고급 C# 프로그래밍

321190
2017년 가을학기
10/10/2017
박경신

Overview

- 속성(Property), 인덱서(Indexer)
- 대리자(Delegate), 이벤트(Event)
- 무명메소드(Anonymous Method)
- 람다식(Lambda Expression)
- 특성(Attribute)
- 제네릭(Generic)
- 컬렉션(Collections)
- 파일입출력 (File I/O)
- 어셈블리 (Assembly)

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

- 속성설명
  - 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);
        }
}
```

Indexer

- 인덱서(indexer)란 내부적으로 객체를 배열처럼 사용할 수 있게 해주는 일종의 연산자
- 속성과 마찬가지로 모습은 필드이지만 실제로는 메소드로 작동
- 인덱서 특징
  - 필드처럼 보이지만 실제로는 메소드처럼 동작
  - get, set 접근자에 의하여 표현
  - this 키워드를 반드시 사용
  - 배열에 접근하는 것처럼 [] 기호를 사용

Indexer

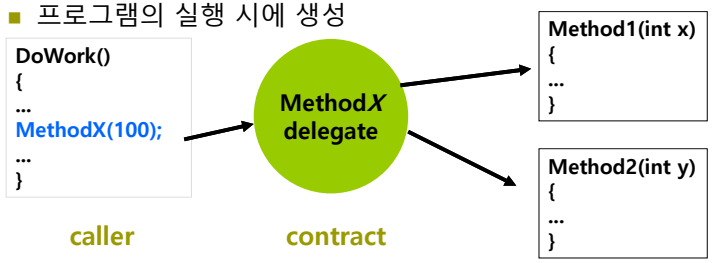
```
public class FavoriteIndexer  {
        private Hashtable myFavorite = new Hashtable();
        public string this[string kind]   {
                get { return (string)myFavorite[kind]; }
                set { myFavorite[kind] = value; }
        }
}
public class IndexerTest   {
        public static void Main(string [] args)  {
                 FavoriteIndexer in = new FavoriteIndexer ();
                // string indexer 의 set 연산자 호출
                in["fruit"] = "apple";
                in["color"] = "blue";
                // string indexer 의  get 연산자 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

- Delegate 정의, 생성 및 사용
  - **delegate** 키워드를 사용하여 정의
  - **new** 키워드를 통하여 생성
  - Delegate 호출로 메소드 간접 호출

```
// 대리자 정의
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
```

간접 호출될 메소드

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 delegate void OnClick(string what);      // 대리자 정의
public class DelegateTest  {
    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;
```

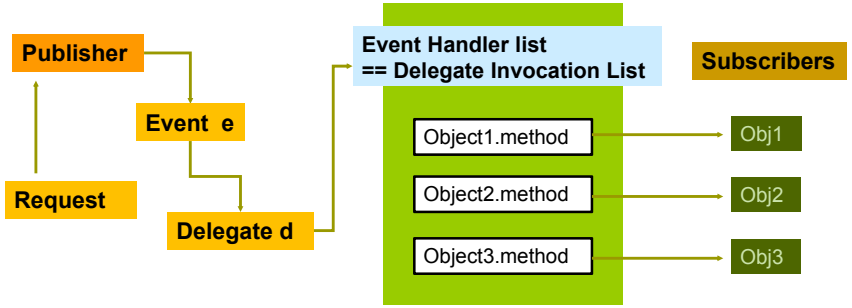
```
    Console.WriteLine("대리자 x 호출");
    x("X");   // Hello X!
    Console.WriteLine("대리자 y 호출");
    y("Y");   // Bye Bye Y!
    Console.WriteLine("대리자 z 호출");
    z("Z");   // Hello Z! Bye Bye Z!
    Console.WriteLine("대리자 w 호출");
    w("W"); // Bye Bye W!
    }
}
```


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 Framework에서의 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;
```

  - **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 **virtual**로 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

```
using System;
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
        }
}
```

