[목 차]

람다 식(Ramda Expression)

람다 식은 람다 대수의 형식을 C#에 구현한 문법이다. C#의 경우 람다 식은 다음과 같이 구별돼 사용된다.

- **코드로서의 람다 식(가장 일반적인 방법)**

  익명 메서드의 간편 표기 용도로 사용된다.

- **데이터로서의 람다 식**

  람다 식 자체가 데이터가 되어 구문 분석의 대상이 된다. 이 람다 식은 별도로 컴파일할 수 있으며, 그렇게 되면 메서드로도 실행할 수 있다.

1. 람다 식 선언 형식 과 예

매개변수목록 => 식

```
delegate int Calculate(int a, int b);

int add(int n1, int n2) {   return n1 + n2; }

static void Main(string[] args)
{
      Calculate calc1 = add;
      Console.WriteLine( calc1(10,20) );

      Calculate calc2 = (int a, int b) => a + b;   //형식 유추:람다 식 매개 변수를 컴파일러가 유추
      Console.WriteLine( calc2(10,20) );
}
```

1. 문 형식의 람다 식

```
매개변수목록 => {
        문장1;
        문장2;
        …
}
```

```
delegate void DoSomething();
static void Main(string[] args) {
    DoSomething DoIt = () => {
        Console.WriteLine("출력1");
        Console.WriteLine("출력2");
    };
    DoIt();
```

2. 코드 로서의 람다 식

익명메서드를 람다 식으로 변환1

```
using System;
using System.Threading;
class Program{
    static void ThreadFun(object obj)    {
        Console.WriteLine("이름이 있는 메서드");
    }
    static void Main(string[] args)    {
        Thread thread = new Thread(ThreadFun);
        thread.Start();

        Thread thread1 = new Thread(
                delegate (object obj)
                {
                    Console.WriteLine("이름이 없는 메서드");
                });
        thread1.Start();

        Thread thread2 = new Thread(
            (obj) =>
            {
                Console.WriteLine("람다식을 이용한 메서드");
            });
        thread2.Start();
    }
}
```

익명메서드를 람다 식으로 변환2

```
using System;
class Program
{
    delegate int? MyDivide(int a, int b);

    static void Main(string[] args)
    {
        MyDivide myFunc = (a, b) =>
        {
            if (b == 0)
            {
                return null;
            }
            return a / b;
        };

        Console.WriteLine("10 / 2 == " + myFunc(10, 2)); // 출력 결과: 10 / 2 == 5
        Console.WriteLine("10 / 0 == " + myFunc(10, 0)); // 출력 결과: 10 / 2 ==
    }
}
```

익명메서드를 람다 식으로 변환2(return 문 생략), return 및 중괄호 생략 가능!

```
using System;

class Program
{
    delegate int MyAdd(int a, int b);

    static void Main(string[] args)
    {
        MyAdd myFunc = (a, b) => a + b;
        Console.WriteLine("10 + 2 == " + myFunc(10, 2)); // 출력 결과: 10 + 2 == 12
    }
}
```

3. FUNC와 ACTION으로 더 간편하게 무명 함수 만들기

단 하나의 익명 메소드/무명 함수를 만들기 위해 매번 델리게이트를 따로 선언해줘야 하는 불편이 남아 있다.

.NET 프레임워크는 프로그래머들의 불편을 덜어주기 위해 2 종류의 델리게이트들을 미리 선언해 두었다.

Func 델리게이트

반환 값이 있는 익명 메소드/무명 함수를 위한 델리게이트

특정 조건으로 필터링하고 그 결과를 반환하는 Where 같은 메소드를 구현할 때 유용하다.

Action 델리게이트

반환 값이 없는 익명 메소드/무명 함수를 위한 델리게이트

네임스페이스: System
어셈블리: System.Core (System.Core.dll)

[Action 델리게이트]

```
delegate void Action ( );
delegate void Action <T> (T arg);
delegate void Action <T1, T2> (T1 arg1, T2 arg2);
delegate void Action <T1, T2, T3> (T1 arg1, T2 arg2, T3 arg3);
delegate void Action <T1 ,T2, T3, T4> (T1 arg1, T2 arg2, T3 arg3, T4 arg4);
...
delegate void Action<T1, T2, T3, T4, T5, T6, T7, T8, T9, T10, T11, T12, T13, T14, T15, T16>(T1 arg1, ..., T16 arg16)
```

