객체지향 프로그래밍

C# - Class

KYUNGSUNG UNIVERSITY SINCE 1955

인덱서

- 인덱서(indexer)
  - 배열 연산자인 '[]'를 통해서 객체를 다룰 수 있도록 함
  - 지정어 this를 사용하고, '[]'안에 인덱스로 사용되는 매개 변수 선언.
  - 겟-접근자 혹은 셋-접근자만 정의할 수 있음.
- 인덱서의 수정자
  - static만 사용할 수 없으며, 의미는 메소드와 모두 같음.
  - 접근 수정자(4개), new, virtual, override, abstract, sealed, extern – 총10개
- 인덱서의 정의 형태

```
[indexer-modifiers] returnType this[parameterList] {
        set {
                // indexer body
        }
        get {
                // indexer body
        }
}
```

```
using System;

namespace IndexerApp
{
    class Color
    {
        private string[] color = new string[5];
        public string this[int index]
        {
            get { return color[index]; }
            set { color[index] = value; }
        }
    }
    class Program
    {
        public static void Main()
        {
            Color c = new Color();
            c[0] = "WHITE";
            c[1] = "RED";
            c[2] = "YELLOW";
            c[3] = "BLUE";
            c[4] = "BLACK";
            for (int i = 0; i < 5; i++)
                Console.WriteLine("Color is " + c[i]);
        }
    }
}
```

```
using System;
namespace StringIndexerApp{
    class StringIndexer    {
        public char this[string str, int index]
        {
            get { return str[index]; }
        }
        public string this[string str, int index, int len]
        {
            get { return str.Substring(index, len); }
        }
    }
    class Program    {
        public static void Main()        {
            string str = "Hello";
            StringIndexer s = new StringIndexer();
            for (int i = 0; i < str.Length; i++)
                Console.WriteLine("{0}[{1}] = {2}", str, i, s[str, i]);
            Console.WriteLine("Substring of {0} is {1}", str, s[str, 2, 3]);
        }
    }
}
```

```
using System;

namespace A112_Indexer
{
  class MyCollection<T>
  {
    private T[] array = new T[100];

    public T this[int i]
    {
      get { return array[i]; }
      set { array[i] = value; }
    }
  }

  class Program
  {
    static void Main(string[] args)
    {
      var myString = new MyCollection<string>();
      myString[0] = "Hello, World!";
      myString[1] = "Hello, C#";
      myString[2] = "Hello, Indexer!";
      for (int i = 0; i < 3; i++)
        Console.WriteLine(myString[i]);
    }
  }
}
```

인덱서(Indexer)는 인스턴스 내의 데이터에 접근하는 방법이다. 용도는 속성(Property)와 똑같은데 다른 점은 배열과 같이 인덱스를 사용할 수 있다는 점이다. 인덱서는 클래스나 구조체의 인스턴스를 배열처럼 인덱싱할 수 있게 한다.
인덱서는 this[] 를 사용하며 겟터와 셋터를 만든다.

겟터에서 어떤 값을 가져오고, 어떤 값을 세팅하는지 정해주면 된다.

연산자의 중복

- 연산자 중복의 의미
  - 시스템에서 제공한 연산자를 재정의 하는 것
  - 클래스만을 위한 연산자로서 자료 추상화가 가능
  - 문법적인 규칙은 변경 불가(연산 순위나 결합 법칙 등)

- 연산자 중복이 가능한 연산자

종 류	연 산 자
단 항	+, -, !, ~, ++, --, true, false
이 항	+, -, *, /, %, &, |, ^, <<, >>, ==, !=, <, >, <=, >=
형 변환	변환하려는 자료형 이름

연산자의 중복

- 연산자 중복 방법
  - 수정자는 반드시 public static.
  - 반환형은 연산자가 계산된 결과의 자료형.
  - 지정어 operator 사용, 연산기호로는 특수 문자 사용.

