HCI 프로그래밍

7. Property, Indexer, Delegate, Event, Anonymous Method, Lambda, Attribute, File IO & Assembly

HCI
Human Computer Interaction

Class

클래스 멤버

- 필드 (Field)
  ✓클래스 범위에서 선언된 변수. 객체의 상태정보 저장
- 상수 (Contant)
  ✓컴파일 시간에 설정된 변경할 수 없는 필드
- 메소드 (Method)
  ✓클래스의 행동특성 정의
- 생성자 (Constructor)
  ✓ 클래스 인스턴스 생성 및 멤버 초기화
- 종료자 (Destructor)
  ✓ 리소스 해제 용도 등 매우 드물게 사용

클래스 멤버

- 연산자 (Operator)
  - ✓오버로드 된 연산자는 형식멤버로 간주.
  - ✓연산자를 오버로드 하는 경유 유형에서 공용 정적 메소드로 정의
- 속성 (Property)
  - ✓필드와 같이 상태정보를 저장하며 내부적으로 상태에 대한 접근 가능한 메소드
  - ✓Smart Field
- 인덱서 (Indexer)
  - ✓인덱서를 사용하면 배열과 유사한 방식으로 객체를 인덱싱
  - ✓Smart Array

클래스 멤버

- 이벤트 (Event)
  ✓어떤 사건이 발생하였을 때 사용자에게 해당사실을 알려주는 방법
- 중첩형식
  ✓내부에 선언된 클래스, 구조체 또는 인터페이스

Operator Overloading

C++와 마찬가지로 C#에도 연산자 오버로딩을 지원

- 즉, +, -, *, / 등과 같은 연산자는 원래의 의미인 사칙연산이 있지만 이것을 재정의하여 사용하게 해주는 것
- 연산자 오버로딩은 일반 메서드 오버로딩과 차이점은 별로 없음
- C#에서 오버로딩할 수 있는 연산자들
  - ✓단항 연산자 : +, -, !, ~, ++, --, true, false
  - ✓이항 연산자 : +, -, *, /, %, &, |, ^, 〈〈, 〉〉
  - ✓비교 연산자 : ==, !=, 〈, 〈=, 〉, 〉=

Operator Overloading

```
// 유리수
public readonly struct Fraction {
    private readonly int num;
    private readonly int den;
    public Fraction(int numerator, int denominator) {
        if (denominator == 0) {
            throw new ArgumentException("Denominator cannot be zero.", "");
        }
        num = numerator;
        den = denominator;
    }
    // 단항 연산자 오버로딩
    public static Fraction operator +(Fraction a) => a;
    public static Fraction operator -(Fraction a) => new Fraction(-a.num, a.den);
    // 이항 연산자 오버로딩
    public static Fraction operator +(Fraction a, Fraction b)
        => new Fraction(a.num * b.den + b.num * a.den, a.den * b.den);
    public static Fraction operator -(Fraction a, Fraction b) => a + (-b);
    public static Fraction operator *(Fraction a, Fraction b)
        => new Fraction(a.num * b.num, a.den * b.den);
```

Operator Overloading

```
public static Fraction operator /(Fraction a, Fraction b) {
        if (b.num == 0) {
            throw new DivideByZeroException();
        }
        return new Fraction(a.num * b.den, a.den * b.num);
    }
    // 비교 연산자 오버로딩
    public static bool operator >=(Fraction a, Fraction b)
        => (a.num/a.den >= b.num/b.den);
    public static bool operator <=(Fraction a, Fraction b)
        => (a.num/a.den <= b.num/b.den);
    public override string ToString() => $"{num} / {den}";
}
```

Operator Overloading

```
public static class OperatorOverloading {
    public static void Main() {
        var a = new Fraction(5, 4);
        var b = new Fraction(1, 2);
        Console.WriteLine(-a); // output: -5 / 4
        Console.WriteLine(a + b); // output: 14 / 8
        Console.WriteLine(a - b); // output: 6 / 8
        Console.WriteLine(a * b); // output: 5 / 8
        Console.WriteLine(a / b); // output: 10 / 4
        Console.WriteLine(a >= b); // True
        Console.WriteLine(a <= b); // false
    }
```

Property

속성이란 클래스의 속성을 함수적 동작에 의하여 표현하는 구성요소

- 필드처럼 보이지만 실제로는 메소드처럼 동작
- get, set 접근자에 의하여 표현
- 클래스의 내부구조를 추상적으로 표현하여 보호
- 내부적으로 메모리가 배정되지 않음
- 메소드처럼 동작하므로
  virtual, static, abstract, override 키워드 사용 가능

Property

속성 설명

- name : private로 설정된 Car 클래스의 멤버 필드
- Name : public으로 설정된 Car 클래스의 속성 (Property)
- get : read기능을 수행하는 메소드
- set : write기능을 수행하는 메소드
- value : set 접근자에게 전달되는 인자 값

```
public class Car
{
        private string name;

        public string Name
        {
                get { return name; }
                set { name = value; }
        }
}
```

Property

속성 종류

- Read-Write 속성 : get, set 접근자 모두를 사용하며
  property의 가장 일반적인 모습
- Read-only 속성 : get 접근자만 프로퍼티에 존재하며
  읽을 수만 있는 읽기전용 속성
- Write-only 속성 : set 접근자만 사용하며 멤버 필드에
  값을 쓰기만 할 수 있는 속성
- Static 속성 : 클래스 레벨에 존재하는 속성으로, 개체를 만들지 않고도
  사용이 가능하며, this 키워드 사용이 불가능

Property

```
public class Car    {
        private string name;
        public string Name
        {
                get { return name; }
                set { name = value; }
        }
}
public class PropertyTest  {
        public static void Main(string[] args) {
                Car c = new Car();
                // set연산자 호출
                c.Name = “아반테";
                // get연산자 호출
                Console.WriteLine(c.Name);
        }
}
```

속성 (Property)

- 은닉화 (Encapsulation)를 위해서, private 멤버 필드에 값을 얻거나 할당하는 목적의 접근 방법을 주는 public 메소드 개념
- 속성의 구조는 get과 set을 가지고 있음 – get만 사용하면 읽기 전용, set만 사용하면 쓰기 전용
- set에서 사용되는 value는 매개변수 값을 디폴트로 들어오는 값으로 사용

```
private string name;
public string Name {
        get {
                return name;  // 속성 반환 구현
        }
        set     {
                name = value; // 속성 설정
        }
}
```

