C# 코딩 연습

컬렉션, 序

2008-03-12

김태현 (kimgwajang@hotmail.com)

이 문서는 세 편으로 기획된 "C# 코딩 연습 – 컬렉션" 시리즈 중 첫 번째입니다. 이 시리즈는 닷넷 프레임웍에서 차지하는 컬렉션의 높은 비중을 이해하고, 이를 효율적으로 사용하여 코드의 품질을 높이는 방법에 대한 고민을 나누는 것을 목적으로 기획되었습니다.

현재 이 시리즈는 총 세 편으로 기획되어 있으며 각각의 주제는 다음과 같습니다.

- 서 : 닷넷 프레임웍의 컬렉션을 이해하기 위한 기반 기술들을 살펴봅니다. 특히 컬렉션의 내부에서 사용되는 인터페이스와 대리자와 클래스 등의 역할에 중점을 둡니다.
- 본 : 가장 빈번히 사용되는 동시에 다른 고급 컬렉션들의 기반이 되는 세 가지 컬렉션들을 직접 만들어 보며 컬렉션의 내부 구조와 로직을 이해합니다. 이는 컬렉션의 내부 구조와 로직을 이해함으로써 컬렉션을 효율적으로 사용하는 데 목적이 있습니다.
- 결 : 닷넷 베이스 클래스 라이브러리에 포함된 컬렉션의 한계를 보완하는 두 개의 공개된 컬렉션 셋을 소개합니다. 이들 컬렉션 셋은 필드에서 바로 사용할 수 있을 만큼의 높은 완성도를 가지고 있습니다.

I. 序의 序

> ***컬렉션 : 서로 밀접하게 관련된 데이터를 그룹화하여 좀 더 효율적으로 처리할 수 있게 한 특수한 클래스 혹은 구조체(MSDN 라이브러리)***

컬렉션을 사용하면 데이터를 효율적으로 처리할 수 있다는 말을 거꾸로 생각해 봅시다. 대표적인 컬렉션인 배열이 없다고 가정을 해볼까요? 배열이 없다면 우리가 일상적으로 작성하는 코드는 어떻게 될까요? 예를 들어 100 명의 시험 점수를 저장해야 한다면 100 개의 변수를 만들어야 할까요? 어떻게든 구현은 된다 하더라도, 그 너저분함은 생각하기도 싫을 만큼 끔찍할 것입니다. 하지만 이런 우울한 상황은 배열이라는 컬렉션을 도입하는 것 만으로 훨씬 효율적으로 변경될 수 있습니다.

그렇다면 배열이 만병통치약일까요? 물론 그렇지는 않습니다. 배열은 나름의 장점을 분명히 가지고 있지만, 동시에 다른 컬렉션들에 비해서 적지 않은 단점도 가지고 있습니다. 대표적으로 요소를 저장할 수 있는 용량이 고정되어 있다는 점을 들 수 있겠습니다. 그래서 경우에 따라서는 배열이 아닌 다른 컬렉션을 선택하여야 하는 상황도 얼마든지 있습니다.

베이스 클래스 라이브러리에는 우리가 사용할 수 있는 수 많은 컬렉션이 있고, 또 서드파티 라이브러리로 제공되는 컬렉션도 부지기수입니다. 물론 사용자가 고유한 컬렉션을 만들 수도 있고요. 그렇다면 이 많은 컬렉션 중에서 상황에 가장 적합한 컬렉션을 선택하는 것이 관건이 되겠습니다.

여기서 '선택'이라는 말이 중요합니다. 대부분의 경우에 있어서 우리는 이미 존재하는 컬렉션 중에서 가장 적합한 것을 고르게 됩니다. 아마도 기존의 컬렉션으로는 도저히 처리할 수 없는 특별한 기능을 가진 컬렉션을 만드는 일은 거의 없을 것입니다.

우리의 목표는 뛰어난 컬렉션을 만드는 것이 아닙니다. 사용 목적에 가장 적합한 컬렉션을 선택하여 정확하게 사용하는 것이 목표입니다. 그렇다면 어떻게 하면 이 '선택'을 잘 할 수 있을까요? '아는 만큼 보인다'는 말이 있지요. 컬렉션의 경우에도 그렇습니다. 우리가 일상적으로 사용하는 컬렉션에 대해서 더 많이 알 수록, 우리는 컬렉션을 더 잘 선택할 수 있을 것입니다. 그때에 컬렉션을 사용하는 것은 확실히 예전과 다르겠지요.

하지만 컬렉션, 특히 그 내부 구현에 대해서 아는 것은 말처럼 간단하지가 않습니다. 베이스 클래스 라이브러리를 포함하여, 공개된 컬렉션 라이브러리들의 복잡함은 그 완성도를 감안할 때 쉬이 짐작할 수 있을 것입니다. 실제로 닷넷 프레임웍의 컬렉션 클래스들의 소스를 보면 정말이지 대단합니다. 수 많은 예외 상황들에 대한 처리는 기본이고, 쓰레드 안정성, 내부 버전 관리, 얕은 복사와 깊은 복사, 거미줄처럼 얽혀 있는 상속과 포함 관계 등을 모두 고려하려면 당연히 복잡할 수 밖에 없겠지요.

다시 한번 강조하지만, 우리의 목표는 베이스 클래스 라이브러리 수준의 컬렉션을 만드는 것이 아닙니다. 실제 필드에서 사용할 목적으로 List<T>나 Dictionary<T>와 같은 컬렉션을 다시 만드는 것은 현명하지 못한 방법입니다. 대신에 우리는 각 컬렉션을 최대한으로 간소화시킨 모델을 이용하여 그 핵심 로직을 이해하고, 그에 대한 이해를 바탕으로 상황에 맞는 컬렉션을 선택하고 또 효율적으로 사용하는 전략에 집중을 할 것입니다.

II. 컬렉션과 인터페이스

모든 닷넷 컬렉션은 적어도 ICollection<T> 인터페이스는 구현을 합니다. 바꾸어 말하면 모든 닷넷 컬렉션은 ICollection<T> 인터페이스에 정의되어 있는 연산을 수행할 수 있습니다. 이에 더하여 몇몇 컬렉션들은 추가적인 인터페이스를 구현하기도 합니다. 예를 들어, Stack<T> 클래스는 ICollection<T>을 구현하므로 ICollection<T>에 정의되어 있는 Add 와 Remove 같은 작업을 수행할 수 있습니다. 반면에 List<T> 클래스의 경우에는 IList<T> 인터페이스를 구현하므로 ICollection<T> 인터페이스와 IList<T> 인터페이스에 있는 연산을 모두 수행할 수 있습니다. IList<T>에 정의되어 있는 Insert 와 RemoveAt 같은 연산은 Stack<T>에서는 지원되지 않고 List<T>에서만 지원이 된다는 이야기이지요.

닷넷 컬렉션의 연산을 정의하고 있는 인터페이스들에 대해서 좀 더 자세히 살펴봅시다. 먼저 다음 클래스 다이어그램을 보지요.

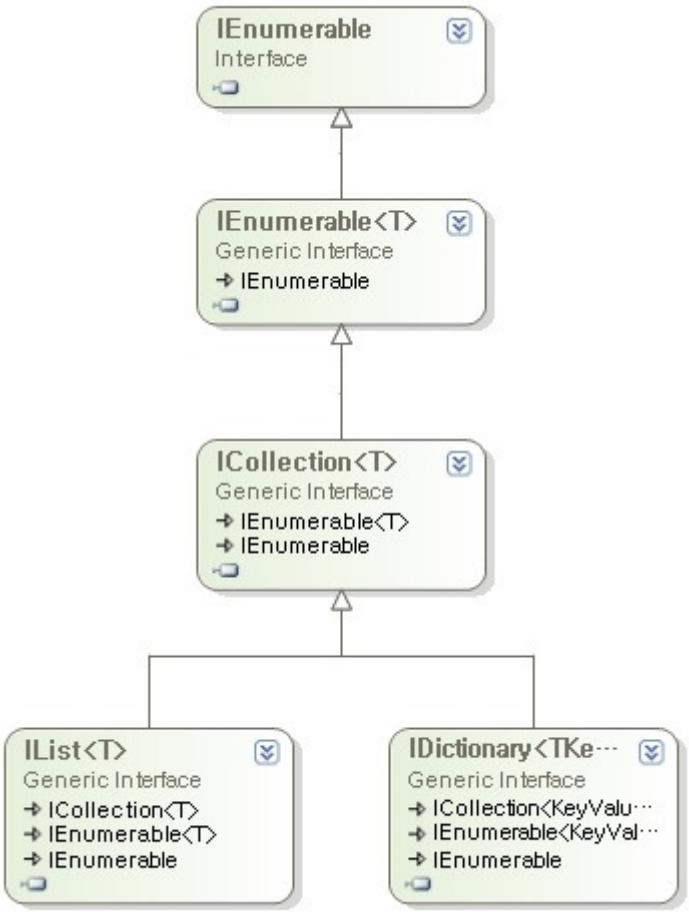

그림 1 주요 컬렉션 인터페이스들의 클래스 다이어그램

물론 베이스 클래스 라이브러리에는 훨씬 더 많은 수의 컬렉션 인터페이스들이 있습니다. 이 그림에서는 최상위에 있는 가장 중요한 몇 가지만을 표시하였습니다.

각각의 인터페이스들은 계층구조를 형성하고 있습니다. 예컨데 ICollection<T> 는 IEnumerable<T>를 상속 하고 있으며, IList<T>는 ICollection<T>를 상속 하고 있습니다. 따라서 IList<T> 역시 IEnumerable<T>를 간접적으로 상속 한다고 할 수 있겠습니다.

그러나 사실은, IList<T>는 ICollection<T>를 통해 IEnumerable<T>를 간접적으로 상속 하는 것이 아니라, 직접 IEnumerable<T>를 상속 합니다. 하지만 ICollection<T>가 IEnumerable<T>의 모든 계약을 상속 하고 있기 때문에, ICollection<T>를 상속 하면 IEnumerable<T>는 자동적으로 상속을 하게 되는 것입니다.

대부분의 컬렉션 클래스들은 적어도 ICollection<T>를 구현 합니다. Stack<T>이나 Queue<T> 등이 이에 속합니다. 조금 더 특수한 컬렉션들은 IList<T>나 IDictionary<TKey, TValue> 혹은 더 특화된 인터페이스를 구현하기도 합니다. 보통 인덱스로 요소에 접근하는 것을 허용하는 컬렉션은 IList<T>를 구현합니다. 배열과 List<T>가 대표적인 IList<T>를 구현하는 컬렉션입니다. 반면에 키-값 쌍으로 데이터를 저장하는 컬렉션은 IDictionary<TKey, TValue>를 구현합니다. Dictionary<TKey, TValue>를 예로 들 수 있겠네요.

IList<T>와 IDictionary<TKey, TValue>가 모두 ICollection<T>을 상속하는 것에 주목하여 주십시오. 따라서 IList<T> 혹은 IDictionary<TKey, TValue>를 구현하는 컬렉션은 동시에 ICollection<T>을 구현한 것이기도 합니다. 즉 IList<T>나 IDictionary<TKey, TValue>을 구현한 컬렉션은 ICollection<T>만 구현한 컬렉션에 비해서 좀 더 특화된 컬렉션이라고 할 수 있겠습니다.

이제 부터는 위 클래스 다이어그램에 등장하는 각각의 인터페이스에 대해서 하나씩 알아보도록 하겠습니다.

1. IEnumerable과 IEnumerable<T>

IEnumerable 인터페이스는 이름이 의미하는 바와 같이, 이 인터페이스를 구현하는 클래스를 열거 가능한 클래스로 만들어 줍니다. 즉 어떠한 클래스이더라도 IEnumerable 만 구현한다면 그 클래스는 **열거 가능한** 클래스가 되어 그 클래스를 foreach 문에서 사용할 수 있다는 이야기입니다.

(물론 인터페이스를 구현할 수 있는 것은 클래스 만이 아니라 구조체도 가능합니다. 서술의 편의상 여기서는, '인터페이스를 구현하는 클래스 혹은 구조체'라는 표현 대신에 그냥 '인터페이스를 구현하는 클래스'라고만 하겠습니다. 이러한 문맥에서는 특별한 언급이 없는 한, 클래스와 구조체를 구별하지 마시기 바랍니다..)