Anonymous Method

- 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);
    }
}
```


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 둔다.

    (input 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.");
}
```

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 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);
    }
}
```


Generic

- 제네릭 (Generic)은 다양한 자료형에 적용될 수 있는 일반적인 클래스를 정의함
- 제네릭은 사용할 자료형을 매개변수로 전달받음

```
public class IntStack {
    int[] items;
    int count;
    public void Push(int item) { ... }
    public int Pop() { ... }
}

public class FloatStack {
    float[] items;
    int count;
    public void Push(float item) { ... }
    public float Pop() { ... }
}
```

```
public class Stack<T> {
    T[] items;
    int count;
    public void Push(T item) { ... }
    public T Pop() { ... }
}
....
Stack<int> istack = new Stack<int>();
istack.Push(3);
int x = istack.Pop();
Stack<float> fstack = new Stack<float>();
fstack.Push(3.0);
float y = fstack.Pop();
```

Collections

- 컬랙션 (Collections)은 객체를 쉽게 다룰 수 있도록 여러 가지 클래스와 인터페이스를 미리 정의한 데이터 구조
  - 데이터 구조란 데이터를 다루는 방식을 정의한 것
  - 프로그래머가 일일이 데이터 구조를 만드는 불편함 감소
  - 컬랙션으로 ArrayList, SortedList, Hashtable, Stack, Queue, NameValueCollection 등이 있음
- 컬랙션 특징
  - **데이터를 보관할 수 있으며 수정, 삭제, 삽입, 검색 등의 기능**
  - 컬랙션은 클래스마다 구현되어지는 알고리즘(예로, LinkedList, Hash, Stack, Queue 등)이 다를 뿐 같은 부류임
  - **동적으로 메모리 확장 가능**

Collections

- System.Collections의 클래스와 인터페이스
  - SortedList
  - Stack, Queue
  - BitArray
  - NameValueCollections
  - IEnumerable, IEnumerator
  - ICollection
  - IList
  - IDictionary, IDictionaryEnumerator
  - ICloneable
  - ISerializable
  - ArrayList
  - Hashtable

SortedList Class

- SortedList는 Hashtable과 ArrayList의 혼합형
  - 내부의 데이터는 키(Key)와 값(Value)로 이루어져 있으며 키를 기준으로 정렬되고 키와 인덱스로 접근가능
  - 내부적으로 정렬된 컬랙션을 유지하고 있는 특징을 가짐

```
public class SortedList: IDictionary, ICollection, IEnumerable, ICloneable
```

- SortedList 특징
  - 키의 목록 또는 값의 목록만 반환하는 메소드 제공
  - 내부적으로 두 개의 배열, 즉 키에 대한 배열과 값에 대한 배열을 유지하여 요소를 목록에 저장
  - SortedList는 각 요소에 대해 키, 값 또는 인덱스의 세가지 방법으로 접근
  - 요소가 삽입되면, 지정된 키가 이미 존재하는 검사 (중복키 허용안함)

SortedList Class

- SortedList 메소드
  - Add() – 키와 값으로 데이터를 삽입
  - Clear() – 모든 요소를 제거
  - Contains() – 특정 키가 들어 있는지 여부를 확인
  - ContainsKey(), ContainsValue() – 특정 키/값이 들어있는지 여부 확인
  - GetByIndex(), GetKey() – 지정한 인덱에서 값/키를 가져옴
  - GetKeyList() – 키 리스트를 가져옴
  - Remove(), RemoveAt() – 지정한 키/인덱스로 요소를 제거
  - GetEnumerator() – IDictionaryEnumerator를 반환

Queue Class

- Queue는 **FIFO(First-In, First-Out)** 컬랙션
  - FIFO – 먼저 들어간 데이터가 제일 먼저 나오는 메모리 구조를 클래스화