Property

```
public class Car    {
        private string color;
        public string Name { // property
            get;
            set;
        }
        public string Color { // property
            get          {
                return color;
            }
            set          {
                color = value;
            }
        }
        public void Print()       {
            Console.WriteLine("Car {0}, {1}", Name, Color); // 속성이용 멤버필드 출력
        }
}
```

Property

```
public class PropertyTest {
    static void Main() {
Car c = new Car();
        c.Name = "Avante";  // property set
        c.Color = "White";  // property set
        Console.WriteLine("Car: {0}, {1}", c.Name, c.Color); // property get
        c.Print();          // method
    }
}
```

Indexer

- 인덱서(indexer)란 내부적으로 객체를 배열처럼 사용할 수 있게 해주는 일종의 연산자
- 속성과 마찬가지로 모습은 필드이지만 실제로는 메소드로 작동

Indexer

인덱서 특징

- 속성처럼 get, set 을 정의
- this 키워드를 반드시 사용
- 배열에 접근하는 것처럼 [ ] 를 사용
- 입력 파라메터인 인덱스도 여러 데이터 타입 사용 가능하나,
  주로 int 나 string 사용하여 인덱스 값을 주는 것이 일반적
- 리턴 데이터 타입이 여러 가지로 지정 가능

Indexer

```
public class MyClass  {
    private const int MAX = 10;
    private int[] data = new int[MAX];
    public int this[int index]   {
        get {
            if (index < 0 || index >= MAX)
                throw new IndexOutOfRangeException();
            else
                return data[index];
        }
        set {
            if (index >= 0 && index < MAX)
                data[index] = value;
        }
    }
}
MyClass cls = new MyClass();
cls[1] = 1024; // set 호출
System.Console.WriteLine(cls[1]);  // get 호출
```

Indexer

```
public class MyFavorite  {
    private Hashtable myFavorite = new Hashtable();
    public string this[string kind]   {
        get {
            return (string)myFavorite[kind];
        }
        set {
            myFavorite[kind] = value;
        }
    }
}

MyFavorite in = new MyFavorite ();
// string indexer 의 set 호출
in["fruit"] = "apple";
in["color"] = "blue";
// string indexer 의  get 호출
System.Console.WriteLine(in["fruit"]);  // apple
System.Console.WriteLine(in["color"]);  // blue
```

Delegate & Event

대리자 (Delegate)

- Delegate 생성
- Delegate +/- Operator

이벤트 (Event)

- Events 동작원리
- Event 구성요소
  - ✓ 이벤트 핸들러 대리자 (Delegate)
  - ✓ 이벤트를 발생시키는 객체 (Publisher)
  - ✓ 이벤트에 응답하는 객체 (Subscriber)
  - ✓ 이벤트 매개변수 (Event Argument)

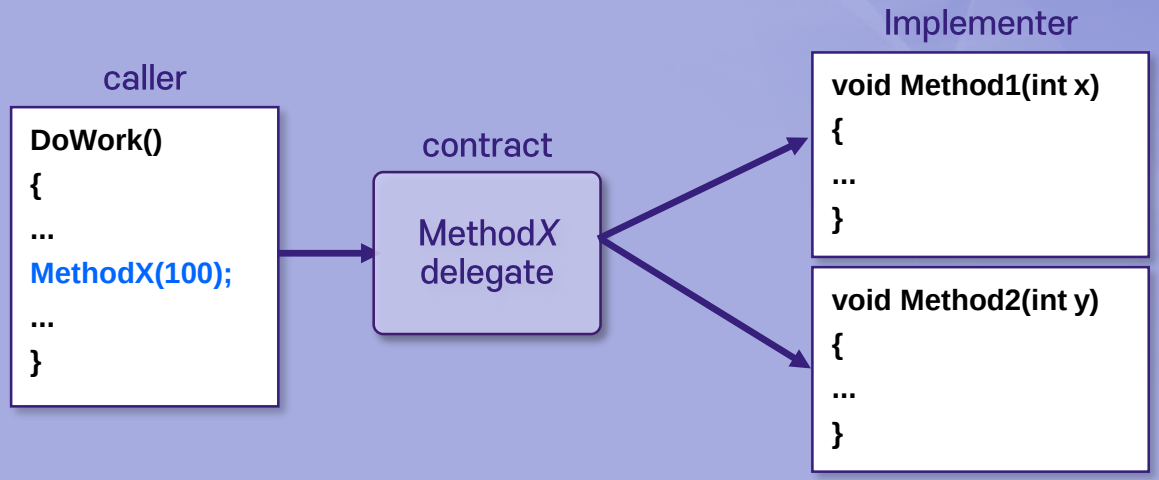
Delegate

대리자 특징

- 메소드를 간접 호출할 때 사용되는 method pointer
- 대리자의 구조는 함수의 포인터(주소)를 저장하는 구조
- 저장되는 함수의 원형과 대리자 함수의 원형은 반드시 같아야 함 (매개변수 리스트와 반환 값이 동일)
- 객체 지향의 속성 중에 다형성과 비슷한 작업을 수행
- 프로그램의 실행 시에 생성

Delegate

대리자 특징



Delegate

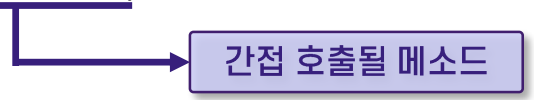
대리자 특징

- 대리자는 C++ 함수 포인터와 유사하지만 멤버 함수에 대해 완전히 객체 지향적임. 즉, 대리자는 개체 인스턴스 및 메서드를 모두 캡슐화함
- 대리자를 통해 메서드를 매개 변수로 전달할 수 있음
- 대리자를 사용하여 콜백 메서드를 정의할 수 있음
- 여러 대리자를 연결할 수 있음. 예를 들어 단일 이벤트에 대해 여러 메서드를 호출할 수 있음
- 람다 식은 인라인 코드 블록을 작성하는 더욱 간단한 방법임