- 연산자 중복 정의 형태

```
public static [extern] returnType operator op (parameter1 [, parameter2]) {
        // ... operator overloading body ...
}
```

연산자의 중복

연산자 중복 정의 규칙

연 산 자	매개변수 형과 반환형 규칙
단항 +, -, !, ~	매개변수의 형은 자신의 클래스, 복귀형은 모든 자료형이 가능함.
++ / --	매개변수의 형은 자신의 클래스, 복귀형은 자신의 클래스이거나 파생 클래스이어야 함.
true / false	매개변수의 형은 자신의 클래스, 복귀형은 bool 형 이어야 함.
shift	첫 번째 매개변수의 형은 클래스, 두 번째 매개변수의 형은 int 형, 복귀형은 모든 자료형이 가능함.
이 항	shift 연산자를 제외한 이항 연산자인 경우, 두 개의 매개변수 중 하나는 자신의 클래스이며, 복귀형은 모든 자료형이 가능함.

연산자의 중복

- 대칭적 방식으로 정의
  - true와 false, ==과 !=, <과 >, <=과 >=
- 형 변환 연산자(type-conversion operator)
  - 클래스 객체나 구조체를 다른 클래스나 구조체 또는 C# 기본 자료형으로 변환
  - 사용자 정의 형 변환(user-defined type conversion)
- 형 변환 연산자 문법 구조

```
public static [extern] explicit operator type-name(parameter1)
        또는
public static [extern] implicit operator type-name(parameter1)
```

연산자의 중복

```
using System;
   class Complex {
       private double realPart;                    // 실수부
       private double imagePart;                   // 허수부
       public Complex(double rVal, double iVal) {
           realPart = rVal;
           imagePart = iVal;
       }
       public static Complex operator+(Complex x1, Complex x2) {
           Complex x = new Complex(0, 0);
           x.realPart = x1.realPart + x2.realPart;
           x.imagePart = x1.imagePart + x2.imagePart;
           return x;
       }
       override public string ToString() {
           return "("+ realPart + "," + imagePart + "i)";
       }
   }
```

```
class OperatorOverloadingApp {
        public static void Main() {
            Complex c, c1, c2;
            c1 = new Complex(1, 2);
            c2 = new Complex(3, 4);
            c = c1 + c2;
            Console.WriteLine(c1 + " + " + c2 + " = " + c);
        }
    }
```

델리게이트

- 델리게이트(delegate)는 메소드 참조 기법
  - 객체지향적 특징이 반영된 메소드 포인터
- 이벤트와 스레드를 처리하기 위한 방법론
- 특징
  - 정적메소드 및 인스턴트 메소드 참조 가능 – 객체지향적
  - 델리게이트의 형태와 참조하고자하는 메소드의 형태는 항상 일치 – 타입안정적
  - 델리게이트 객체를 통하여 메소드를 호출 – 메소드참조
- VS. 함수포인터(C/C++)
  - 메소드 참조 기법면에서 유사
  - 객체지향적이며 타입 안정적

델리게이트의 정의

- 정의 형태

```
[modifiers] delegate returnType DelegateName(parameterList);
```

- 수정자
  - 접근 수정자
    - public, protected, internal, private
  - new
  - 클래스 밖에서는 public과 internal 만 가능

- 델리게이트 정의 시 주의점
  - 델리게이트 할 메소드의 메소드 반환형 및 매개변수의 개수, 반환형을 일치시켜야 함

델리게이트의 정의

- 델리게이트 정의 예

```
delegate void SampleDelegate(int param);    // 델리게이트 정의
    class DelegateClass {
    public void DelegateMethod(int param) { // 델리게이트할 메소드
        // ...
    }
}
```

델리게이트 객체 생성

- 델리게이트를 사용하기 위해서는 델리게이트 객체를 생성하고 대상 메소드를 연결해야 함
  - 해당 델리게이트의 매개변수로 메소드의 이름을 명시
  - 델리게이트 객체에 연결할 수 있는 메소드는 형태가 동일하면 인스턴스 메소드뿐만 아니라 정적 메소드도 가능
- 델리게이트 생성(인스턴스 메소드)
  - 델리게이트할 메소드가 포함된 클래스의 객체를 먼저 생성
  - 정의된 델리게이트 형식으로 델리게이트 객체를 생성
  - 생성된 델리게이트를 통하여 연결된 메소드의 호출
- 델리게이트 객체 생성 예

```
DelegateClass obj = new DelegateClass();
SampleDelegate sd = new SampleDelegate(obj.DelegateMethod);
```

