



# C# -CHAPTER7-

SOUL SEEK



# 목차

---

## 1. LINQ



The LINQ logo is centered on a dark blue background with a subtle radial gradient. The logo consists of the letters "LINQ" in a white, sans-serif font. The letter "I" is a simple vertical line, while "N", "Q", and "L" have a small horizontal bar at their base. The "Q" features a small tail that curves upwards and to the right. The background is decorated with faint, light blue circuit-like patterns in the corners, composed of lines and small circles.

LINQ

# 1. LINQ

## LINQ

- **Language Integrated Query**
- **C#**언어에 통합된 데이터 **Query**(질의)기능을 말한다.
- **Query(질의)** : “육하원칙에 의거해서 대답하시요...”랑 비슷한 의미
  - **From** : 어떤 데이터 집합에서 찾을 것인가?
  - **Where** : 어떤 값의 데이터를 찾을 것인가?
  - **Select** : 어떤 항목을 추출할 것인가? 와 같은 기능을 가지고 있다.

일반 적으로 데이터를 기입하는 구문

```
class Profile
```

```
{  
    public string    Name { get; set; }  
    public int       Height { get; set; }  
}
```

```
Profile[] arrProfile = {  
    new Profile(){Name= “나연”, Height= 163},  
    new Profile(){Name= “사나”, Height= 164},  
    new Profile(){Name= “모모”, Height= 162},  
    new Profile(){Name= “채영”, Height= 159}  
}
```

# 1. LINQ

앞의 구문을 기반으로 **164**미만인 데이터만 골라 새 컬렉션으로 추출해야 한다면?

```
List<Profile> profiles = new List<Profile>();
```

```
foreach(Profile profile in arrProfile)  
{  
    if(profile.Height < 164)  
        profiles.Add(profile);  
}
```

조건자 함수를 익명 메소드를 이용해서 구현하였다.

```
profiles.Sort(  
    (profile1, profile2) =>  
    {  
        return profile1.Height - profile2.Height;  
    });
```

```
foreach(var profile in profile)  
    Console.WriteLine("{0}, {1}", profile.Name, profile.Height);
```

# 1. LINQ

앞의 구문을 **LINQ**를 이용해서 구현 한다면?

```
var profiles = from profile in arrProfile
               where profile.Height < 175
               orderby profile.Height
               select profile;
```

어떠한 행동을 해야 할지를 지정하는 것이기 때문에 **foreach**처럼 직접적인 데이터를 담아내지 않는다.

```
Foreach(var profile in profiles)
    Console.WriteLine("{0}, {1}", profile.Name, profile.Height);
```

**기본구문 : from, where, orderby, select**

## from

- **Query**식은 항상 **from**으로 시작하여야 한다.
- 쿼리식의 대상이 될 데이터 원본과 데이터 원본 안에 들어 있는 각 요소 데이터를 나타내는 범위변수(쿼리변수 - **Query Variable**)를 **from**에서 지정해 줘야한다.
  - **From**의 원본은 아무 형식이나 사용할 수는 없고 **IEnumerable<T>** 인터페이스를 상속하고 있어야 한다. 즉, 배열, 컬렉션, 컬렉션 제너릭이 사용 가능하고 사용자가 직접 정의한 **IEnumerable<T>** 클래스도 사용 가능하다.

# 1. LINQ

**from** <범위 변수> **in** <데이터 원본>의 형식 사용

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

```
var result =    from n in numbers  
                ...
```

**where**

- **Filter** 역할을 하는 연산자.
- **From** 절이 데이터 원본으로 부터 뽑아 낸 범위 변수가 가져야 하는 조건을 **where** 연산자에 매개 변수로 입력하면 해당 조건에 부합하는 데이터만을 걸러낸다.

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

```
var result =    from      n in numbers  
                where     n < 5  
                ....
```

# 1. LINQ

## orderby

- 데이터의 정렬을 수행하는 연산자.
- 정렬기준을 제시해주면 된다.
- 기본 적으로 오름 차순 정렬을 지원하지만 내림차순 또는 오름차순으로 각각 지정하고 싶을때 → **ascending, desending** 키워드를 붙여주면 된다.

