQ این قابلیت را دارد که با تمامی این منابع داده کار کند . علاوه بر استخراج و اجرای کوئری بر روی منابع داده، بوسیله LINQ شما امکان تغییر و دستکاری یک منبع داده مانند یک بانک SQL Server را خواهید داشت. برای شروع با یک مثال کوچک شروع میکنیم:

فرض کنید که یک آرایه از نوع رشته داریم که یکسری نام را داخل آن قرار دادیم، حالا می خواهیم نام هایی که با حروف Mo شروع می شوند را جستجو کرده و به کاربر نمایش دهیم . برای اینکار باید یک آرایه ایجاد کرده و با یک دستور foreach بر روی تک تک عناصر آرایه عملیات مقایسه را انجام داده و نتیجه را به کاربر نشان دهیم:

```
string[] names = { "Ali", "Hasan", "Mojtaba", "Hamid", "Morteza", "Reza",
"Mohammad" };
foreach (string name in name)
    if (name.StartsWith("Mo"))
        Console.WriteLine(name);
```

با اجرای کد بالا تمامی نام هایی که با حروف Mo شروع می شوند در خروجی چاپ خواهند شد. اما اینکار را می توان با نوشتن یک کوئری بسیار ساده LINQ نیز انجام داد. قطعه کد زیر یک کوئری LINQ است که نام هایی که با Mo شروع می شوند را برای ما استخراج کرده و بر روی خروجی نمایش میدهد:

```
string[] names = { "Ali", "Hasan", "Mojtaba", "Hamid", "Morteza", "Reza",
"Mohammad" };
IEnumerable<string> query = from n in names
                            where n.StartsWith("Mo")
                            select n;
foreach (string name in query)
    Console.WriteLine(name);
```

در کد بالا دستوراتی وجود دارد که شاید با آنها نا آشنا باشید، اما نگران نباشید، در ادامه این مقاله با تک تک دستورات بالا و کاربرد هر یک آشنا خواهید شد . اما مهمترین چیزی که در دستورات بالا توجه یک برنامه نویس را به خود جلب میکند، شباهت ساختار کوئری نوشته شده با کوئری های SQL است. وقتی این کوئری اجرا می شود یک شئ از نوع اینترفیس IEnumerable<string> که یک اینترفیس جنریک است ایجاد شده و نتایج داخل آن ذخیره می شود. این اینترفیس در داخل فضای نام System.Collections.Generic قرار دارد و قابلیت حرکت در میان نتایج کوئری را بوسیله دستور foeach فراهم می آورد (Iterator ها). برای آشنایی با Iterator ها و کلاسهای جنریک میتوانید مقالات زیر را مطالعه کنید:

Generics :: حمید قادر
Iterator ها :: حسین احمدی

برای مقایسه عملگرهای LINQ با دستوراتی که قبلا" برای گرفتن کوئری ها از اشیاء استفاده می شد، به سراغ کلاس List<T> می رویم. مثال بعدی را با نوشتن یک کلاس به نام Person شروع میکنیم:

```
public class Person
{
    public Person() { }
    public Person(string firstName, string lastName, int age)
    {
        this.FirstName = firstName;
        this.LastName = lastName;
        this.Age = age;
    }

    public string FirstName { get; set; }
    public string LastName { get; set; }
    public int Age { get; set; }

    public override string ToString()
    {
        return string.Format("FirstName: {0}\n{LastName: {1}\nAge: {2}",
this.FirstName,
                    this.LastName, this.Age);
    }
}
```

کلاس بالا شامل سه فیلد، دو متد سازنده و متد override شده ToString است. در قدم بعدی ما یک لیست از اشیاء Person ایجاد میکنیم. (List<T> یک شئ جنریک است که در فضای نام System.Collections.Generic قرار دارد)

```
List<Person> persons = new List<Person>()
{
    new Person() { FirstName = "Ali", LastName = "Jamali", Age = 42 },
    new Person() { FirstName = "Hosein", LastName = "Ahmadi", Age = 23 },
    new Person() { FirstName = "Mohamamd", LastName = "Sadeghi", Age = 17
},
    new Person() { FirstName = "Reza", LastName = "Jamali", Age = 24 },
    new Person() { FirstName = "Nima", LastName = "Karami", Age = 18 }
};
```

حال می خواهیم از مجموعه بالا تمامی افرادی که سنشان زیر ۲۰ سال است را انتخاب کرده و در خروجی نمایش دهیم. کلاس List<T> متدی دارد به نام FindAll که برای پارامتر ورودی آدرس تابعی را میگیرد که عملیات مقایسه ما را بر روی شئ Person انجام می دهد و سپس نتیجه را بر میگرداند. در اینجا ما از متد FindAll و Anonymous Methods برای فرستادن کد مورد نظر برای مقایسه سن اشخاص به تابع FindAll استفاده میکنیم:

```
List<Person> query = persons.FindAll(delegate(Person p) { return p.Age <
20; });
foreach (Person person in query)
{
    Console.WriteLine(person.ToString());
```

```
}
```

با اجرای کد بالا، متد FindAll شئ جدیدی از نوع List<Person> برگردانده و ما می توانیم با یک دستور foreach نتایج را در خروجی چاپ کنیم . اما کار بالا را می توان با یک کوئری LINQ به سادگی انجام داد:

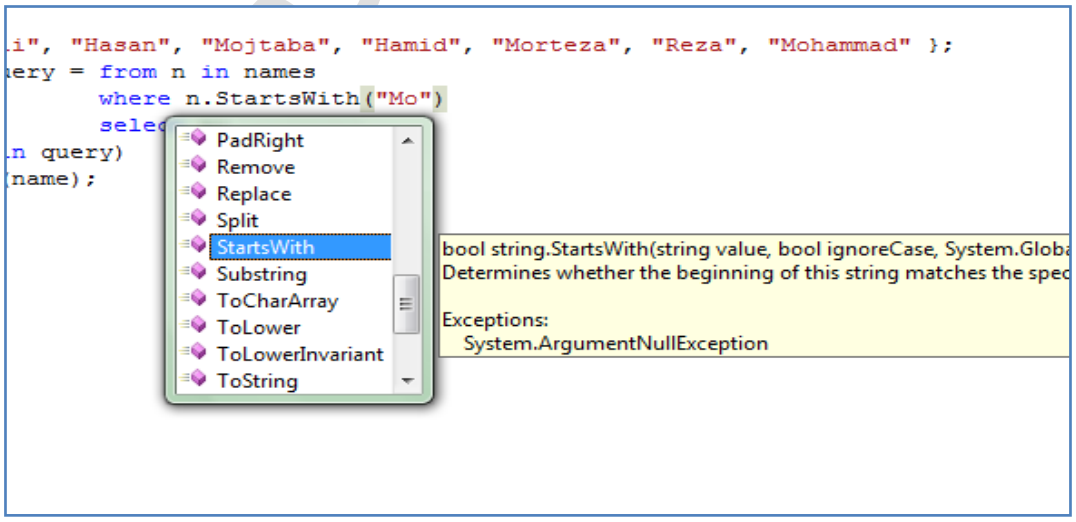
```
IEnumerable<Person> query = from p in persons
                            where p.Age < 20
                            select p;
foreach (Person person in query)
{
    Console.WriteLine(person.ToString());
}
```

حال اگر بخواهیم شرط را تغییر بدهیم، بطوریکه علاوه بر سن افراد، نا م آنها نیز مورد مقایسه قرار بگیرد کافیست کوئری خود را به صورت زیر تغییر دهیم:

```
IEnumerable<Person> query = from p in persons
                            where p.Age < 20 && p.FirstName.StartsWith("M")
                            select p;
```

تا اینجای مقاله در کوئری هایی که نوشتیم تنها از عملگر where برای فیل تر سازی اطلاعات استفاده کردیم، در LINQ عملگرهای دیگری برای مرتب سازی نتایج، گروه بندی، ادغام و ... وجود دارد که در بخش های بعدی این مقاله به تفصیل در مورد آنها صحبت خواهیم کرد.

اما یکی از قابلیت های جالب در نوشتن کوئری های LINQ که در محیط Visual Studio 2008 به چشم می آید قابلیت IntelliSense در نوشتن Query ها است که نوشتن آن ها را بسیار آسان می کند. همانگونه که شما وقتی شئ ای از نوع string ایجاد می کنید و می توانید به وسیله IntelliSense به تمامی متدهای آن دسترسی داشته باشید، در کوئری های LINQ نیز عملیات به همین گونه است . شما در هنگام نوشتن کوئری ها به تمامی متد هایی که برای یک شئ تعریف شده است دسترسی دارید مانند متد StartWith کلاس string که در کوئری های بالا از آن استفاده کردیم.



در ابتدای مقاله گفتیم که LINQ قابلیت کار با چندین منبع داده را دارد که در زیر به طور خلاصه به بررسی هر یک از این منابع می پردازیم:

۱. LINQ to Object: قابلیت اجرای کوئری ها بر روی اشیاء موجود در حافظه را برای ما فراهم میکند. مانند کوئری بالا که بر روی لیست افراد List<Person> اجرا کردیم. این قابلیت بر روی تمامی کلاس های مجمو عه ای که اینترفیس IEnumerable را پیاده سازی کرده باشند قابل اجرا است.