델리게이트 객체 호출

- 델리게이트 객체의 호출은 일반 메소드의 호출과 동일
- 델리게이트를 통하여 호출할 메소드가 매개변수를 갖는다면 델리게이트를 호출하면서 ( )안에 매개변수를 기술

```
using System;

namespace DelegateCallApp
{
    delegate void DelegateOne();       // delegate with no params
    delegate void DelegateTwo(int i); // delegate with 1 param
    class DelegateClass
    {
        public void MethodA()
        {
            Console.WriteLine("In the DelegateClass.MethodA ...");
        }
        public void MethodB(int i)
        {
            Console.WriteLine("DelegateClass.MethodB, i = " + i);
        }
    }
    class Program
    {
        public static void Main()
        {
            DelegateClass obj = new DelegateClass();
            DelegateOne d1 = new DelegateOne(obj.MethodA);
            DelegateTwo d2 = new DelegateTwo(obj.MethodB);
            d1();             // invoke MethodA() in DelegateClass
            d2(10);           // invoke MethodB(10) in DelegateClass
        }
    }
}
```

```
using System;

namespace Delegate
{
    delegate int MyDelegate( int a, int b);

    class Calculator
    {
        public int Plus(int a, int b)
        {
            return a + b;
        }

        public static int Minus(int a, int b)
        {
            return a - b;
        }
    }
```

```
    class MainApp
    {
        static void Main(string[] args)
        {
            Calculator Calc = new Calculator();
            MyDelegate Callback;

            Callback = new MyDelegate( Calc.Plus );
            Console.WriteLine(Callback(3, 4));

            Callback = new MyDelegate(Calculator.Minus);
            Console.WriteLine(Callback(7, 5));
        }
    }
}
```

Delegate의 기본
배열에서 홀수와 짝수 찾기

```
using System;

namespace A113_DelegateExample
{
  class Program
  {
    delegate bool MemberTest(int a);

    static void Main(string[] args)
    {
      int[] arr = new int[] { 3, 5, 4, 2, 6, 4, 6, 8, 54, 23, 4, 6, 4 };

      Console.WriteLine(Count(4));
      Console.WriteLine(Count(arr, 4));
      Console.WriteLine("짝수의 갯수: " + EvenCount(arr));
      Console.WriteLine("홀수의 갯수: " + OddCount(arr));

      Console.WriteLine("짝수의 갯수: " + Count(arr, IsEven));
      Console.WriteLine("홀수의 갯수: " + Count(arr, IsOdd));
    }
```

```
    static int EvenCount(int[] a)
    {
      int cnt = 0;
      foreach (var n in a)
      {
        if (n % 2 == 0)
          cnt++;
      }
      return cnt;
    }

    static int OddCount(int[] a)
    {
      int cnt = 0;
      foreach (var n in a)
      {
        if (n % 2 == 1)
          cnt++;
      }
      return cnt;
    }
```

```
static int Count(int[] a, MemberTest testMethod)
{
  int cnt = 0;
  foreach (var n in a)
  {
    if (testMethod(n) == true)
      cnt++;
  }
  return cnt;
}

static public bool IsOdd(int n) { return n % 2 != 0; }
static public bool IsEven(int n) { return n % 2 == 0; }

private static int Count(int v)
{
  var nums = new[] { 3, 5, 4, 2, 6, 4, 6, 8, 54, 23, 4, 6, 4 };
  int cnt = 0;
  foreach (var n in nums)
  {
    if (n == v)
      cnt++;
  }
  return cnt;
}
```

```
    private static int Count(int[] nums, int v)
    {
      int cnt = 0;
      foreach (var n in nums)
      {
        if (n == v)
          cnt++;
      }
      return cnt;
    }
  }
}
```

```
using System;

namespace A114_AnonymousDelegate{
  class Program  {
    delegate bool MemberTest(int x);

    static void Main(string[] args)
    {
      var arr = new[] { 3, 34, 6, 34, 7, 8, 24, 3, 675, 8, 23 };

      int n = Count(arr, delegate (int x) { return x % 2 == 0; });
      Console.WriteLine("짝수의 갯수 : " + n);

      n = Count(arr, delegate (int x) { return x % 2 != 0; });
      Console.WriteLine("홀수의 갯수 : " + n);
    }

    //private static int Count(int[] arr, Func<int, bool> testMethod)
    private static int Count(int[] arr, MemberTest testMethod)
    {
      int cnt = 0;
      foreach (var n in arr)
      {
        if (testMethod(n))
          cnt++;
      }
      return cnt;
    }
  }
}
```