Delegate 정의, 생성 및 사용

- **delegate** 키워드를 사용하여 정의
- **new** 키워드를 통하여 생성
- 대리자 호출로 메소드 간접 호출

```
// 대리자 정의 - 컴파일러에 의해 MulticastDelegate에 파생된 클래스로 변환
public delegate void DelegateMethod(int x);
// 대리자가 지칭할 메소드 구현
public static void CallbackMethod(int x) {  Console.WriteLine(x);  }
// 대리자 생성 및 초기화
DelegateMethod dm = new DelegateMethod(CallbackMethod);
// 또는 DelegateMethod dm = CallbackMethod;
...
// 대리자를 이용한 메소드 간접 호출
dm(100);      // CallbackMethod(100) 호출
```



Delegate

```
public class MethodClass {
    public void Method1(string msg) { … }
    public void Method2(string msg) { … }
}
public delegate void Del(string msg);  // 대리자 정의
public class DelegateTest {
    public static void DelegateMethod(string msg)   {
        System.Console.WriteLine(msg);
    }
    public static void MethodWithCallback(int x, int y, Del callback)   {
        callback("The number is " + (x + y).ToString());
    }
    public static void Main(string[] args)  {
        Del handler = DelegateMethod; // 대리자 생성
        handler("Hello"); // DelegateMethod("Hello") 호출
        MethodWithCallback(1, 2, handler); // DelegateMethod("The number is" + (1 + 2));
        var mc = new MethodClass();
        Del d1 = mc.Method1;
        Del d2 = mc.Method2;
        Del d3 = d1 + d2;
    }
}
```

Delegate

```
public class Click  {
    public void MouseClick(string what)   {
        System.Console.WriteLine("마우스의 {0} 버튼이 클릭됐습니다.", what);
    }
    public void KeyBoardClick(string what) {
        System.Console.WriteLine("키보드의 {0} 버튼이 클릭됐습니다.", what);
    }
}
public class DelegateTest  {
    public delegate void OnClick(string what);      // 대리자 정의
    public static void Main(string[] args)  {
        Click c = new Click();
        OnClick dm = new OnClick(c.MouseClick);
        dm("왼쪽");    // c.MouseClick(“왼쪽”) 호출
        dm = new OnClick(c.KeyBoardClick);
        dm("스페이스");  // c.KeyboardClick(“스페이스”) 호출
     }
```

Delegate 사용 예

```
using System;

class NumClass {
    public int number;

    public NumClass() {
        this.number = 0;
    }

    public void Plus(int value) {
        this.number += value;
    }

    public void Minus(int value) {
        this.number -= value;
    }

    public static void PrintHello(int value) {
      for(int i=0;i<value;i++)
        Console.WriteLine("Hello");
    }
}
```

```
// Callback할 메소드 형식으로 대리자 선언
public delegate void Handler(int value);
class DelegateTest {
  public static void Main() {
    NumClass c = new NumClass();
    // 인스턴스 메소드 위임
    Handler h = new Handler(c.Plus); // c.Plus 대리자 생성
    h(10);    // 대리자로 c.Plus(10) 호출 => 10
    Console.WriteLine("h(10)={0}",c.number); // h(10)=10
    c.Plus(20); // 30
    Console.WriteLine("c.Plus(20)={0}",c.number); //c.Plus(20)=30
    h = new Handler(c.Minus);  // c.Minus 대리자 생성
    h(10);  // 대리자로 c.Minus(10) 호출 => 20
    Console.WriteLine("h(10)={0}",c.number); // h(10)=20
    c.Minus(20); // 0
    Console.WriteLine("c.Minus(20)={0}",c.number); //c.Minus(20)=0
    //정적 메소드 위임
    h = new Handler(NumClass.PrintHello); // 대리자 생성
    h(3); // 대리자로 NumClass.PrintHello(3) 호출 – Hello 3번 출력
  }
}
```

Multiple Delegate

Delegate Operator

- Combine (+) : 대리자를 결합하는 연산자
- Remove (-) : 대리자를 제거하는 연산자

```
public static void Main() {
        NumClass c = new NumClass();
        Handler  h = new Handler(c.Plus) + new Handler(c.Minus); // 대리자 Combine
        //h = (Handler) Delegate.Combine(new Handler(c.Plus), new Handler(c.Minus));
        h(10);  // c.Plus(10)와 c.Minus(10)를 함께 호출
        Console.WriteLine("c.number={0}",c.number);          // c.number=0
        h = h - new Handler(c.Minus);                        // 대리자 Remove
        //h = (Handler) Delegate.Remove(h, new Handler(c.Minus));
        h(10);  // c.Plus(10)만 호출
        Console.WriteLine("c.number={0}",c.number);          // c.number=10
        h += new Handler(NumClass.PrintHello);
        h(5);   // c.Plus(5)와 NumClass.PrintHello(5)를 함께 호출
        Console.WriteLine("c.number={0}",c.number);          // c.number=15
}
```

Multiple Delegate

```
delegate void StringHandler(string s);
class DelegateTest {
        public static void Hello(string s) {
            Console.WriteLine("Hello {0}!", s);
        }

        public static void Bye(string s)    {
            Console.WriteLine("Bye Bye {0}!", s);
        }

        public static void Main() {
                StringHandler x, y, z, w;
// Hello 메소드를 참조하는 x delegate 객체 생성
                x = new StringHandler(Hello);
// Bye 메소드를 참조하는 y delegate 객체 생성
                y = new StringHandler(Bye);
// delegate x, y를 결합하여 z delegate에 대입
                z = x + y;
// 결합된 delegate z에서 x를 제거한 w delegate
                w = z - x;
 Console.WriteLine("대리자 x 호출");
  x("X");          // Hello X!
  Console.WriteLine("대리자 y 호출");
  y("Y");          // Bye Bye Y!
  Console.WriteLine("대리자 z 호출");
  z("Z");          // Hello Z! Bye Bye Z!
  Console.WriteLine("대리자 w 호출");
  w("W");          // Bye Bye W!
  }
}
```

Event

Event란 발생한 사건을 알리기 위해 보내는 메시지

- 이벤트는 마우스 클릭, 키 누름 등 동작의 발생을 알리기 위해 개체에서 보내는 메시지
- GUI 컨트롤, 특정 객체의 상태 변화를 알리는 신호로 사용

이벤트, 이벤트 발생기, 이벤트 처리기로 구성됨

- 이벤트 송신기(Publisher)에서 이벤트를 발생시키고, 이벤트 수신기(Subscriber) 에서 이벤트를 캡처하여 이벤트에 응답

NET Framework에서 이벤트는 delegate 모델에 기반

- **delegate**는 이벤트에 대해 등록된 **이벤트 처리기(event handler) 목록을 유지하여** 이벤트를 발생시킨 객체의 **발송자 역할** 담당