또한 IEnumerable<T>는 IEnumerable 의 제네릭 버전이고 IEnumerable 를 상속 합니다. 따라서 여기서는 IEnumerable<T>에 대해서만 다루도록 하겠습니다.

A. 열거

먼저 코드부터 보면서 IEnumerable<T> 인터페이스가 왜 필요한지에 대해서 생각해 봅시다. 여자 친구들의 이름을 저장하는 GirlFriendCollection 라는 현실성 없는 클래스를 생각해 보지요.

```
1 public class GirlFriendCollection
2 {
3     private string[] _names;
4
5     public GirlFriendCollection(params string[] names)
6     {
7         _names = names;
8     }
9 }
```

코드 1

5 번 라인에서 params 키워드를 사용하는 것 말고는 흔한 코드이네요. 배열을 매개 변수로 전달할 때 params 키워드를 사용하면 좀 더 편리하게 활용할 수 있습니다.

말이 나온 김에 params 키워드에 대해 잠시 알아보도록 하겠습니다. 예를 들어 아래와 같은 메서드가 있다고 합시다.

```
1 void Foo(int[] array)
2 {
3 }
4
5 void CallFoo()
6 {
7     Foo(new int[]{1, 2});
8 }
```

7 번 라인에서 보듯이 Foo 메서드에 1 과 2 를 넘기고자 한다면 먼저 배열을 만든 후 여기에 1 과 2 를 채워서 넘겨주어야 합니다. 하지만 params 키워드를 이용하면,

```
01 void Foo(params int[] array)
02 {
03 }
04
05 void CallFoo()
```

```
06 {
07     Foo(1, 2);
08     Foo(1, 2, 3);
09 }
```

와 같이 그냥 1 과 2 만 넘기면 컴파일러가 자동으로 배열을 만들어 줍니다. 8 번 라인 처럼 1, 2, 3 세 값을 넘기는 경우에도 그냥 주욱 넘기면 됩니다.

다음으로는 아래와 같이 여자 친구들의 이름을 foreach 문을 통해 출력을 하며 이국소녀들의 이름을 불러봅시다.

```
1 private static void Main(string[] args)
2 {
3     GirlFriendCollection myGirls = new GirlFriendCollection("패", "경", "옥");
4
5     foreach (string girl in myGirls)
6         Console.WriteLine(girl);
7 }
```

코드 2

5 번 라인을 확인하시기 바랍니다. 의도대로 myGirls 를 foreach 문을 통해 열거하기 위해서는 GirlFriendCollection 클래스를 **열거 가능하게** 만들어야 합니다. 이는 이름이 의미하는 것처럼 바로 IEnumerable<T>를 구현하는 것입니다.

B. 반복기를 사용하지 않는 IEnumerable<T> 구현

열거 가능한 클래스로 만들기 위해 구현하여야 할 IEnumerable<T> 인터페이스의 멤버는 다음과 같습니다. (MSDN 라이브러리)

메서드	
IEnumerator<T> GetEnumerator ()	컬렉션을 반복하는 열거자를 반환합니다.

표 1 IEnumerable<T>의 멤버

GirlFriendCollection 가 IEnumerable<T> 인터페이스를 구현하는 코드는 이런 형태가 되겠지요.

```
01 public class GirlFriendCollection : IEnumerable<string>
02 {
03     private string[] _names;
04
```

```
05     public GirlFriendCollection(params string[] names)
06     {
07         _names = names;
08     }
09
10     #region IEnumerable<string> Members
11     IEnumerator<string> IEnumerable<string>.GetEnumerator()
12     {
13         throw new System.NotImplementedException();
14     }
15     #endregion
16
17     #region IEnumerable Members
18     public IEnumerator GetEnumerator()
19     {
20         return ((IEnumerable<string>) this).GetEnumerator();
21     }
22     #endregion
23 }
```

코드 3

위에서 이야기했듯이 IEnumerable<T>는 IEnumerable 인터페이스를 상속합니다. 따라서 IEnumerable<T>를 구현하기 위해서는 IEnumerable 의 멤버인 GetEnumerator 도 같이 구현하여야 합니다. 그것이 18 ~ 21 번 라인입니다. 실제 구현은 11 ~ 14 번 라인의 IEnumerable<string>.GetEnumerator 를 다시 호출하고 있습니다.

또한 11 번 라인처럼 인터페이스를 명시적으로 구현하는 방법에 대해서도 유념하시기 바랍니다. 코드 3 의 경우에는 이름이 동일한 GetEnumerator 메서드가 두 개 있습니다. (각각 IEnumerable 와 IEnumerable<T>의 멤버) 따라서 아래와 같이 두 인터페이스 멤버를 모두 암시적으로 구현을 하게 되면 이름 충돌이 발생하게 됩니다.

```
01 #region IEnumerable<string> Members
02 public IEnumerator<string> GetEnumerator()
03 {
04     throw new System.NotImplementedException();
05 }
06 #endregion
07
08 #region IEnumerable Members
09 public IEnumerator GetEnumerator()
10 {
11     return ((IEnumerable<string>) this).GetEnumerator();
12 }
13 #endregion
```

코드 4

이를 해결하기 위해서 코드 3 에서는 IEnumerable<T>의 GetEnumerator 를 명시적으로 구현하고 있는 것입니다.

코드 4 의 11 번 라인과 같이 IEnumerable.GetEnumerator()가 IEnumerable<T>.GetEnumerator()를 호출하는 코드도 유심히 보시기 바랍니다. 그냥 GetEnumerator() 라고 해서는 어떤 GetEnumerator()를 말하는지 알 수가 없습니다. 따라서 위 코드에서는 this. GetEnumerator() 라고 호출하는 것이 아니라 this 를 IEnumerable<string>로 형변환 한 후 GetEnumerator()를 호출하고 있습니다. 어려운 문법은 아니지만 생각을 좀 해봐야 하는 부분이네요.

코드 3 에서 IEnumerable 의 GetEnumerator 는 IEnumerable<T>의 GetEnumerator 를 호출하는 것으로 이미 구현이 되어 있기 때문에, 우리는 IEnumerable<T>의 GetEnumerator 만 구현을 하면 되겠습니다. 인터페이스의 정의에 의하면 이 메서드는 열거자를 반환하는데, 이 열거자는 IEnumerator <string>를 구현하여야 합니다. 그렇다면 우리는 IEnumerator<string> 인터페이스를 구현하는 클래스나 구조체도 작성하여야 한다는 말이 되겠네요.

여기서 잠시 IEnumerable(혹은 IEnumerable<T>)와 IEnumerator(혹은 IEnumerator<T>)에 대해서 정리를 하는 게 좋을 것 같습니다. 둘 다 클래스를 열거 가능하게 하는데 사용되는 인터페이스이긴 한데, 어떤 차이점이 있을까요?

IEnumerable 은 단어의 의미대로 이 인터페이스를 구현하면 **열거가 가능한** 클래스가 된다는 의미입니다. 이에 반해 IEnumerator 인터페이스를 구현한 클래스는 열거 가능한 클래스가 되는 것이 아니라 **열거자 클래스**가 된다는 뜻이고요.

우리의 예에서 GirlFriendCollection 클래스는 열거가 가능한 클래스이지, 열거자 클래스는 아닙니다. 따라서 GirlFriendCollection 는 IEnumerable 를 구현합니다. 그렇다면 IEnumerator 는 언제 사용하느냐? 위에서 보신 것처럼 IEnumerable 의 GetEnumerator() 메서드는 반환값으로

IEnumerator, 즉 IEnumerator 나 이를 구현한 열거자 클래스를 반환하여야 합니다. 곧 보시겠지만, 우리는 GirlFriendCollection 의 열거자 클래스인 GirlFriendEnumerator 클래스를 만들고, 이를 GetEnumerator()의 반환값으로 반환할 것입니다.

참고로 닷넷 프레임웍에는 이렇게 –able 과 –er(or)로 끝나는 인터페이스 쌍이 몇 가지 더 있습니다. IComparable 과 IComparer 가 대표적인 예이지요. 만약에 GirlFriend 라는 클래스가 있고 이를 비교가 가능한 클래스로 만들고 싶다면 GirlFriend 가 IComparable 를 구현하도록 만들어야 하고, GirlFriend 를 비교하는 클래스를 만들고 싶다면 IComparer 를 구현하는 GirlFriendComparer 클래스를 만들어야할 것입니다.

IEnumerator<T> 인터페이스는 아래와 같은 멤버를 가지고 있습니다. (MSDN 라이브러리)

속성	
`T Current { get; }`	컬렉션에서 열거자의 현재 위치에 있는 요소입니다.

표 2 IEnumerator<T>의 멤버

그리고 IEnumerator<T> 인터페이스는 IEnumerator 와 IDisposable 을 상속하기 때문에, 우리가 작성하려고 하는 열거자는 아래와 같은 IEnumerator 와 IDisposable 의 멤버도 구현을 하여야 합니다. (MSDN 라이브러리)

속성	
`Object Current { get; }`	컬렉션의 현재 요소입니다.
메서드	
`bool MoveNext ()`	열거자를 컬렉션의 다음 요소로 이동합니다.
`void Reset ()`	컬렉션의 첫 번째 요소 앞의 초기 위치에 열거자를 설정합니다.

표 3 IEnumerator 의 멤버

메서드	
`void Dispose ()`	관리되지 않는 리소스의 확보, 해제 또는 다시 설정과 관련된 응용 프로그램 정의 작업을

수행합니다.

표 4 IDisposable의 멤버

그래서 우리가 작성하려고 하는 열거자의 이름이 GirlFriendEnumerator 라면, GirlFriendEnumerator 는 다음과 같이 IEnumerator<T>와 IEnumerator 와 IDisposable 의 멤버를 모두 구현하하여야 합니다.

```
01 public class GirlFriendCollection : IEnumerable<string>
02 {
03     private string[] _names;
04 
05     public GirlFriendCollection(params string[] names)
06     {
07         _names = names;
08     }
09 
10     #region IEnumerable<string> Members
11     IEnumerator<string> IEnumerable<string>.GetEnumerator()
12     {
13         GirlFriendEnumerator enumerator = new GirlFriendEnumerator(this);
14         return enumerator;
15     }
16     #endregion
17 
18     #region IEnumerable Members
19     public IEnumerator GetEnumerator()
20     {
21         return ((IEnumerable<string>) this).GetEnumerator();
22     }
23     #endregion
24 
25     private class GirlFriendEnumerator : IEnumerator<string>
26     {
27         private readonly GirlFriendCollection _girlFriends;
28 
29         public GirlFriendEnumerator(GirlFriendCollection girlFriends)
30         {
31             _girlFriends = girlFriends;
32         }
33 
34         private int _index = -1;
35 
36         #region IDisposable Members
37         public void Dispose()
38         {
39         }
40         #endregion
41 
42         #region IEnumerator Members
43         public bool MoveNext()
44         {
```

```
45            _index++;
46            return _index < _girlFriends._names.Length;
47        }
48
49        public void Reset()
50        {
51            _index = -1;
52        }
53
54        public object Current
55        {
56            get { return ((IEnumerator<string>)this).Current; }
57        }
58        #endregion
59
60        #region IEnumerator<string> Members
61        string IEnumerator<string>.Current
62        {
63            get
64            {
65                try
66                {
67                    return _girlFriends._names[_index];
68                }
69                catch (IndexOutOfRangeException)
70                {
71                    throw new InvalidOperationException();
72                }
73
74            }
75        }
76        #endregion
77    }
78 }
```

코드 5

먼저 GirlFriendEnumerator 를 GirlFriendCollection 의 내포 클래스로 구현한 것을 유심히 보아 주십시오. GirlFriendEnumerator 가 반드시 GirlFriendCollection 의 내포된 클래스로 구현되어야 하는 것은 아닙니다. 여기서는 GirlFriendEnumerator 클래스를 GirlFriendCollection 내부에서만 사용하기 때문에 외부로 노출시키지 않았습니다. 필드가 아닌 클래스 레벨의 캡슐화 원칙이라고 할 수 있겠네요.