이름 없는 델리게이트
Anonymous Delegate

```
using System;

namespace A115_FuncAndAction
{
  class Program
  {
    static void Main(string[] args)
    {
      var arr = new[] { 3, 34, 6, 34, 7, 8, 24, 3, 675, 8, 23 };

      int n = Count(arr, delegate (int x) { return x % 2 == 0; });
      Console.WriteLine("짝수의 갯수 : " + n);

      n = Count(arr, delegate (int x) { return x % 2 != 0; });
      Console.WriteLine("홀수의 갯수 : " + n);
    }

    private static int Count(int[] arr, Func<int, bool> testMethod)
    {
      int cnt = 0;
      foreach (var n in arr)
      {
        if (testMethod(n))
          cnt++;
      }
      return cnt;
    }
  }
}
```

Func와 Action으로 델리게이트를 더 간단히 만들기

델리게이트를 사용하려면 우선 delegate를 선언해야 하는데 이것도 사실은 번거로운 일이다. .NET에서는 Func와 Action 델리게이트를 미리 만들어서 제공한다. 이를 사용하면 delegate를 선언할 필요가 없다. Func 델리게이트는 결과를 반환하는 메소드를 참조하기 위해서, Action 델리게이트는 반환값이 없는 메소드를 참조한다.

람다식(Lambda Expression)

```
using System;
namespace A116_LambdaExpression{
  class Program  {
    static void Main(string[] args)
    {
      var arr = new[] { 3, 34, 6, 34, 7, 8, 24, 3, 675, 8, 23 };

      int n = Count(arr, x => x % 2 == 0);
      Console.WriteLine("짝수의 갯수 : " + n);

      n = Count(arr, x => x % 2 == 1);
      Console.WriteLine("홀수의 갯수 : " + n);
    }

    private static int Count(int[] arr, Func<int, bool> testMethod)
    {
      int cnt = 0;
      foreach (var n in arr)
      {
        if (testMethod(n))
          cnt++;
      }
      return cnt;
    }
  }
}
```

람다식은 익명 메소드를 간단하게 표현할 수 있는 방법이다. 람다식은 개체로 처리되는 코드 블록(식 또는 블록)이며 메소드와 같이 매개변수와 리턴 값을 갖는다. 람다식은 인수를 메소드에 전달할 수 있으며 값을 반환할 수 있다. 람다식은 델리게이트로 표현될 수 있는 코드이다. 람다식의 델리게이트형은 리턴 값이나 파라미터의 개수에 따라 정해진다. 리턴 값이 없는 람다식은 Action 델리게이트에 해당하고 리턴값이 있는 람다식은 Func 델리게이트에 해당한다.

람다식(Lambda Expression)의 사용

```
using System;

namespace A117_LambdaExamples
{
  class Program
  {
    delegate double CalcMethod(double a, double b);
    delegate bool IsTeenAger(Student student);
    delegate bool IsAdult(Student student);

    static void Main(string[] args)
    {
      Func<int, int> square = x => x * x;
      Console.WriteLine(square(5));

      int[] numbers = { 2, 3, 4, 5 };
      var squaredNumbers = numbers.Select(x => x * x);
      Console.WriteLine(string.Join(" ", squaredNumbers));

      Action line = () => Console.WriteLine();
      line();

      CalcMethod add = (a, b) => a + b;
      CalcMethod subtract = (a, b) => a - b;
```

```
        Console.WriteLine(add(10, 20));
        Console.WriteLine(subtract(10.5, 20));

        IsTeenAger isTeenAger = delegate (Student s) { return s.Age > 12 && s.Age < 20; };

        Student s1 = new Student() { Name = "John", Age = 18 };
        Console.WriteLine("{0}은 {1}.", s1.Name, isTeenAger(s1) ? "청소년입니다" : "청소년이 아닙니다");

        IsAdult isAdult = (s) => {
          int adultAge = 18;
          return s.Age >= adultAge;
        };

        Student s2 = new Student() { Name = "Robin", Age = 20 };
        Console.WriteLine("{0}은 {1}.", s2.Name, isAdult(s2) ? "성인입니다" : "성인이 아닙니다");
    }

    public class Student
    {
      public string Name { get; set; }
      public int Age { get; set; }
    }
  }
}
```