۲. LINQ to XML: قابلیت اجرای کوئری ها بر روی مستندات XML را فراهم میکند. همچنین عملگرهایی را در اختیار شما قرار میدهد که به XPath وابسته بوده و به شما این امکان را می دهد در داخل مستندات XML حرکت کرده و گره های مختلف را مورد پیمایش قرار دهید . پشتیبان LINQ to XML از کلاس های فضای نام System.Xml و System.Xml.Linq استفاده میکند. مخصوصا" Reader و Writer برای اجرای کوئری های. دو کلاسی که نقش مهمی را در LINQ to XML بازی می کنند، کلاس های XElement و XAttribute هستند. کلاس XElement نمایش دهنده المان های XML هستند و LINQ to XML از آنها برای ایجاد گره های مربوط به المان و فیلتر سازی اطلاعات استفاده می کند. هر XElement شامل یکسری صفات برای هر المان XML است که این صفات بوسیله کلاس XAttribute مشخص می شوند. مثال های زیر خلاصه ای از نحوه استفاده از امکانات LINQ to XML را نشان می دهد:

کاربرد کلاس XElement را در کد زیر مشاهده میکنید:

```
XElement x = new XElement(
    "Person",
    new XElement("FirstName", "Hosein"),
    new XElement("LastName", "Ahmadi"));
```

حاصل قطعه کد بالا تگ های XML ایجاد شده زیر است:

```
<Person>
  <FirstName>Hosein</FirstName>
  <LastName>Ahmadi</LastName>
</Person>
```

حال اگر بخواهیم از XAttribute نیز در کلاس XElement استفاده کنیم، کد بالا را به صورت زیر تغییر می دهیم:

```
XElement x = new XElement("Person",
    new XAttribute("PersonID", "10"),
    new XElement("FirstName", "Hosein"),
    new XElement("LastName", "Ahmadi"));
```

حال المان ایجاد شده توسط کد بالا به صورت زیر خواهد بود:

```
<Person PersonID="10">
  <FirstName>Hosein</FirstName>
  <LastName>Ahmadi</LastName>
</Person>
```

اما قدرت اصلی LINQ to XML در این قسمت معلوم می شود که میتوانیم پارامتر های ورودی جهت قرار گیری در المان XML تعریف شده را توسط یک Query انتخاب کنیم:

```
string[] names = { "Hosein", "Hamid", "Reza", "Mohammad", "Hadi" };
XElement x = new XElement("SelectedNames",
    from n in names
    where n.StartsWith("H")
    select new XElement("Name", n));
```

المان ایجاد شده توسط کد بالا به صورت زیر است:

```
<SelectedNames>
  <Name>Hosein</Name>
  <Name>Hamid</Name>
  <Name>Hadi</Name>
</SelectedNames>
```

در بخش های بعدی مقالات به طور مفصل در باره LINQ to XML صحبت خواهیم کرد.

۳. LINQ to ADO.NET: این بخش در رابطه با اجرای کوئری های LINQ بر روی پایگاه های داده مانند SQL Server و همچنین دستکاری داده های آنان است. خود این بخش را می توان به سه بخش LINQ to SQL، LINQ to Dataset و LINQ to Entities تقسیم بندی کرد که هر کدام مبحث کاملی را می طلبند . به دلیل اینکه حجم مطالب مربوط به این بخش بسیار زیاد بوده و همچنین نیاز به آشنایی با مفاهیم پایه ای مانند ORM را دارد، توضیحات کامل در بخش مربوطه مطرح خواهند شد. تنها موردی که در اینجا می توان به آن اشاره کرد این است که در LINQ to SQL و LINQ to Entities برای موجودیت های بانک (جداول) در داخل برنامه اشیاء ای ایجاد می شوند که این اشیاء به واسطه یکسری Attribute ها توانایی تقابل با بانک و جداول مربوطه را دارند . کوئری های که برای خواندن اطلاعات از جدول به صورت کوئری های LINQ نوشته می شوند، به کوئری های SQL تبدیل شده و به سرویس SQL فرستاده می شوند تا نتایج مورد نظر گرفته شده و به کاربر نمایش داده شود . برای مثال کلاس زیر نمایش معادل یک جدول از بانک SQL است که در زبان C# بصورت یک کلاس تعریف شده است:

```
[Table(Name = "Person.Contact")]
public class Contact
{
    [Column(DBType = "nvarchar(50) not null")]
    public string FirstName;
    [Column(DBType = "nvarchar(50) not null")]
    public string LastName;
    [Column(DBType = "nvarchar(50) not null")]
    public string EmailAddress;
}
```

به استفاده از Attribute ها برای Map کردن جدول به صورت یک شئ توجه کنید . بوسیله LINQ to SQL می توان از روی این شئ بر روی جدول معادل در بانک اطلاعاتی کوئری هایی را اجرا کرد و یا رکورد هایی را دستکاری کرد.

مطالبی که تا اینجا بیان شد، خلاصه ای از توانایی ها و ویژگی های LINQ بود که در ادامه این مقاله با تفصیل در مورد هر یک از این ویژگی ها بحث خواهیم کرد . در بخش بعد در مورد ویژگی های جدید C# ۳.۰ که کاربرد های فراوانی در LINQ دارند بحث خواهیم کرد . این ویژگی ها به شرح زیر هستند:

۱. Implicitly Typed Variables

۲. Anonymous Types
۳. Initialization Object
۴. Lambda Expressions
۵. Properties Automatic Implemented
۶. Extension Methods

فصل دوم :: آشنایی با ویژگی های جدید C# نسخه ۳.۰

در ادامه مقالات آموزشی LINQ به بررسی ویژگی های جدید C# ۳.۰ خواهیم پرداخت و در بخش های بعدی با نحوه استفاده از این ویژگی ها در کوئری LINQ آشنا خواهیم شد . به دلیل اینکه مقالاتی در این باره داخل سایت وجود دارد، فقط در مورد Lambda Expressions به صورت کامل توضیح خواهیم داد و بر روی سایر ویژگی ها به صورت خیلی مختصر بحث خواهیم کرد. ویژگی هایی که در این مقاله بررسی خواهیم کرد به شرح زیر است:

۱. Implicitly Typed Variables
۲. Expressions Lambda
۳. Extension Methods
۴. Anonymous Types

۱. اولین ویژگی که میخواهیم در مورد آن بحث کنیم ویژگی Implicitly Typed Variables است. این ویژگی به شما این امکان رو میده تا انتخاب نوع داده تعریفی داخل کد را به عهده کامپایلر بزارید. برای مثال قطعه کد زیر رو در نظر بگیرید،

```
int i = 25
```

یک متغییر int تعریف کردیم و مقدار آن را مساوی ۲۵ قرار دادیم، در C# نسخه ۳.۰ با استفاده از کلمه کلیدی var انتخاب نوع متغییر را به عهده کامپایلر بزاریم . انتخاب نوع بر اساس مقداری که داخل متغییر در هنگام تعریف رخته می شود (Initialization) انتخاب می شود:

```
var i = 12;
var s = "Welcome";
var d = 2.6;
```

در کد های بالا، متغیر i از نوع int، متغیر s از نوع string و متغیر d از نوع double در نظر گرفته می شود. فقط نکته ای که باید در نظر بگیرید این است که هنگام استفاده از کلمه var باید متغییر مقدار دهی اولیه شود. در غیر این صورت با پیغام خطا مواجه خواهید شد.

۲. ویژگی بعدی Lambda Expressions هستند، این عبارات چیزی معادل Anonymous methods در نسخه های قبلی C# هستند. در بخش قبلی درباره متد FindAll کلاس جنریک List<T> صحبت کردیم،. این متد آدرس یک متد برای مقایسه عناصر لیست را میگیرد. این کار در نسخه های قبلی در دو حالت امکان پذیر بود . یکی تعریف یک متد و ارسال متد به عنوان پارامتر به متد FindAll:

```
static void Main(string[] args)
{
    List<int> list = new List<int>() { 7, 9, 3, 1, 5, 6, 7 };
    List<int> query = list.FindAll(Compare);
    foreach (int i in query)
        Console.WriteLine(i);
}

static bool Compare(int a)
{
    return a > 5;
}
```

همانطوری که در کد بالا مشاهده میکنید، متدی با عنوان Compare تعریف شده و به عنوان پارامتر به متد FindAll جهت مقایسه فرستاده می شود . اما روش بعدی استفاده از Anonymous Methods که با استفاده از کلمه کلیدی delegate می توان از آن استفاده کرد:

```
static void Main(string[] args)
{
    List<int> list = new List<int>() { 7, 9, 3, 1, 5, 6, 7 };
    List<int> query = list.FindAll(delegate(int a) { return a > 5; });
    foreach (int i in query)
        Console.WriteLine(i);
}
```

کاربرد دیگو Anonymous Methods در استفاده از Event ها است که میتوان کد مربوط به یک Event را با استفاده از Anonymous Methods به صورت درجا تعریف کرد:

حالت اول:

```
public Form1()
{
    InitializeComponent();
    button1.Click += Button1_Click;
}

public void Button1_Click(object sender, EventArgs e)
{
    MessageBox.Show("Welcome to C# world");
}
```

و با استفاده از Anonymous Methods:

```
public Form1()
{
    InitializeComponent();
    button1.Click += delegate(object sender, EventArgs e)
    {
        MessageBox.Show("Welcome To C#");
    };
}
```

اما از Lambda Expressions چگونه می توان استفاده کرد . عبارات Lambda همان کار Anonymous Methods را انجام می دهند با این تفاوت که دیگر نیازی به تعیین نوع پارامترهای ورودی تابع مورد نظر نیست! برای روش تر شدن قضیه به همان مثال متد FindAll توجه کنید، در کد زیر ما همان کد را با عبارات Lambda پیاده سازی میکنیم:

```
List<int> query = list.FindAll(delegate(int a) { return a > 5; }); //
anonymous methods with delegate keyword
List<int> query = list.FindAll((n) => { return n > 5; }); // anonymous
methods with Lambda Expressions
```

در کد بالا، در خط اول با استفاده از کلمه کلیدی delegate و در خط دوم با استفاده از Lambda Expressions، متد بی نام مورد نظر را پیاده سازی کردیم . تفلوت موجود در ساختار ها، عدم نیاز به تعیین نوع پارامتر های ورودی متد و استفاده از عملگر => برای نشان دادن عبارات Lambda می باشد.

مثال دیگر همان تعیین متد مورد نظر برای ایونت Click کنترل Button است که در کد زیر با استفاده از عبارات Lambda آن را پیاده سازی کردیم:

```
button1.Click += (sender, e) =>
{
    MessageBox.Show("Welcome To C#");
};
```

نکات:

* در صورتی که تعداد پارامتر های ورودی یکی باشد، می تواند پرانتز ها را حذف کرد، اما در صوروتی که تعداد پارامترها بیش از یکی باشد، باید آنها را داخل پرانتز قرار داد.

```
List<int> query = list.FindAll(n => { return n > 5; });

button1.Click += (sender, e) =>
{
    MessageBox.Show("Welcome To C#");
};
```

* در صورتی که تعداد دستورات داخل بدنه متد تنها یک دستور باشد، می توان از نوشتن {} خودداری کرد:

```
button1.Click += (sender, e) => MessageBox.Show("Welcome To C#");
```

* در صورتی که متد مورد نظر، مقداری را برگرداند (مانند متد FindAll) و تعداد دستورات داخل بدنه متد تنها یک دستور باشد که حاصل را برمیگرداند، با حذف {} ها باید کلمه کلیدی return را نیز حذف کرد:

```
List<int> query = list.FindAll(n => n > 5); // valid
List<int> query = list.FindAll(n => return n > 5); // wrong
```

یکی از کاربرد های مهم عبارات Lambda در نوشتن کوئری های LINQ است که در بخش های بعدی مقالات با آنها بیشتر آشنا خواهیم شد.

