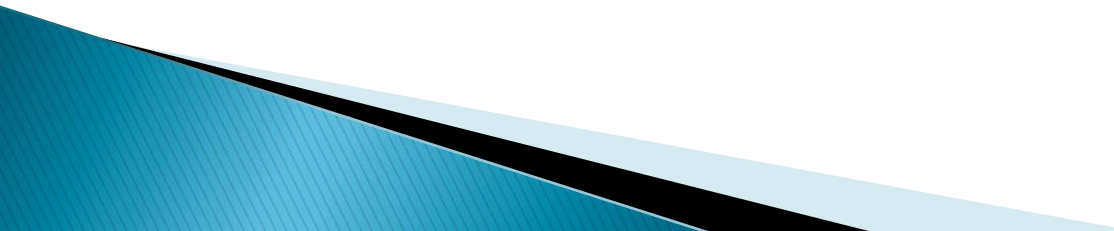


C#

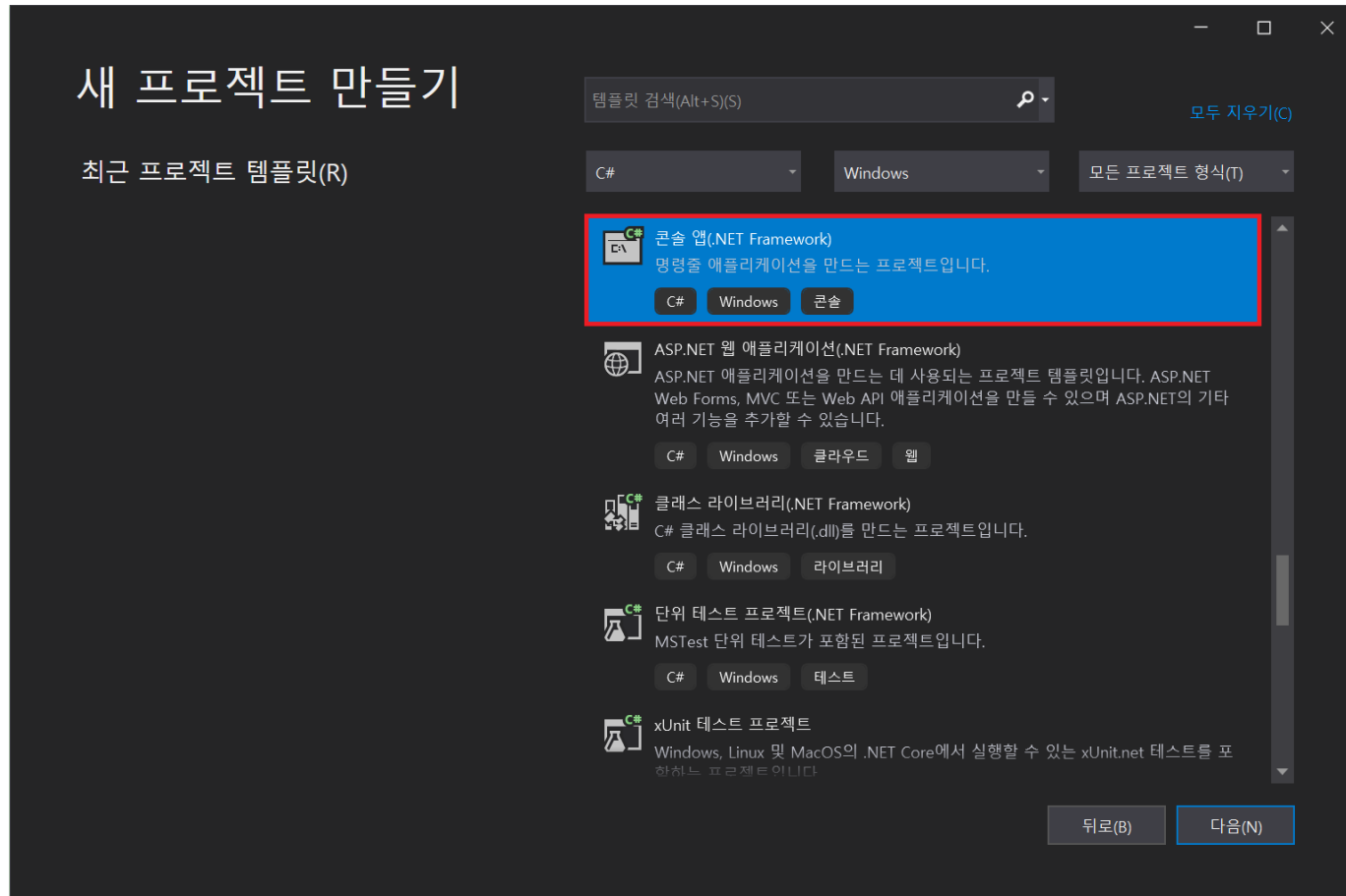
목차

1. C# 프로젝트 생성 . . . 4p
 2. 문자 출력 . . . 7p
 3. 변수 . . . 9p
 4. 반복문과 Containers . . . 14p
 5. 함수(Method) . . . 15p
- 

목차

- 6. 프로퍼티(속성) . . . 18p
 - 7. 제네릭(Generic) . . . 20p
 - 8. 클래스 . . . 22p
 - 9. Delegate, Event . . . 24p
 - 10. 람다식(Lambda Expression) . . . 26p
- 

C# 프로젝트 생성



C# 프로젝트 생성

▶ C#의 특징

- C#은 객체 지향 언어로 파일을 class로 나누며 파일명이 class명이 된다
- C/C++와 달리 헤더(*.h)와 소스(*.cpp) 파일로 나누지 않고 소스(*.cs) 파일만을 지닌다
- 전역 변수의 개념이 없으며 멤버 변수를 static을 이용하여 공용화 한다
- .Net Framework에서 지원하는 정수 형식을 사용한다
- 기본적인 문법 등은 C/C++와 크게 다르지 않다
- internal 접근 한정자 : 같은 소스 내에서는 public으로 적용되고 다른 소스에서는 private으로 적용 된다
- struct는 Value 타입이고 class는 Reference 타입이다
- class는 여러 클래스를 상속 받는 것(다중 상속)이 안되지만 인터페이스를 다중 상속 받는 것은 가능하다
- struct는 구조체 상속이 안되며 인터페이스 상속은 가능하다
- GC(Garbage Collection) : 힙(Heap) 영역의 메모리가 일정 시간 사용되지 않으면 알아서 메모리를 해제하여 준다
- GC(가비지 컬렉션)이 메모리를 해제할 때 사용되는 비용이 크기 때문에 GC 발생을 최대한 줄여주는 코딩을 하는 것이 좋다