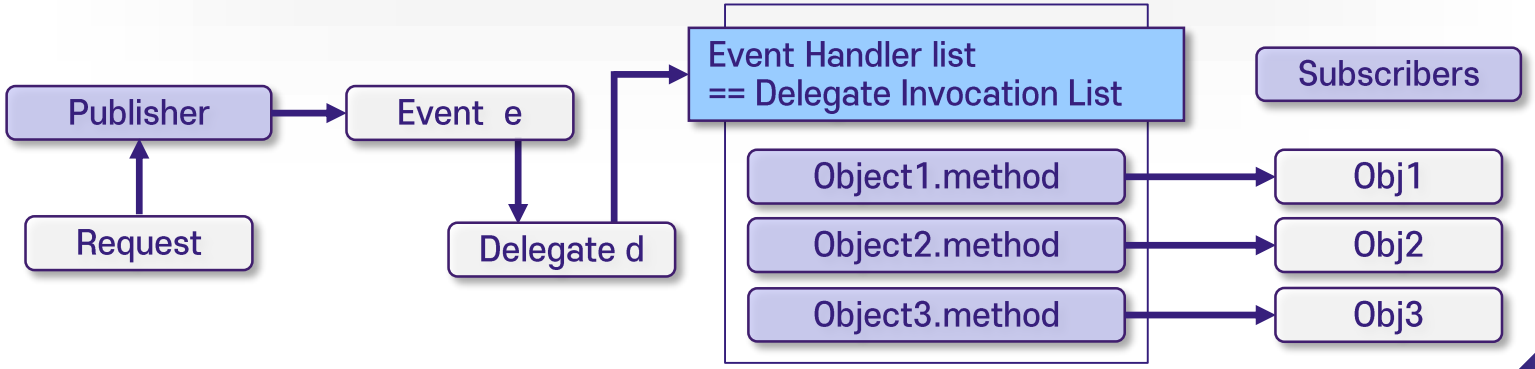
Event 동작원리

이벤트 송신기 (Publisher)

- 이벤트(Event)를 발생하여 특정 객체(Subscriber)에게 통지

이벤트 수신기 (Subscriber)

- 특정 이벤트 발생 시 이벤트에 반응하여 처리할 개체
- Publisher로부터 호출 되어질 이벤트 처리 메소드 (Event Handler)를 등록한 객체



Event 구성요소

이벤트 핸들러 대리자 (Delegate)

- **이벤트에 응답하는 메소드를 가리키는 대리자**
- 닷넷 프레임워크에서 정의된 EventHandler 대리자 형식으로 선언

```
public delegate void EventNameEventHandler (Object sender, EventArgs e)
```

- **Object sender** – 이벤트 발생 객체
- **EventArgs e** – 이벤트 발생 시 넘겨 줄 추가정보,
  EventArgs 클래스로부터 상속
- 추가정보를 사용하지 않는 이벤트에 대해서는
  닷넷 프레임워크에서 정의된 EventHandler 대리자 이용

Event 구성요소

이벤트를 발생시키는 객체 (Publisher)

- **이벤트 선언**
  - ✓event 키워드를 사용하여 대리자 형식의 필드처럼 사용
  - ✓인터페이스에서 정의 가능

```
public event EventNameEventHandler EventName;
```

Event 구성요소

이벤트를 발생시키는 객체 (Publisher)

- **이벤트를 발생시키는 내용을 포함하는 메소드(On*EventName*) 정의**
  - ✓이벤트를 선언한 클래스에서만 이벤트 호출(발생)
  - ✓하위 클래스에서 재정의하거나 호출 가능하도록 **virtual**로 정의

```
public class EventPub {
    public event MyEventEventHandler MyEvent;  // 이벤트 선언
    protected virtual void OnMyEvent(EventArgs e)  {
        if (MyEvent != null)
         MyEvent(this, e);              //이벤트 호출(발생)
    }
```

Event 구성요소

이벤트에 응답하는 객체 (Subscriber)

- 이벤트 처리 (Event Handler) 메소드 정의
  - ✓이벤트 핸들러 대리자 형식
  - ✓하위 클래스에서 재정의하거나 호출 가능하도록 virtual로 정의
- 이벤트 처리 메소드를 이벤트 대리자에 연결(등록)

```
class EventSub {
   virtual void MyEventH(Object sender, EventArgs e){...} // 이벤트 핸들러
   …
}
 …
 EventPub p = new EventPub();                    // publisher object
 EventSub s = new EventSub();                    // subscriber object
 p.MyEvent += new MyEventEventHandler(s.MyEventH);// 이벤트핸들러 등록
 …
 p.OnMyEvent(args);                              // 이벤트 요청
 …
```

Event 구성요소

이벤트 매개변수(Event Argument)

- 이벤트 발생시 추가할 정보
- 이벤트 핸들러로 넘겨줄 데이터
- EventName EventArgs 클래스 정의
- System.EventArgs에서 파생

```
// 이벤트 매개변수 클래스 정의 (System.EventArgs에서 상속)
class MyEventEventArgs : EventArgs  { …}
…
// 이벤트 매개변수 객체 생성
MyEventEventArgs args = new MyEventEventArgs(..);

// 이벤트 발생시 매개변수로 처리
p.OnMyEvent(args);
```

Delegate를 통한 이벤트 처리

```
namespace EventTest {
    public delegate void MyEventEventHandler(); // 대리자 이벤트 모델
    class MyButton { // myButton 클래스는 윈도우 응용 프로그램이 제공하는 버튼 컨트롤
        public event MyEventEventHandler MyEvent; // 이벤트 정의
        public void OnMyEvent() { if (MyEvent != null) MyEvent(); } // 이벤트 발생 메소드
    }
    class EventTest {
        public EventTest() {
            // 이벤트를 가지고 있는 MyButton을 객체로 만듦
            MyButton button1 = new MyButton();
            // 이벤트에 button1_Click을 위임
            button1.MyEvent += new MyEventEventHandler(this.button1_Click);
            // 이벤트에 button1의 이벤트를 발생
            button1.OnMyEvent();
        }
        void button1_Click() {
            Console.WriteLine("버튼에서 이벤트 발생");
        }
        static void Main(string[] args) {
            EventTest e = new EventTest(); // 버튼에서 이벤트 발생
        }
    }
}
```