GirlFriendEnumerator 를 내포된 클래스로 선언하는 25 번 라인을 보면 한 가지 재미있는 곳이 있습니다. 클래스의 접근 지정자가 private 으로 되어 있습니다. 클래스가 다른 클래스에 내포되지 않고 네임 스페이스에서 바로 정의될 경우에 지정할 수 있는 접근 지정자는 두 가지 밖에

없습니다. public 과 internal 이지요. 그 외의 다른 세 가지 접근 지정자(private, protected, protected internal)은 지정할 수 없습니다. 하지만 클래스가 다른 클래스에 내포가 된다면 다섯 가지의 접근 지정자를 모두 사용할 수 있습니다. 왜 일까요?

아마도 C#을 설계한 사람들은 이 문제에 대해서 많은 고민을 하고 결정을 내렸을 것입니다. 이 글을 읽는 분들도 C#을 만든 사람들의 마음을 생각하며 같은 고민을 해보시기 바랍니다. 그러한 고민을 많이 할수록 언어에 대한 이해의 폭이 늘어나는 것이 아닌가 합니다. (비슷한 예로 추상 클래스의 생성자는 오직 protected 만이 의미가 있습니다. 왜 그런지에 대한 답을 스스로 고민하여 찾아낼 수 있다면, 추상 클래스, 더 나아가서 OOP 에 대한 이해가 부쩍 증가한 걸 느끼실 수가 있을 것입니다.)

다시 GirlFriendEnumerator 의 이야기를 계속 하자면, GirlFriendEnumerator 는 GirlFriendCollection 형의 _girlFriends 와 int 형의 _index, 두 개의 필드를 가지고 있습니다. _girlFriends 는 실제로 열거를 할 객체에 대한 참조이고요, _index 는 여자 친구들의 이름이 담긴 배열의 인덱스를 나타냅니다.

11 번 라인의 IEnumerable<string>.GetEnumerator 메서드의 구현은 GirlFriendEnumerator 객체를 생성하여 반환하는 것으로 되어 있습니다. 이때 GirlFriendEnumerator 의 생성자에 매개변수로 GirlFriendCollection 의 객체를 넘기는 것을 확인하시기 바랍니다.

37 번 라인에서는 IDisposable.Dispose 를 구현하는데, 여기서는 아무런 일도 하지 않습니다. 43 번 라인의 MoveNext 메서드는 현재의 인덱스를 뒤로 한 칸 옮기고, 남아 있는 요소가 있으면 true, 없으면 false 를 반환합니다. 그리고 Reset 메서드는 현재의 인덱스를 -1 로 설정을 합니다. 인덱스가 0 이라면 첫번째 요소를 가리키는 것이므로, -1 은 첫번째 요소의 앞을 의미합니다. 마지막으로 Current 속성은 현재 요소를 반환하는데, 위 코드에서는 단순히 IEnumerator<T>.Current 를 반환하고 있습니다.

61 번 라인부터 시작하는 IEnumerator<T>.Current 속성이 실제로 값을 열거하는 코드인데요, 구현은 간단합니다. 단순히 이름 배열의 _index 번째 요소를 반환합니다. 또한 여기서도 IEnumerator.Current 와 IEnumerator<T>.Current 의 이름이 중복되기 때문에 IEnumerator<T>.Current 를 명시적으로 구현한 것을 볼 수 있습니다.

이제 코드 2 는 정상적으로 실행이 됩니다.

```
패
경
옥
Press any key to continue . . .
```

여기까지가 GirlFriendCollection 클래스에 열거 기능을 추가하기 위한 작업이었습니다. 간단한 열거 기능을 구현하기 위해서 꽤 많은 양의 코드를 작성하였습니다. 경우에 따라서는 지루하고 반복적인 작업이라서 실수를 할 가능성도 있습니다. 하지만 반갑게도 C# 2.0 에는 이러한 반복 작업을 대폭 줄일 수 있는 기능이 언어에 추가되었습니다. 바로 반복기를 사용하는 것입니다.

C. 반복기를 사용하는 IEnumerable<T> 구현

C# 2.0 에서 도입된 반복기를 사용하면 열거자를 구현하는 작업을 획기적으로 줄일 수 있습니다. 한 두줄의 코드만 추가해 주면 나머지 코드는 컴파일러가 컴파일 타임에 자동으로 만들어 냅니다.

반복기를 지원하기 위해서 C# 2.0 에는 yield 라는 새로운 키워드가 도입되었습니다. 바로 예제를 보면서 이야기를 하지요. 아래는 반복기를 사용하여 GirlFriendCollection 에 열거 기능을 추가한 코드입니다.

```
01 public class GirlFriendCollection : IEnumerable<string>
02 {
03     private string[] _names;
04
05     public GirlFriendCollection(params string[] names)
06     {
07         _names = names;
08     }
09
10     #region IEnumerable<string> Members
```

```
11      IEnumerator<string> IEnumerable<string>.GetEnumerator()
12      {
13          for (int i = 0; i < _names.Length; i++)
14              yield return _names[i];
15      }
16      #endregion
17
18      #region IEnumerable Members
19      public IEnumerator GetEnumerator()
20      {
21          return ((IEnumerable<string>) this).GetEnumerator();
22      }
23      #endregion
24  }
```

코드 6

GirlFriendEnumerator 와 같은 열거자를 만드는 코드는 몽땅 빠지고 대신에 13 ~ 14 라인만이 추가되었습니다. 앞에서 했던 작업들에 비하면 허무할 만큼 코드가 간단해졌습니다. 14 번 라인에서 return 문 앞에 yield 키워드가 붙으면 현재 위치를 저장하고 현재 요소의 값을 반환합니다. 이후 다음 반복에서는 저장된 현재 위치를 한 칸 뒤로 옮기고 나서 새로운 현재 요소의 값을 반환합니다. 즉 GirlFriendEnumerator 와 동일한 역할을 하는 것입니다.

C# 2.0 이상을 사용할 수 있는 환경이라면 아마도 반복기를 사용하지 않고 열거 가능한 클래스를 만드는 경우는 없을 것입니다. 그렇다고 해서 코드 5 와 같이 반복기를 사용하지 않는 코드가 의미가 없다는 의미는 아닙니다. 코드 5 는 이른바 이터레이터 패턴입니다. 이터레이터 패턴의 구조를 숙지하고 있다면 yield 키워드에 의해 컴파일러가 생성해내는 코드에 대해서도 쉽게 추정이 가능할 것입니다. 즉 닷넷 프레임웍의 내부 동작에 대해 더 많은 것을 이해하게 되고, 그 만큼 더 정확하고 효율적으로 닷넷 프레임웍을 사용할 수 있다는 의미가 되겠습니다.

2. ICollection<T>

두번째로 살펴볼 인터페이스는 ICollection<T>입니다. ICollection<T> 인터페이스는 자신을 구현하는 클래스 혹은 구조체를 컬렉션으로 만들어줍니다. 어떤 객체를 우리가 컬렉션이라고 부르려면 기본적으로 몇 가지 연산을 수행할 수 있어야 합니다. 예를 들어 새로운 요소를 추가 / 삭제 한다든지, 요소의 개수를 반환한다든지 하는 연산을 들 수 있습니다. ICollection<T>

인터페이스는 바로 이러한 연산을 구체적으로 정의해 놓은 것입니다. ICollection<T>의 멤버는 다음과 같습니다.(MSDN 라이브러리)

속성	
`int Count { get; }`	ICollection 에 포함된 요소 수를 가져옵니다.
`bool IsReadOnly { get; }`	ICollection 가 읽기 전용인지 여부를 나타내는 값을 가져옵니다.
메서드	
`void Add (T item)`	ICollection 에 항목을 추가합니다.
`void Clear ()`	ICollection 에서 항목을 모두 제거합니다.
`bool Contains (T item)`	ICollection 에 특정 값이 들어 있는지 여부를 확인합니다.
`void CopyTo (T[] array, int arrayIndex)`	특정 Array 인덱스부터 시작하여 ICollection 의 요소를 Array 에 복사합니다.
`bool Remove (T item)`	ICollection 에서 맨 처음 발견되는 특정 개체를 제거합니다.

표 5 ICollection<T>의 멤버

간략히 정리를 하자면, ICollection<T> 인터페이스를 구현한 컬렉션은 요소를 추가, 삭제 할 수 있으며, 특정 요소를 가지고 있는지 검사하거나, 전체 요소를 모두 지울 수도 있습니다. 그리고 현재 요소의 개수를 반환할 수도 있습니다.

또한 ICollection<T>는 IEnumerable<T>를 상속하고 있습니다. 따라서 ICollection<T> 인터페이스를 구현하는 클래스는 IEnumerable<T>의 멤버도 구현하여야 하는데, 이는 이 클래스가 열거 가능한 클래스가 된다는 말이기도 합니다.

3. IList<T>

IList<T> 인터페이스는 인덱스를 사용하여 요소에 개별적으로 접근할 수 있는 컬렉션을 정의합니다. 멤버는 다음과 같습니다.(MSDN 라이브러리)

속성	
`T this [int index] { get; set; }`	지정한 인덱스에 있는 요소를 가져오거나 설정합니다.

메서드	
`int IndexOf (T item)`	`IList` 에서 특정 항목의 인덱스를 확인합니다.
`void Insert (int index, T item)`	항목을 `IList` 의 지정한 인덱스에 삽입합니다.
`void RemoveAt (int index)`	지정한 인덱스에서 `IList` 항목을 제거합니다.

표 6 IList<T> 멤버

1 개의 속성과 3 개의 메서드가 있는데, 모두 인덱스와 관련이 있습니다. 인덱서를 통해 요소의 값을 읽거나 쓸 수 있고, 특정 요소의 인덱스 번호를 알 수도 있습니다. ICollection<T>으로부터 상속 받은 Add 와 Remove 를 사용하여 요소를 추가, 삭제 할 수 있지만, 이에 더하여 특정 인덱스에 요소를 삽입하거나 특정 인덱스에 있는 요소를 삭제할 수 있는 메서드도 가지고 있습니다.

4. IDictionary<TKey, TValue>

IDictionary<TKey, TValue> 인터페이스는 키-값 쌍으로 데이터를 그룹화 하는 컬렉션을 정의합니다. 또한 IDictionary<TKey, TValue> 인터페이스는 이때까지 등장한 인터페이스와는 달리 제네릭 형식 매개 변수를 두 개를 가지고 있습니다. 따라서 IDictionary<TKey, TValue>는 IList<T> 처럼 ICollection<T>를 상속하지만 서로 약간 다릅니다. IList<T>의 정의가 아래와 같음에 비해,

```
public interface IList<T> : ICollection<T> {…}
```

IDictionary<TKey, TValue>의 정의는 이렇습니다.

```
public interface
IDictionary<TKey,TValue> : ICollection<KeyValuePair<TKey,TValue>> {…}
```

키-값 쌍을 나타내는 제네릭 구조체인 KeyValuePair 를 ICollection<T>의 형식 매개 변수로 지정하고 있습니다.

IDictionary<TKey, TValue> 멤버는 다음과 같습니다.

속성	
`TValue this [TKey key] { get; set; }`	지정된 키가 있는 요소를 가져오거나 설정합니다.
`ICollection<TKey> Keys { get; }`	`IDictionary` 의 키를 포함하는 `ICollection` 을

	가져옵니다.
ICollection<TValue> Values { get; }	IDictionary 의 값을 포함하는 ICollection 을 가져옵니다.
메서드	
void Add (TKey key, TValue value)	제공된 키와 값이 있는 요소를 IDictionary 에 추가합니다.
bool ContainsKey (TKey key)	IDictionary 에 지정된 키가 있는 요소가 포함되어 있는지 여부를 확인합니다.
bool Remove (TKey key)	IDictionary 에서 지정한 키를 가지는 요소를 제거합니다.
bool TryGetValue (TKey key, out TValue value)	지정된 키와 연결된 값을 가져옵니다.