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

```
var result =      from      n in numbers  
                  where      n < 5;  
                  orderby    n ascending 또는 desending  
                  ...
```

## Select

최종 결과를 추출하는 쿼리식.

결과물을 넘겨주기 이전에 어떤 형식을 넘겨줄지 결정된다.

→ 데이터를 대기중인 **var** 형식은 **select**에 의해 어떤 형식이 될지 결정 된다.

```
var result =      from      n in numbers  
                  where      n < 5;  
                  orderby    n ascending 또는 desending  
                  select    n
```



# 1. LINQ

- 멤버 변수의 형식이 될 수도 있다.

```
var profiles = form      profile in arrProfile  
                  where    profile.Height < 175;  
                  orderby profile.Height  
                  select  profile.Name;
```

→ var은 멤버인 Name의 형식인 string이 된다.

- 새로운 데이터 형식의 인스턴스를 생성해서 넘겨 줄 수도 있다.

```
var profiles = form      profile in arrProfile;  
                  where    profile.Height < 175;  
                  orderby profile.Height;  
                  select  new {Name = profile.Name, InchHeight =  
                               profile.Height * 0.393};
```

**Name, InchHeight**라는 두개의 **sting, float**형 멤버를 가지는 인스턴스를 생성해서 **profiles**에 넘겨준다. 여기서 var은 무명 클래스의 인스턴스형이 되는 것이다.

# 1. LINQ

여러 개의 데이터 원본에 접근하는 쿼리 식(Query Expression)

- **from**을 이용해서 반복문을 중첩해서 사용하듯이 사용하면 된다.

```
class Classroom
{
    public string    Name { get; set; }
    public int[]     Score { get; set; } // 배열.
}
```

```
Class[] arrClass =
{
    new Class(){Name="연두반", Score=new int[]{99, 80, 70, 24}},
    new Class(){Name="분홍반", Score=new int[]{69, 45, 87, 72}},
    new Class(){Name="파랑반", Score=new int[]{92, 30, 85, 94}},
    new Class(){Name="노랑반", Score=new int[]{90, 88, 0, 17}},
};
```

```
Var classes = from c in arrClass
               from s in c.Score
               where s < 60
               select new {c.Name, Lowest = s};
```

첫 번째 데이터 원본

두 번째 데이터 원본

# 1. LINQ

여러 개의 데이터 원본에 접근하는 쿼리 식(Query Expression)

- **from**을 이용해서 반복문을 중첩해서 사용하듯이 사용하면 된다.

```
class Classroom
```

```
{  
    public string    Name { get; set; }  
    public int[]     Score { get; set; } // 배열.  
}
```

```
Class[] arrClass =
```

```
{  
    new Class(){Name="연두반", Score=new int[]{99, 80, 70, 24}},  
    new Class(){Name="분홍반", Score=new int[]{69, 45, 87, 72}},  
    new Class(){Name="파랑반", Score=new int[]{92, 30, 85, 94}},  
    new Class(){Name="노랑반", Score=new int[]{90, 88, 0, 17}},  
};
```

```
Var classes = from c in arrClass
```

```
    from s in c.Score
```

```
    where s < 60
```

```
    select new {c.Name, Lowest = s};
```

첫 번째 데이터 원본

두 번째 데이터 원본

# 1. LINQ

## group by로 데이터 분류

- 분류기준에 따라 데이터를 그룹화하여 관리할 수 있게 해준다.