Delegate를 통한 이벤트 추가 및 제거

```
class EventTest {
    public EventTest() {
        // 이벤트를 가지고 있는 MyButton을 객체로 만듦
        MyButton button1 = new MyButton();
        Console.WriteLine("클릭 이벤트 추가 후 버튼 클릭");
        // 이벤트에 button1_Click을 위임
        button1.MyEvent += new MyEventEventHandler(this.button1_Click);
        // 이벤트에 button1의 이벤트를 호출
        button1.OnMyEvent();    // button1_Click만 호출
        Console.WriteLine("변경 이벤트 추가 후 버튼 클릭");
        // 이벤트에 button1_Change를 위임
        button1.MyEvent += new MyEventEventHandler(this.button1_Change);
        //이벤트에  button1의 이벤트를 호출 (Click + Change 호출)
        button1.OnMyEvent();     // button1_Click & button1_Change 함께 호출
    }
    void button1_Click() { Console.WriteLine("->클릭 이벤트 발생"); }
    void button1_Change() { Console.WriteLine("->변경 이벤트 발생");}
    static void Main(string[] args) {
            EventTest e = new EventTest();
    }
}
```

```
클릭 이벤트 추가 후 버튼 클릭
->클릭 이벤트 발생
변경 이벤트 추가 후 버튼 클릭
->클릭 이벤트 발생
->변경 이벤트 발생
```

Event에 인자 전달

```
delegate void ClickEventHandler(string label); // delegate event
class MyButton { // MyButton 클래스는 윈도우 응용 프로그램이 제공하는 버튼 컨트롤
    public event ClickEventHandler Click; // 이벤트 정의
    public void OnClick() { if (Click != null) Click(label); } // 이벤트 발생 메소드
    public string Label { set { label = value;} get { return label; } }
    string label;
}
class EventTest {
    public EventTest() {
        // 이벤트를 가지고 있는 MyButton을 객체로 만듦
        MyButton button1 = new MyButton();
        // 이벤트에  button1_Label을 위임
        button1.Label = "테스트";
        button1.Click += new ClickEventHandler(this.button1_Label);
        // 이벤트에  button1의 이벤트를 호출
        button1.OnClick();          // button1_Label 호출
    }
    void button1_Label(string label) { Console.WriteLine("-> 클릭 이벤트 발생: " + label); }
    static void Main(string[] args) {
        EventTest e = new EventTest(); // -> 클릭 이벤트 발생: 테스트
    }
}
```

무명 메소드

무명 메소드(Anonymous Method)를 사용하여 메소드 이름이 아닌 코드 블록 자체를 대리자(Delegate)의 매개변수로 사용할 수 있음

- 별도의 메소드를 생성할 필요가 없으므로 대리자를 객체화하는데 따르는 코딩 오버헤드를 줄일 수 있음

```
public MyForm() {
    button1.Click += new ClickEventHandler(button1_Click);
}
void button1_Click(object sender, EventArgs e) {
    MessageBox.Show(textBox1.Text);
}
```

```
public MyForm() {
    button1.Click += delegate(object sender, EventArgs e) {
        MessageBox.Show(textBox1.Text);
    }
}
```

무명 메소드

대리자 비교: 명명된 메소드 vs. Anonymous Method

```
delegate void Del(int i, double j);
class MathClass {
    void Multiply(int i, double j) {
        Console.WriteLine( (i * j).ToString());
    }
    static void Main(string[] args) {
        MathClass m = new MathClass();
        Del d = m.Multiply;
        d(1, 2.0); // m.Multiply(1, 2.0) 과 동일함 결과는 2.0
        Del d2 = delegate(int i, double j) { // 무명 메소드 사용
            Console.WriteLine( (i + j).ToString());
        };
        d2(1, 2.0); // 결과는 3.0
    }
}
```

Lambda Expression

람다 식 (Lambda expression)를 사용하여 더욱 간단히 바꿀 수 있음

```
public MyForm() {
    button1.Click += delegate(object sender, EventArgs e) {
        MessageBox.Show(textBox1.Text);
    }
}
```

```
public MyForm() {
    button1.Click += (sender, e) => MessageBox.Show(textBox1.Text);
}
```

C# Lambda Expression

람다 식 (Lambda expression)은 ⇒ (“이동”) 연산자 사용

- 람다 식은 anonymous method와 비슷하게 익명함수
- ⇒ 왼쪽에 입력 인자 (0~N개)를, 오른쪽에 실행 식/문장을 둠

  i(nput parameters) => expression

  (input parameters) => { statement }
- 람다 식의 실행 문장 블록이 1개 일 경우 { } 괄호를 생략 가능

```
(x, y) => x == y
(int x, string s) => s.Length >x // 컴파일러가 입력형식을 유추 못할 때 명시
() => Write(“No”); // 입력 매개 변수가 0개 이면, 빈 괄호를 지정
(p) => Write(p);
str => { MessageBox.Show(str); } // 문자열 하나를 받아 메시지박스에 출력
delegate int del(int i);
del myDelegate =  x => x*x;
int j = myDelegate(5); // j=25
```

C# Delegate 발전 과정

- C# 1.0 – 명시적 대리자 (Delegate) 사용
- C# 2.0 – 무명 메소드 (Anonymous method) 사용
- C# 3.0 – 람다 식 (Lambda expression) 사용

```
delegate void TestDelegate(string s);
static void M(string s) {      Console.WriteLine(s);     }
static void Main(string[] args) {
    // C# 1.0
    TestDelegate testDeleA= new TestDelegate(M);
    // C# 2.0
    TestDelegate testDeleB= delegate(string s) { Console.WriteLine(s) };
    // C# 3.0
    TestDelegate testDeleC= (s) => { Console.WriteLine(s) };
    // invoke the delegates
    testDeleA("Hello. My name is M and I write lines");
    testDeleB("That's nothing. I'm anonymous and ");
    testDeleC("I'm a famous author.");
}
```