IDictionary<TKey, TValue>는 키를 통해서 특정 요소에 접근할 수 있습니다. 문법은 IList<T>와 같이 인덱서를 사용하지만, IList<T>는 인덱서의 매개 변수가 int 형인 인덱스인 반면, IDictionary<TKey, TValue>는 인덱서의 매개 변수가 TKey 형인 키라는 것이 다릅니다. 또한 각각 키 혹은 값 만으로 된 ICollection<T> 객체를 반환할 수도 있습니다. (Keys 속성과 Values 속성)

두 개의 형식 매개 변수가 있기 때문에 Add 와 Remove 메서드 역시 시그니처가 다릅니다. 또한 컬렉션이 특정 키 값을 가지고 있는지를 알아볼 수도 있습니다.

III. 컬렉션과 메서드, 그리고 대리자

C# 1.0 에서 가장 많이 사용된 컬렉션은 아마도 ArrayList 일 것입니다. List<T>는 닷넷 프레임웍 2.0 에서 추가된 ArrayList 의 제네릭 버전입니다. ArrayList 는 List<T>에 비하면 어떠한 장점도 가지고 있지 않습니다. 따라서 C# 2.0 이상에서 코딩을 하는 경우라면, 하위 호환성을 고려해야하는 등의 특수한 사정이 있지 않다면, ArrayList 를 사용하는 경우는 없을 것입니다. 그래서 아마도 C# 2.0 이상에서 가장 많이 사용할 컬렉션은 List<T>가 될 것입니다.

또한 List<T>는 ArrayList 에서 object 를 T 로 일반화시킨 것 이상입니다. 새로 도입된 제네릭의 도움을 받아 다수의 메서드와 대리자를 제공하기도 합니다. 이 단원에서는 새로 도입된 헬퍼 메서드와 대리자에 대해서 살펴보도록 하겠습니다.

(註 : 이 단원에서 다루는 대부분의 메서드와 대리자는 배열, 즉 Array 객체에서도 사용할 수 있습니다. 단지 Array 클래스는 제네릭이 아니므로, List<T>의 메서드와 동일한 이름의 메서드가 정적 제네릭 메서드로 구현되어 있습니다.)

(註의 註 : List<T>. Sort 는 제네릭 클래스의 메서드이지 제네릭 메서드가 아닙니다. 반면에 Array.Sort<T>()는 제네릭 메서드가 맞습니다.)

1. 컬렉션의 헬퍼 메서드

List<T> 컬렉션의 멤버 중에는 몇 가지 흥미로운 메서드가 있습니다. ForEach 나 TrueForAll 과 같은 이름을 가진 이 메서드들을 살펴 보기 전에 먼저 아래 예제를 보도록 합시다.

여자 친구의 나이와 이름을 표현하는 GirlFriend 객체를 몇 개 만들어 이를 List<T> 컬렉션에 담는 코드를 생각해 보지요.

```
01 public class GirlFriend
02 {
03     private readonly string _name;
04 
05     public string Name
06     {
07         get { return _name; }
08     }
09 
10     private readonly int _age;
11 
12     public int Age
13     {
14         get { return _age; }
15     }
16 
17     public GirlFriend(string name, int age)
18     {
19         _name = name;
20         _age = age;
21     }
22 
23     public override string ToString()
24     {
25         return string.Format("이름 : {0}\t 나이 : {1}", _name, _age);
26     }
27 }
```

코드 7

```
1 private static void Main(string[] args)
2 {
3     List<GirlFriend> myGirls = new List<GirlFriend>();
4     myGirls.Add(new GirlFriend("패", 19));
5     myGirls.Add(new GirlFriend("경", 21));
6     myGirls.Add(new GirlFriend("옥", 20));
7 }
```

코드 8

이제 myGirls 컬렉션에 포함된 세 이국 소녀가 모두 19 세 이상인지를 확인하는 코드를 추가합시다.

```
01 private static void Main(string[] args)
02 {
03     List<GirlFriend> myGirls = new List<GirlFriend>();
04     myGirls.Add(new GirlFriend("패", 19));
05     myGirls.Add(new GirlFriend("경", 21));
06     myGirls.Add(new GirlFriend("옥", 20));
07
08     bool result = true;
09     foreach (GirlFriend myGirl in myGirls)
10     {
11         if (myGirl.Age < 19)
12         {
13             result = false;
14             break;
15         }
16     }
17 }
```

코드 9

8 ~ 16 라인의 로직은 두 부분으로 나눌 수 있습니다. 1) myGirls 컬렉션 객체의 각 요소를 열거하고, 2) 각 요소가 특정 조건을 만족하는지를 검사하는 것이 그것입니다.

이번에는 요구 사항을 조금 수정하여 세 소녀의 이름이 모두 외자인자를 체크하도록 코드를 변경해 봅시다. 8 ~ 16 라인의 로직이 두 부분으로 이루어져 있고, 새로운 요구 사항은 이 중 2) 부분에만 변경이 일어나고 있습니다. 그렇다면 위 01 private static void Main(string[] args)

```
02 {
03     List<GirlFriend> myGirls = new List<GirlFriend>();
04     myGirls.Add(new GirlFriend("패", 19));
05     myGirls.Add(new GirlFriend("경", 21));
06     myGirls.Add(new GirlFriend("옥", 20));
```

```
07
08     bool result = true;
09     foreach (GirlFriend myGirl in myGirls)
10     {
11         if (myGirl.Age < 19)
12         {
13             result = false;
14             break;
15         }
16     }
17 }
```

코드 9 의 11 번 라인만 아래와 같이 바꾸면 되지 않을까요.

```
01 private static void Main(string[] args)
02 {
03     List<GirlFriend> myGirls = new List<GirlFriend>();
04     myGirls.Add(new GirlFriend("패", 19));
05     myGirls.Add(new GirlFriend("경", 21));
06     myGirls.Add(new GirlFriend("옥", 20));
07
08     bool result = true;
09     foreach (GirlFriend myGirl in myGirls)
10     {
11         if (myGirl.Name.Length != 1)
12         {
13             result = false;
14             break;
15         }
16     }
17 }
```

코드 10

```
01 private static void Main(string[] args)
02 {
03     List<GirlFriend> myGirls = new List<GirlFriend>();
04     myGirls.Add(new GirlFriend("패", 19));
05     myGirls.Add(new GirlFriend("경", 21));
06     myGirls.Add(new GirlFriend("옥", 20));
07
08     bool result = true;
09     foreach (GirlFriend myGirl in myGirls)
10     {
11         if (myGirl.Age < 19)
12         {
13             result = false;
14             break;
15         }
16     }
17 }
```

코드 9 과 01 private static void Main(string[] args)

```
02 {
03     List<GirlFriend> myGirls = new List<GirlFriend>();
04     myGirls.Add(new GirlFriend("패", 19));
05     myGirls.Add(new GirlFriend("경", 21));
06     myGirls.Add(new GirlFriend("옥", 20));
07
08     bool result = true;
09     foreach (GirlFriend myGirl in myGirls)
10     {
11         if (myGirl.Name.Length != 1)
12         {
13             result = false;
14             break;
15         }
16     }
17 }
```

코드 10 는 11 번 라인을 제외하면 완전히 동일합니다. 그렇다면 11 번 라인을 제외한 공통인 부분을 재사용할 수는 없을까요? 메서드로 묶을 수 있을 겉 같은데, 이 메서드는 List<T>의 각 요소를 열거하며 어떤 일을 하고 있으니까, 아예 List<T>의 메서드로 만드는 것이 어떨까요? 한번 만들어 볼까요?

```
01 public delegate bool Predicate<T>(T item);
02
03 public class List<T> : …
04 {
05     ...
06
07     public bool TrueForAll(Predicate<T> match)
08     {
09         bool result = true;
10         foreach (T item in this)
11         {
12             if (match(item) == false)
13             {
14                 result = false;
15                 break;
16             }
17         }
18     }
19 }
```

코드 11

코드 11 은 List<T> 클래스의 TrueForAll 메서드의 구현을 추정을 해서 만들어 본 것입니다. 먼저 1 번 라인에서 T 형의 형식 매개 변수를 가지는 제네릭 대리자를 선언하였습니다. 이 대리자는 T 형의 객체를 매개 변수로 받고 bool 형을 반환하고 있습니다. 7 번 라인에서는

TrueForAll 이라는 메서드를 정의하고 있는데, 이름이 의미하는 바와 같이, 컬렉션의 모든 원소에 대해 매개 변수로 받은 대리자를 실행시켰을 때 (12 번 라인) 참이라는 결과가 나오는지를 검사하는 일을 합니다.

이제 01 private static void Main(string[] args)
02 {
03 List<GirlFriend> myGirls = new List<GirlFriend>();
04 myGirls.Add(new GirlFriend("패", 19));
05 myGirls.Add(new GirlFriend("경", 21));
06 myGirls.Add(new GirlFriend("옥", 20));
07
08 bool result = true;
09 foreach (GirlFriend myGirl in myGirls)
10 {
11 if (myGirl.Age < 19)
12 {
13 result = false;
14 break;
15 }
16 }
17 }

코드 9 과 01 private static void Main(string[] args)
02 {
03 List<GirlFriend> myGirls = new List<GirlFriend>();
04 myGirls.Add(new GirlFriend("패", 19));
05 myGirls.Add(new GirlFriend("경", 21));
06 myGirls.Add(new GirlFriend("옥", 20));
07
08 bool result = true;
09 foreach (GirlFriend myGirl in myGirls)
10 {
11 if (myGirl.Name.Length != 1)
12 {
13 result = false;
14 break;
15 }
16 }
17 }

코드 10 는 아래와 같이 바뀌게 됩니다.

01 private static void Main(string[] args)
02 {
03 List<GirlFriend> myGirls = new List<GirlFriend>();
04 myGirls.Add(new GirlFriend("패", 19));
05 myGirls.Add(new GirlFriend("경", 21));
06 myGirls.Add(new GirlFriend("옥", 20));

```
07
08      // 모든 소녀가 19 세 이상인지를 검사
09      bool result1 = myGirls.TrueForAll(IsAdult);
10
11      // 모든 소녀가 외자 이름인지를 검사
12      bool result2 = myGirls.TrueForAll(delegate(GirlFriend myGirl)
13                                        {
14                                            return myGirl.Name.Length == 1;
15                                        });
16  }
17
18  private static bool IsAdult(GirlFriend myGirl)
19  {
20      return myGirl.Age >= 19;
21  }
```

코드 12

9 번과 12 번 라인에서 각각 모든 소녀가 19 세 이상인지, 혹은 외자 이름을 가지고 있는지를 검사하고 있습니다. 01 private static void Main(string[] args)

```
02  {
03      List<GirlFriend> myGirls = new List<GirlFriend>();
04      myGirls.Add(new GirlFriend("패", 19));
05      myGirls.Add(new GirlFriend("경", 21));
06      myGirls.Add(new GirlFriend("옥", 20));
07
08      bool result = true;
09      foreach (GirlFriend myGirl in myGirls)
10      {
11          if (myGirl.Age < 19)
12          {
13              result = false;
14              break;
15          }
16      }
17  }
```

코드 9 와 01 private static void Main(string[] args)

```
02  {
03      List<GirlFriend> myGirls = new List<GirlFriend>();
04      myGirls.Add(new GirlFriend("패", 19));
05      myGirls.Add(new GirlFriend("경", 21));
06      myGirls.Add(new GirlFriend("옥", 20));
07
08      bool result = true;
09      foreach (GirlFriend myGirl in myGirls)
10      {
11          if (myGirl.Name.Length != 1)
12          {
13              result = false;
14              break;
15          }
```

```
16      }
17  }
```

코드 10 에 비하면 재사용을 통해 코드의 양이 대폭 줄어든 것을 확인할 수 있습니다.

또한 9 번 라인과 12 번 라인은 대리자 메서드를 전달할 때 이름 있는 메서드와 이름 없는 메서드(무명 메서드)를 각각 사용하고 있습니다. 메서드가 비교적 간단하다면 무명 메서드를 사용하는 것이 코드의 가독성을 높이는 방법이 될 수 있습니다.

코드 11 에서 사용된 Predicate 대리자나 List<T>.TrueForAll 메서드는 사실 이미 베이스 클래스 라이브러리에 존재하는 대리자와 메서드입니다. 베이스 클래스 라이브러리의 컬렉션 클래스는 이 외에도 컬렉션 작업에서 유용하게 사용할 수 있는 대리자와 메서드를 다수 제공하고 있습니다. 다음은 이런 메서드의 목록입니다.