۳. یکی دیگر از ویژگی های جدید نسخه ۳ زبان C#، متدهای توسعه یا Extension Methods هستند. این قابلیت به شما این امکان را می دهد تا به کلاس های موجود، متد های مورد نیاز خودتان را اضافه کنید. برای مثال، نوع داده int را فرض کنید. می خواهیم به این نوع داده، متدی با نام Negative را اضافه کنیم تا منفی عددی که داخل آن ذخیره شده است را برای ما برگرداند. برای این کار باید یک کلاس static ایجاد کرده و داخل آن متد های توسعه مورد نظر خود را بنویسیم. توجه کنید که خود متد توسعه نیز باید از نوع int باشد. برای درک بهتر این موضوع به کد زیر توجه کنید:

```
class Program
{

    static void Main(string[] args)
    {
        int myInt = 25;
        Console.WriteLine(myInt.Negative());
    }
```

```
}

static class Extensions
{
    public static int Negative(this int i)
    {
        return -i;
    }
}
```

با قطعه کد بالا، در داخل کلاس Extensions، ما یک متد با نام Negative برای نوع داده int تعریف کردیم. ساختار نوشتن متدهای Extensions مانند نوشتن سایر متدها می باشد، با این تفاوت که اولین پارامتر ورودی این تابع، نشان دهنده یک Instance از نوع داده ای است که می خواهیم متد مورد نظر به آن اضافه شود . همچنین این متد ها همیشه باید به صورت Static نوشته شوند. در بخش بالای کد و داخل متد Main ما از این متد استفاده کردیم. توجه کنید که متدهای توسعه تنها از طریق Instance های ایجاد شده از روی یک نوع داده قابل دستری هستند. نحوه نمایش متد های توسعه با سایر متدها، در IntelliSence محیط VS فرق دارد. در کنار این متد ها، یک فلش آبی رنگ به سمت پایین وجود دارد که نشان دهنده یک متد توسعه است.

۴. ویژگی بعدی که در مورد آن صحبت خواهیم کرد نوع های بی نام یا Anonymous Types هستند. این نوع ها با کمک کلمه کلیدی var تعریف شده و می توانند دارای یکسری خصوصیات باشند. برای مثال، می توانیم یک نوع بدون نام که دارای خصوصیات FirstName و LastName باشد را تعریف کنیم، بدون اینکه نیازی به تعریف یک کلاس جدید باشد . به مثال زیر توجه کنید:

```
var myAnonymousType = new { FirstName = "Hosein", LastName = "Ahmadi" };
Console.WriteLine(myAnonymousType.FirstName + myAnonymousType.LastName);
```

با استفاده از کد بالا یک نوع بدون نام ایجاد شد . انتخاب نام این نوع به عهده کامپایلر است . اگه به کد IL ایجاد شده توسط کامپایلر نگاهی بندازیم می بینیم که خود کامپایلر نامی برای این نوع در نظر میگیره:

```
.method private hidebysig static void Main(string[] args) cil managed
{
    .entrypoint
    .maxstack 3
    .locals init (
        [0] class <>f__AnonymousType0`2<string, string> myAnonymousType)
    L_0000: nop
    L_0001: ldstr "Hosein"
    L_0006: ldstr "Ahmadi"
    L_000b: newobj instance void <>f__AnonymousType0`2<string,
string>::.ctor(!0, !1)
    L_0010: stloc.0
    L_0011: ldloc.0
    L_0012: callvirt instance !0 <>f__AnonymousType0`2<string,
string>::get_FirstName()
    L_0017: ldloc.0
    L_0018: callvirt instance !1 <>f__AnonymousType0`2<string,
string>::get_LastName()
    L_001d: call string [mscorlib]System.String::Concat(string, string)
    L_0022: call void [mscorlib]System.Console::WriteLine(string)
    L_0027: nop
    L_0028: ret
```

```
}
```

نکته قابل توجه دیگر اینجاست که اگه دو نوع بدون نام دارای خواص یکسان باشند، می توانند به صورت Implicit به یکدیگر Cast شوند، مثال:

```
var myAnonymousType1 = new { FirstName = "Hosein", LastName = "Ahmadi" };
var myAnonymousType2 = new { FirstName = string.Empty, LastName =
string.Empty };
myAnonymousType2 = myAnonymousType1;
Console.WriteLine(myAnonymousType2.FirstName + myAnonymousType2.LastName);
```

نوع های بدون نام کابردهای زیادی در کوئری های LINQ دارن که در بخش های بعدی با آ نها آشنا خواهیم شد.

این چهار ویژگی، مهمترین ویژگی هایی بودند که در کوئری های LINQ کاربرد دارند، چندین ویژگی جدید دیگر نیز به نسخه ۳ زبان C# اضافه شده اند که در اینجا تنها به ذکر نام آنها اکتفا کرده و دوستان می توانند برای توضیحات اضافی به مقاله هایی که دو ستان دیگر داخل سایت قرار دادند مراجعه کنند:

۱. Implemented Properties Automatic
۲. Object Initialization
۳. Initialization Collection

در بخش بعدی مقالات آموزشی LINQ، در مورد ساختار کوئری های LINQ و همچنین نحوه اجرای کوئری ها صحبت خواهیم کرد.

فصل سوم :: ساختار کوئر های LINQ و نحوه اجرای کوئری ها

بخش سوم آموزش LINQ رو با هم شروع میکنیم . در این بخش از سری مقالات آموزشی، درباره ساختار کوئری های LINQ و همچنین نحوه اجرای کوئری های صحبت خواهیم کرد.

مباحث این بخش به شرح زیر می باشند:

۱. ساختار کوئری های LINQ و اجزای تشکیل دهنده هر کوئری
۲. استفاده از ساختار کوئری و ساختار متد در نوشتن کوئری ها
۳. استفاده از Anonymous Types در نوشتن کوئری های LINQ
۴. آشنایی با مفهوم Deferred Query Execution (نحوه اجرای کوئر ها)

خوب، تو اولین مرحله می خواییم با ساختار و اجزای تشکیل دهنده یک کوئری آشنا بشیم . اگه بخواییم ساختار کلی یک کوئری LINQ رو نمایش بدیم، ساختار یک کوئری LINQ به صورت زیر می باشد:

```
from <element> in <data_source>
[query standard operators]
select <element type in returned sequence>
```

همانطور که در بالا مشاهده میکنید کوئری های LINQ از بخش های مختلفی تشکیل شده اند. (قسمت هایی که داخل <> قرار گرفته توسط کاربر تعیین شده و اجباری می باشند و قسمت هایی که داخل [] قرار گرفته اند اختیاری می باشند، باقی اجزاء که داخل نشانه ای قرار نگرفته اند بخش های پیش فرض کوئری های LINQ هستند)

from: تمامی کوئری های LINQ با کلمه کلیدی from آغاز می شوند. این کلمه نشان دهنده آغاز یک کوئری LINQ است.

<element>: این قسمت نمایش دهنده یکی از عناصر موجود در source data ایست که می خواهیم از آن کوئری بگیریم . انتخاب نام برای این قسمت آزاد است و می توان هر نامی را برای آن انتخاب کرد . برای مثال اگر لیستی داشته باشیم که عناصر موجود در آن از نوع int باشند، زمان اجرای کوئری element از نوع int در نظر گرفته خواهد شد.

in: بخش بعدی یک کوئری LINQ کلمه کلیدی in است و نوشتن آن در تمام کوئری های LINQ اجباریست، بوسیله این کلمه که قبل از نام data_source می آید، تعیین می کنیم که کوئری بر روی کدام منبع داده باید اجرا شود.

<data_source>: در این بخش نام منبع داده ای که می خواهیم کوئری بر روی آن اجرا شود را می نویسیم.

<query standard operators>: این بخش حاو ی تمامی operator های استاندارد LINQ است که برای عملیات فیل تر کردن اطلاعات، مرتب سازی، گروه بندی و یا ادغام از آنها استفاده می شوند. درباره اپراتور های استاندارد LINQ در بخش بعدی صحبت خواهیم کرد.

select: این کلمه کلیدی بعد از بخش اپراتور های استاندارد می آید و نوشتن آن در تمام کوئری های LINQ اجباری است. این کلمه نشان دهنده به پایان رسیدن یک کوئری LINQ است و وظیفه انتخاب عناصر مورد نظر از منبع داده را دارد.

<element type in returned sequence>: این قسمت نوع عناصر موجود در لیست ایجاد شده توسط کوئری را تعیین میکند، می توان نوع خروجی های کوئری را به طور دلخواه تعیین کرد.

برای درک بهتر هر یک از بخش های بالا با یک مثال جلو میریم:

```
int[] nums = { 4, 1, 7, 5, 9, 6, 2 };

var query = from n in nums
            where n > 4
            select n.ToString();

foreach (string s in query)
    Console.Write(s);
```

همانطور که گفته شد، کوئری با کلمه کلیدی from آغاز می شو، n در کوئری بالا نشان دهنده یکی از عناصر موجود در منبع داده است که از آن برای عملیات های فیل تر کردن استفاده می کنیم، بعد از n، کلمه کلیدی in و بعد از آن منبع داده مورد نظر است که در این کوئری منبع داده مورد نظر ما آرایه nums است. در بخش بعدی از اپراتور استاندارد where که برای فیل تر کردن اطلاعات استفاده می شود، استفاده کردیم، برای فیل تر سازی اطلاعات، از n که در خط اول کوئری تعریف کردیم استفاده کردیم، در خط بع دی هم بعد از کلمه کلیدی select نوع عناصر موجود در query را تعیین می کنیم . در اینجا با استفاده از متد ToString تعیین کردیم که عناصر موجود در کوئری از نوع string باشند. اگر به جای n.ToString()، تنها n را می نوشتیم، عناصر موجود در لیست برگردانده شده توسط کوئر ی از نوع int در نظر گرفته می شد.