[Func 델리게이트]

```
delegate TResult Func <TResult> ( );
delegate TResult Func <T, TResult> (T arg);
delegate TResult Func <T1, T2, TResult> (T1 arg1, T2 arg2);
delegate TResult Func <T1, T2, T3, TResult> (T1 arg1, T2 arg2, T3 arg3);
delegate TResult Func <T1, T2, T3, T4, TResult> (T1 arg1, T2 arg2, T3 arg3,T4 arg4);
...
delegate TResult Func<T1, T2, T3, T4, T5, T6, T7, T8, T9, T10, T11, T12, T13, T14, T15, T16, TResult>(T1, arg1, ..., T16 arg16)
```

Func 사용 예

```
static void Main(string[] args)
{
    // 입력 매개 변수는 없으며, 반환 타입int  10을 반환
    Func<int> func1 = () => 10;
    Console.WriteLine("func1() : {0}", func1());   //10

    // 입력 매개 변수는 int 형식 하나, 반환 형식도 int
    Func<int, int> func2 = (x) => x * 2;
    Console.WriteLine("func2(4) : {0}", func2(4));        //6

    // 입력 매개 변수는 double, double 형식 두개, 반환 형식도 double
    Func<double, double, double> func3 = (x, y) => x / y;
    Console.WriteLine("func3(22/7) : {0}", func3(22, 7));      //6
}
```

Action 사용 예

```
Action act1 = () => Console.WriteLine("Action()");
act1();
```

```
Action<double, double> act3 =
(x, y) =>
{
        double pi = x / y;
        Console.WriteLine("Action<T1, T2>({0}, {1}) : {2}", x, y, pi);
};
act3(22.0, 7.0);
```

사용 예제

```
using System;
using System.Collections.Generic;
class Program{
    static void Main(string[] args)
    {
        Action<string> logOut =
            (txt) =>
            {
                Console.WriteLine(DateTime.Now + ": " + txt);
            };
        logOut("This is my world!");

        Func<double> pi = () => 3.141592;
        Console.WriteLine(pi());
    }
}
```

Action

```
static void Main(string[] args)          {
            Action act1 = () => Console.WriteLine("Action()");
            act1();
            int result = 0;
            Action<int> act2 = (x) => result = x * x;
            act2(3);
            Console.WriteLine("result : {0}", result);

            Action<double, double> act3 =
            (x, y) =>   {
                double pi = x / y;
                Console.WriteLine("Action<T1, T2>({0}, {1}) : {2}", x, y, pi);
            };
            act3(22.0, 7.0);
 }
```

예를 하나 살펴 보자. 숫자로 구성된 배열이 존재하고, 다양한 조건으로 배열에서 값을 가져오는 메서드

를 정의하고 싶다고 가정하자. 여기서는 "다양한 조건으로 검색이 가능하도록"이 중요한 포인트이다. 검색을 다양하게 하려면 호출되는 시점에 필터링 조건을 주면 된다. 해당 값이 조건이 맞는지 확인해서 bool 형식을 반환하는 코드가 필요하므로 "Func"를 사용하여 다음과 같이 메소드를 정의 하였다.

```
private static List<int> FilterOfInts(int[] source, Func<int, bool> filter)
        {
            List<int> result = new List<int>();
            foreach (int i in source) {
                if (filter(i)) { result.Add(i); }
            }
            return result;
        }
```

[숫자 배열에서 여러 조건으로 필터링 할 수 있는 메서드 정의]

이제 정의된 메서드를 통해, 명시적으로 정의된 메서드 혹은 익명메서드, 람다식 모두를 사용하여 원하는 결과를 볼 수 있다. 아래 예제에서는 FilterOfInts를 모두 호출하지만 람다식으로 다양한 조건을 파라미터로 넘기고 있다. 각 조건에 맞는 메소드를 모두 구현하는 것이 아니라 단일 메소드를 호출 하므로 보다 유연한 모듈 개발이 가능한 것을 확인 할 수 있다.