C# 프로젝트 생성

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace 프로젝트명으로_이름이_자동으로_지정
{
    class 파일(*.cs)명으로_지정
    {
        static void Main(string[] args)
        {
        }
    }
}
```

문자 출력

▶ 문자 출력 함수

- `using System`에서 지원한다
- `Console.Write()` : 문자열을 출력하고 줄을 변경하지 않는다
- `Console.WriteLine()` : 문자열을 출력하고 줄을 변경한다
- `Console.ReadKey()` : 누른 키 정보를 알아오며 `bool` 형 매개 변수를 지닌다

```
namespace HelloWorld
{
    class Program
    {
        static void Main(string[] args)
        {
            // Hello World!! 문자 출력.
            Console.WriteLine("Hello World!!");

            // 인수는 default로 false
            // false: 누른 키를 콘솔창에 표시한다. true: 누른 키를 콘솔창에 표시하지 않는다.
            // 콘솔 창이 바로 종료되지 않기 위해 추가한 코드.
            Console.ReadKey();
        }
    }
}
```

문자 출력

▶ 형식 문자열

- C/C++의 문자 포맷(%d %s ...)과 형식이 다르다
- `String.Format("{0} {1} ... {n}", ...);`
- 형식 지정 시 **서식을 정하지 않으면** `.ToString()`의 결과 값이 들어가게 된다