- public List<TOutput> ConvertAll<TOutput>(Converter<T, TOutput> converter)

  List<TInput> 객체의 각 원소를 TOutput 형으로 변환하여 List<TOutput>을 반환합니다.

- public bool Exists(Predicate<T> match)

  리스트에 있는 모든 원소 중 match 조건을 만족하는 원소가 있는지를 검사합니다.

- public T Find(Predicate<T> match)

  리스트에 있는 모든 원소 중 match 조건을 만족하는 첫번째 원소를 반환합니다.

- public List<T> FindAll(Predicate<T> match)

  리스트에 있는 모든 원소 중 match 조건을 만족하는 모든 원소를 반환합니다.

- public int FindIndex(Predicate<T> match

리스트에 있는 모든 원소 중 match 조건을 만족하는 첫번째 원소의 인덱스를 반환합니다.

- public int FindLastIndex(Predicate<T> match)

  리스트에 있는 모든 원소 중 match 조건을 만족하는 마지막 원소의 인덱스를 반환합니다.

- public void ForEach(Action<T> action)

  리스트의 모든 원소에 대해 action 을 수행합니다.

- public bool TrueForAll(Predicate<T> match)

  리스트에 있는 모든 원소가 match 조건을 만족하는지 검사합니다.

2. 컬렉션의 헬퍼 대리자

앞 단원에서 거론된 모든 메서드는 자체적으로 기능을 수행할 수 없고, 다른 메서드의 도움을 받아야 합니다. 예를 들어 ForEach 메서드의 경우, List<T>의 모든 원소를 순회하면서 어떤 작업을 하는데, 이 '어떤 작업'에 대한 정보를 매개 변수로 받습니다. '작업'이 의미하는 것 처럼 이는 메서드를 나타냅니다. 결국 ForEach 메서드는 다른 메서드를 매개 변수로 받습니다. 메서드를 매개 변수로 전달하기 위해서는 대리자를 사용합니다. 아래 목록은 컬렉션의 작업을 편리하게 하는 제네릭 대리자의 목록입니다.

- public delegate void Action<T>(T obj)

  T 형의 매개변수를 하나 받고 반환값이 없는 메서드를 나타냅니다.

- public delegate TOutput Converter<TInput, TOutput>(TInput input)

TInput 형의 매개변수를 받고 이를 TOutput 형을 변환하여 반환하는 메서드를 나타냅니다.

- public delegate bool Predicate<T>(T obj)

  T 형의 매개변수를 받아 그것이 특정 조건을 만족하는지를 반환하는 메서드를 나타냅니다.

- public delegate int Comparison<T>(T x, T y)

  두 객체를 비교하는 메서드를 나타냅니다. x 가 y 보다 작으면 음수, 같으면 0, 크면 양수를 반환합니다.

이러한 대리자를 사용하는 예는 코드 12 를 참조하시기 바랍니다.

IV. 순서 비교자

1. Sort 메서드

List<T>에는 Sort 메서드가 있어 쉽게 정렬이 가능합니다. 코드를 보면서 이야기를 해봅시다.

GirlFriend 라는 클래스가 있다고 하지요.

```
01 public class GirlFriend
02 {
03     private readonly string _name;
04
05     public string Name
06     {
07         get { return _name; }
08     }
09
10     private readonly int _age;
11
12     public int Age
13     {
14         get { return _age; }
15     }
16
17     public GirlFriend(string name, int age)
18     {
19         _name = name;
```

```
20          _age = age;
21      }
22
23      public override string ToString()
24      {
25          return string.Format("이름 : {0}\t 나이 : {1}", _name, _age);
26      }
27  }
```

코드 13

여자 친구 몇 명을 List<T>에 추가 하고 정렬을 해보도록 합시다.

```
01 private static void Main(string[] args)
02 {
03     List<GirlFriend> myGirls = new List<GirlFriend>();
04     myGirls.Add(new GirlFriend("패", 19));
05     myGirls.Add(new GirlFriend("경", 21));
06     myGirls.Add(new GirlFriend("옥", 20));
07
08     myGirls.Sort();
09
10     foreach (GirlFriend girl in myGirls)
11         Console.WriteLine(girl);
12 }
```

코드 14

안타깝게도 InvalidOperationException 예외가 발생합니다. 어떻게 보면 당연한 이야기입니다. 아무런 기준이나 근거도 없이 여자 친구들을 정렬한다는게 어불성설이지요. 하마 못해, 나이 혹은 이름 순으로 정렬하겠다는 정보라도 있어야 정렬을 할 수 있지 않겠습니까?

이야기를 진행시키기 전에 List<T>. Sort 메서드의 시그니처를 살펴 봅시다.

```
public void Sort()
public void Sort(Comparison<T> comparison)
public void Sort(IComparer<T> comparer)
public void Sort(int index, int count, IComparer<T> comparer)
```

네 가지 오버로드가 있는데요, 세 번째와 네 번째는 동일한 로직입니다. 그렇다면 List<T>를 정렬하는 방법에는 세 가지가 있다고 하겠습니다. 각각에 대해서 알아봅시다.

A. Sort()의 경우

첫 번째 방법은 Sort 메서드에 아무런 매개 변수를 요구하지 않습니다. 그럼 나이 혹은 이름 순으로 정렬하라는 것과 같은 정보를 어디서 찾는 것일까요?

바로 List<T>의 원소가 이러한 정보를 제공하여야 합니다. 좀 더 정확히 이야기하면 List<T>의 원소가 비교 가능한 객체이어야 한다는 말입니다. (여러 개의 객체가 있을 때 먼저 이들을 서로 비교할 수 있어야 정렬이 가능한 법이니까요.) 우리는 List< GirlFriend>를 사용하니까 바로 GirlFriend 객체가 비교 가능한 객체가 되어야 합니다.

그렇다면 GirlFriend 객체를 어떻게 비교 가능한 객체로 만들 수 있을까요? 또 Sort 메서드는 어떻게 두 개의 GirlFriend 객체를 서로 비교할 수 있는 것일까요? 예컨데 GirlFriend 객체에 CompareWithOther 과 같은 비교 메서드가 있다고 해도, Sort 가 이 메서드의 존재를 어떻게 알고 호출을 할 수 있을까요?

짐작하셨겠지만, 바로 여기서 계약으로서의 인터페이스가 등장을 합니다. 즉, 매개변수가 없는 Sort 메서드(void Sort())는 List<T>의 원소인 객체와 계약을 맺었는데, 그 계약의 내용은 List<T>의 원소인 객체는 int CompareTo(T *other*)와 같은 메서드를 가지고 있다는 것입니다. 따라서 실제 List<T>의 원소의 객체가 무엇이든지 간에 Sort 메서드는 그 객체의 CompareTo 메서드를 호출하면 정렬을 수행할 수 있습니다.

계약의 다른 당사자인 List<T>의 원소인 객체는 int CompareTo(T *other*)를 구현하여야 합니다. 이러한 계약을 문법으로 표시한 것이 바로 인터페이스 입니다.

위 예를 들어 설명을 하자면 Sort 메서드와 GirlFriend(List<T>의 원소) 사이에는 GirlFriend 가 int CompareTo(GirlFriend *other*) 메서드를 구현하여야 한다는 계약이 성립되어 있는 것이고, 이 계약을 C# 문법으로 표현하자면, int CompareTo(T other) 메서드를 멤버로 가지는 인터페이스를 정의하고 GirlFriend 가 이 인터페이스를 구현하면 되겠습니다.

이 인터페이스는 이미 닷넷 프레임웍 라이브러리에 정의가 되어 있기 때문에 새로 만들 필요가 없습니다. 바로 IComparable(제네릭 버전은 IComparable<T>) 인터페이스인데, 멤버는 메서드 하나 밖에 없습니다.

```
int CompareTo(T other)
```

이 메서드는 현재 객체가 비교할 객체(other) 보다 작으면 음수, 같으면 0, 크면 양수를 반환합니다. 음수, 양수가 헷갈리신다면 T를 int 형이라 생각하시고, 현재 객체에서 매개 변수인 객체(other)를 뺀 값이 반환값이라고 생각을 하시면 쉽습니다.

이제 이 IComparable<T> 인터페이스를 구현하여 GirlFriend 객체를 비교 가능한 객체로 만들어 보겠습니다.

```
01 public class GirlFriend : IComparable<GirlFriend>
02 {
03     private readonly string _name;
04
05     public string Name
06     {
07         get { return _name; }
08     }
09
10     private readonly int _age;
11
12     public int Age
13     {
14         get { return _age; }
15     }
16
17     public GirlFriend(string name, int age)
18     {
19         _name = name;
20         _age = age;
21     }
22
23     public override string ToString()
24     {
25         return string.Format("이름 : {0}\t 나이 : {1}", _name, _age);
26     }
27
28     #region IComparable<GirlFriend> Members
29     public int CompareTo(GirlFriend other)
30     {
31         return Name.CompareTo(other.Name);
32     }
33     #endregion
```

```
34 }
```

코드 15

1 번 라인에서 GirlFriend 클래스가 IComparable<GirlFriend>를 구현한다고 하였습니다. 그리고 int CompareTo(GirlFriend other) 메서드를 구현하고 있습니다. int CompareTo(GirlFriend other)는 현재 객체의 Name 과 other 객체의 Name 을 비교한 결과를 반환합니다.

이제 GirlFriend 객체는 비교 가능한 객체가 되어 List<GirlFriend>는 정렬을 할 수 있게 되었습니다. 코드 14 을 다시 실행시킨 결과는 아래와 같습니다.

```
이름 : 경        나이 : 21
이름 : 옥        나이 : 20
이름 : 패        나이 : 19
계속하려면 아무 키나 누르십시오 . . .
```

의도한 대로 여자 친구들이 이름 순으로 정렬이 되었습니다.

B. Sort(Comparison<T> comparison)의 경우

위 코드 15 는 제대로 동작하지만, 이번에는 이름이 아니라 나이 순으로 정렬을 하는 경우를 생각해 봅시다. 앞의 방법 대로라면 코드 15 의 int CompareTo(GirlFriend other)가 이름이 아니라 나이를 기준으로 정렬하도록 수정하여야 합니다.

```
1 public int CompareTo(GirlFriend other)
2 {
3     return Age.CompareTo(other.Age);
4 }
```

코드 16

물론 제대로 동작은 하지만 GirlFriend 객체를 수정하여야 하는 한계가 있습니다. 만일 List<GirlFriend> 객체를 런타임의 입력에 따라 이름 순 혹은 나이 순으로 정렬을 해야 한다면 이러한 방법으로는 불가능합니다. (리플렉션이나 동적코드생성 기법을 사용하면 가능하지만 여기서는 논외로 하겠습니다.)

그래서 List<T>를 정렬하는 두 번째 방법이 등장하는데요. 이번에는 List<T>의 원소를 비교 가능한 객체라고 전제 하는 것이 아니라, Sort 메서드의 매개 변수로 아예 List<T>의 원소들을 비교하는 메서드를 전달하는 것입니다.

메서드의 매개 변수로 다른 메서드를 전달한다, 그렇습니다, 바로 대리자를 사용하는 것입니다. 물론 닷넷 프레임웍 라이브러리에는 이런 대리자도 이미 정의되어 있습니다.

```
public delegate int Comparison<T>(T x, T y)
```

Comparison 대리자의 의미는 x 가 y 보다 작으면 음수, 같으면 0, 크면 양수를 반환한다는 것입니다.