제네릭 대리자

.NET에서 제공하는 Generic Delegate

- public delegate void Action<in T>(T object)
  - T 매개변수를 받아 반환값이 없는 함수에 사용
- Public delegate TOutput Converter<TInput, TOutput> (TInput object)
  - TInput 매개변수를 받아 TOuput으로 변환하여 반환
- public delegate bool Predicate<T>(T object)
  - T 매개변수를 받아 특정 조건을 만족하는지 반환
- public delegate int Comparison<T>(T x, T y)
  - x, y 두 객체를 비교하여 x가 y보다 크면 양수, 작으면 음수, 같으면 0 반환
- public delegate TResult Func<out TResult>()
  - TResult 매개변수에 지정된 형식의 값을 반환

제네릭 대리자

public delegate void Action<in T>(T object)

- Action 대리자는 반환형 없고 매개변수는 0-16개를 받아들임

```
static void Output(string s) {
    Console.WriteLine(s);
}
static void Main(string[] args) {
    // 일반 메소드 사용
    Action<string> act = Output;
    act("hello"); // hello
    // 무명 메소드 사용
    Action<string, int> act2 = delegate(string msg, int num) {
        Console.WriteLine(msg + " num=" + num);
    };
    act2("data", 3); // data num=3
    // 람다 사용
    Action<int, int, int> act3 = (x, y, z) => Console.WriteLine("x+y+z=" + (x+y+z));
    act3(5, 6, 7); // x+y+z=18
}
```

제네릭 대리자

public delegate TResult Func<T, out TResult>()

- Func 대리자는 반환형이 있어야하고 매개변수는 0-16개 지원

```
static bool IsValidRange(int n) {
    return n >= 10;
}
static void Main(string[] args) {
    // 일반 메소드 사용
    Func<int, bool> f1 = IsValidRange;
    bool result = f1(20); // True
    // 무명 메소드 사용
    Func<int, bool> f2 = delegate(int n) {
        return n >= 10;
    };
    result = f2(-1); // False
    // 람다 사용
    Func<int, bool> f3 = n => n >= 10;
    result = f3(5); // False
```

제네릭 대리자

public delegate bool Predicate<T>(T object)

- Predicate 대리자는 반환형이 bool이고 매개변수가 1개

```
static bool IsNameLengthTwo(Person person) {
    return person.Name.Length == 2;
}
static void Main(string[] args) {
    // 일반 메소드 사용
    Predicate<Person> p1 = IsNameLengthTwo;
    bool result = p1(dooly); // False
    // 무명 메소드 사용
    Predicate<int> p2 = delegate(int n) {
        return n >= 10;
    };
    result = p2(dooly.Age); // True
    // 람다 사용
    Predicate<string> p3 = s => s.StartsWith("A");
    result = p3(dooly.Name); // False
}
```

Attribute

- 특성 (Attribute)은 미리 정의된 시스템 정보나 사용자 지정 정보를 대상요소 (어셈블리, 클래스, 구조체, 메소드 등)와 연결시켜 주는 기능을 가짐
- Attribute 정보는 Assembly에 Metadata 형식으로 저장됨

Attribute

Attribute 형식

```
[attribute명 (“positional_parameter”, named_parameter=value,..)]
```

- Attribute는 [] 를 사용하여, [] 안에 Attribute 이름,
  지정위치 매개변수와 명명 매개변수를 기입
- 지정위치 매개변수 (positional_parameter)
  – 필수적인 정보, 생성자 매개변수에 해당, “ ”을 사용하여 값을 기입
- 명명 매개변수 (named_parameter)
  – 선택적인 정보, 속성에 해당, ‘=‘를 사용하여 멤버필드와 값을 기입

내장 특성에 Conditional, DllImport, AttributeUsage 등이 있음

Conditional Attribute

Conditional 특성은 조건부 메소드를 생성할 때 사용

- **특정 전처리 식별자에 의해 실행되는 조건부 메소드**
- C++에서 #if *conditional* … #endif 전처리기 지시문과 유사
- #define 유무에 따라서 호출이 결정되는 조건부 메소드에 사용
- 반드시 **using System.Diagnostics**를 사용해야 함
- 조건부 메소드의 반환형이 **void** 형을 사용해야 함

```
#define DEBUG      // 만약 #undef DEBUG를 하면 Conditional 부분이 지나감
using System.Diagnostics;
class ConditionalAttributeTest {
    public static void Main() {
        [Conditional("DEBUG")]
        public static void DebugInfoPrint() { …. } // 요부분만 호출
        [Conditional("REGULAR")]
        public static void InfoPrint() { …. }
    }
}
```

DllImport Attribute

DllImport 특성은 닷넷 응용프로그램에서 관리되지 않는 DLL 함수 또는 메소드를 사용할 수 있게 하는 특성

- 반드시 **using System.Runtime.InteropService**를 사용해야 함
- 아래 예는, Win32 API의 MessageBox 함수를 호출하는 경우

```
using System.Runtime.InteropServices;
class DllImportAttributeTest {
    [DllImport("User32.dll", CharSet = CharSet.Auto)]
    public static extern int MessageBox(int h, String text, String title, uint type);
    public static void Main() {
        MessageBox(0, "Test Win32 MessageBox", "DllImportTest", 2);
    }
}
```

FILE I/O

스트림 (Stream)

- 자료의 입출력을 도와주는 추상적인 개념의 중간매체
  - ✓입력 스트림(Input Stream)은 데이터를 스트림으로 읽어들임
  - ✓출력 스트림(Output Stream)은 데이터를 스트림으로 내보냄
- 스트림을 사용하는 곳
  - ✓파일
  - ✓키보드, 모니터, 마우스
  - ✓메모리
  - ✓네트워크
  - ✓프린트

FILE I/O

입출력 스트림 (Input/Output Stream) 클래스

- FileStream – 파일에 스트림을 생성하는 클래스
- BufferedStream – 버퍼기능을 가진 바이트스트림
- MemoryStream – 메모리에 입출력 바이트스트림
- TextReader & TextWriter – 문자스트림 입출력 추상클래스
- StringReader & StringWriter – string 입출력 스트림
- BinaryReader & BinaryWriter – 데이터타입의 메모리 사이즈에 따른 바이너리 입출력 스트림
- StreamReader & StreamWriter 클래스 – 문자스트림

파일 (File)과 디렉토리 (Directory) 클래스

- FileSystemInfo – 파일 시스템 객체를 나타내는 기본 클래스
- Directory, DirectoryInfo – 디렉토리를 나타내는 기본 클래스
- File, FileInfo – 파일을 나타내는 기본 클래스
- Path – 경로 클래스
- DriveInfo – 드라이브를 나타내는 클래스