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

namespace Sample
{
    class Program
    {
        private static List<int> FilterOfInts(int[] source, Func<int, bool> filter)
        {
            List<int> result = new List<int>();
            foreach (int i in source)
            {
                if (filter(i))
                {
                    result.Add(i);
                }
```

```
        }

        return result;
    }

    static void Main(string[] args)
    {
        int[] source = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

        //===홀수 Filter===
        List<int> oddNumbers = FilterOfInts(source, i => ((i & 1) == 1));

        Console.WriteLine("=== 홀수 Filter ===");
        foreach (var item in oddNumbers)
        {
            Console.WriteLine(item);
        }

        //===짝수 Filter===
        List<int> evenNumbers = FilterOfInts(source, i => ((i & 1) == 0));

        Console.WriteLine("=== 짝수 Filter ===");
        foreach (var item in evenNumbers)
        {
            Console.WriteLine(item);
        }

        //===소수 Filter===
        List<int> primeNumbers = FilterOfInts(source,
        checkNumber =>
        {
            BitArray numbers = new BitArray(checkNumber + 1, true);
            for (int i = 2; i < checkNumber + 1; i++)
            {
                if (numbers[i])
                {
                    for (int j = i * 2; j < checkNumber + 1; j += i) { numbers[j] = false; }
                    if (numbers[i]) { if (checkNumber == i) { return true; } }
```

```
                    }
                }
                return false;
            });

            Console.WriteLine("=== 소수 Filter ===");
            foreach (var item in primeNumbers)
            {
                Console.WriteLine(item);
            }
        }
    }
}
```

델리게이션 활용 & Func 활용 비교

```
using System;
namespace Sample{
    class Program    {
        delegate string Concatenate(string[] args);
        static void exam1()          {
            string[] strs = { "aaa", "bbb", "ccc", "ddd" };
            Concatenate concat =
                (arr) =>
                {
                    string result = "";
                    foreach (string s in arr)
                        result += s;
                    return result;
                };
            Console.WriteLine(concat(strs));
        }
        static void exam2()          {
            string[] strs = { "aaa", "bbb", "ccc", "ddd" };
            Func<string[], string> concat =
                (arr) =>
                {
                    string result = "";
                    foreach (string s in arr)
                        result += s;

                    return result;
                };

            Console.WriteLine(concat(strs));
        }
        static void Main(string[] args)          {
            exam2();
        }
    }
}
```

4. 컬렉션과 람다 식

확장메서드, 람다 식, Action, Func 은 기존 컬렉션의 기능을 더욱 풍부하게 개선했다.

- List<T> 에 정의된 ForEach : public void ForEach<T> action);
- Array에 정의된 ForEach : public static void ForEach<T>(T[]array, Action<T> action);

⇨ Foreach 메서드는 Action<T> 델리게이트를 인자로 받아 컬렉션의 모든 요소를 열람하면서 하나씩 Action<T>의 인자로 요소 값 전달.

```
using System;
using System.Collections.Generic;
class Program{

    static void Main(string[] args)
    {
        List<int> list = new List<int> { 3, 1, 4, 5, 2 };

        foreach (var item in list)
        {
            Console.WriteLine(item + " * 2 == " + (item * 2));
        }

        Console.WriteLine();

        list.ForEach((elem) => { Console.WriteLine(elem + " * 2 == " + (elem * 2)); });

        Console.WriteLine();

        // 또는 Array.ForEach로 나타낼 수도 있고
        Array.ForEach(list.ToArray(), (elem) => { Console.WriteLine(elem + " * 2 == " + (elem * 2)); });

        Console.WriteLine();

        // 또는 람다 식이 아닌 익명 메서드로도 나타낼 수 있음
        list.ForEach(delegate (int elem) { Console.WriteLine(elem + " * 2 == " + (elem * 2)); });
    }
}
```

예제1) 특정 조건을 만족하는 요소만 반환

```
using System;
using System.Collections.Generic;
class Program{
    delegate int MyAdd(int a, int b);

    static void Main(string[] args)
    {
        List<int> list = new List<int> { 3, 1, 4, 5, 2 };

        // 방법 1
        {
            List<int> evenList = new List<int>();

            foreach (var item in list)
            {
                if (item % 2 == 0) // 짝수인지 판정해서 evenList에 추가한다.
                {
                    evenList.Add(item);
                }
            }
            foreach (var item in evenList)
            {
                Console.Write(item + ","); // 출력 결과: 4, 2,
            }
        }
        Console.WriteLine();

        // 방법 2
        {
            List<int> evenList = list.FindAll((elem) => elem % 2 == 0);
            evenList.ForEach((elem) => { Console.Write(elem + ","); }); // 출력 결과: 4, 2,
        }
    }
}
```

* public List<T> FindAll(Predicate<T> match);

Predicate 델리게이트는 한 개의 인자를 받고 bool형을 반환하는데, 결국 Func<T, bool> 과 동일하다.

예제2) 특정 조건을 만족하는 요소만 반환