```
float fValue = 1.23f;
```

```
int nValue = 123;
```

```
Console.WriteLine(String.Format("{0} | {1}", fValue, nValue));
```

```
Console.WriteLine(String.Format("{0:C}", fValue));
```

```
Console.WriteLine(String.Format("{0:D}", nValue));
```

```
Console.WriteLine(String.Format("{0:E}", fValue));
```

```
Console.WriteLine(String.Format("{0:F}", fValue));
```

```
Console.WriteLine(String.Format("{0:N}", fValue));
```

```
Console.WriteLine(String.Format("{0:P}", fValue));
```

```
Console.WriteLine(String.Format("{0:X}", nValue));
```

서식	구분	예제
{0:C} {0:c}	통화	입력 값 : 123.456 {0:C} ₩123 {0:C1} ₩123.5 {0:C3} ₩123.456
{0:D} {0:d}	10진수	입력 값 : 123456 {0:D} 123456 {0:D3} 123456 {0:D10} 0000123456
{0:E} {0:e}	지수	입력 값 : 123.456 {0:E} 1.23456E+002 {0:E3} 1.235E+002 {0:E10} 1.2345600100E+200
{0:F} {0:f}	소수점	입력 값 : 123.456 {0:F} 123.45 {0:F3} 123.456 {0:F10} 123.4560000000
{0:N} {0:n}	숫자	입력 값 : 123456.789 {0:N} 123,456.80 {0:N3} 123,456.800 {0:F10} 123,456.8000000000
{0:P} {0:p}	백분율	입력 값 : 0.123456 {0:P} 12.35% {0:P3} 12.346% {0:P10} 12.3456000000%
{0:X} {0:x}	16진수	입력 값 : 123456 {0:X} 1E240 {0:x} 1e240 {0:X8} 0001E240

변수

▶ C/C++의 차이점

- 초기화가 필수이며 쓰레기 값을 지니고 있다면 **컴파일 에러를 발생**한다
- C#에는 **포인터가 없다**, *를 간접 참조 연산자라 하여 C/C++의 포인터와 다르다
- 변수는 **Value**와 **Reference** 타입으로 나뉜다
- Value : 스택(Stack) 영역에 할당되며, **코드 블록이 끝날 때 메모리에서 해제** 된다
- Reference : 힙(Heap) 영역에 할당되어 **GC(가비지 콜렉션)**이 메모리를 **해제**하여 준다

▶ object

- 모든 데이터 타입은 **object**를 상속 받는다
- 어떤 데이터라도 받을 수 있다
- **Reference** 타입이다
- 사용 할 때마다 **박싱(Boxing)**, **언박싱(Unboxing)**이 발생하여 **많은 시간을 소모**하게 된다

변수

▶ 박싱(Boxing)과 언박싱(Unboxing)

- 데이터를 **object**로 변환되는 것을 박싱이라 하며 힙 영역에 저장된다
- **object** 타입으로 저장된 데이터를 값에 맞는 타입으로 되돌리는 것을 언박싱이라 한다
- 언박싱 과정에서 해당 **데이터 타입으로 변경이 가능한지 확인**을 위하여 **is** 연산자를 사용한다

▶ as, is 연산자

- **as** : 형 변환에 사용, 일반적인 캐스트와 다른 점은 **변환이 가능하지 않으면 null**을 반환한다는 것
따라서 **null**을 허용하는 객체에 사용이 가능하다
- **is** : 특정 객체와 호환이 가능한지 확인, 가능하면 **true**를 반환하고 아니라면 **false**를 반환한다

```
static void Main(string[] args)
{
    int a = 123;
    object b = a;    // Boxing
    if (b is int)
    {
        int c = (int)b; // Unboxing
    }
}
```

변수

▶ char

- 문자를 받으며 **배열을 이용하여 문자열을 받을 수 없다**

▶ string

- 문자열을 받으며, **Reference 타입**이다
- Split() : **지정한 문자 기준으로 나뉜 문자열의 배열을 반환**한다
- Trim() : 문자열의 **앞, 뒤 공백을 제거** 하거나 **지정한 문자를 제거하여 문자열을 반환**한다
- ToUpper(), ToLower() : 문자열을 **대문자, 소문자로 변환하여 반환**한다

▶ 문자열에 문자열 더하기

- 문자열을 **+연산자를 이용하여 더하면 GC가 발생**하기 때문에 +연산자를 이용하지 않는 것이 좋다
- **System.Text.StringBuilder를 사용하는 것이 좋다**(**using System.Text**)
- StringBuilder의 Append()가 **기본 데이터 타입들을 매개 변수로 지원**하기 때문에 **int 등의 데이터를 string으로 변환하지 않고 문자열에 더할 수 있다**

변수

▶ var

- C/C++에서의 auto처럼 사용
- **지역 변수로만 사용이 가능**

▶ 문자를 숫자로 변환

- 각 숫자 **데이터 타입**에는 Parse()와 TryParse()함수가 있어 쉽게 **문자를 데이터 타입에 맞는 숫자로 변경** 가능하다

```
int value = int.Parse("123");  
int value; if (int.TryParse("123", out value)) ...
```

▶ 숫자를 문자로 변환

- **기본으로 지원**하는 ToString()함수를 이용하여 **문자열로 변환**이 가능하다