گرفتن کوئری از منابع داده به دو صورت امکان پذیر است:

۱. استفاده از کوئری های LINQ
۲. استفاده از متدهای کوئری و Expressions Lambda

در مورد روش اول در بخش های قبلی صحبت کردیم . در روش دوم از Extension Method هایی استفاده می کنیم که در net. نسخه ۳.۵ اضافه شده اند. استفاده از این Method ها برای نوشتن کوئری انعطاف پذیری بیشتری رو در نوشتن کوئری ها به ما میدن. نوشتن کوئری بالا با استفاده از متد های کوئری به صورت زیر است:

```
int[] nums = { 4, 1, 7, 5, 9, 6, 2 };
var query = nums.Where(n => n > 4).Select(n => n.ToString());

foreach (string s in query)
    Console.Write(s);
```

اما گفتیم که استفاده از متد ها در نوشتن کوئری انعطاف پذیری بیشتری دارند .یک نمونه از این انعطاف پذیری انتخاب ایندکس های دلخواه در کوئری گرفته شده است . برای مثال فرض کریم بخواهیم در کوئری گرفته شده، فقط ایندکس های زوج را برگردانیم:

```
int[] nums = { 4, 1, 7, 5, 9, 6, 2 };
var query = nums.Where((n, index) => n > 4 && index % 2 == 0).Select(n =>
n.ToString());

foreach (string s in query)
```

```
    Console.Write(s);
```

پارامتر دومی که داخل متد where با نام index تعریف شده، نشان دهنده ایندکس مربوط به هر یک از عناصر نتیجه برگردانده شده از شرط قبل از شر index است. در ایجاد تمامی ایندکس های زوج برگردانده می شوند . متدهایی که در نوشتن کوئری استفاده می شوند پارامتر های ورودی دیگری نیز دارند که برای آشنایی با آنها می توانید از MSDN استفاده کنید. در بخش های بعدی مقاله با استفاده از این متدها بیشتر آشنا خواهیم شد.

اما امکان بعدی که در مورد آن می خواهیم بحث کنیم قابلیت استفاده از Anonymous Types در نوشتن کوئری ها است. همانطور که در بخش قبلی گفتیم Anonymous Types نوع هایی بی نام هستند که می توانند دارای یکیسری خصوصیات باشند . در کوئری های LINQ میتوانیم تعیین کنیم که خروجی ما از نوع Anonymous Types باشند. به مثال زیر توجه کنید، می خواهیم خروجی کوئری نوعی که تنها حاوی خصوصیت FullName است باشد، FullName از ترکیب خصوصیات FirstName و LastName در شئ Person تشکیل شده است (در کد زیر از کلاس Person که در بخش اول ایجاد کردیم استفاده می کنیم):

```
List<Person> persons = new List<Person>()
{
    new Person() { FirstName = "Ali", LastName = "Jamali", Age = 42 },
    new Person() { FirstName = "Hosein", LastName = "Ahmadi", Age = 23 },
    new Person() { FirstName = "Mohamamd", LastName = "Sadeghi", Age = 17
},
    new Person() { FirstName = "Reza", LastName = "Jamali", Age = 24 },
    new Person() { FirstName = "Nima", LastName = "Karami", Age = 18 }
};

var query = from p in persons
            where p.Age > 20
            select new { FullName = p.FirstName + " " + p.LastName };

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

نوشتن کوئری بالا با استفاده از متدهای کوئری نیز امکان پذیر است:

```
var query = persons.Where(p => p.Age > 20)
        .Select(n => new { FullName = n.FirstName + " " + n.LastName });
```

اما نحوه اجرای کوئری ها در LINQ به چه صورت است . وقتی شما داخل کد برنامه ای که نوشتید از کوئری LINQ استفاده کردید، هنگام اجرا، وقتی برنامه به کوئری LINQ می رسد، در اصل کوئری اجرا نمی شود، بلکه هنگامی کوئری اجرا می شود که از نتایج کوئری استفاده شود. در مثال های بالا کوئری های نوشته شده زمانی اجرا می شوند که برنامه به دستور foreach می رسد. به همین دلیل به اجرای کوئری های LINQ در زبان انگلیسی Deferred-Query-Execution یا اجرای کوئری با تاخیر نیز می گویند . البته می توان کوئری ها را در همان خطی که کوئری نوشته شده است اجرا کرد . این کار به استفاده از متد های ToList، ToArray، ToDictionary و ToLookup امکان پذیر است. این متد ها نتیجه کوئری را به یکی از مجموعه های List، Array، Dictionary و یا LookUp تبدیل می کنند. برای درک بهتر به مثال زیر دقت کنید:

```
List<Person> query = persons.Where(p => p.Age > 20).ToList();
```

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

اگر بخواهیم کوئری بالا را با استفاده از دستورات LINQ بنویسیم باید به صورت زیر عمل کنیم:

```
List<Person> query = (from p in persons
                      where p.Age > 20
                      select p).ToList();
```

همانطور که مشاهده میکنید در کوئری بالا، بلافاصله نتیجه کوئری با استفاده از متد ToList داخل یک مجموعه List ریخته می شود و سپس با دستور foreach این لیست مورد پیمایش قرار می گیرد . دستور ToDictionary برای ایجاد یک Dictionary که حاوی یک کلید است استفاده می شود:

```
Dictionary<string, Person> query = persons.Where(p => p.Age >
20).ToDictionary(n => n.FirstName);
Console.WriteLine(query["Hosein"].ToString());
```

هر یک از مباحثی که در بالا عنوان شد، در نوشتن کوئری های قدرتمند LINQ نقش دارند. در بخش های بعدی مقالات، با استفاده از مثال های گوناگون به صورت جزئی تر با هر یک از این مباحث آشنا خواهید شد. در بخش بعدی در مورد اپراتورهای استاندارد LINQ صحبت خواهیم کرد.

فصل چهام :: عملگرهای استاندارد LINQ (بخش اول)

در این بخش درباره عملگرهای استانداردی که در LINQ برای ایجاد کوئری ها می تونیم از اونها استفاده کنیم صحبت خواهیم کرد . در بخش های قبلی با یکی از این عملگر ها به نام where آشنا شدیم که وظیفه فیل تر سازی اطلاعات در کوئری های LINQ رو داره. در این بخش به طور مفصل در باره تک تک این عملگرها بحث خواهیم کرد.

کلا" دو دسته از عملگرهای استاندارد در LINQ وجود دارند، اپراتورهای استانداری که بر روش اشیاء ای از نوع اینترفیس IEnumerable<T> عمل می کنند و دسته بعدی عملگرهایی هستند که بر روی اشیاء ای از نوع اینترفیس IQueryable<T> عمل می کنند. هر کدام از این عملگرها اعضاء Static از کلاس های Enumerable و Queryable هستند. (این کلاس ها در داخل فضای نام System.Linq قرار دارند). این متدها از نوع متدهای Extension تعریف شده اند. (در باره این متد ها در بخش دوم مقالات توضیح داده شد).

کد زیر مثالی در باره استفاده از عملگرهای استاندارد LINQ است:

```
string myStr = "This text is only for test";
string[] words = myStr.Split(' ');

var query = from word in words
            group word.ToLower() by word.Length into gr
            orderby gr.Key
            select new { Length = gr.Key, Words = gr };
foreach (var obj in query)
{
    Console.WriteLine(obj.Length);
    foreach (string word in obj.Words)
    {
        Console.WriteLine(word);
    }
}
```

در کد بالا، ابتدا رشته ای تعریف شده و کلمات رشته داخل یک آرایه قرار میگیرند . سپس با یک کوئری LINQ کلمات به ترتیب طول مرتب شده و دسته بندی می شن، بعد از دسته بندی حروف با یک دستور foreach مقادیر خروجی نمایش داده می شن . عملگرهایی که در بالا استفاده شده است، یکی عملگر group by برای دسته بندی خروجی و یکی هم عملگر orderby برای مرتب سازی خروجی می باشد. خروجی کد بالا به صورت زیر خواهد بود:

```
2
is
3
for
4
this
text
only
test
```

در ادامه به بررسی هر یک از عملگرهای LINQ خواهیم پرداخت:

۱. عملگرهای مرتب سازی: اولین دسته از عملگرهایی که بررسی خواهیم کردیم مربوط به مرتب سازی خروجی کوئری می باشد. این عملگرها عبارتند از:

۱.۱. عملگر OrderBy: که عملیات مرتب سازی لیست به صورت صعودی را انجام می دهد . (LINQ Syntax = orderby fieldname):

```
int[] numbers = { 4, 1, 6, 5, 2, 8, 3, 7, 9 };

IEnumerable<int> query = from n in numbers
                         orderby n
                         select n;

IEnumerable<int> query2 = numbers.OrderBy(n => n);  // query using OrderBy
extention method

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

output:
1
2
3
4
5
6
7
8
9
```

۱.۲. عملگر OrderByDescending: عملیات مرتب سازی لیست به صورت نزولی را انجام می دهد. (LINQ Syntax = orderby fieldname descending):

```
int[] numbers = { 4, 1, 6, 5, 2, 8, 3, 7, 9 };

IEnumerable<int> query = from n in numbers
                         orderby n descending
                         select n;

IEnumerable<int> query2 = numbers.OrderByDescending(n => n);

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

output:
9
8
7
6
5
4
3
2
1
```

۱.۳. عملگر ThenBy: عملیات مرتب سازی ثانویه را به صورت صعودی انجام می دهد (LINQ Syntax = orderby field1, field2, ...):

```
 List<Person> persons = new List<Person>()
{
    new Person() { FirstName = "Hosein", LastName = "Ahmadi", Age = 23 },
    new Person() { FirstName = "Hamid", LastName = "Asgari", Age = 27 },
    new Person() { FirstName = "Reza", LastName = "Karimi", Age = 42 },
    new Person() { FirstName = "Mohammad", LastName = "Zamani", Age = 31 },
    new Person() { FirstName = "Reza", LastName = "Ahmadi", Age = 26 },
    new Person() { FirstName = "Ali", LastName = "Kiani", Age = 17 },
    new Person() { FirstName = "Saman", LastName = "Skandari", Age = 25 },
    new Person() { FirstName = "Karim", LastName = "Heydari", Age = 19 }
};

var query = from p in persons
            orderby p.FirstName, p.Age
            select p;

var query2 = persons.OrderBy(n => n.FirstName).ThenBy(n => n.Age);

foreach (Person p in query2)
{
    Console.WriteLine(p.ToString());
    Console.WriteLine("--------------------");
}

output:
FirstName: Ali
LastName: Kiani
Age: 17
--------------------
FirstName: Hamid
LastName: Asgari
Age: 27
--------------------
FirstName: Hosein
LastName: Ahmadi
Age: 23
--------------------
FirstName: Karim
LastName: Heydari
Age: 19
--------------------
FirstName: Mohammad
LastName: Zamani
Age: 31
--------------------
FirstName: Reza
LastName: Ahmadi
Age: 26
--------------------
FirstName: Reza
LastName: Karimi
Age: 42
--------------------
FirstName: Saman
```