### group A by B into C

→ **A**에는 QueryVainary를 **B**에는 분류 기준을, **C**는 그룹 변수를 위치 시키면 된다.

```
Profile[] arrProfile =  
{  
    new Profile(){Name= "정우성", Height=186},  
    new Profile(){Name= "김태희", Height=158},  
    new Profile(){Name= "고현정", Height=172},  
    new Profile(){Name= "이문세", Height=178},  
}
```

정우성, 186
김태희, 158
고현정, 172
이문세, 178
하하, 171

group by를 이용해서 분류하면..

```
var listProfile = form profile in arrProfile  
group profile by profile.Height < 175 into g  
select new { GroupKey = g.Key, Profiles = g };
```

김태희, 158 고현정, 172 하하, 171	} < 175	
정우성, 186 이문세, 178		} >= 175

# 1. LINQ

## Join - 데이터 원본을 연결하기

- 각 데이터 원본에서 특정 필드의 값을 비교하여 일치하는 데이터 끼리 연결을 수행

### 내부조인

- 교집합과 비슷하다
- 일치하는 데이터들만 연결한 후 반환한다.
  - 두 데이터의 일치한 부분을 파악하고 합쳐준다.

#### [Name, Height]

정우성, 186
김태희, 158
고현정, 172
이문세, 178
하하, 171

+

#### [Product, Star]

비트, 정우성
CF 다수, 김태희
아이리스, 김태희
모래시계, 고현정
Solo 예찬, 이문세

=

A.Name == B.Star조건  
으로 내부 조인

정우성, 비트,	186
김태희, CF 다수,	158
김태희, 아이리스,	158
고현정, 모래시계,	172
이문세, Solo 예찬,	178

# 1. LINQ

```
from a in A  
join b in B on a.XXXX equals b.YYYY
```

- **a**는 **from** 절에서 뽑아낸 범위 변수이고, 연결 대상 데이터 **b**는 **join** 절에서 뽑아낸 변수이다.
- **join** 절의 **on** 키워드는 조인조건을 수반한다.
- **on** 절의 조인 조건은 “동등(==, Equality)” 비교만 가능하고 “크거나 작음(<, >)” 비교는 불가능하다.

```
var listProfile =  
    form    profile    in arrProfile  
    join    product    in arrProduct on profile.Name equals product.Star  
    select new  
    {  
        Name  = profile.Name,  
        Work  = product.Title,  
        Height = profile.Height  
    }
```

# 1. LINQ

## 외부조인

- 조인 결과에 기준이 되는 데이터 원본은 모두 다 포함된다.
- **SQL(Structured Query Language)**의 왼쪽 조인과 같은 동작을 한다.
  - **LINQ**는 원래 **DBMS**에서 사용하던 **SQL**를 본떠 프로그래밍 언어 안에 통합한 것이다.
  - 외부조인도 **SQL**에서 본뜬 것이고, **SQL**에서는 왼쪽조인, 오른쪽조인, 완전외부조인 이렇게 세가지 인데 왼쪽조인은 왼쪽 원본 데이터를 기준으로 오른쪽 조인은 오른쪽 원본 데이터를 기준으로 완전 외부조인은 왼쪽과 오른쪽 데이터 원본 모두를 기준으로 한다.
  - **LINQ**의 외부조인은 이중에 왼쪽 조인 기능만 가지고 있다.

**[Name, Height]**

정우성, 186
김태희, 158
고현정, 172
이문세, 178
하하, 171

+

**[Product, Star]**

비트, 정우성
<b>CF</b> 다수, 김태희
아이리스, 김태희
모래시계, 고현정
<b>Solo</b> 예찬, 이문세

=

**A.Name == B.Star**조건  
으로 외부 조인

정우성, 비트,	186
김태희, <b>CF</b> 다수,	158
김태희, 아이리스,	158
고현정, 모래시계,	172
이문세, <b>Solo</b> 예찬,	178
하하,	, 171



# 1. LINQ

```
var listProfile =  
    from    profile in arrProfile  
    join    product in arrProduct on profile.Name equals product.Star into ps  
    form    product in ps.DefaultIfEmpty(new Product(){Title = “그런거 없음”})  
    select new  
    {  
        Name  = profile.Name,  
        Work  = product.Title,  
        Height = profile.Height  
    }
```