```
public class Queue: ICollection, IEnumerable, ICloneable
```

- Queue 메소드
  - Enqueue() 메소드는 큐의 첫 위치에 요소를 삽입
  - Dequeue() 메소드는 큐의 마지막 위치의 요소를 반환하고 삭제 (반환되는 데이터형은 object형)
  - Peek() 메소드는 마지막 위치의 요소를 제거하지 않고 반환 (반환되는 데이터형은 object형)

Stack Class

- Stack는 **LIFO(Last-In, First-Out)** 컬랙션
  - LIFO – 제일 마지막에 들어간 데이터가 제일 먼저 나오는 메모리 구조를 클래스화

```
public class Stack: ICollection, IEnumerable, ICloneable
```

- Stack 메소드
  - Push() 메소드는 스택의 맨 위에 요소를 삽입
  - Pop() 메소드는 스택의 맨 위에 있는 요소를 삭제하고 데이터 반환 (반환되는 데이터형은 object형)
  - Peek() 메소드는 스택의 맨 위에 있는 요소를 제거하지 않고 반환 (반환되는 데이터형은 object형)

ArrayList Class

- ArrayList는 **ILIST**를 구현한 대표적인 클래스
  - ArrayList는 데이터를 삽입했을 때 순서대로 삽입되며 중간삽입이나 제거 또한 가능

```
public class ArrayList: IList, ICollection, IEnumerable, ICloneable
```

- ArrayList 메소드
  - Add(), AddRange() 메소드는 데이터/데이터 리스트 삽입
  - Insert() 메소드는 중간에 데이터 삽입
  - Remove(), RemoveAt(), RemoveRange() 메소드는 해당 요소 제거 또는 인덱스로 요소 제거 또는 범위만큼 요소 제거
  - Sort() 메소드는 요소 정렬
  - GetEnumerator() 메소드는 IEnumerator를 반환

Hashtable Class

- Hashtable은 **IDictionary**를 구현한 대표적인 클래스
  - 내부의 데이터는 키(Key)와 값(Value)을 이용