```
LastName: Skandari
Age: 25
--------------------
```

۱.۴. عملگر ThenByDescending: عملیات مرتب سازی ثانویه به صورت نزولی انجام می دهد (LINQ Syntax = orderby field1,field2,... descending):

```
List<Person> persons = new List<Person>()
{
    new Person() { FirstName = "Hosein", LastName = "Ahmadi", Age = 23 },
    new Person() { FirstName = "Hamid", LastName = "Asgari", Age = 27 },
    new Person() { FirstName = "Reza", LastName = "Karimi", Age = 42 },
    new Person() { FirstName = "Mohammad", LastName = "Zamani", Age = 31 },
    new Person() { FirstName = "Reza", LastName = "Ahmadi", Age = 26 },
    new Person() { FirstName = "Ali", LastName = "Kiani", Age = 17 },
    new Person() { FirstName = "Saman", LastName = "Skandari", Age = 25 },
    new Person() { FirstName = "Karim", LastName = "Heydari", Age = 19 }
};

var query = from p in persons
            orderby p.FirstName ascending, p.Age descending
            select p;

var query2 = persons.OrderBy(n => n.FirstName).ThenByDescending(n =>
n.Age);

foreach (Person p in query)
{
    Console.WriteLine(p.ToString());
    Console.WriteLine("--------------------");
}
```

۱.۵. عملگر Reverse: ترتیب خروج کوئری را برعکس می کند (این عملگر معادلی در ساختار LINQ ندارد):

```
 int[] numbers = { 4, 1, 7, 5, 3, 8, 2, 9 };

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

query.ForEach(n => Console.WriteLine(n));

9
2
8
3
5
7
1
4
```

عملگرهایی که در بالا در موردشان صحبت کردیم برای مرتب سازی نتایج کوئری ها استفاده می شوند. در بخش بعدی در مورد عملگر های مجموعه ای (Set Operators) صحبت خواهیم کرد.

۲. عملگرهای مجموعه ای: این عملگر ها جهت انجام عملیات هایی مانند اشتراک، اجتماع و ... بر روی مجموعه ها استفاده می شوند.

۲.۱. عملگر Distinct: جهت حذف المان های تکراری در یک مجموعه استفاده می شود (معادل در کوئری های LINQ ندارد)

```
int[] set1 = { 4, 1, 7, 5, 2, 8, 6, 4, 5, 1, 8, 9 };

var query = set1.Distinct();

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

output:
4
1
7
5
2
8
6
9
```

۲.۲. عملگر Except: المان های تکراری دو مجموعه حذف شده و تنها المان های غیر تکراری در نتیجه ظاهر می شوند (معادل LINQ ندارد)

```
int[] set1 = { 3, 6, 1, 2, 7, 8 };
int[] set2 = { 5, 2, 3, 7, 6 };

var query = set1.Except(set2);

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

output:
1
8
```

۲.۳. عملگر Intersect: خروجی حاوی اشتراک دو مجموعه خواهد بود (معادل LINQ ندارد)

```
 int[] set1 = { 3, 6, 1, 2, 7, 8 };
int[] set2 = { 5, 2, 3, 7, 6 };

var query = set1.Intersect(set2);

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

out:
3
6
2
7
```

۲.۴. عملگر Union: خروجی حاوب اجتماع دو مجموعه خواهد بود.

```
int[] set1 = { 3, 6, 1, 2, 7, 8 };
```

```
int[] set2 = { 5, 2, 3, 7, 6 };

var query = set1.Union(set2);

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

output:
3
6
1
2
7
8
5
```

۳. عملگرهای فیل تر کردن اطلاعات : عملگرهای بعدی که در مورد آنها صحبت خواهیم کرد برای فیل تر کردن اطلاعات به کار می روند. دو نوع عملگر برای اینکار وجود دارد:

۳.۱. OfType: برای آشنایی بیشتر با این متد مثالی با استفاده از یک ArrayList خواهیم زد، ArrayList شئ ایست که می تواند هر نوع داده ای را در خود ذخیره کند. حال اگر ما بخواهیم از داخل یک ArrayList که حاوی چندین نوع می باشد، یک نوع خاص را استخراج کنیم می توانیم از این عملگر استفاده کنیم:

```
ArrayList list = new ArrayList() { 2, 7, "A", 'C', 2.7, 4, "D" };

IEnumerable<int> query1 = list.OfType<int>();
IEnumerable<string> query2 = list.OfType<string>();

foreach (int i in query1)
    Console.WriteLine(i);
foreach (string s in query2)
    Console.WriteLine(s);

output:
2
7
4
A
D
```

۳.۲. عملگر where: تا اینجای کار چند مثال در مورد عملگر where زده شده. در زیر با مثالی دیگری این متد را بررسی می کنیم:

```
string[] s = { "Hello LINQ!", "Hello C#!", "LINQ is very flexible","Hello
Generics!"};

IEnumerable<string> query = from w in s
                            where w.ToLower().Contains("linq")
                            select w;
foreach (string linq in query)
    Console.WriteLine(linq);

output:
Hello LINQ!
LINQ is very flexible
```

در این م قاله در مورد سه گروه از عملگر های استاندارد صحبت کردیم، عملگرهای مرتب سازی، عملگرهای مجموعه ای و عملگرهای فیل تر سازی . به دلیل حجم زیاد این عملگرها، در طول چند بخش در مورد عملگرها صحبت خواهم کرد . در بخش بعدی در مورد عملگرهای Quantifier، عملگرهای Projection و عملگرهای Partitioning صحبت خواهیم کرد.

فصل پنجم :: عملگر های استاندارد LINQ (بخش دوم)

در ادامه سری مقالات آموزش LINQ به ادامه بررسی عملگرهای استاندارد LINQ خواهیم پرداخت. در بخش چهارم درباره برخی از این عملگره ا صحبت کردیم. عملگرهایی که در بخش پنجم مورد بررسی قرار خواهند گرفت به شرح زیر می باشند:

۱. عملگر های Quantifier
۲. عملگر های Projection
۳. عملگر های Partitioning

۱. عملگر های Quantifier: عملگر های این مقداری Boolean را بر می گردانند، زمانی که تعدادی یا کل المان های موجود در مجموعه، در شرایط تعیین شده صدق کنند . عملگر های این بخش عبارتند از:

۱.۱: عملگر All: این عملگر زمانی مقدار true بر می گرداند که تمامی اعضای مجموعه در شرایط تعیین شده صدق کنند. به مثال زیر توجه کنید:

```
string[] strArray1 = { "a", "b", "d", "t", "l", "o" };

string[] strArray2 = { "a", "a", "a", "a", "a", "a" };

bool flag1 = strArray1.All(n => n.Equals("a"));
bool flag2 = strArray1.All(n => n.Equals("a"));

Console.WriteLine(flag1);
Console.WriteLine(flag2);
```

کد بالا مقادیر false و true را به ترتیب چاپ می کند . زیرا لیست اول همه عناصر مساوی کاراکتر a نیستند، ولی لیست دوم همه عناصر برابر کاراکتر a هستند.

```
bool flag = strArray.All(n => n.Equals("a"));

Console.WriteLine(flag);
```

۲.۱: عملگر Any: این عملگر زمانی مقدار true بر می گرداند که حداقل یکی از المان های مجموعه دارای شرایط تعیین شده باشد. این عملگر به دو صورت به کار گرفته می شود:

۱. استفاده بدون پارامتر ورودی که تعیین می کند لیست هیچ عضوی دارد یا خیر.
۲. استفاده با پارامتر که تعیین می کند حداقل یکی از المان های لیست دارای شرایط تعیین شده می باشد یا خیر:

```
string[] strArray = { "a", "b", "d", "t", "l", "o" };

bool flag1 = strArray.Any(n => n.Equals("p"));
bool flag2 = strArray.Any(n => n.Equals("l"));

Console.WriteLine(flag1);
Console.WriteLine(flag2);
```

در بالا خروجی ابتدا false و سپس true می باشد.

۲.۳: عملگر Contains: این عملگر تعیین می کند که لیست دارای عضوی برابر با عضو تعیین

شده می باشد یا خیر . این عملگر به جای یک شرط برای پارامتر ورودی، یک مقدار از نوع المان های لیست می گیرد. به مثال زیر توجه کنید:

```
string[] strArray = { "a", "b", "d", "t", "l", "o" };
bool flag1 = strArray.Contains("b");

Console.WriteLine(flag1);
```

کد بالا مقدار True را برای خروجی چاپ می کند . این عملگر یک یک پارامتر از نوع IEqualityComarer نیز می گیرد که جهت مقایسه المان ها مورد استفاده قرار می گیرد.

۲. عملگر های Projection: عملگر های این بخش برای تعیین نوع خروجی و تبدیل آن به نوع مورد نظر استفاده می شوند. از عملگر Select تا این لحظه چندین بار استفاده کردیم و با آن آشنایی داریم، اما عملگر SelectMany! از این عملگر برای استفاده چندین from در یک query استفاده می شود. به مثال زیر دقت کنید (به کلاس Person یک آرایه با نام Friends از نوع string اضافه شده است)

```
List<Person> persons = new List<Person>()
{
    new Person() { FirstName = "Ali", LastName = "Jamali", Age = 42,
Friends = new string[] { "Hosein","Reza"}},
    new Person() { FirstName = "Hosein", LastName = "Ahmadi", Age =
23,Friends = new string[] { "Mohammad","Nima"} },
    new Person() { FirstName = "Mohamamd", LastName = "Sadeghi", Age =
17,Friends = new string[] { "Reza","Nima"} },
    new Person() { FirstName = "Reza", LastName = "Jamali", Age =
24,Friends = new string[] { "Hosein","Mohammad"} },
    new Person() { FirstName = "Nima", LastName = "Karami", Age =
18,Friends = new string[] { "Reza","Hosein"} }
};

var query1 = from p1 in persons
             from p2 in p1.Friends
             where p1.Age > 20
             select p2;

var query2 = persons.Where(n => n.Age > 20).SelectMany(friend =>
friend.Friends);

query2.ToList().ForEach(n => Console.WriteLine(n));
```

هر دو کوئری نوشته شده در بالا، یک کار را ان جام می دهد . در کوئری بالا، ابتدا تمام اشخاصی که سنشان بالای ۲۰ سال است انتخاب شده و سپس لیست دوستان آنها به صورت یک مجموعه آرایه قرار داده شده است . یکی از مهمترین کاربرد های SelectMany زمانی است که شما می خواهید لیستی را که داخل هر یک از المان های نتیجه کو ئری قرار دارد با یکدیگر Merge کنید. در بخش LINQ to SQL با این کاربرد بیشتر آشنا خواهید شد.

۳. عملگر های Partitioning: از این عملگر ها برای تقسیم بندی خروجی کوئری استفاده می شود. عملگر های این دسته عبارتند از:

۳.۱: عمگر Skip: با استفاده از این عملگر می توان تعداد رکورد دلخواه در ابتدای نتیجه کوئری را در نظر نگرفت. مثال:

```
int[] numbers = { 4, 6, 9, 3, 1, 2, 7, 5, 10, 8, 21, 7 };
```

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

query.ToList().ForEach(n => Console.WriteLine(n));
```

در خروجی بالا، دو المان ابتدایی کوئری در نظر گرفته نشده و المان سوم خروجی کوئری بر روی خروجی ظاهر می گردد.