멀티 캐스트

- 하나의 델리게이트 객체에 형태가 동일한 여러 개의 메소드를 연결하여 사용 가능
  - C# 언어는 델리게이트를 위한 +와 – 연산자(메소드 추가/제거)를 제공

- 멀티캐스트 델리게이션(multicast delegation)
  - 델리게이트 연산을 통해 하나의 델리게이트 객체에 여러 개의 메소드가 연결되어 있는 경우, 델리게이트 호출을 통해 연결된 모든 메소드를 한번에 호출
  - 델리게이트를 통하여 호출되는 순서는 등록된 순서와 동일

```
using System;

namespace MultiCastApp
{
    delegate void MultiCastDelegate();
    class Schedule
    {
        public void Now()
        {
            Console.WriteLine("Time : " + DateTime.Now.ToString());
        }
        public static void Today()
        {
            Console.WriteLine("Date : " + DateTime.Today.ToString());
        }
    }
    class Program
    {
        public static void Main()
        {
            Schedule obj = new Schedule();
            MultiCastDelegate mcd = new MultiCastDelegate(obj.Now);
            mcd += new MultiCastDelegate(Schedule.Today);
            mcd();
        }
    }
}
```

```
using System;

namespace DelegateOperationApp
{
    delegate void MultiDelegate();
    class DelegateClass
    {
        public void MethodA()
        {
            Console.WriteLine("In the DelegateClass.MethodA ...");
        }
        public void MethodB()
        {
            Console.WriteLine("In the DelegateClass.MethodB ...");
        }
        public void MethodC()
        {
            Console.WriteLine("In the DelegateClass.MethodC ...");
        }
    }
```

```
class Program
    {
        public static void Main()
        {
            DelegateClass obj = new DelegateClass();// 클래스 객체 생성
            MultiDelegate dg1, dg2, dg3;              // 델리게이트 선언
            dg1 = new MultiDelegate(obj.MethodA);   // 델리게이트 객체 생성
            dg2 = new MultiDelegate(obj.MethodB);   // 델리게이트 객체 생성
            dg3 = new MultiDelegate(obj.MethodC);   // 델리게이트 객체 생성
            // ...
            dg1 = dg1 + dg2;                          // 메소드 추가
            dg1 += dg3;                              // 메소드 추가
            dg2 = dg1 - dg2;                          // 메소드 제거
            dg1();
            Console.WriteLine("After dg1 call ...");
            dg2();
            Console.WriteLine("After dg2 call ...");
            dg3();
        }
    }
}
```

이벤트

- 이벤트(event)
  - 사용자 행동에 의해 발생하는 사건
  - 어떤 사건이 발생한 것을 알리기 위해 보내는 메시지
  - C#에서는 이벤트 개념을 프로그래밍 언어 수준에서 지원
- 이벤트 처리기(event handler)
  - 발생한 이벤트를 처리하기 위한 메소드
- 이벤트-주도 프로그래밍(event-driven programming)
  - 이벤트와 이벤트 처리기를 통하여 객체에 발생한 사건을 다른 객체에 통지하고 그에 대한 행위를 처리하도록 시키는 구조를 가짐
  - 각 이벤트에 따른 작업을 독립적으로 기술
  - 프로그램의 구조가 체계적/구조적이며 복잡도를 줄일 수 있음

이벤트 정의

- 정의 형태

```
[event-modifier] event DelegateType EventName;
```

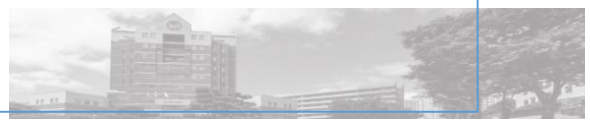
- 수정자
  - 접근 수정자
  - new, static, virtual, sealed, override, abstract, extern
  - 이벤트 처리기는 메소드로 배정되기 때문에 메소드 수정자와 종류/의미가 같음