## LINQ 표준 연산자

- **LINQ**는 **.NET** 언어 중에 **C#**과 **VB**에서만 사용 가능 하다.
- **LINQ** 구문은 실제 내부에서 표준 연산자들을 사용할 수 있도록 키워드를 지정해 준다고 생각하면 된다.
- 직관적으로 보기 편리하게 만들기 위해 **LINQ**의 구문을 생각해낸 것이고, **LINQ** 구문은 **CLR**이 알아 볼 수 있도록 **C#** 컴파일러가 변형해 준다.
- **LINQ** 표준 연산자는 총 **53**개를 지원하지만 **C#**에서는 **11**개만 사용 된다.



# 1. LINQ

```
var listProfile =  
    from    profile in arrProfile  
    join    product in arrProduct on profile.Name equals product.Star into ps  
    form    product in ps.DefaultIfEmpty(new Product(){Title = “그런거 없음”})  
    select new  
    {  
        Name  = profile.Name,  
        Work   = product.Title,  
        Height = profile.Height  
    }
```

## LINQ 표준 연산자

- **LINQ**는 **.NET** 언어 중에 **C#**과 **VB**에서만 사용 가능 하다.
- **LINQ** 구문은 실제 내부에서 표준 연산자들을 사용할 수 있도록 키워드를 지정해 준다고 생각하면 된다.
- 직관적으로 보기 편리하게 만들기 위해 **LINQ**의 구문을 생각해낸 것이고, **LINQ** 구문은 **CLR**이 알아 볼 수 있도록 **C#** 컴파일러가 변형해 준다.
- **LINQ** 표준 연산자 메소드 중 **C#**의 쿼리식에서 지원하는 것은 **11**개 뿐이다.  
→ 쿼리식으로 직접 사용할 수 없어도 메소드는 사용 할 수 있기 때문에 이를 활용한다면 얼마든지 활용 할 수 있다.

# 1. LINQ

## C#에 지원되는 쿼리식의 메소드

종류	메소드 이름	설명	C# 쿼리식 문법
정렬	<b>OrderBy</b>	오름차순으로 값을 정렬	<b>orderby</b>
	<b>OrderByDescending</b>	내림차순으로 값을 정렬	<b>orderby ... descending</b>
	<b>ThenBy</b>	오름차순으로 <b>2차</b> 정렬	<b>orderby ... , ...</b>
	<b>ThenByDescending</b>	내림차순으로 <b>2차</b> 정렬	<b>orderby ... , ... descending</b>
필터링	<b>Where</b>	필터링할 조건을 평가하는 함수를 통과하는 값들만 추출	<b>where</b>
데이터 추출	<b>Select</b>	값을 추출하여 시퀀스를 만든다.	<b>select</b>
	<b>SelectMany</b>	여러 개의 데이터 원본으로부터 값을 추출하여 하나의 시퀀스를 만든다. 여러 개의 <b>from</b> 절을 사용한다.	
데이터 결합	<b>Join</b>	공통 특성을 가진 서로 다른 두 개의 데이터 소스의 객체를 연결, 공통 특성을 ( <b>Key</b> )로 삼아, 키가 일치하는 두 객체를 쌍으로 추출	<b>join .. in .. on .. equals ..</b>
	<b>GroupJoin</b>	기본적으로 <b>Join</b> 연산자와 같은 일을 하되, 조인 결과를 그룹으로 만들어 놓는다.	<b>join .. in .. on .. equals .. into</b>
데이터 그룹화	<b>GroupBy</b>	공통된 특성을 공유하는 요소들을 각 그룹으로 묶는다, 각 그룹은 <b>IGrouping&lt;( Of &lt;( TKey, TElement )&gt; )&gt;</b> 객체로 표현 된다.	<b>group .. by</b> 또는 <b>group .. by .. into ..</b>
형식변환	<b>Cast</b>	컬렉션의 요소들을 특정형식으로 변환한다.	범위 변수를 선언 할 때 명시적으로 형식을 지정하면 된다.