FILE I/O

File 클래스

- I/O 기본 클래스로 파일에 관련된 정보를 제공하거나, FileStream의 객체를 생성하여 파일의 I/O작업을 수행
- sealed 키워드를 사용하여 클래스의 상속을 막음
- 멤버 메소드들이 public static으로 선언
- using System.IO를 사용

FILE I/O

File 클래스

- 파일관련 메소드 제공
  - ✓Create, Copy, Move, Delete – 파일을 생성, 복사, 이동, 삭제
  - ✓Open, OpenRead, OpenText, OpenWrite – 파일 열기
  - ✓AppendText – 유니코드 텍스트를 추가하는 StreamWriter 생성
  - ✓Exists – 파일 존재 여부를 확인
  - ✓SetCreationTime, GetCreationTime – 파일이 생성된 날짜와 시간
  - ✓SetAttributes, GetAttributes – 파일의 지정된 FileAttributes 등

Directory 클래스

- 디렉토리 생성, 이동, 삭제, 디렉토리 존재여부, 하위 디렉토리들의 이름, 디렉토리 내의 파일 이름의 정보를 알아내는데 사용하는 클래스
- sealed 키워드를 사용하여 클래스의 상속을 막음
- 멤버 메소드들이 public static으로 선언
- **디렉토리 관련 메소드 제공**
  - ✓CreateDirectory, Delete – 디렉토리 생성, 이동, 삭제
  - ✓Exists – 디렉토리 존재 여부를 확인
  - ✓GetFiles – 디렉토리에 있는 파일목록 배열 반환
  - ✓GetDirectories – 디렉토리에 있는 하위 디렉토리 목록 배열 반환

Path 클래스

- 파일이나 디렉토리의 경로 (Path)를 확장 및 변경, 수정하는 클래스
- sealed 키워드를 사용하여 클래스의 상속을 막음
- 멤버 필드와 메소드들이 public static으로 선언
- 경로 관련 메소드 제공
  - ✓DirectorySeparatorChar – 디렉토리 구분자 캐릭터
  - ✓Combine – 경로들 결합
  - ✓GetFileName – 경로에서 파일이름을 얻기

FILE I/O

- using System.IO
- FileStream 클래스
  - 파일 입출력 뿐만 아니라 파일과 관련된 다른 운영체제의 핸들 (파이프, 표준입력, 표준출력 등)을 읽고 쓰는데도 유용하게 사용
- 바이트스트림이 아닌 문자스트림을 사용하기 위해서는 FileStream을 StreamReader와 StreamWriter로 변환하여 사용
- 바이트스트림을 사용하기 위해서는 FileStream을 BinaryReader와 BinaryWriter로 변환하여 사용

FileStream 예제

```
string path = @“C:/Test.txt";
if (File.Exists(path)) { // Delete the file if it exists.
    File.Delete(path);
}
using (FileStream fs = File.Create(path)) { //Create the file.
    AddText(fs, "This is some text");
    AddText(fs, "\r\nand this is on a new line");
    for (int i=1; i < 100; i++) AddText(fs, Convert.ToChar(i).ToString());
}
using (FileStream fs = File.OpenRead(path)) { //Open the stream
    byte[] b = new byte[1024];
    UTF8Encoding temp = new UTF8Encoding(true);
    while (fs.Read(b,0,b.Length) > 0) {
        Console.WriteLine(temp.GetString(b));
    }
}
static void AddText(FileStream fs, string value) {
    byte[] b = new UTF8Encoding(true).GetBytes(value);
    fs.Write(info, 0, info.Length);
}
```

StreamReader & StreamWriter 예제

```
using (StreamReader r = new StreamReader(@“C:/Test.txt”)) {
    while ((line = r.ReadLine()) != null) {
        Console.WriteLine(line);
    }
}
```

```
FileStream fs = File.OpenRead(@“C:/Test.txt”);
StreamReader r = new // 한글 처리시 필요
                 StreamReader(fs, System.Text.Encoding.Default);
r.BaseStream.Seek(0, SeekOrigin.Begin);
while(r.Peek() > -1) { r.ReadLine(); …. }
r.Close();
```

StreamReader & StreamWriter 예제

```
using (StreamWriter w = new StreamWriter(@"C:/Test.txt")) {
    w.WriteLine("Line 1");
    w.WriteLine("Line 2");
    w.WriteLine(new char[5] {'A', 'B', 'C', 'D', 'E'});
```

```
string path = @"C:\Output.txt";
FileStream fs = new FileStream(path, FileMode.Create);
StreamWriter w = new // 한글 처리시 필요
                StreamWriter(fs, System.Text.Encoding.Default);
//w.BaseStream.Seek(0, SeekOrigin.Begin);
w.BaseStream.Seek(0, SeekOrigin.End); // 맨 뒤에 추가
w.WriteLine(s);        //…. 중간생략
w.Flush();     // 스트림에 기록한 모든 데이터를 밀어내는 역할
w.Close();
```

BinaryReader & BinaryWriter 예제

```
FileStream fs = new FileStream(@“C:/Data.bin”,
                    FileMode.CreateNew, FileAccess.Write);
BinaryWriter bw = new BinaryWriter(fs);
int i = 10;
float f = 123.4f;
double d = 567890;
string s = “test”;
bw.Write(i);
bw.Write(f);
bw.Write(d);
bw.Write(s);
bw.Close();
fs.Close();
```

BinaryReader & BinaryWriter 예제

```
FileStream fs = new FileStream(@“C:/Data.bin”,
                    FileMode.Open, FileAccess.Read);
BinaryReader br = new BinaryReader(fs);
int i = br.ReadInt32();
float f = br.ReadSingle();
double d = br.ReadDouble();
string s = br.ReadString();
Console.WriteLine(“i=“ + i + “ f=“ + f + “ d=“ + d + “ s=“ + s);
br.Close();
fs.Close();
```