```
int value = 123;  
value.ToString();  
또는  
(123).ToString();
```

변수

▶ 배열

- C/C++와 선언 방식이 다르다
- Length를 이용하여 배열의 크기를 알 수 있다
- Rank를 이용하여 차원 수를 알 수 있다
- GetLength() 함수로 지정한 차원의 크기를 알 수 있다

자료형[] 변수명 = new 자료형[크기];

```
int[] arr = new int[3];
```

```
int[] arr = new int[3] { 0, 1, 2 };
```

```
int[] arr = new int[] { 0, 1, 2 };
```

```
int[] arr = { 0, 1, 2 };
```

```
arr.Length; // Length:3
```

```
arr.Rank; // Rank:1
```

자료형[,] 변수명 = new 자료형[행, 열];

```
int[,] multi_arr = new int[2, 3] { { 0, 1, 2 }, { 3, 4, 5 } };
```

```
multi_arr.Length; // Length:6
```

```
multi_arr.Rank; // Rank:2
```

```
multi_arr.GetLength(0); // value:2
```

```
multi_arr.GetLength(1); // value:3
```

반복문과 Containers

▶ foreach

- 다른 반복문, (for, while)에 비하여 **느리다** (치명적이지는 않으나 되도록 쓰지 않는 것이 좋다)
- **힙(Heap) 영역의 데이터를 순회하기에 적합하다**
- **순서가 정해져 있지 않은 데이터 집합** (Dictionary, Hashtable) **순회에 자주 사용된다**
- C++에서의 범위 기반 **for**문과 같다

```
int[] a = new int[] { 0, 1, 2 };  
foreach (var value in a) { }
```

▶ System.Collections

- **C#에서 사용 가능한 STL Containers**를 지원해 준다.
- ArrayList, Hashtable, Stack, Queue...
- List는 System.Collections.Generic에서 지원한다

▶ Dictionary와 Hashtable

- C/C++에서의 **STL의 map**과 같이 사용한다
- Dictionary : key와 value 데이터 타입을 선언 시 **직접 정한다**
- Hashtable : key와 value 데이터 타입이 **object**다

함수(Method)

▶ Method

- C/C++의 Function과 같으며 언어마다 함수를 칭하는 명칭이 다르다
- C#에서는 함수를 메소드(Method)라 칭한다
- 변수와 같이 멤버 함수로 만든다

▶ 레퍼런스 매개 변수

- 레퍼런스 타입 지정은 **ref**와 **out** 키워드를 사용
- **ref** :
 - 함수 내에서 매개 변수에 값을 넣어주지 않아도 된다
 - 기존 변수를 함수 내에서 수정할 때 사용
- **out** :
 - 함수 내에서 매개 변수에 **값을 넣어주지 않으면 에러가 발생**한다
 - 함수 내에서 생성된 값을 사용할 때 사용

```
static public void Reference_Out(out int value) { value = 10; } // value = 10; 구문이 없으면 에러가 발생한다.  
static public void Reference_Ref(ref int value) { value = 20; } // value = 20; 구문이 없어도 에러가 발생하지 않는다.  
static void Main(string[] args)  
{  
    int value;  
    Reference_Out(out value);  
    Console.WriteLine(value.ToString());  
    Reference_Ref(ref value);  
    Console.WriteLine(value.ToString());  
    Console.ReadKey();  
}
```

함수(Method)

- ▶ 가변 길이 매개 변수
 - `params` 키워드로 배열을 만든다
 - 인수의 수를 가변으로 사용 가능하다

```
public int SUM(params int[] args)
{
    int result = 0;
    for (int i = 0; args.Length > i; i++)
    {
        result += args[i];
    }