۳.۲: SkipWhile این عملگر المان ها را تا زمانی که شرط تعیین شده بر قرار باشد در خروجی در نظر نمی گیرد:

```
 int[] numbers = { 4, 6, 9, 3, 1, 2, 7, 5, 10, 8, 21, 7 };
var query = numbers.OrderBy(n => n).SkipWhile(n => n < 10);

query.ToList().ForEach(n => Console.WriteLine(n));
```

در کد بالا ابتدا لیست به ترتیب صعودی مرتب شده و سپس اعداد تا زمانی که کوچکتر از ۱۰ باشند در خروجی در نظر گرفته نمی شوند و اعداد ۱۰ و ۲۱ در خروجی چاپ خواهند شد.

۳.۳: عملگر های Take و TakeWhile: این عملگر ها عکس عملگر های Skip و SkipWhile عمل میکنند. یعنی المان های تا محل تعیین شده را بر می گردانند. مثال:

```
int[] numbers = { 4, 6, 9, 3, 1, 2, 7, 5, 10, 8, 21, 7 };
var query = numbers.Take(2);

query.ToList().ForEach(n => Console.WriteLine(n));
```

کد بالا ۲ المان ابتدای لیست را بر می گرداند.

```
int[] numbers = { 4, 6, 9, 3, 1, 2, 7, 5, 10, 8, 21, 7 };
var query = numbers.OrderBy(n => n).TakeWhile(n => n < 10);

query.ToList().ForEach(n => Console.WriteLine(n));
```

کد بالا ابتدا لیست را مرتب کرده و تا زمانی که اعداد کوچکتر از ۱۰ باشند در خروجی نمایش داده می شوند.

در این بخش با تعداد دیگری از عملگر های استاندارد LINQ آشنا شدیم. در بخش بعدی با عملگر های Join، عملگر های Grouping، عملگر های Generation، عملگر های Equality و عملگر های Element آشنا خواهیم شد.

فصل ششم :: عملگر های استاندارد LINQ (بخش سوم)

در مقالات قبلی در مورد تعدادی از عملگر های LINQ صحبت کردیم، در ادامه به بررسی عملگر های زیر خواهیم پرداخت:

۱. عملگر های Join که برای ادغام دو لیست بر اساس یک یا چندین کلید استفاده می شوند.
۲. عملگر های Grouping که برای گروه بندی لیست بر اساس یک کلید خواص استفاده می شود.
۳. عملگر های Generation که برای تهیه لیستی جدید از مقادیر استفاده می شوند.
۴. عملگر های Equality که برای مقایسه دو لیست استفاده می شوند.
۵. عملگر های Element که برای گرفتن یکی از آیتم های لیست استفاده می شود.

بیشتر روی عملگر های Join و عملگر های Grouping که کاربرد های خیلی زیادی هم دارند تمرکز خواهیم کرد.

۱. عملگر های Join: افرادی که با کوئری های SQL کار کرده باشند با مفهوم Join آشنایی دارند. عملگر های Join جهت اقدام دو لیست مبتنی بر یک کلید خواص استفاده می شود که شبیه به عملیات inner join در SQL می باشد. برای مثال، فرض کنیم دو لیست داریم، داخل یکی از لیست ها، مشتریان و داخل لیست دیگر سفارشات مربوط به هر یک از مشتریان را نگهداری می کنیم. حال اگر بخواهیم این دولیست با یکدیگر ادغام شده و هر سفارش به همراه نام سفارش دهنده نمایش داده شود از Join استفاده می کنیم. برای آشنایی بیشتر به یک مثال توجه کنید، فرض کنیم می خواهیم اطلاعات مربوط به سفارشات مشتریان را ذخیره کنیم، برای اینکار دو کلاس با نام های زیر در برنامه تعریف می کنیم:

```
class Customer
{
    public int CustomerID { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
}

class Orders
{
    public int CustomerID { get; set; }
    public int OrderID { get; set; }
    public string OrderDescription { get; set; }
}
```

کلاس Customer اطلاعات مشتری را نگهداری کرده و کلاس Order اطلاعات سفارشرا نگهداری می کند، کلاس سفارشات یک خصوصیت با نام CustomerID دارد که برای ارتباط بین مشتری و سفارش استفاده می شود که همان فیلد کلید است (طراحی انجام شده برای این مثال با توجه به اصول شئ گرایی صحیح نمی باشد، اما برای نزدیکی ساختار لیست ها به جداول بانک اطلاعاتی طراحی به این صورت انجام شده است)، در ادامه می خواهیم یک کوئری اجرا کنیم که اطلاعات را به صورت نام مشتری، کد سفارش و توضیحات سفارش برای برگرداند، در این قسمت ما می توانیم از عملگر های Join استفاده کنیم:

```
List<Customer> customers = new List<Customer>
{
    new Customer() { CustomerID = 1, FirstName = "Mohammad", LastName =
"Sadeghi" },
    new Customer() { CustomerID = 2, FirstName = "Ali", LastName = "Rezaee"
}
};

List<Orders> orders = new List<Orders>
{
    new Orders() { CustomerID = 1, OrderID = 1, OrderDescription = "Order1"
},
    new Orders() { CustomerID = 1, OrderID = 2, OrderDescription = "Order2"
},
    new Orders() { CustomerID = 1, OrderID = 3, OrderDescription = "Order3"
},
    new Orders() { CustomerID = 2, OrderID = 1, OrderDescription = "Order1"
},
    new Orders() { CustomerID = 2, OrderID = 2, OrderDescription = "Order2"
},
};

var joinQuery = from c in customers
                join j in orders
                on c.CustomerID equals j.CustomerID into orderDetails
                from d in orderDetails
                select new { CustomerName = c.FirstName + " " + c.LastName,
OrderID = d.OrderID, OrderDesc = d.OrderDescription };

foreach (var item in joinQuery)
{
    Console.WriteLine(string.Format("Name:{0} OrderID:{1} OrderDesc:{2}",
item.CustomerName, item.OrderID, item.OrderDesc));
}

Console.ReadKey();
```

همانطور که در کد بالا مشاهده می کنید در کوئری از عملگر join برای ادغام سفارشات با اطلاعات مشتری استفاده شده است، ساختاری عملگر های Join به صورت زیر است:

from <element> in <list1>
join <joinName> in <list2>
on <keyField in list1> equals <keyField in list2>
from r in <joinName>
select <query result projection>

در بخش بالا خط سوم تعیین کننده شرط ما برای ادغام دو لیست می باشد که این مقایسه باید بر روی کلید مورد نظر انجام شود. امکان انجام مقایسه بین دو کلید و یا بیشتر نیز وجود دارد، برای مقایسه بین تعداد فیلد های بیشتر باید به صورت زیر عمل کرد:

from <element> in <list1>
join <joinName> in <list2>
on new {<keyField in list1>,<keyField in list1>, ...} equals {<keyField in list2>,<keyField in list2>,...}
from r in <joinName>
select <query result projection>

با استفاده از الگوی بالا می توان از چندین فیلد کلید برای نوشتن کوئری استفاده کرد. برای روشن تر شدن یک مثال دیگر رو با هم بررسی می کنیم، فرض کنیم داخل کلاس سفارش می خواهیم اطلاعات کالای سفارش داده شده را نیز نگهداری کنیم، برای این منظور کلاسی با نام Product تعریف می کنیم و کلاس Order را به صورت زیر تغییر می دهیم:

```
class Orders
{
    public int CustomerID { get; set; }
    public int OrderID { get; set; }
    public int ProductID { get; set; }
    public string OrderDescription { get; set; }
}

class Product
{
    public int ProductID { get; set; }
    public string ProductName { get; set; }
    public decimal Price { get; set; }
}
```

همانطور که مشاهده می کنید، در کلاس Orders خصوصیتی با نام ProductID تعریف شده که تعیین کننده کد کالای سفارش داده شده می باشد. حالا می خواهیم از لیست سفارشات مشتریان کوئری بنویسیم که نام مشتری، به همراه کد سفارش، نام کالا و قیمت کالا را را نمایش دهد، در این قسمت باید از عملگر join استفاده کنیم که از چندین فیلد برای کلید استفاده می کند:

```
List<Customer> customers = new List<Customer>
{
    new Customer() { CustomerID = 1, FirstName = "Mohammad", LastName =
"Sadeghi" },
    new Customer() { CustomerID = 2, FirstName = "Ali", LastName = "Rezaee"
}
};

List<Product> products = new List<Product>
{
    new Product() { ProductID = 1, ProductName = "Product1", Price = 1000
},
    new Product() { ProductID = 2, ProductName = "Product2", Price = 2500
},
    new Product() { ProductID = 3, ProductName = "Product3", Price = 4000
},
    new Product() { ProductID = 4, ProductName = "Product4", Price = 3700
},
};
```

```
List<Orders> orders = new List<Orders>
{
    new Orders() { CustomerID = 1, OrderID = 1, ProductID = 1,
OrderDescription = "Order1" },
    new Orders() { CustomerID = 1, OrderID = 2, ProductID = 3,
OrderDescription = "Order2" },
    new Orders() { CustomerID = 1, OrderID = 3, ProductID = 2,
OrderDescription = "Order3" },
    new Orders() { CustomerID = 2, OrderID = 1, ProductID = 1,
OrderDescription = "Order1" },
    new Orders() { CustomerID = 2, OrderID = 2, ProductID = 4,
OrderDescription = "Order2" },
};

var joinQuery = from c in customers
                from p in products
                join j in orders
                on new { c.CustomerID, p.ProductID } equals new {
j.CustomerID, j.ProductID } into orderDetails
                from d in orderDetails
                select new
                {
                    PersonName = c.FirstName + " " + c.LastName,
                    OrderID = d.OrderID,
                    ProductName = p.ProductName,
                    Price = p.Price,
                    OrderDesc = d.OrderDescription
                };
```