이벤트 정의

- 이벤트 정의 순서
  ① 이벤트 처리기의 형태와 일치하는 델리게이트를 정의
  (또는 System.EventHandler 델리게이트를 사용)
  ② 델리게이트를 이용하여 이벤트를 선언
  (미리 정의된 이벤트인 경우에는 생략)
  ③ 이벤트 처리기를 작성
  ④ 이벤트에 이벤트 처리기를 등록
  ⑤ 이벤트를 발생
  (미리 정의된 이벤트는 사용자 행동에 의해 이벤트가 발생)

- 이벤트가 발생되면 등록된 메소드가 호출되어 이벤트를 처리
  - 미리 정의된 이벤트 발생은 사용자의 행동에 의해서 발생
  - 사용자 정의 이벤트인 경우에는 명시적으로 델리게이트 객체를 호출함으로써 이벤트 처리기를 작동

```
using System;
namespace EventHandlingApp{
    public delegate void MyEventHandler();                    // 1. 이벤트를 위한 델리게이트 정의
    class Button
    {
        public event MyEventHandler Push;                     // 2. 이벤트 선언
        public void OnPush()
        {
            if (Push != null)
                Push();                                       // 5. 이벤트 발생
        }
    }
    class EventHandlerClass    {
        public void MyMethod()                                // 3. 이벤트 처리기 작성
        {
            Console.WriteLine("In the EventHandlerClass.MyMethod ...");
        }
    }
    class Program    {
        public static void Main()        {
            Button button = new Button();
            EventHandlerClass obj = new EventHandlerClass();
            button.Push += new MyEventHandler(obj.MyMethod);    // 4. 등록
            button.OnPush();
        }
    }
}
```



이벤트 정의

- 이벤트 처리기 등록
  - 델리게이트 객체에 메소드를 추가/삭제하는 방법과 동일
  - 사용 연산자
    - = : 이벤트 처리기 등록
    - + : 이벤트 처리기 추가
    - - : 이벤트 처리기 제거

```
Event = new DelegateType(Method);  // 이벤트 처리기 등록
Event += new DelegateType(Method); // 이벤트 처리기 추가
Event -= new DelegateType(Method); // 이벤트 처리기 제거
```

이벤트의 활용

- C# 언어에서의 이벤트 사용
  - 프로그래머가 임의의 형식으로 델리게이트를 정의하고 이벤트를 선언할 수 있도록 허용
  - .NET 프레임워크는 이미 정의된 System.EventHandler 델리게이트를 이벤트에 사용하는 것을 권고
  - System.EventHandler

```
delegate void EventHandler(object sender, EventArgs e);
```

- 이벤트와 윈도우 환경
  - 이벤트는 사용자와 상호작용을 위해 주로 사용
  - 윈도우 프로그래밍 환경에서 사용하는 폼과 수많은 컴포넌트와 컨트롤에는 다양한 종류의 이벤트가 존재
  - 프로그래머로 하여금 적절히 사용할 수 있도록 방법론을 제공한

구조체

- 구조체(struct)
  - 클래스와 동일하게 객체의 구조와 행위를 정의하는 방법
  - 클래스 – 참조형, 구조체 – 값형
  - 예제 StructApp.cs – 구조체를 선언하여 활용한 예제
- 구조체의 형태

```
[struct-modifiers] struct StructName {
        // member declarations
}
```

- 구조체의 수정자
  - public, protected, internal, private, new

구조체

- 구조체와 클래스 차이점

① 클래스는 참조형이고 구조체는 값형이다.

② 클래스 객체는 힙에 저장되고 구조체 객체는 스택에 저장된다.

③ 배정 연산에서 클래스는 참조가 복사되고 구조체는 내용이 복사된다.

④ 구조체는 상속이 불가능하다.

⑤ 구조체는 소멸자를 가질 수 없다.

⑥ 구조체의 멤버는 초기값을 가질 수 없다.

Reference

- ✓ C# 프로그래밍 입문, 오세만 외4, 생능출판
- ✓ 초보자를 위한 C# 200제, 강병익, 정보문화사
- ✓ 프랙티컬 C#, 이데이 히데유키, 김범준, 위키북스
- ✓ C#언어 프로그래밍 바이블, 김명렬 외1, 홍릉과학출판사
- ✓ C# and the .NET Platform, Andrew Troelsen, 장시혁, 사이텍미디어
- ✓ https://docs.microsoft.com/ko-kr/learn/browse/?products=dotnet&terms=c%23