    return result;
}
```

```
SUM(1, 2);
SUM(1, 2, 3);
SUM(1, 2, 3, ...);
```


수학 함수

- ▶ `Math.Pow(double x, double y)` 함수
 - `x`의 `y`승을 반환한다
 - 제곱 연산자, **C#에서는 ^연산자는 비트(XOR) 연산자**로 사용처가 다르다
- ▶ `Math.DivRem(int a, int b, out int result)` 함수
 - `a`를 `b`로 나눈 몫을 반환하고 나머지 값을 `out result`로 반환한다
 - C#에서도 C/C++에서의 %연산자를 사용 가능하며, 추가로 몫을 알기 위해서 해당 함수를 사용한다
- ▶ `Math.Round(double a)` 함수
 - 소수 값을 가장 가까운 정수 값으로 반올림
- ▶ `Math.Ceiling(double a)` 함수
 - 소수 값을 정수 값으로 올림
- ▶ `Math.Truncate(double d)` 함수
 - 소수점 아래를 버린다

프로퍼티(속성)

▶ get, set 속성 접근자

- C/C++에서는 정보 은닉을 위하여 멤버 변수를 `private`으로 선언하여 `Get()`, `Set()` 함수를 만들어 사용하는데 C#에서는 프로퍼티를 이용하여 쉽고 간단하게 구현이 가능하다
- 프로퍼티는 **멤버 변수처럼 사용하며 함수의 형태로 구현되는** C# 언어의 요소이다
- `get` : 속성 값을 반환한다
- `set` : 값을 할당 받는다, **암묵적 매개 변수** `value`를 가진다

```
public int property { get; private set; }
```

또는

```
private int m_nProperty;  
public int property { get { return m_nProperty; } }
```

또는

```
private int m_nProperty;  
public int property { set { m_nProperty = value; } }
```

또는

```
private int m_nProperty;  
public int property  
{  
    get { return m_nProperty; }  
    set  
    {  
        m_nProperty = value;  
    }  
}
```

프로퍼티(속성)

▶ Indexer(인덱서)

- 클래스나 구조체의 인스턴스를 배열처럼 인덱싱할 수 있다

```
public class PropertiesIndexer
{
    private List<int> list = new List<int>();
    private Dictionary<string, int> keyValues = new Dictionary<string, int>();

    public PropertiesIndexer()
    {
        list.Add(1);
        keyValues.Add("key", 0);
    }

    public int this[int index] { get { return list[index]; } set { list[index] = value; } }
    public int this[string key] { get { return keyValues[key]; } }
}

static void Main(string[] args)
{
    PropertiesIndexer arrayProperties = new PropertiesIndexer();
    Console.WriteLine(arrayProperties[0]);
    Console.WriteLine(arrayProperties["key"]);
    arrayProperties[0] = 10;
    Console.WriteLine(arrayProperties[0]);
    Console.ReadKey();
}
```

제네릭(Generic)

▶ 일반화(Generic) 프로그래밍

- C/C++의 **template**처럼 여러 데이터 타입을 받을 수 있는 함수나 클래스 등을 만드는 것을 말한다

접근한정자 `class` 클래스명<타입명>{}

접근한정자 반환타입 함수명<타입명>(타입명 매개변수명){}

▶ 타입 제약

- 타입에 **조건을 걸어 해당 형식의 타입만을 사용**하도록 한다

...<타입명> **where** 타입명 : 제약조건

...<타입명>(타입명 매개변수명) **where** 타입명 : 제약조건

제네릭(Generic)

제약 조건	설명
where T : struct	값(Value) 형식
where T : class	참조(Reference) 형식
where T : new()	함수에 매개 변수가 없고 new T()가 있어야 한다
where T : base_class_name	지정된 기본 클래스 또는 기본 클래스에서 파생된 클래스 형식
where T : base_interface	지정된 인터페이스 또는 지정된 인터페이스를 구현
where T : U	U : 또 다른 제네릭 타입 / U에 대해 제공하는 인수 또는 U에서 파생

```
class MyClass<T> where T : struct
MyClass<int> myClass = new MyClass<int>();
```

```
class MyClass<T> where T : class
MyClass<string> myClass = new MyClass<string>();
```

```
class MyClass<T> where T : new()
{
    public T MyMethod()
    {
        return new T();
    }
}
```

```
class MyClass<U>
{
    public void MyMethod<T>(T value) where T : U { }
}
```

```
class MyClass<T> where T : System.Enum
```

클래스

▶ 객체 복사

- C#에서 **class**는 참조 타입이기 때문에 단순 대입으로는 값을 복사한 새로운 객체를 만드는 것이 아닌 **참조 형식의 얕은 복사**를 하게 된다
- **깊은 복사를 위한 구현이 필요**하며, .Net Framework의 유틸리티 클래스나 다른 프로그래머가 작성한 코드와 호환되도록 하기 위해서는 **ICloneable** 인터페이스를 상속하도록 하여 구현 한다

// 얕은 복사.

```
class MyClass
```

```
{
```

```
    public int value;
```

```
}
```

```
MyClass a = new MyClass();
```

```
a.value = 1;
```

```
MyClass b = a;
```

```
b.value = 2; // a.value = 2가 된다.
```

// 깊은 복사.