# 1. LINQ

이외에 전체적으로 지원하는 메소드.. **C#**에서 이 녀석들은 메소드로 사용하면 된다.

종류	메소드 이름	설명
정렬	<b>Reverse</b>	컬렉션 요소의 순서를 거꾸로 뒤집습니다.
집합	<b>Distinct</b>	중복 값을 제거한다.
	<b>Except</b>	두 컬렉션 사이의 차집합을 반환, 다시 말해 임의의 한 컬렉션( <b>a, b, c, e</b> )에는 존재하는데 다른 한 컬렉션( <b>a, d, f</b> )에는 존재하지 않는 요소들( <b>b, e</b> )를 반환한다.
	<b>Intersect</b>	두 컬렉션 사이의 교집합을 반환한다.
	<b>Union</b>	두 컬렉션 사이의 합집합을 반환한다. 예를 들어 한쪽 컬렉션이 <b>a, b, c, d</b> 요소를 갖고 있고 다른 한쪽 컬렉션이
필터링	<b>OfType</b>	메소드의 형식 매개 변수로 형식 변환이 가능한 값들만 추출한다.
수량 연산	<b>ALL</b>	여러 개의 데이터 원본으로부터 값을 추출하여 하나의 시퀀스를 만든다. 여러 개의 <b>from</b> 절을 사용한다.
	<b>Any</b>	모든 요소 중 단 하나의 요소라도 임의의 조건을 만족시키는지를 평가한다, 결과는 <b>true</b> 이거나 <b>false</b> , 둘 중 하나
	<b>Contains</b>	명시한 요소가 포함되어 있는지를 평가한다. 역시 결과는 <b>true</b> 이거나 <b>false</b> , 둘 중 하나
데이터 분할	<b>Skip</b>	시퀀스에서 지정한 위치까지 요소들을 건너뛴다.
	<b>SkipWhile</b>	입력된 조건 함수를 만족시키는 요소들을 건너뛴다.
	<b>Take</b>	시퀀스에서 지정한 요소까지 요소들을 취한다.
	<b>TakeWhile</b>	입력된 조건 함수를 만족시키는 요소들을 취한다.

# 1. LINQ

종류	메소드 이름	설명
데이터 그룹화	<b>ToLookup</b>	키( <b>Key</b> ) 선택 함수를 이용하여 골라낸 요소들을 <b>Lookup&lt;(Of&lt;(TKey, TElement)&gt;)&gt;</b> 형식의 객체에 삽입한다.(이 형식은 하나의 키에 여러 개의 객체를 대응시킬 때 사용하는 컬렉션이다.)
생성	<b>DefaultIfEmpty</b>	빈 컬렉션을 기본값이 할당된 싱글톤 컬렉션으로 바꿉니다. 싱글톤( <b>Singleton</b> )이란, 해당형식의 객체를 오직 단 하나만 만들고 이 객체를 전역에서 접근할 수 있도록 하는 디자인 기법, 기본값이 할당된 컬렉션은 참조용으로만 사용할 것이니 여러 개의 인스턴스가 필요 없고, 싱글톤을 이용하면 메모리 낭비를 줄일 수 있다.
	<b>Empty</b>	비어 있는 컬렉션을 반환
	<b>Range</b>	일정 범위의 숫자 시퀀스를 담고 있는 컬렉션을 생성한다.
	<b>Repeat</b>	같은 값이 반복되는 컬렉션을 생성한다.
동등 여부 평가	<b>SequenceEqual</b>	두 시퀀스가 서로 일치하는지를 평가한다.
요소접근	<b>ElementAt</b>	컬렉션으로부터 임의의 인덱스에 존재하는 요소를 반환한다.
	<b>ElementAtOrDefault</b>	컬렉션으로부터 임의의 인덱스에 존재하는 요소를 반환하되, 인덱스가 컬렉션의 범위를 벗어날 때 기본값을 반환한다.
	<b>First</b>	컬렉션의 첫 번째 요소를 반환한다. 조건식이 매개 변수로 입력되는 경우 이 조건을 만족시키는 첫 번째 요소를 반환한다.
	<b>FirstOrDefault</b>	<b>First</b> 연산자와 같은 기능을 하되, 반환할 값이 없는 경우 기본값을 반환한다.
	<b>Last</b>	컬렉션의 마지막 요소를 반환한다. 조건식이 매개 변수로 입력되는 경우 이 조건을 만족시키는 마지막 요소를 반환한다.
	<b>LastOrDefault</b>	<b>Last</b> 연산자와 같은 기능을 하되, 반환할 값이 없는 경우 기본값을 반환한다.
	<b>Single</b>	컬렉션의 유일한 요소를 반환한다. 조건식이 매개 변수로 입력되는 경우 이 조건을 만족시키는 유일한 요소를 반환한다.