همانطور که در کوئری بالا مشاهده می کنید از ترکیب دو کلید CustomerID و ProductID برای انجام عملیات Join استفاده شده که این قابلیت به ما این امکان رو میده تا کوئری های پیچیده ای را بنویسیم. از عملگر های Join در Method Based Query هم می توان استفاده کرد. برای مثال، اولین کوئری نوشته شده در این بخش را می توان به صورت زیر نیز نوشت:

```
var query = customers.Join(orders, n => n.CustomerID, n => n.CustomerID,
    (l1, l2) => new
    {
        PersonName = l1.FirstName + " " + l1.LastName,
        OrderID = l2.OrderID,
        OrderDesc = l2.OrderDescription
    });
```

نوع دیگری از عملگر Join، استفاده از GroupJoin است، این عملگر برای ادغام یک لیست با مقادیر متناظر در لیست دیگر استفاده می شود، این عملگر برای کار با ساختاری های Hierarchical استفاده می شود. برای آشنایی بیشتر، فرض کنیم لیستی داریم شامل نام کارخانه های خودرو سازی و لیستی دیگر که نام خودرو به همراه شرکت تولید کننده را نگهداری می کند، حال می خواهیم یک کوئری بنویسیم که نتیجه آن نام شرکت و لیستی از خودرو های آن شرکت می باشد، به کد زیر توجه کنید:

```
    class Program
    {
        static void Main(string[] args)
        {
```

```
            List<Company> companies = new List<Company>()
            {
                new Company() { Name = "IranKhodro" },
                new Company() { Name = "Siapa" },
                new Company() { Name = "Hyundai" }
            };

            List<Car> cars = new List<Car>()
            {
                new Car() { CompanyName = "IranKhodro", Name = "Peikan" },
                new Car() { CompanyName = "IranKhodro", Name = "P 405" },
                new Car() { CompanyName = "IranKhodro", Name = "P Pars" },
                new Car() { CompanyName = "Siapa" , Name = "Pride" },
                new Car() { CompanyName = "Siapa" , Name = "Xantia" },
                new Car() { CompanyName = "Hyundai" , Name = "Azera" },
                new Car() { CompanyName = "Hyundai" , Name = "Sonata" },
                new Car() { CompanyName = "Hyundai" , Name = "Santafeh" },
                new Car() { CompanyName = "Hyundai" , Name = "Genesis" }
            };

            var query = companies.GroupJoin(cars, company => company.Name,
car => car.CompanyName,
                (company, car) => new { company.Name, Cars = car.Select(n
=> n.Name) });

            foreach (var co in query)
            {
                Console.WriteLine(co.Name);
                foreach (var car in co.Cars)
                    Console.WriteLine("   {0}", car);
            }
            Console.ReadKey();
        }
    }

    class Company
    {
        public string Name { get; set; }
    }


    class Car
    {
        public string CompanyName { get; set; }
        public string Name { get; set; }
    }
```

با اجرای کد بالا، خروجی زیر بر روی صفحه Console ظاهر خواهد شد:

IranKhodro
   Peikan
   P 405
   P Pars
Siapa
   Pride
   Xantia

Hyundai
    Azera
    Sonata
    Santafeh
    Genesis

بیشترین کاربرد عملیات های Join در نوشتن کوئری ها بر روی بانک های اطلاعاتی می باشد که در بخش آشنایی با LINQ2SQL و LINQ2Entities با این کار بیشتر آشنا خواهیم شد.

۲. عملگر های Grouping: عملگر های بعدی که در مورد آن توضیح خواهیم داد عملگر های Grouping یا گروه بندی می باشند، فرض کنیم لیستی از مبلغ های سفارش به همراه تاریخ سفارش را داریم، می خواهیم لیستی را که شامل جمع مبلغ های هر روز می باشد را تهیه کنیم. برای این کار باید نتیجه را بر اساس تاریخ گروه بندی کرده و جمع مبلغ هر روز را حساب کرده و در خروجی نمایش دهیم:

```
List<OrdersHistory> orderHistory = new List<OrdersHistory>
{
    new OrdersHistory() { OrderDate  = "1388/01/25", OrderPrice = 125000 },
    new OrdersHistory() { OrderDate  = "1388/01/25", OrderPrice = 18500 },
    new OrdersHistory() { OrderDate  = "1388/01/12", OrderPrice = 2500000
},
    new OrdersHistory() { OrderDate  = "1388/02/12", OrderPrice = 17500 },
    new OrdersHistory() { OrderDate  = "1388/01/17", OrderPrice = 41000 },
    new OrdersHistory() { OrderDate  = "1388/01/28", OrderPrice = 250000 },
    new OrdersHistory() { OrderDate  = "1388/02/06", OrderPrice = 60000 },
    new OrdersHistory() { OrderDate  = "1388/02/08", OrderPrice = 281000 },
    new OrdersHistory() { OrderDate  = "1388/03/02", OrderPrice = 236000 },
    new OrdersHistory() { OrderDate  = "1388/03/02", OrderPrice = 782400 },
};

var query = from e in orderHistory
            group e by e.OrderDate into dateGrouped
            orderby dateGrouped.Key
            select new { Date = dateGrouped.Key, TotalPrice =
dateGrouped.Sum(n => n.OrderPrice) };

foreach (var item in query)
{
    Console.WriteLine("{0}    {1}", item.Date, item.TotalPrice);
}

Console.ReadKey();
```

با اجرای این دستور خروجی زیر چاپ خواهد شد:

1388/01/12 2500000
1388/01/17 41000
1388/01/25 143500
1388/01/28 250000
1388/02/06 60000

1388/02/08 281000
1388/02/12 17500
1388/03/02 1018400

۳. عملگرهای Generation: از این عملگر ها برای مجموعه های جدید از روی مقادیر مجموعه های موجود استفاده می شود. در این بخش به بررسی عملگر های Generation خواهیم پرداخت:

۳.۱. عملگر DefaultIfEmpty: این عملگر در صورتی که مجموعه مورد نظر خالی باشد، مقدار پیش فرض نوع اعضای آن مجموعه را بر خواهد گرداند، برای مثال:

```
List<string> names = new List<string>
{
    "Ali","Mohammad","Mahdi","Abbas","Behzad","Behrooz","Naser"
};
Console.WriteLine(names.Where(n => n.StartsWith("D")).First());
Console.WriteLine(names.Where(n =>
n.StartsWith("D")).DefaultIfEmpty().First());
```

با اجرای دستور شماره ۱، به دلیل اینکه لیست مورد نظر خالی می باشد، یک Exception ایجاد خواهد شد، زیرا لیست خالی بوده و متد First هیچ عنصری را در لیست نخواهد یافت، ولی با اجرای دستور شماره ۲، به دلیل اینکه از متد DefaultIfEmpty استفاده شده است، به دلیل اینکه نتیجه کوئری یک لیست خالی می باشد، یک لیست که تنها دارای یک مقدار و آنهم مقدار پیش فرض نوع string می باشد، بر خواهد گرداند که متد First آن مقدار را بر می گرداند. از این متد در مواقعی که نتیجه کوئری قابل پیش بینی نیست می توان استفاده کرد.

۳.۲. Empty: این متد یک لیست خالی را بر می گرداند. این متد به صورت مستقیم از داخل کلاس Enumerable صدا زده می شود:

```
IEnumerable<string> result = Enumerable.Empty<string>();
```

۳.۳. عملگر Range: لیستی حاوی اعداد مشخص شده بر می گرداند، این متد به صورت مستقیم از طریق کلاس Enumerable صدا زده می شود:

```
IEnumerable<int> numbers = Enumerable.Range(1, 10);
```

۳.۴. عملگر Repeat: برای ایجاد یک مجموعه حاوی مقدار مشخص به تعداد مشخص شده استفاده می شود، این متد به صورت مستقیم از طریق کلاس Enumerable صدا زده می شود:

```
IEnumerable<string> list = Enumerable.Repeat("Hi", 10);
```

این متد لیستی حاوی ۱۰ عنصر با مقدار "Hi" بر می گرداند.

۴. عملگر های Equality: این قسمت تنها یک عملگر دارد، عملگر SequenceEqual، این عملگر چک می کند دو مجموعه داده شده مقادیرشان به ترتیب با یکدیگر برابر می باشد یا خیر:

```
string[] list1 = { "A", "D", "B", "H", "K" };
string[] list2 = { "D", "Q", "H", "A", "B" };
string[] list3 = { "A", "D", "B", "H", "K" };
Console.WriteLine(list1.SequenceEqual(list2));
Console.WriteLine(list1.SequenceEqual(list3));
Console.ReadKey();
```

اجرای قطعه کد بالا، ابتدا مقدار False و سپس مقدار True را بر می گرداند.

۵. عملگر های Element: از این عملگر ها برای گرفتن یکی از عناصر لیست استفاده می شود:

۵.۱. ElementAt: از این متد برای گرفتن یک عنصر در اندیس مشخص استفاده می شود:

```
string[] list1 = { "A", "D", "B", "H", "K" };
Console.WriteLine(list1.ElementAt(2)); // out B
```

۵.۲. ElementAtOrDefault: مانند متد ElementAt عمل می کند، با این تفاوت که اگر اندیس داده شده خارج از بازه لیست باشد، مقدار پیش فرض نوع داده لیست را بر می گرداند.