```
class MyClass
```

```
{
```

```
    public int value;
```

```
    public MyClass Clone()
```

```
    {
```

```
        MyClass temp = new MyClass();
```

```
        temp.value = this.value;
```

```
        return temp;
```

```
    }
```

```
}
```

```
MyClass a = new MyClass();
```

```
a.value = 1;
```

```
MyClass b = a.Clone();
```

```
b.value = 2; // b.value = 2, a.value = 1
```

클래스

// ICloneable 인터페이스 상속.

```
class MyClass : ICloneable
{
    public int value;

    public object Clone()
    {
        MyClass temp = new MyClass();
        temp.value = this.value;
        return temp;
    }
}
```

```
MyClass a = new MyClass();
a.value = 1;
MyClass b = a.Clone() as MyClass;
// MyClass b = (MyClass)a.Clone();
b.value = 2; // b.value = 2, a.value = 1
```

Delegate, Event

▶ delegate(델리게이트)

- C/C++의 함수 포인터처럼 사용하며 메소드에 대한 참조이다
- 인스턴스 메소드, 정적 메소드 모두 참조 가능하다
- 델리게이트 체인(delegate chain)으로 **하나의 델리게이트에 여러 메소드를 등록(+=), 해제(-=)**할 수 있다
- 이름을 제외하여 함수(익명 메소드)를 구현할 수 있다
- `Action<T...>` : 반환 타입이 `void`이고 매개 변수 타입 `T`를 지정할 수 있는 미리 정의 된 delegate
- `Func<T..., R>` : 반환 타입 `R`과 매개 변수 타입 `T`를 지정할 수 있는 미리 정의 된 delegate
- ❖ Action과 Func은 `using System;` 해야 사용 가능

접근한정자 `delegate` 반환형식 델리게이트명(매개변수_목록);

```
public delegate void WriteWord();
static void Print1() { Console.WriteLine("Print1!!"); }
static void Print2() { Console.WriteLine("Print2!!"); }
static void Print3() { Console.WriteLine("Print3!!"); }
WriteWord writeWord = Print1;
writeWord += Print2;
writeWord += Print3;
writeWord(); // Print1!! Print2!! Print3!!
writeWord -= Print1;
writeWord -= Print3;
writeWord(); // Print2!!
writeWord += delegate () { Console.WriteLine("Print4!!"); }; // 익명(무명) 메소드.
writeWord(); // Print2!! Print4!!
```


Delegate, Event

▶ event

- `delegate`와 사용 방법은 크게 차이는 없다
- 델리게이트에 `event`를 수식해서 선언한 것에 불과한 형태지만 델리게이트와 달리 **이벤트를 외부에서 직접적으로 호출이 불가능**하다
- 이벤트가 **특정한 상황에서만 발생해야 할 경우** 사용하여 이벤트 발생 처리에 대한 신뢰성을 높인다

접근한정자 `event` 델리게이트명 이벤트명;

```
public delegate void EventHandler(string msg);
class Notifier
{
    public event EventHandler OnEvent;
    public void DoSomething(int value)
    {
        if (0 > value) { OnEvent(String.Format("이벤트 발생!! vlaue:{0}", value)); }
    }
}
static void MyHandler(string msg) { Console.WriteLine(msg); }
static void Main(string[] args)
{
    Notifier notifier = new Notifier();
    notifier.OnEvent += MyHandler;
    for (int i = 5; -5 < i; i--)
    {
        notifier.DoSomething(i);
        //notifier.OnEvent("") 호출 불가.
    }
}
```

람다식(Lambda Expression)

▶ 람다식

- 익명(무명) 메소드와 비슷한 형태를 지닌다
- => 연산자 : C# 3.0부터 지원을 하는 연산자로 람다식을 표현할 때 사용된다
- `delegate` 또는 익명(무명) 메소드 보다 더 간략하게 표현할 수 있다
- 형식 : (입력_파라미터) => { 실행_문장_블럭 };

// 입력 파라미터가 없는 경우.

```
() => Console.WriteLine("Hello World!!");
```

// 입력 파라미터가 1개 이상.

```
(a, b) => { ... };
```

// 입력 파라미터 타입을 명시.

```
(string a, int b) => { ... };
```

```
Action<int> Act = (value) => { Console.WriteLine(value); };
```

```
Act(1);
```