```
public class Hashtable: IDictionary, ICollection, IEnumerable,
                        ISerializable, IDeserializationCallback, ICloneable
```

- Hashtable 메소드
  - Add() 메소드는 (키, 변수)로 된 데이터 삽입
  - Clear() 메소드는 모든 요소 제거
  - Remove() 메소드는 키를 확인하여 요소 삭제
  - ContainsKey(), ContainsValue() 메소드는 특정 키/값을 포함하는지 확인
  - CopyTo() 메소드는 해쉬테이블에 있는 원소를 1차원 배열로 복사
  - Keys, Values 속성은 ICollection으로 반환
  - GetEnumerator() 메소드는 IEnumerator를 반환

Collections Interface

- IEnumerable 인터페이스
  - GetEnumerator() – IEnumerator 개체를 반환
- IEnumerator 인터페이스
  - 내부에서 IEnumerable을 사용하여 데이터 검색 기능 제공
  - Current 속성 – 컬렉션에서 현재 객체에 대한 참조를 반환
  - MoveNext() – 다음 요소로 이동
  - Reset() – Current 포인터를 컬렉션의 처음 앞으로 설정
- ICollection 인터페이스
  - Count 속성 – 컬렉션의 객체 수를 반환
  - IsSynchronized 속성 - 다중 스레드된 액세스를 위해 컬렉션에 대한 액세스를 동기화한 경우 true 반환
  - SyncRoot 속성 – 하나 이상의 코드 문장이 동시에 한 스레드에만 실행되는 것을 확실하게 하기 위해 잠그거나 해제
  - CopyTo() – 지정한 배열 위치부터 컬렉션 요소를 배열로 복사

Collections Interface

- IList 인터페이스
  - ICollection 인터페이스에서 파생된 것으로 IEnumerable과 ICollection 기능을 모두 포함
  - IsFixedSize 속성 – 리스트가 고정 길이 리스트인지 확인
  - IsReadOnly 속성 – 리스트가 읽기전용인지 확인
  - 인덱서 속성 – 인덱스 값으로 데이터를 얻거나 추가
  - Add() – 리스트 끝에 데이터를 추가
  - Clear() – 리스트 내의 모든 데이터를 제거
  - Contains() – 어떤 데이터가 리스트 내에 존재하는지 여부 확인
  - IndexOf() – 리스트 내의 특정 데이터의 위치를 반환
  - Insert() – 리스트 내의 특정 위치에 데이터를 삽입
  - Remove() – 매개변수로 입력된 객체를 리스트 내에서 제거
  - RemoveAt() – 지정한 인덱스의 데이터를 제거

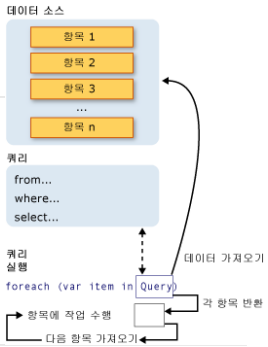
Collections Interface

- IDictionary 인터페이스
  - 순서에 의존하는 IList와 달리 키와 값으로 대응시켜 데이터를 추출
  - IsFixedSize 속성 – 컬렉션의 크기가 정해져 있는지 검사
  - IsReadOnly 속성 – 컬렉션이 읽기전용인지 확인
  - Keys 속성 – 컬렉션 내의 모든 키를 나열
  - Values 속성 – 컬렉션 내의 모든 값을 나열
  - Add() – 키와 값을 전달하여 데이터를 컬렉션에 추가
  - Clear() – 컬렉션의 모든 데이터를 제거
  - Contains() – 특정 키가 데이터와 연관되어 있는지 검사
  - GetEnumerator() – IDictionaryEnumerator 반환
  - Remove() – 삭제할 값의 키를 전달하여 데이터를 컬렉션에서 제거
- IDictionaryEnumerator 인터페이스
  - DictionaryEntry 속성 – 열거 요소 내의 키와 값을 가져옴
  - Key 속성 – 열거 요소 내의 키를 가져옴
  - Value 속성 – 열거 요소 내의 값을 가져옴

LINQ

- C# 3.5 LINQ(Language Integrated Query) 쿼리
  - 쿼리는 데이터 소스에서 데이터를 검색하는 식
  - 배열/데이터베이스에서 조건에 맞는 자료만 뽑는 기능을 제공
  - **IEnumerable<T> (instance) = from (자동변수)**
    in (배열/DB)
    where (조건)
    select (리턴값)

```
// 1. Data source
int[] numbers = new int[7] { 0, 1, 2, 3, 4, 5, 6 };
// 2. Query creation (짝수만 선택)
var numQuery = from num in numbers
               where (num % 2) == 0 select num;
// 3. Query execution
foreach (int num in numQuery)
    Console.WriteLine("{0,1}", num);
```



LINQ

- 데이터 소스 (DataSource)
  - 데이터소스는 IEnumerable<T>나 IQueryable<T> 인터페이스에서 파생된 것이면 LINQ를 사용하여 쿼리 가능한 형식임
  - 데이터소스가 쿼리 가능한 형식으로 존재하지 않는 경우 LINQ 공급자가 소스를 나타내야 함

```
// Create a data source from an XML document using System.Xml.Linq
XElement contacts = XElement.Load(@"C:₩myContactList.xml");

// LINQ to SQL
Northwnd db = new Northwnd(@"C:₩northwnd.mdf");
IQueryable<Customer> custQuery =
                    from cust in db.Customers
                    where cust.City == "London" select cust;
```

LINQ

- 필터링
  - 가장 일반적인 쿼리 작업은 boolean 형식으로 필터를 적용하는 것임
  - 결과는 **where** 절을 사용하여 생성됨
  - **&&** 및 **||** 연산자를 사용하여 where 절에 필요한 만큼의 필터 식을 적용가능
- 정렬
  - **orderby**절은 반환된 시퀀스의 요소가 정렬하고 있는 형식의 기본 비교자에 따라 정렬됨