۵.۳. First: اولین عنصر لیست را بر می گرداند.

۵.۴. FirstOrDefault: مانند عملگر First عمل کرده، با این تفاوت که اگر لیست خالی باشد مقدار پیش فرض را بر می گرداند.

۵.۵. Last: آخرین عنصر لیست را بر می گرداند.

۵.۶. LastOrDefault: مانند عملگر Last عمل کرده، با این تفاوت که اگر لیست خالی باشد مقدار پیش فرض را بر می گرداند.

۵.۷. Single: یکی از مقادیر لیست را بر اساس شرط مشخص شده بر می گرداند:

```
string[] list1 = { "A", "D", "B", "H", "K" };
string element = list1.Single(n => n.Equals("B"));
```

۵.۸. SingleOrDefault: مانند عملگر Single عمل کرده، با این تفاوت که در صورتی که شرط داده شده لیست خالی بر گرداند، مقدار پیش فرض را بر خواهد گرداند.

در این بخش به بررسی تعدادی دیگر از عملگر های LINQ پرداختیم، در بخش بعدی سایر عملگر های Converting، Concatenation و Aggregation را بررسی خواهیم کرد که مباحث مربوط به عملگر های LINQ به پایان برسد.

فصل هفتم :: عملگر های استاندارد LINQ (بخش چهارم)

در این بخش، به بررسی آخرین دسته از عملگر های استاندارد LINQ خواهیم پرداخت که با اتمام این بخش صحبت بر روی عملگر های LINQ به پایان خواهد رسید. عملگر هایی که در این بخش در مورد آنها صحبت خواهیم کرد عبارتند از:

۱. عملگر های Converting: برای تغییر نوع اشیاء استفاده می شوند.
۲. عملگر های Concatenation: برای الحاق دو لیست به یکدیگر مورد استفاده قرار می گیرند.
۳. عملگر های Aggregation: برای استخراج یک مقدار از داخل یک لیست مورد استفاده قرار می گیرند، مانند مجموع، کوچکترین مقدار و ...

۱. عملگر های Converting: از این عملگر ها برای تبدیل اشیاء به نوع های دیگر مورد استفاده قرار می گیرند. این قسمت شامل هشت عملگر می باشد که در زیر به بررسی یکایک این عملگر ها خواهیم پرداخت:

۱.۱. عملگر AsEnumerable: از این عملگر برای تبدیل یک نوع از IEnumerable<T> به نوع دیگری از IEnumerable<T> استفاده می شود. یکی از ویژگی های این عملگر، استفاده از شرط های Where در Table های LINQ to SQL می

باشد، زیرا برخی از دستورات .NET. داخل شرط های Table ها در LINQ to SQL قابلیت استفاده را ندراد. در بخش LINQ 2 SQL با این قابلیت بیشتر آشنا خواهیم شد.

۱.۲. عملگر AsQueryable: از این دستور برای تبدیل یک نوع از IEnumerable<T> به IQuerable<T> استفاده می شود، بیشترین کاربرد این متد در مواقعی است که از Expression Trees برای نوشتن کوئری های داینامیک استفاده می

شود. در بخش های بعدی با عملکرد این متد بیشتر آشنا خواهیم شد.

۱.۳. عملگر Cast: از این عملگر در مواقعی که بخواهیم عناصر یک لیست را به نوعی دیگر تبدیل کنیم استفاده می شود. مثال:

```
System.Collections.ArrayList list = new System.Collections.ArrayList { 1,
8, 3, 6, 4, 2, 3 };
var query = list.Cast<int>().Select(n => n);
foreach (int i in query)
    Console.WriteLine(i);
```

در مثال بالا یک ArrayList تعریف شده است که با عملگر Case عناصر آن را به نوع int تبدیل کردیم.

۱.۴. عملگر OfType: از این عملگر برای استخراج عناصر از یک نوع خاص از یک لیست استفاده می شود، برای مثال اگر یک ArrayList با چندین عنصر از چندین نوع داده داشته

باشیم، می توانیم نوع مورد نظر را از داخل لیست استخراج

کنیم:

```
System.Collections.ArrayList list = new System.Collections.ArrayList { 1,
"test", "A", 6, 'c', "D", 3 };
var numbers = list.OfType<int>();
var characters = list.OfType<char>();
var strings = list.OfType<string>();
```

۱.۵. عملگر ToArray: این عملگر، عناصر یک لیست از نوع IEnumerable را به آرایه تبدیل می کند:

```
List<int> numbers = new List<int> { 4, 8, 9, 6, 3, 1, 2, 8 };
int[] array = numbers.ToArray();
```

۱.۶. عملگر ToDictionary: این عملگر لیست داده شده را به نوع Dictionary<Tkey,TValue> تبدیل می کند، کلید بر اساس یک تابع که به تابع Key Selector شناخته می شود مشخص می شود، برای مثال کلاس Customer را در نظر بگیرید، فرض کنید می خواهیم لیستی از اشیاء Customer را یک Dictionary تبدیل کنیم که کلید این Dictionary فیلد CustomerID می باشد:

```
List<Customer> customers = new List<Customer>
{
    new Customer() { CustomerID = 1, FirstName = "Mohammad", LastName =
"Sadeghi" },
    new Customer() { CustomerID = 2, FirstName = "Ali", LastName = "Rezaee"
},
    new Customer() { CustomerID = 3, FirstName = "Hosein", LastName =
"Mohammad" }
};
var dictionary = customers.ToDictionary(n => n.CustomerID);
Console.WriteLine(dictionary[2].FirstName);
```

۱.۷. عملگر ToList: مجموعه موجود از نوع IEnumerable<T> را به نوع List<T> تبدیل می کند:

```
int[] numbers = { 7, 9, 8, 2, 3, 1, 6, 5 };
numbers.ToList().ForEach(n => Console.WriteLine(n));
```

۱.۸. عملگر ToLookup: این عملگر لیست داده شده را به نوع Lookup<Tkey,TElement> تبدیل می کند، کلید بر اساس یک تابع که به تابع Key Selector شناخته می شود مشخص می شود، فرق Lookup با Dictionary در این است که کلاس Lookup از نوع One-To-Many می باشد، یعنی یک کلید می تواند به چندین Element اشاره کنید:

```
List<CityInfo> citites = new List<CityInfo>
{
    new CityInfo() { Province = "Tehran", CitiyName = "Karaj" },
    new CityInfo() { Province = "Tehran", CitiyName = "Ray" },
```

```
    new CityInfo() { Province = "Tehran", CitiyName = "Hashtgerd" },
    new CityInfo() { Province = "Mazandaran", CitiyName = "Babolsar" },
    new CityInfo() { Province = "Tehran", CitiyName = "Noor" },
    new CityInfo() { Province = "Tehran", CitiyName = "Nowshahr" }
};
var query = citites.ToLookup(n => n.Province);
foreach (CityInfo info in query["Tehran"])
    Console.WriteLine(info.CitiyName);
```

۲. عملگر های Concatenaion: این بخش دارای یک عملگر با نام Concate می باشد که دو لیست را با یکدیگر ادغام می کند:

```
int[] list1 = { 1, 2, 3, 4, 5 };
int[] list2 = { 5, 6, 7, 8, 9, 10 };
IEnumerable<int> concated = list1.Concat(list2);
foreach (int i in concated)
    Console.WriteLine(i);
```

۳. عملگر های Aggregation: از این عملگر ها برای استخراج یک مقدار از مجموعه داده شده استفاده می شود، عملگر های این مجموعه به شرح زیر می باشند:

۳.۱. عملگر Average: میانگین مقادیر یک لیست را به دست می آورد:

```
int[] list1 = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
Console.WriteLine(list1.Average());
```

۳.۲. عملگر Count: از این عملگر برای بدست آوردن تعداد عناصر موجود در لیست استفاده می شود.

```
int[] list1 = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
Console.WriteLine(list1.Count());
```

۳.۳. عملگر LongCount: مانند عملگر Count عمل می کند، با این تفاوت که عدد برگردانده شده از نوع long می باشد، استفاده از این عملگر برای گرفتن تعداد عناصر مجموعه های خیلی بزرگ پرکاربرد می باشد.

۳.۴. عملگر Max: این عملگر بزرگترین عنصر موجود در لیست را بر می گرداند:

```
int[] list1 = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
Console.WriteLine(list1.Max());
```

۳.۵. عملگر Min: این عملگر کوچکترین عنصر موجود در لیست را بر می گرداند:

```
int[] list1 = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
Console.WriteLine(list1.Min());
```

۳.۶. عملگر Sum: این عملگر حاصل مجموع کلیه عناصر موجود در لیست را بر می گرداند:

```
int[] list1 = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
Console.WriteLine(list1.Sum());
```

** نکته :: برای متد های ذکر شده می توان یک تابع نیز به عنوان ورودی تعیین کرد که انتخاب کننده فیلد مورد نظر برای محاسبه می باشد، مثال:

```
List<OrdersHistory> orderHistory = new List<OrdersHistory>
{
    new OrdersHistory() { OrderDate  = "1388/01/25", OrderPrice = 125000 },
    new OrdersHistory() { OrderDate  = "1388/01/25", OrderPrice = 18500 },
    new OrdersHistory() { OrderDate  = "1388/01/12", OrderPrice = 2500000
},
    new OrdersHistory() { OrderDate  = "1388/02/12", OrderPrice = 17500 },
    new OrdersHistory() { OrderDate  = "1388/01/17", OrderPrice = 41000 },
    new OrdersHistory() { OrderDate  = "1388/01/28", OrderPrice = 250000 },
    new OrdersHistory() { OrderDate  = "1388/02/06", OrderPrice = 60000 },
    new OrdersHistory() { OrderDate  = "1388/02/08", OrderPrice = 281000 },
    new OrdersHistory() { OrderDate  = "1388/03/02", OrderPrice = 236000 },
    new OrdersHistory() { OrderDate  = "1388/03/02", OrderPrice = 782400 },
};

Console.WriteLine(orderHistory.Sum(n => n.OrderPrice));

Console.ReadKey();
```

۳.۷. عملگر Aggregate: از این عملگر برای اعمال یک عملیات دلخواه بر روی عنصر لیست استفاده می شود، برای مثال پیاده سازی عملیات Sum با کمک عملگر Aggregate به صورت زیر می باشد:

```
int[] list1 = { 2, 3, 4, 5, 6, 7 };
Console.WriteLine(list1.Aggregate((current, next) =>
{
    return current + next;
}));
```

خوب، تا اینجای مقاله ما با مقدمات LINQ و عملگر های آن آشنا شدیم، با استفاده از مباحث مطرح شده تا این بخش قادر خواهیم بود تا LINQ را به بخش های دیگری بسط دهیم، در بخش های بعدی با مباحث پیشرفته تری مانند LINQ 2 XML و LINQ 2 SQL آشنا خواهیم شد.