이 방법으로 정렬하는 예를 보도록 하겠습니다. GirlFriend 클래스는 더 이상 IComparable< GirlFriend >를 구현할 필요가 없습니다. 따라서 코드 13 과 동일하므로 생략합니다.

```
01 private static int CompareGirlFriendByName(GirlFriend x, GirlFriend y)
02 {
03     return x.Name.CompareTo(y.Name);
04 }
05
06 private static int CompareGirlFriendByAge(GirlFriend x, GirlFriend y)
07 {
08     return x.Age.CompareTo(y.Age);
09 }
10
11 private static void Main(string[] args)
12 {
13     List<GirlFriend> myGirls = new List<GirlFriend>();
14     myGirls.Add(new GirlFriend("패", 19));
15     myGirls.Add(new GirlFriend("경", 21));
16     myGirls.Add(new GirlFriend("옥", 20));
17
18     myGirls.Sort(CompareGirlFriendByName);
19     //  myGirls.Sort(CompareGirlFriendByAge);
20
21     foreach (GirlFriend girl in myGirls)
22         Console.WriteLine(girl);
23 }
```

코드 17

먼저 Comparison 대리자의 시그니처와 일치하는 두 개의 메서드를 정의합니다. 각각 이름과 나이 순으로 두 개의 GirlFriend 객체 간의 비교 결과를 반환합니다. 18 번 라인에서 Sort 메서드를 호출하는데 매개변수로 Comparison 대리자를 전달합니다. 아시겠지만 C# 2.0 부터는 아래 두 줄은 동일합니다.

```
myGirls.Sort(CompareGirlFriendByName);
myGirls.Sort(new Comparison<GirlFriend>(CompareGirlFriendByName));
```

18 번 라인을 주석으로 묶고 19 번 라인의 주석을 풀면 이번에는 이름이 아니라 나이 순으로 정렬을 합니다. 앞의 방법에 비하면 런타임에도 정렬 방법을 바꿀 수 있는 등 한결 유연해진 모습입니다.

C. Sort(IComparer<T> comparer)의 경우

의미상으로 대리자와 인터페이스는 물론 분명히 서로 다른 목적을 가지고 있지만, 일반적으로 인터페이스는 대리자 대용으로 사용할 수 있습니다. 둘다 '위임' 구조에 기반하고 있기 때문에 이것이 가능합니다. Sort 메서드의 세 번째 오버로드에서 이러한 예를 확인할 수 있습니다.

이번에는 Sort 가 IComparer<T> 인터페이스를 매개 변수로 받습니다. 이 인터페이스도 멤버 메서드를 하나만 가지고 있는 간단한 인터페이스입니다.

```
int Compare(T x, T y)
```

의미는 Comparison<T> 대리자와 동일합니다.

코드 17 을 대리자가 아니라 인터페이스를 사용하는 형태로 고치면 다음과 같습니다.

```
01 public class GirlFriendNameComparaer : IComparer<GirlFriend>
02 {
03     public int Compare(GirlFriend x, GirlFriend y)
04     {
05         return x.Name.CompareTo(y.Name);
06     }
07 }
08
09 public class GirlFriendAgeComparaer : IComparer<GirlFriend>
10 {
11     public int Compare(GirlFriend x, GirlFriend y)
```

```
12     {
13         return x.Age.CompareTo(y.Age);
14     }
15 }
16
17 internal class Program
18 {
19     private static void Main(string[] args)
20     {
21         List<GirlFriend> myGirls = new List<GirlFriend>();
22         myGirls.Add(new GirlFriend("패", 19));
23         myGirls.Add(new GirlFriend("경", 21));
24         myGirls.Add(new GirlFriend("옥", 20));
25
26         myGirls.Sort(new GirlFriendNameComparaer());
27         // myGirls.Sort(new GirlFriendAgeComparaer());
28
29         foreach (GirlFriend girl in myGirls)
30             Console.WriteLine(girl);
31     }
32}
```

코드 18

대리자 대신 IComparer<T> 인터페이스를 구현하는 두 개의 비교자 클래스를 만든 것 말고는 코드 17 과 매우 유사합니다.

2. BinarySearch 메서드

자료 구조나 알고리즘을 공부하신 분들은 익숙하시겠지만, 이진 검색이라는 것이 있습니다. O(logN)의 성능을 보장하는, 현재까지 알려진 검색 알고리즘 중에서는 가장 빠른 검색 방법 중 하나 입니다.

O(logN)에 대해 첨언을 하자면, 정확하게는 $O(\log_2 N)$ (← 2N 이 아니라 2 를 밑수로 가지는 logN 입니다.) 이라고 해야 할 것인데, 자료 구조의 성능을 표현하는 이른바 big O 표현식에서는 밑수가 큰 의미가 없기 때문에 관례적으로 O(logN)으로 표기를 합니다. 어쨌거나 이게 어떤 의미인가 하면, 예컨데 100 개의 원소가 있는 컬렉션이 있고, 이 중 한 원소를 검색하는 상황을 생각해 봅시다. 순차 검색의 경우라면 첫 원소 부터 차례대로 검색을 해 나가는 수 밖에 없습니다. 따라서 최악의 경우에는 100 번의 비교를 하여야 합니다. 반면에 이 컬렉션에 대해

이진 검색을 한다면, $\log_2 100 = 7.XXX$ 이므로 8 번만 비교를 하면 됩니다. 원소의 개수가 만일 1,000,000 개 라면요? 1,000,000 vs. 20 이라는 경이적인 성능 차이가 발생하게 됩니다.

List<T>에는 이런 이진 검색을 하는 BinarySearch 메서드가 있습니다. 따라서 컬렉션에서 특정 원소를 찾는 검색 작업이 매우 효율적으로 이루어질 수 있습니다.

BinarySearch 에 대해 본격적인 이야기를 시작하기 전에, 몇 가지 흥미로운 사항부터 짚어보도록 하겠습니다.

먼저 List<T>에서 BinarySearch 를 수행하기 위해서는 반드시 List<T> 객체가 **정렬된 상태**로 있어야 합니다. 이는 List<T>. BinarySearch 뿐만 아니라 모든 이진 검색의 전제 조건이기도 합니다. 정렬되지 않은 컬렉션에 대해 이진 검색을 수행한 결과는 엉터리 입니다. 그렇다면, 이진 검색을 수행하고자 하는 컬렉션에 원소가 삽입된다면 이 컬렉션을 다시 정렬된 상태로 만들기 위한 조작이 필요하다는 말이 됩니다. 그렇습니다. 그리고 이 문제는 자료 구조 과목에서 다루는 큰 이슈이기도 합니다. 그래서 일반적으로 자료의 삽입이 빈번하게 일어나고 동시에 이진 검색을 지원해야 하는 경우에는 List<T> 컬렉션은 좋은 선택이 아닙니다. 이러한 경우에는 Red-Black 트리나 SkipList 같은 컬렉션이 좋은 대안이 될 것 입니다.

또 한가지 흥미로운 점은 BinarySearch 의 반환값 입니다. BinarySearch 메소드의 반환값이 찾는 원소의 인덱스라는 건 직관적으로 알 수 있겠는데, 흥미로운 것은 컬렉션 내에 찾는 원소가 없는 경우 입니다. 순차 검색을 하는 List<T>.IndexOf() 메서드의 경우에는 찾는 원소가 컬렉션에 없으면 항상 -1 을 반환합니다. 여기서 -1 은 0 보다 작은 값이라는 의미 밖에 없습니다. 왜냐하면 반환값이 0 보다 크거나 같다면 찾는 원소가 컬렉션에 있다는 의미가 되니까, 0 보다 작은 값이라면 어떤 것이라도 상관이 없고, List<T>.IndexOf()을 설계한 개발자는 -1 을 선택한 것입니다.

반면에 BinarySearch 메서드의 경우에는 이 반환값에 부가적인 정보가 들어 있습니다. 코드를 보면서 이야기해볼까요?

```
01 private static void Main(string[] args)
```

```
02 {
03     List<string> list = new List<string>();
04     list.Add("a");
05     list.Add("c");
06     list.Add("e");
07
08     int returnValue = list.BinarySearch("b");
09
10     Console.WriteLine(returnValue);
11     Console.WriteLine(~ returnValue);
12 }
```

코드 19

결과는 다음과 같습니다.

```
-2
1
계속하려면 아무 키나 누르십시오 . . .
```

8 번 라인에서 list 에 없는 "b"를 이진 검색한 반환값을 returnValue 에 저장합니다. 10 번과 11 번 라인에서는 returnValue 와 returnValue 의 비트 보수 연산(~ 연산자)한 결과를 각각 출력합니다. 비트 보수 연산에 대해서는 부록 : 비트 보수 연산과 정수의 표현을 참고하시기 바랍니다. 그렇다면 위 결과로 나온 -2 와 1 의 의미는 무엇일까요?

일단 -2 의 의미는 0 보다 작기 때문에 list 컬렉션에 "b" 객체가 포함되어 있지 않다는 의미입니다. 그럼 1 의 의미는 무엇일까요? 1 은 만일 검색한 원소인 "b"를 list 에 삽입하고자 할 때 list 의 정렬 상태를 깨뜨리지 않고 "b"를 삽입할 수 있는 위치(인덱스)를 나타냅니다.

인덱스	0	1	2
원소	a	c	e

그림 2 원소 b 를 삽입하기 전의 list (정렬되어 있음)

인덱스	0	**1**	2	3
원소	a	**b**	c	e

그림 3 원소 b 를 삽입한 후의 list (정렬 상태가 유지됨)

그림 2 와 그림 3 은 각각 원소 b 를 삽입하기 전과 후의 list 입니다. 원소 b 를 삽입할 때 list 의 정렬상태를 깨뜨리지 않고 삽입할 수 있는 가장 낮은 인덱스는 1 이며, 이것은 바로 BinarySearch 의 반환값을 비트 보수 연산한 값입니다.

BinarySearch 의 반환값을 이용하여 List<T>에 원소를 삽입하는 예제는 다음과 같습니다.

```
01 private static void Main(string[] args)
02 {
03     List<string> list = new List<string>();
04     list.Add("a");
05     list.Add("c");
06     list.Add("e");
07
08     string newItem = "b";
09
10     int returnValue = list.BinarySearch(newItem);
11
12     list.Insert(~ returnValue, newItem);
13
14     foreach (string item in list)
15         Console.WriteLine(item);
16 }
```

코드 20

실행 결과는 다음과 같습니다.

```
a
b
c
e
계속하려면 아무 키나 누르십시오 . . .
```

결과를 보면 원소 b 가 삽입되고 나서도 list 는 여전히 정렬상태가 유지되고 있음을 알 수 있습니다.

결과적으로 BinarySearch 의 반환값은 두 가지의 정보를 동시에 표현하고 있습니다. 통상적인 메서드 설계 지침에 의하면 이는 권장되지 않는 사항입니다. 하나의 반환값은 하나의 정보 만을 표현하는 것이 정석입니다. 반환해야 할 값의 정보가 두 가지 이상이라면, 1) 반환값 대신 두 개 이상의 out 매개 변수를 사용하거나, 2) 반환할 정보를 필드로 가지고 있는 클래스 혹은 구조체를 반환하는 것이 일반적인 패턴입니다.

BinarySearch 의 반환값의 경우에도 1)검색하는 원소의 인덱스와 2)새 원소를 삽입할 위치라는 두 가지의 정보는, 비록 서로 논리적으로 배타적인 관계를 형성하고는 있지만, 분명히 서로 다른 별개의 정보로 간주되어야 하고, out 매개 변수를 받거나 클래스 혹은 구조체를 반환하는 형태로 디자인하는 것이 더 원칙에 충실한 디자인이 되지 않았을까 하는 생각을 해봅니다. 제가 보기에 BinarySearch 메서드는 원칙 보다는 실용성을 선택한 것 같습니다.

BinarySearch 에 대한 서론이 길었습니다. BinarySearch 는 세 개의 오버로드를 가지고 있습니다.

```
public int BinarySearch(T item)
public int BinarySearch(T item, IComparer<T> comparer)
public int BinarySearch(int index, int count, T item, IComparer<T> comparer)
```

세 번째 오버로드는 두 번째 오버로드와 동일한 것으로 간주해도 무방하므로 처음과 두 번째 오버로드에 대해서만 살펴보도록 하겠습니다.

메서드의 시그니처를 보고 이미 짐작하셨겠지만, BinarySearch 에도 Sort 와 동일한 이슈가 있습니다. 즉, BinarySearch 메서드를 사용하기 위해서는 1) List<T> 객체의 원소가 비교 가능한 객체이거나, 2) List<T>의 원소를 비교하는 일을 하는 비교자 객체를 BinarySearch 의 매개 변수로 전달하여야 합니다. Sort 와 다른 점은 매개 변수로 대리자를 전달하는 오버로드가 없다는 점 말고는 사용법이 거의 동일합니다.