```
var queryLondonCustomers =
                    from cust in db.Customers
                    where cust.City == "London" && cust.Name == "Devon"
                    orderby cust.Name ascending
                    select cust;
```

LINQ

- 그룹화
  - **group**절을 사용하면 지정한 키에 따라 결과를 그룹화 가능
  - 그룹화 결과를 참조해야하는 경우 **into** 키워드 사용하여 계속 쿼리할 수 있는 식별자를 만들어야 함
- 조인
  - 데이터 소스에서 명시적으로 모델링 않 된 시퀀스 간의 연결 생성
  - **join**절은 DB 테이블에 직접 작업하는 대신 개체 컬랙션에 대해 작업
- 선택 (프로젝션)
  - **select**절에서 쿼리 결과를 생성하고 각 반환된 요소의 형식 지정

```
var queryCustomers =
    from cust in db.Customers
    group cust by cust.City into custGroup
    where custGroup.Count() > 2 // 3명이상의 고객을 포함하는 그룹만 반환
    orderby custGroup.Key
    select custGroup;
```

LINQ

- 표준 쿼리 연산자 확장 메서드
  - LINQ의 선언적 쿼리 구문은 컴파일 할 때 CLR에 대한 메서드 호출으로 변환됨
  - 이러한 메서드 호출은 같은 이름을 사용하는 표준 쿼리 연산자 **Where, Select, GroupBy, Join, Max 및 Average** 메서드 구문을 직접 사용가능

```
int[] numbers = new int[7] { 0, 1, 2, 3, 4, 5, 6 };
// Query Syntax
var numQuery1 = from num in numbers
                where (num % 2) == 0 orderby num select num;
// Method Syntax
var numQuery2 = numbers.Where(num => num%2 == 0).OrderBy(n=>n);
```

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

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
  - **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
    - 등등

FILE I/O

- 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

FILE I/O

- 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 필드와 메소드 제공**
    - **DirectorySeparatorChar** – 디렉토리 구분자 캐릭터
    - **Combine** – 경로들 결합
    - **GetFileName** – 경로에서 파일이름을 얻기

FILE I/O

- FileStream 클래스
  - 파일 입출력 뿐만 아니라 파일과 관련된 다른 운영체제의 핸들 (파이프, 표준입력, 표준출력 등)을 읽고 쓰는데도 유용하게 사용
  - 바이트스트림이 아닌 **문자스트림**을 사용하기 위해서는 FileStream을 **StreamReader**와 **StreamWriter**로 변환하여 사용

```
using System.IO;
class FileTest {
    public static void Main() {
        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();
    }
}
```

FILE I/O

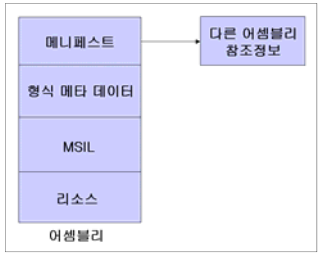
```
using System.IO;
class FileTest {
    public static void Main() {
        string path = @"C:₩Output.txt";
        FileStream fs = new FileStream(path, FileMode.Create);
        StreamWriter w = new // 한글 처리시 필요
                        StreamWriter(fs, System.Text.Encoding.Default);
        //w.BaseStream.Seek(0, 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();
    }
}
```

Assembly

- COM
  - 이미 사용 중인 검증된 코드의 재사용을 위해 COM 활용
  - 서로 다른 언어로 작성된 Binary 타입 공유를 위해 COM 서버 생성
    - Client에서 COM 서버를 호출할 때 COM 서버의 버전을 확인할 수 없음
    - COM 서버 위치나 이름이 바뀌게 되면 Registry의 변경이 쉽지 않음
- Assembly
  - 어셈블리란 .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

Assembly

- 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



Assembly

- 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



작업한 어셈블리 참조추가하기

Assembly

- 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