COM

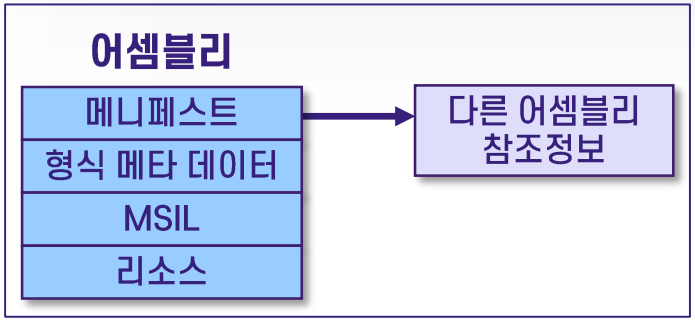
- 이미 사용 중인 검증된 코드의 재사용을 위해 COM 활용
- 서로 다른 언어로 작성된 Binary 타입 공유를 위해 COM 서버 생성
- 멤버 필드와 메소드들이 public static으로 선언
  - ✓Client에서 COM 서버를 호출할 때 COM 서버의 버전을 확인할 수 없음
  - ✓COM 서버 위치나 이름이 바뀌게 되면 Registry의 변경이 쉽지 않음

Assembly

- 컴파일을 통해서 나온 결과 파일을 어셈블리(Assembly)라고 함
- .exe, .dll과 같은 프로그램의 제일 작은 실행 단위
- 배포의 단위로써 코드 재사용 및 버전관리를 가능하게 하는 단위
- Class 접근 제한자인 internal의 허용 단위
- 같은 COM DLL에 대한 서로 다른 버전 동시 제공가능
  (즉, Client가 원하는 버전을 파악하여 해당버전을 로딩)
- 어셈블리 내에 자신에 대한 메타데이터를 포함하여 배포는 해당 파일을 원하는 위치에 복사함 (즉, Registry에 등록하지 않음)
- Native Code가 아니라 MSIL이라는 중간코드

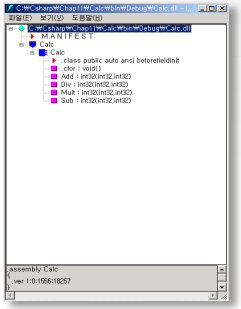
어셈블리의 구성

- Manifest – 다른 어셈블리를 참조한다면 그 정보가 필요함
- Type Metadata – 어셈블리 안에 존재하는 클래스, 속성, 메소드, 변수 등에 대한 정보
- MSIL – 컴파일 했을 때 만들어지는 실제 실행 파일
- Resource – 어셈블리가 사용하는 리소스이며 해당 프로그램의 이미지나 텍스트, 아이콘 등



ILDASM 도구

- Visual C++의 dumpbin.exe나 Turbo C의 tdump.exe에 해당하는 유틸리티로 실행 파일의 내부 구조를 보여줌
- 메니페스트 메타정보를 표시
- .assembly
- .class : 해당 어셈블리 내에 존재하는 클래스 파일의 정보를 보여줌
- .method : 해당 어셈블리 내에 존재하는 메소드의 정보를 보여줌
- .ctor : constructor 파일의 약자로 생성자를 의미



Assembly

C# DLL 생성

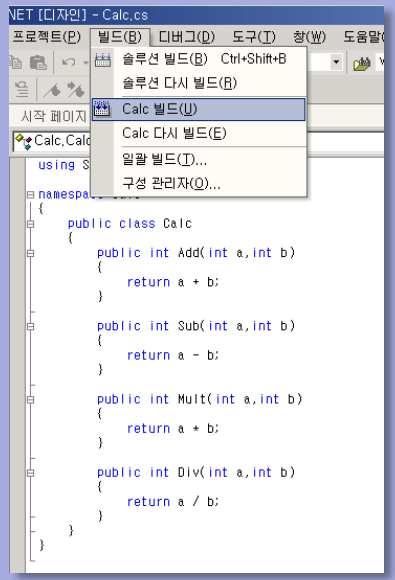
- 새프로젝트 → 템플렛 → 클래스 라이브러리 선택
- 어셈블리로 사용할 기능은 *.cs 파일로 작성
- 컴파일은 *.dll 파일로 작성

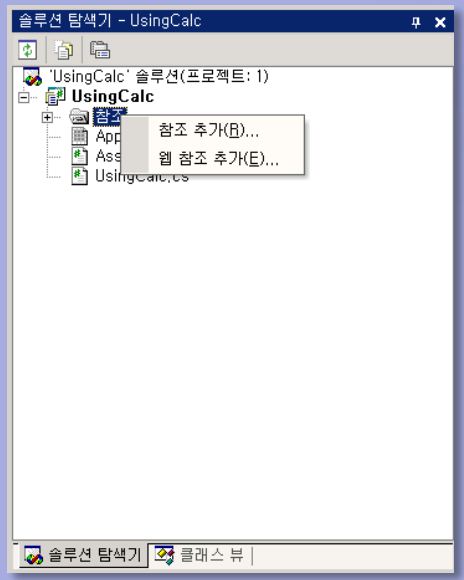
C# DLL 활용

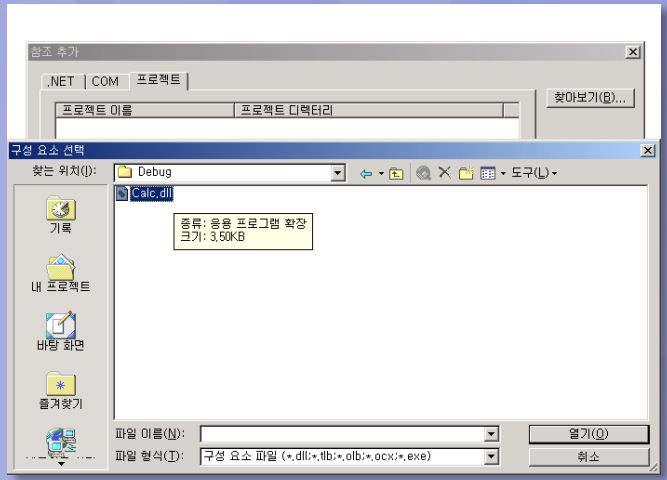
- 해당 어셈블리가 필요한 곳에서 DLL 참조하여 사용
- 해당 어셈블리의 네임스페이스 using
- 어셈블리의 속성, 메소드 사용

Assembly

작업한 어셈블리 참조추가하기







csc 컴파일러를 사용한 C# DLL 생성 및 활용 예

- Point 클래스와 Point3D 클래스를 PointLib 라이브러리로 생성

  csc /out:PointLib.dll /t:library Point.cs Point3D.cs

- PointTest 클래스에서 PointLib 라이브러리 사용

  csc /out:PointTest.exe /reference:PointLib.dll PointTest.cs