```
using System;
using System.Collections.Generic;
using System.Linq;
class Program{
    static void Main(string[] args)
    {
        List<int> list = new List<int> { 3, 1, 4, 5, 2 };

        // 방법 1 : Foreach문 사용
        {
            int count = 0;

            foreach (var item in list)
            {
                if (item > 3)
                {
                    count++;
                }
            }

            Console.WriteLine("3보다 큰 수는 " + count + "개 있음.");
        }

        // 방법 2 : Count 확장 메서드 사용
        {
            int count = list.Count((elem) => elem > 3);
            Console.WriteLine("3보다 큰 수는 " + count + "개 있음.");
        }
    }
}
```

* 기존의 컬렉션 크기만 단순하게 반환하는 Count 속성은 Enumerable 타입의 확장 메서드를 통해 특정 조건을 만족하는 개수를 반환할 수 있게 됐다.

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

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

예제3) 특정 조건을 만족하는 요소만 반환

```
using System;
using System.Collections.Generic;
using System.Linq;
class Program{
    static void Main(string[] args)     {
        List<int> list = new List<int> { 3, 1, 4, 5, 2 };
        {
            IEnumerable<int> enumList = list.Where((elem) => elem % 2 == 0);
            Array.ForEach(enumList.ToArray(), (elem) => { Console.WriteLine(elem); });
        }

        Console.WriteLine();
        {
            IEnumerable<int> enumList = list.WhereFunc();    //where 유사 구현 예
            Array.ForEach(enumList.ToArray(), (elem) => { Console.WriteLine(elem); });
        }
    }
}
static class ExtensionSample {
    public static IEnumerable<int> WhereFunc(this IEnumerable<int> inst) //지연된 평가
    {
        foreach (var item in inst)          {
            if (item % 2 == 0)
                yield return item;    //단계적으로 실행.
            }
    }
}
```

* Enumerable 타입에 추가된 Where 확장 메서드는 Func<T, bool> 타입을 인자로 받아, 열거형 형태의 반환을 한다.

Where의 경우 메서드가 실행됐을 때는 어떤 코드도 실행되지 않은 상태이고, 이후 열거자를 통해 요소를 순회했을 때에 비로소 람다 식이 하나씩 실행된다.(지연된 평가(lazy evaluation)이라 함)

IEnumerable 반환타입은 모두 지연된 평가를 함

* FindAll(함수 종료시 결과 반환) 메서드와 결과는 동일하지만 과정이 다름(반환타입이 다르기 때문)

실행이 완료되는 순간 람다 식의 컬렉션의 모든 요소를 대상으로 실행되어 조건을 만족하는 결과반환

예제4) 컬렉션의 개별 요소를 다른 타입으로 변환

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

class Person{
      public int Age;
      public string Name;
}

class Program{
      static void Main(string[] args)      {
            List<int> list = new List<int> { 3, 1, 4, 5, 2 };

            {
                  IEnumerable<double> doubleList = list.Select((elem) => (double)elem);
                  Array.ForEach(doubleList.ToArray(), (elem) => { Console.WriteLine(elem); });
            }
            Console.WriteLine();

            //객체 반환
            IEnumerable<Person> personList = list.Select(
                (elem) => new Person { Age = elem, Name = Guid.NewGuid().ToString() });
            Array.ForEach(personList.ToArray(), (elem) => { Console.WriteLine(elem.Name); });

            Console.WriteLine();

           //익명 객체 반환
            var itemList = list.Select((elem) => new { TypeNo = elem, CreatedDate = DateTime.Now.Ticks });
            Array.ForEach(itemList.ToArray(), (elem) => { Console.WriteLine(elem.TypeNo); });
      }
}
```

* Select 역시 Where과 마찬가지로 IEnumerable<T> 타입을 반환하므로 지연 평가에 해당한다.

FindAll의 지연평가 버전이 Where 메서드인 것처럼 ConvertAll의 지연평가 버전이 Select이다.