# 1. LINQ

종류	메소드 이름	설명
요소 접근	<b>SingleOrDefault</b>	<b>Single</b> 연산자와 같은 기능을 하되, 반환할 값이 없거나 유일한 값이 아닌 경우 주어진 기본값을 반환한다.
형식 변환	<b>AsEnumerable</b>	매개 변수를 <b>IEnumerable&lt;(Of&lt;(T)&gt;)&gt;</b> 로 형식 변환하여 반환한다.
	<b>AsQueryable</b>	(일반화) <b>IEnumerable</b> 객체를 (일반화) <b>IQueryable</b> 형식으로 변환한다.
	<b>OfType</b>	특정 형식으로 형식 변환할 수 있는 값만을 걸러낸다.
	<b>ToArray</b>	컬렉션을 배열로 변환한다. 이 메소드는 강제로 쿼리를 실행한다.
	<b>ToDictionary</b>	키 선택 함수에 근거해서 컬렉션의 요소를 <b>Dictionary&lt;( Of &lt;(TKey, Tvalue)&gt; )&gt;</b> 에 삽입한다. 이 메소드는 강제로 쿼리를 실행한다.
	<b>ToList</b>	컬렉션을 <b>List&lt;(Of&lt;(T)&gt;)&gt;</b> 형식으로 변환한다. 이 메소드는 강제로 쿼리를 실행한다.
	<b>ToLookup</b>	키 선택 함수에 근거해서 컬렉션의 요소를 <b>Lookup&lt;( Of &lt;(TKey, TElement)&gt; )&gt;</b> 에 삽입한다. 이 메소드는 강제로 쿼리를 실행한다.
연결	<b>Concat</b>	두 시퀀스를 하나의 시퀀스로 연결한다.
집계	<b>Aggregate</b>	컬렉션의 각 값에 대해 사용자가 정의한 집계 연산을 수행한다.
	<b>Average</b>	컬렉션의 각 값에 대한 평균을 계산한다.
	<b>Count</b>	컬렉션에서 조건에 부합하는 요소의 개수를 센다.
	<b>LongCount</b>	<b>Count</b> 와 동일한 기능을 하지만, 매우 큰 컬렉션을 대상으로 한다는 점이 다르다.
	<b>Max</b>	컬렉션에서 가장 큰 값을 반환한다.
	<b>Min</b>	컬렉션에서 가장 작은 값을 반환한다.
	<b>Sum</b>	컬렉션 내의 값의 합을 계산한다.

# 1. LINQ

메소드들을 사용하는 예를 들어보자.

**Average** 사용..

```
Profile[] arrProfile =  
{  
    new Profile() { Name = "정우성", Height = 186 },  
    new Profile() { Name = "김태희", Height = 158 },  
    new Profile() { Name = "고현정", Height = 172 },  
}
```

```
var profile = from    profile in arrProfile  
              where    profile.Height < 180  
              select  profile;
```

```
double Average = profiles.Average(profile => profile.Height);  
Console.WriteLine(Average);
```

이 마저도 줄일 수 있다.

```
Double Average = (from    profile in arrProfile  
                  where    profile.Height < 180  
                  select  profile).Average(profile => profile.Height);
```