3. Comparer<T>

이번 단원의 제목은 순서 비교자입니다. 그런데 왜 쌩뚱맞게 Sort 와 BinarySearch 메서드에 대해, 그것도 각각의 오버로드까지 거론하면서 길게 이야기를 하였을까요? 또 순서 비교자와 Sort, BinarySearch 메서드가 어떤 관계가 있는 것일까요? 답은 두 메서드의 오버로드들 중 첫번째 형태, 즉 매개 변수 없는 형태의 Sort 와 BinarySearch 가 순서 비교자를 사용한다는 것입니다.

(순서 비교자와 관련해서는 Sort 와 BinarySearch 를 구별할 필요가 없으므로 여기서는 Sort 에 대해서만 이야기를 할 것인데, 특별한 언급이 없으면 BinarySearch 도 동일합니다.)

Comparer<T>는 이름이 의미하는 것 처럼 비교자입니다. 정확하게 이야기하면 순서 비교자이므로 그 이름도, 다음 단원에서 이야기할 같음 비교자의 이름이 EqualityComparer 인 것과 대구를 맞춰, OrderComparer 정도가 되면 더 명확했을 것 같습니다. 대체로 완벽에 가까운 닷넷 프레임웍의 작명에 있어서 옥의 티 라고나 할까요. Comparer 가 설계될 당시에는 EqualityComparer(2.0 에서 추가)가 존재하지 않았기 때문에 빚어진 사소한 아쉬움이라고 하겠습니다.

Comparer 의 상속 관계를 먼저 보겠습니다.

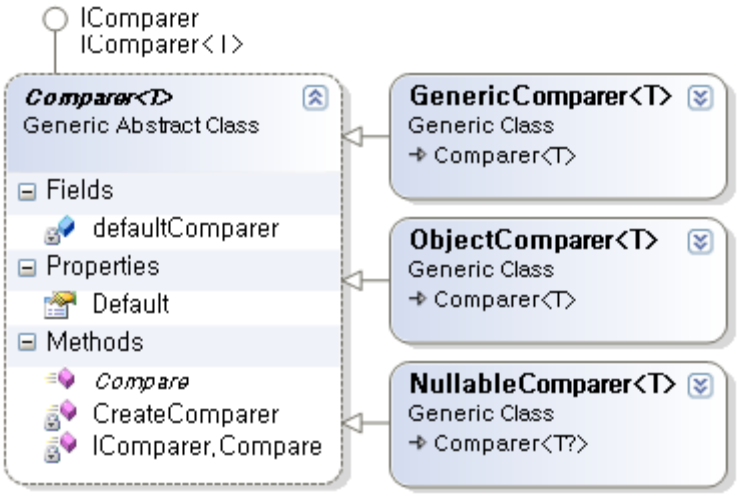

그림 4Comparer 의 상속 관계

Comparer 는 Comparer<T>를 구현합니다. 따라서 int Compare(T *x*, T *y*) 메서드를 구현하여야 합니다. 매개 변수가 없는 형태의 Sort()는 List<T>의 각 원소를 정렬할 때 바로 이 Compare 메서드를 호출하여 두 원소 사이의 순서를 결정합니다.

그런데 Comparer 는 추상 클래스입니다. 인스턴스를 만들 수 없다는 말이지요. 대신 Comparer 를 상속 받은 세 개의 클래스 중 하나의 인스턴스를 만듭니다. 그런데 이 세 개의 자식 클래스들의 접근 지정자가 모두 internal 이어서 어셈블리 외부에서는 접근할 수가 없습니다. 바꿔 말하면 이 자식 클래스들은 모두 닷넷 프레임웍 내부에서만 사용되는 클래스라는 이야기입니다. 그래서 문서화도 되어 있지 않아 MSDN 에도 나오지 않습니다. 오직 닷넷 프레임웍의 소스를

통해서만 그 존재를 확인할 수 있습니다. 위 다이어그램도 닷넷 프레임웍의 소스를 복사하여 생성한 것입니다.

다시 정리를 하자면 Sort 메서드는 Comparer 의 Compare 메서드를 호출하여야 합니다. 그런데 Compare 메서드는 Comparer 에서는 추상 메서드로 선언되어 있으며, 실제 구현은 Comparer 의 자식 클래스들에 들어 있습니다. 그런데 이 자식 클래스들은 internal 이라서 Sort 메서드가 접근하지 못합니다.

대신 Comparer 메서드는 Default 라는 속성을 제공하고 있습니다.

```
public static Comparer<T> Default { get; }
```

Default 속성은 T 의 형식에 따라 GenericComparer, NullableComparer, ObjectComparer 중 한 하나의 인스턴스를 생성하여 Comparer 형으로 반환합니다. 이를 좀 더 자세히 설명하면, T 가 IComparable<T>를 구현하면 GenericComparer 를, 그렇지 않고 T 가 IComparable<U>를 구현한 Nullable<U> 형이라면 NullableComparer 를, 그 외에는 ObjectComparer 의 인스턴스를 반환합니다.

IComparer<T>의 Compare 메서드는 이 자식 클래스들에서 각각 오버라이드 되어 있는데, GenericComparer 의 Compare 는 다음과 같이 재정의되어 있습니다.

```
01 internal class GenericComparer<T> : Comparer<T> where T : IComparable<T>
02 {
03     public override int Compare(T x, T y)
04     {
05         if (x != null)
06         {
07             if (y != null) return x.CompareTo(y);
08             return 1;
09         }
10         if (y != null) return -1;
11         return 0;
12     }
13     ...
14 }
```

1 번 라인에서 T 가 Comparable<T>를 구현한다고 제약 조건이 지정되어 있기 때문에 7 번 라인에서 보는 바와 같이 Comparable<T>의 CompareTo 메서드의 결과를 반환하고 있습니다.

반면에 ObjectComparer 의 Compare 는 다음과 같이 정의되어 있습니다.

```
1 internal class ObjectComparer<T> : Comparer<T>
2 {
3     public override int Compare(T x, T y)
4     {
5         return Comparer.Default.Compare(x, y);
6     }
7     ...
8 }
```

5 번 라인에서 제네릭이 아닌 Comparer 클래스가 나오는데요, 이의 Default 속성은 언제나 제네릭이 아닌 Comparer 입니다. 그래서 결국 5 번 라인은 제네릭이 아닌 Comparer 클래스의 Compare 메서드를 호출하는 것인데요. 이는 닷넷 프레임웍에 다음과 같이 정의되어 있습니다. (사실은 Shared Source Common Language Infrastructure 의 소스인데 아마도 실제 닷넷 프레임웍의 소스와 다르지는 않을 것입니다.)

```
01 public int Compare(Object a, Object b)
02 {
03     if (a == b) return 0;
04     if (a == null) return -1;
05     if (b == null) return 1;
06     if (m_compareInfo != null)
07     {
08         String sa = a as String;
09         String sb = b as String;
10         if (sa != null && sb != null)
11             return m_compareInfo.Compare(sa, sb);
12     }
13
14     IComparable ia = a as IComparable;
15     if (ia != null)
16         return ia.CompareTo(b);
17
18     throw new
ArgumentException(Environment.GetResourceString("Argument_ImplementIComparable"));
19 }
```

중간에 m_compareInfo 와 같은 필드가 등장하긴 하는데 무시하시고, 14 번부터 16 번 라인까지만 보시면 되겠습니다. 이 부분을 문장으로 표현하자면, 만일 매개 변수로 받은 a 가 IComparable 인터페이스를 구현하고 있다면 IComparable 의 CompareTo 메서드를 실행하고

그렇지 않다면 예외를 발생시킨다는 것입니다. Sort 메서드를 시작할 때 든 예제 코드(코드 14)에서 예외가 발생한 것을 기억하시나요? 그것이 바로 여기서 발생한 예외였던 것입니다.

위 단락의 마지막 한 문장을 설명하기 위해서 참으로 많은 이야기를 했습니다. 하지만 아직 한 가지가 더 남았습니다. 순서 비교자를 했으니 그와 쌍을 이루어 다니는 같음 비교자에 대한 이야기까지 해야 제대로 마무리가 될 것 같습니다. 지겹더라도 조금만 더 가봅시다.

V. 같음 비교자

1. Contains 메서드

이번에는 List<T>의 Contains 메서드에 대해서 이야기를 해보겠습니다. Contains 는 매개 변수도 하나 없이 워낙에 간단하기 때문에 별로 주목을 받지 못하는 경우가 많은데, 사실은 여기에 상당한 함정이 도사리고 있습니다.

아래는 Contains 메서드의 시그니처입니다.

```
public bool Contains (T item)
```

누구나 알다 시피, Contains 는 List<T>의 각 원소를 돌면서 item 과 **같은** 원소가 발견되면 true, 그렇지 않으면 false 를 반환합니다. 여기서 문제가 되는 부분은 바로 이 '같다'는 말이 됩니다.

예를 들어 봅시다. 아래와 같이 여자 친구의 인스턴스를 두 개 만듭니다.

```
GirlFriend girl1 = new GirlFriend("패", 19);
GirlFriend girl2 = new GirlFriend("패", 19);
```

girl1 과 girl2 는 이름과 나이, 즉 모든 필드가 동일합니다. 그렇다면 girl1 과 girl2 는 서로 같다고 할 수 있을까요? 물론 두 인스턴스는 같지 않습니다. 왜냐하면 GirlFriend 는 클래스이며 또 클래스는 참조형인데, 닷넷 프레임웍에서 참조형의 인스턴스는 그 필드의 값이 아니라 참조 변수의 값(객체가 생성된 메모리의 주소)이 같을 경우에 같은 객체로 정의를 하기 때문입니다. (참고로 값형의 경우에는 필드의 값이 모두 같으면 같은 객체로 간주합니다.)

하지만 경우에 따라서는 참조형의 경우에도 참조 변수의 값이 아니라 다른 기준에 의해서 두 객체가 같다고 정의해야 하는 경우가 있습니다. 코드를 보면서 이야기해보도록 하겠습니다. 다음 코드는 Contains 메서드를 사용하는 예입니다.

```
01 private static void Main(string[] args)
02 {
03     List<GirlFriend> myGirls = new List<GirlFriend>();
04     myGirls.Add(new GirlFriend("패", 19));
05     myGirls.Add(new GirlFriend("경", 21));
06     myGirls.Add(new GirlFriend("옥", 20));
07
08     GirlFriend targetGirl = new GirlFriend("패", 19);
09
10     bool found = myGirls.Contains(targetGirl);
11
12     Console.WriteLine(found);
13 }
```

코드 21

myGirls 컬렉션에서 이름이 패이고 나이가 19 살인 소녀를 찾고 있습니다. 이런 코드라면 아마도 true 가 출력되는 것이 자연스러울 것 같습니다. 하지만 결과는 그렇지 않습니다. 두 소녀의 같음을 이름과 나이로 판단하는 것이 아니라, GirlFriend 객체가 생성된 메모리 주소로 판단하기 때문에 그렇습니다.

이 코드를 수정하여 결과가 true 가 출력되게, 즉 소녀의 이름과 나이가 같으면 같은 소녀로 간주하도록 해봅시다.

이는 Sort 메서드의 경우와 유사할 것 같습니다. 즉 정렬을 하기 위해서는 세 가지 옵션이 있었는데, 1) List<T>의 원소를 비교 가능한 객체로 만들거나, 2) 두 원소를 비교하여 순서를 결정하는 대리자를 Sort 의 매개 변수로 전달하거나, 3) 두 원소를 비교하여 순서를 결정하는 비교자의 인스턴스를 전달하는 방법이 그것이었습니다. 하지만 Contains 의 경우는 다른 오버로드가 없기 때문에 2) 와 3)의 방법을 사용할 수 없고 1)번 방법 밖에 사용할 수가 없습니다. 즉 바꿔 말하면 GirlFriend 를 **같음 비교 가능한 객체**로 만드는 것입니다.

같은 비교 가능한 객체로 만들기 위해서는 IEquatable<T> 인터페이스를 구현하면 됩니다. IEquatable<T> 인터페이스 역시 단 한 개의 메서드를 가지고 있습니다.

```
bool Equals (T other)
```

Equals 는 현재 객체와 other 을 비교하여 같으면 true, 다르면 false 를 반환합니다.