WhereEnumerable

```
class NameCard
    {
        public string Name { get; set; }
        public string Phone { get; set; }
    }

    class MainApp
    {
        static void Main(string[] args)
        {
            List<NameCard> list = new List<NameCard>();

            list.Add(new NameCard() { Name = "박상현", Phone = "010-5162-1234" });
            list.Add(new NameCard() { Name = "이지은", Phone = "010-9712-3361" });
            list.Add(new NameCard() { Name = "정가은", Phone = "010-2133-8871" });
            list.Add(new NameCard() { Name = "김승수", Phone = "010-5512-2003" });

            IEnumerable<NameCard> result =
                list.Where<NameCard>(namecard => namecard.Name.EndsWith("은"));

            foreach (NameCard namecard in result)
                Console.WriteLine("{0} : {1}", namecard.Name, namecard.Phone);
        }
    }
```

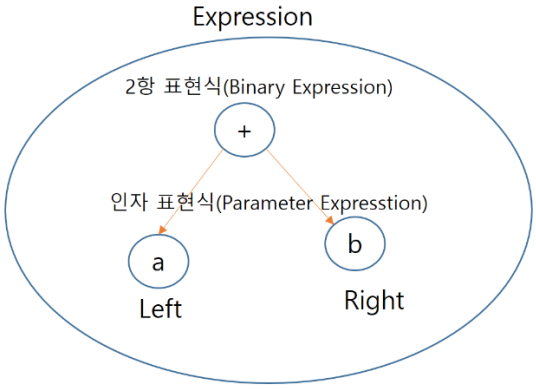
5. 데이터로서의 람다 식

람다 식을 실행코드가 아닌 그 자체로 "식을 표현한 데이터"로도 사용할 수 있다.

이처럼 데이터 구조로 표현된 것을 "식 트리(expression tree") 라고 한다. 식 트리로 담긴 람다 식은 익명 메서드의 대체물이 아니기 때문에 델리게이트 타입으로 전달되는 것이 아니라 식에 대한 구문 분석을 할 수 있는 System.Linq.Expressions.Expression 타입의 인스턴스가 된다.

즉, 람다 식이 코드가 아니라 Expression 객체의 인스턴스 데이터의 역할을 하는 것이다.

Expression<Func<int, int, int>> exp = (a, b) => a + b;




```
using System;
using System.Linq.Expressions;
class Program{
    static void Main(string[] args)     {
        Expression<Func<int, int, int>> exp = (a, b) => a + b;

        // 람다 식 본체의 루트는 2항 연산자인 + 기호
        //BinaryExpression : 이항연산자가 있는 식.
        //exp.Body : 람다식의 본문을 가져옴.
        BinaryExpression opPlus = exp.Body as BinaryExpression;
        Console.WriteLine(opPlus.NodeType); // 출력 결과: Add

        // 2항 연산자의 좌측 연산자의 표현식
        //ParameterExpression : 명명된 매개변수 식
        ParameterExpression left = opPlus.Left as ParameterExpression;
        Console.WriteLine(left.NodeType + ": " + left.Name); // 출력 결과: Parameter: a

        // 2항 연산자의 우측 연산자의 표현식
        ParameterExpression right = opPlus.Right as ParameterExpression;
```

```
        Console.WriteLine(right.NodeType + ": " + right.Name); // 출력 결과: Parameter: b
        //람다식으로 대리자 생성
        Func<int, int, int> func = exp.Compile();
        Console.WriteLine(func(10, 2)); // 출력 결과: 12
    }
}
```

Expression<T> 객체를 람다 식으로 초기화하지 않고 직접 코드와 관련된 Expression 객체로 구성할 수 있다.

```
using System;
using System.Linq.Expressions;
class Program{
    static void Main(string[] args)     {
        ParameterExpression leftExp = Expression.Parameter(typeof(int), "a");
        ParameterExpression rightExp = Expression.Parameter(typeof(int), "b");

        BinaryExpression addExp = Expression.Add(leftExp, rightExp);

        //--정보 출력
        Console.WriteLine(addExp.NodeType); // 출력 결과: Add

        ParameterExpression left = addExp.Left as ParameterExpression;
        Console.WriteLine(left.NodeType + ": " + left.Name); // 출력 결과: Parameter: a

        ParameterExpression right = addExp.Right as ParameterExpression;
        Console.WriteLine(right.NodeType + ": " + right.Name); // 출력 결과: Parameter: b

        //람다식 획득
        Expression<Func<int, int, int>> addLambda =
            Expression<Func<int, int, int>>.Lambda<Func<int, int, int>>
                (addExp, new ParameterExpression[] { leftExp, rightExp });
        Console.WriteLine(addLambda.ToString()); // 출력 결과: (a, b) => (a + b)
        Func<int, int, int> addFunc = addLambda.Compile();
        Console.WriteLine(addFunc(10, 2)); // 출력 결과: 12
    }
}
```