이제 GirlFriend 클래스가 IEquatable<T> 인터페이스를 구현하도록 수정을 하여야 합니다.

```
01 public class GirlFriend : IEquatable<GirlFriend>
02 {
03     private readonly string _name;
04
05     public string Name
06     {
07         get { return _name; }
08     }
09
10     private readonly int _age;
11
12     public int Age
13     {
14         get { return _age; }
15     }
16
17     public GirlFriend(string name, int age)
18     {
19         _name = name;
20         _age = age;
21     }
22
23     public override string ToString()
24     {
25         return string.Format("이름 : {0}\t 나이 : {1}", _name, _age);
26     }
27
28     public bool Equals(GirlFriend other)
29     {
30         return Name.Equals(other.Name) && Age.Equals(other.Age);
31     }
32 }
```

30 번 라인에서 현재 GirlFriend 객체와 other 객체를 비교하는데, Name 과 Age 가 모두 같으면 같은 객체로 정의하고 있습니다. 이제 코드 21 을 실행시키면 true 가 반환됩니다.

2. EqualityComparer<T>

EqualityComparer는 앞 단원에서 이야기한 Comparer를 이해하였다면 쉽게 이해할 수 있습니다. 먼저 EqualityComparer의 상속 관계를 보도록 합시다.

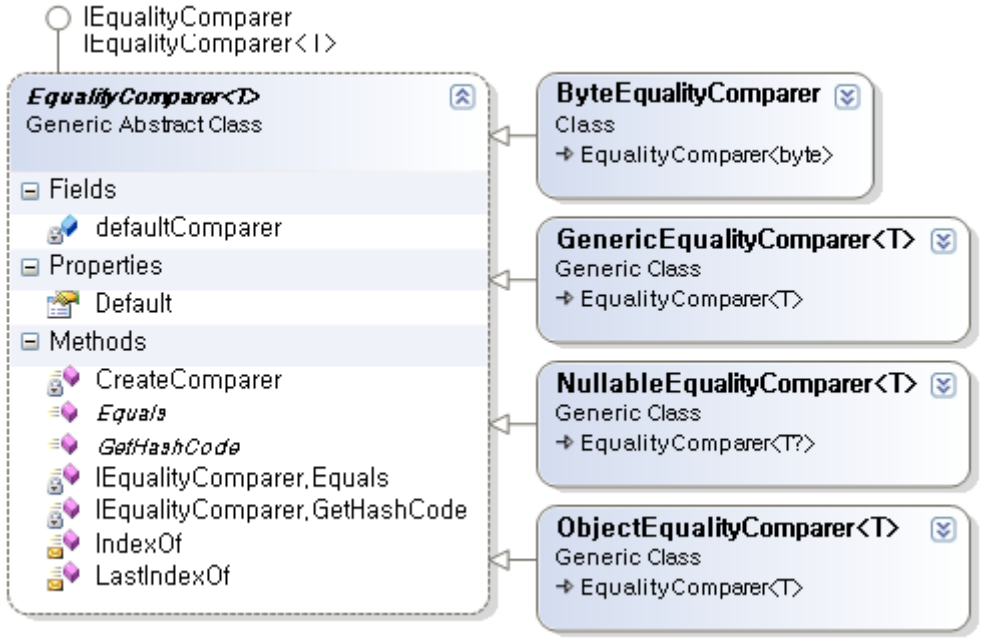

그림 5EqualityComparer의 상속 관계

Comparer와 동일한 구조임을 알 수 있습니다. 다만 ByteEqualityComparer라는 것이 추가되었는데, 이는 닷넷 프레임웍에 달려있는 주석에 의하면 byte 형 간의 비교를 할 때 성능을 위하여 특별히 고안된 같음 비교자라고 합니다. 소스를 보면 C의 런타임 함수인 memchr를 호출하는 unsafe 코드로 되어 있습니다. 그 외는 Comparer와 거의 동일합니다.

Contains 메서드가 매개 변수로 받은 객체와 List<T>의 각 원소가 같은 지를 검사할 때 IEqualityComparer의 bool Equals (T x, T y) 메서드를 호출합니다. 물론 이 Equals 메서드는 IEqualityComparer 인터페이스로부터 구현 상속 받은 것이며, EqualityComparer는 추상 클래스이기 때문에 Equals 메서드를 추상 메서드로 선언만 하고 실제 구현은 그 자식 메서드들에게 맡기고 있습니다.

이제 GenericEqualityComparer 의 경우에는 굳이 소스를 확인하지 않아도, T 가 IEquatable<T>를 구현한 형이기 때문에 IEquatable<T>의 Equals 메서드를 호출할 것이라는 사실은 짐작이 가능합니다.

하지만 ObjectEqualityComparer 의 Equals 메서드 재정의는 Comparer 와 약간 다릅니다.

```
01 internal class ObjectEqualityComparer<T> : EqualityComparer<T>
02 {
03     public override bool Equals(T x, T y)
04     {
05         if (x != null)
06         {
07             if (y != null) return x.Equals(y);
08             return false;
09         }
10         if (y != null) return false;
11         return true;
12     }
13     ...
14 }
```

제네릭이 아닌 EqualityComparer 와 같은 클래스는 없기 때문에 바로 object 클래스에서 정의된, 혹은 하위 클래스 중 어딘가에서 재정의된 Equals 메서드를 호출합니다. 따라서 기본 순서 비교자인 Comparer<T>.Default 는 T 가 IComparable 인터페이스를 구현하지 않으면(구체적으로 IComparable 의 멤버인 CompareTo 메서드를 구현하지 않으면) 예외를 발생시키는 것에 반해, 기본 같음 비교자인 EqualityComparer<T>.Default 는 T 가 Equals 메서드만 구현하면 괜찮습니다. Equals 는 최상위 클래스인 object 에서 이미 가상이나마 구현이 되어 있긴 때문에 EqualityComparer<T>.Default 는 특수한 상황이 아니라면 예외를 발생시키지 않습니다.

바로 위 문장에서 말하는 특수한 상황에는 어떠한 것이 있을까요? 한 가지 예를 들자면 T 가 object 의 Equals 메서드를 재정의 하는데, 예외를 발생시키도록 재정의 된 상황을 들 수 있겠습니다. (물론 재정신을 가진 프로그래머라면 이런 짓을 할 리가 없겠지요?) 그 외에도 많은 예를 들 수 있을 것입니다. 다른 예들을 생각해낼 수 있다면, 제 생각에는 아마도 이 글의 내용을 충분히 이해한 것이라고 생각하셔도 좋을 것 같습니다.

지금까지 많은 이야기를 하였습니다. 한 가지 주제만 놓고 직선을 달려온 것이 아니라, 중간 중간 주제와 직접적인 연관이 없는 샛길로도 왕왕 빠지다 보니 더 에둘러 온 것 같은 느낌도 듭니다. 하지만 이 시리즈의 큰 제목은 "C# 코딩 연습"이며, 동시에 그것이 제가 이 글을 쓰는 최종 목표이기도 합니다. 즐거운 연습이 되셨기를 바랍니다.

VI. 부록 : 비트 보수 연산과 정수의 표현

비트 보수 연산 자체는 피연산자의 각 비트를 반전시키는 간단한 연산이지만, 이는 컴퓨터에서 정수를 표현하는 방법과 밀접한 연관이 있습니다.

컴퓨터에서 정수를 표현하는 방법을 설명하기 위해 4 비트 짜리 부호 있는 정수형을 가정합시다. 첫번째 비트는 부호를 나타내고(0 은 양수, 1 은 음수) 나머지 3 비트는 숫자를 표현합니다. 따라서 표현 가능한 숫자의 개수는, 3 개의 비트를 사용할 수 있고 각 비트는 0 또는 1 의 두 가지 값을 가질 수 있으므로 2 의 3 승 = 8 개가 됩니다. 8 개의 숫자를 표현할 수 있으므로 양수의 경우 1 이 아니라 0 부터 시작해서 0 ~ 7 까지의 값을 가질 수 있습니다. 또한 부호 비트가 1 이 되어 음수가 되는 경우에도 마찬가지로 -7 ~ 0 까지의 값을 표시할 수 있습니다. 이 때 0 은 양수와 음수 양쪽에 모두 포함되어 있으므로, 음수에서는 표현할 수 있는 8 개 숫자의 범위를 1 씩 내립니다. 즉 -8 ~ -1 이 됩니다. 결과적으로 부호 있는 4 비트 정수형의 표현 가능한 범위는 -8 ~ 7 이 됩니다. ()

양수 7 의 경우는 비트로 표현하면 부호 비트는 0, 나머지 비트가 모두 켜져 있는 상태이므로 0111 이 될 것입니다. 그럼 음수 7 은 어떻게 표현할까요? 단순히 부호 비트만 1 로 바꾼 1111 이 될까요? 그렇다면 음수 8 을 표현할 수 없게 됩니다. 또한 0000 과 1000 이 각각 +0 과 -0 을 의미하는 것이라면 이는 중복이 됩니다. 따라서 양수와 음수의 표현에는 다른 방법을 사용합니다. 바로 비트 보수 연산을 한 후 1 을 더하는 것입니다. 예를 들어 양수 1 이 0001 이면 음수 1 은 비트 보수 연산을 한 1110 에다 1 을 더한(비트에 대한 연산이므로 정확하게는 | 비트 연산) 1111 이 된다는 것입니다. 양수 1 과 음수 1 을 더하면 0 이 되어야 하는데, 이를 비트로 표현해도 0001 | 1111 이니까 0000 즉 0 이 맞습니다.

아래는 부호 있는 4 비트 정수의 가능한 모든 값입니다.

비트	0000	0001	0010	0011	0100	0101	0110	0111	
양수	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	
음수		**-1**	**-2**	**-3**	**-4**	**-5**	**-6**	**-7**	**-8**
비트		1111	1110	1101	1100	1011	1010	1001	1000

표 7 부호 있는 4 비트 정수의 비트 값

이 표에는 일련의 규칙이 있습니다. 0 과 -8 을 제외한 다른 수들은, 자신의 비트를 반전한 후 1 을 더하면 부호가 변경됩니다. 예컨데 3 의 경우 0011 을 반전하면 1100, 여기에서 1 을 더하면 1101, 즉 -3 이 됩니다. 거꾸로 -3 의 경우를 볼까요? 1101 을 반전하면 0010, 1 을 더하면 0011 다시 3 이 됩니다.

3 이 0011 이면 -3 은 그냥 1011 로 표현할 일이지 왜 이렇게 복잡하게 만들어 놓았을까요? 단순히 중복되는 0 의 표현을 제거하여 숫자 하나(4 비트에서는 -8)를 더 표현하기 위해서요? 물론 이것도 맞는 말이지만 진짜 이유는 비트 연산을 통하여 덧셈과 뺄셈 연산을 처리하기 위해서입니다.

잠시 0 과 1, 즉 비트만 아는 기계의 입장에서 생각해 봅시다.

```
1 + 2
```

이것을 기계는 어떻게 이해할까요? 이 식을 우리의 부호 있는 4 비트 정수의 비트 표현으로 고치면 이렇습니다.

```
0001 + 0010
```

결과는 0011 즉 3 이 나옵니다.

이번에는 뺄셈을 볼까요?

```
1 - 2
```

물론 비트 표현으로 다시 쓰면,

```
0001 - 0010
```

이 되고 그 결과는 1111 이니까 -1 이 맞습니다. 하지만 위 식을 1 + (-2)로 보고 뺄셈이 아닌 덧셈을 해도 결과는 동일합니다.

```
0001 + 1110
```

이제 결론을 이야기할 때가 됐습니다. 위 표 7 과 같은 식으로 양수와 음수의 비트가 설정되어 있다면, 덧셈 연산과 뺄셈 연산을 동일하게 덧셈 연산으로만 처리할 수 있습니다. 또한 덧셈 연산은 기계의 연산 중에서 비트 연산 다음으로 빠른 연산입니다. 결국, 정수를 기계에서 표현하는 방법은 우리의 직관과는 달리 성능과 효율성을 위해 고도로 정교하게 고안되어 있는 것니다.

인용 자료

MSDN 라이브러리. http://msdn2.microsoft.com/ko-kr/library/default.aspx.