C# vs Java

C-Sharp, Java

Java 개발자를 위한 C#!

- Java 개발자가 당장 C# 개발을 할 수 있도록 작성된 문서입니다.
- 모든 글은 "Java" 관점에서 서술합니다. 기본 문법 지식이 부족하면 이해하기 힘들 수 있습니다.
- 직접적인 **개발 적용**을 목적으로 하기 때문에 더 자세한 사항을 알고 싶다면 여기를 참고하세요.

출력문 Parameter 사용법 (string.Format)

- https://msdn.microsoft.com/ko-kr/library/s8s7t687.aspx

```
//%d, %s  같이 지정할 필요없이 자동으로 형변환되어 출력 가능.
Console.WriteLine("{0} {1} {2}", 77, 3.141592, "Test");

//%05d, %.10f와 같은 표현
Console.WriteLine("{0:D5} {1:F10}", 77, 3.141592);
```

Data Type

Primitive/Reference Type 상관없이 모두 **소문자**로 시작!

Primitive Types

- C#은 모든 Type이 객체!
- Java는 모든 Data Type이 부호 있는(Signed) 수이다. 그에 반해 C#은 부호 없는(unsigned) 수를 지원한다!
- byte는 signed를 지원한다.

Data Type	Size	Range
sbyte	1byte	-128 ~ 127
ushort	2byte	0 ~ 65535
uint	4byte	0 ~ 4294967295
ulong	2byte	0 ~ 18446744073709551615

- Java와 마찬가지로 byte(1byte)를 지원하고, 공통적으로 char는 2byte이다.
- 큰 수를 다루는 Type에 차이가 있다. (Java는 BigDecimal(32byte), BigInteger(56bytes)가 존재)

Data Type	Size	Range
Decimal	16byte	±1.0*1-e=28 ~ ±7.9*10e28

수↔문자열 메소드

Java	C#
Integer.parseInt(“”);	int.Parse(“”);
s.toString();	s.ToString();

상수

- Java는 **final**, C#은 **const** (C를 계승하는 듯.)

Nullable 변수

- Primitive Type 데이터에 아무 값도 가지지 않는 변수가 필요할 때 (선언만 한 것과는 다르다!)
- **Primetive Type**뒤에 **?**를 붙인다.

```
int? a = null;
int b;
int? c;

Console.WriteLine(a);
//Console.writeLine(b); /* Error! */
//Console.writeLine(c); /* Error! */
```

var : Dynamic Type

- Javascript와 같은 “약한 Type 검사” 지원. 자동으로 컴파일러가 자동으로 데이터의 해당 형 변환.
- object와는 다르다.

enum

int to Enum

- (MYENUM) Enum.ToObject(typeof(MYENUM), INTVALUE);
- ((MYENUM[]) Enum.GetValues(typeof(MYENUM)))[INTVAULE];

Method 활용

기본적으로 Java와 동일하게 Method의 Argument로 전달되는 값은 Call by value이다! 단, C#은 키워드를 통해 Call by reference를 이용할 수 있다!

여러 개의 반환 값이 필요할 때

- 아래 두 방법 모두 Parameter로 넘긴 값을 반환 받지 않고도 저장 가능하다!

Call by Reference

- 넘겨주는 쪽, 받는 쪽 모두 앞에 "**ref**" 키워드를 붙인다!

```
static void Main(string[] args)
{
    int a = 10, b = 20;

    Swap(ref a, ref b);
    Console.WriteLine(a+" "+b);
}

static void Swap(ref int a, ref int b)
{
    int temp = a;
    a = b;
    b = temp;
}
```

출력 전용 Parameter

- "Call by Reference"의 "**ref**" 키워드를 **"out"**으로 바꿔주면 된다.
- Write Only ⇒ 반드시 값을 저장해주어야 한다! ⇒ **안전하다!**

```
static void Divide(int x, int y, out int quotient, out int remainder)
{
    quotient = x / y;
    remainder = x % y;
}

static void Main(string[] args)
{
    int x = 5;
    int y = 3;

    int q, r;
    Divide(x, y, out q, out r);
    Console.WriteLine(q+" "+r);
}
```

Method 이름은 같고 Argument 개수가 다를 때

Method Overloading

- Type이 다르거나 Argument 개수가 **유한**적으로 다를 때 사용. (Java와 동일)

Variable Length Arguments (가변 길이 매개변수)

- Type은 같으나 Argument 개수가 가변적인 경우 사용.
- Method argument에 **params TYPE[] args** 이용.

```
static void Main(string[] args)
{
    int result = Sum(1, 2, 3, 4, 5);
    Console.WriteLine(result);
}

static int Sum(params int[] args)
{
    int sum = 0;
    foreach (int n in args)
        sum += n;

    return sum;
}
```

Argument 자동 매칭 및 초기화

명명된 Arguments

- **순서 상관 없이** Argument **이름**에 근거해서 데이터를 할당하는 기능

```
static void Main(string[] args)
{
    PrintProfile(age : 33, name : "James");
}

static void PrintProfile(string name, int age)
{
    Console.WriteLine(name+" / "+age);
}
```

선택적 Arguments

- Argument에 Default값 지정으로 초기화 가능.

- 호출 시 Default값을 가진 Argument는 생략 가능.
- **항상 필수 매개 변수 뒤에 와야 한다!**

Class 활용

- 상속, Overriding, this() 생성자 이용은 "**:**"를 이용한다. (C++ 방식)
- 상속, 방지는 "**sealed**". (= Java에서 **final**과 같음.)

this() 생성자

- 여러 생성자 안에 중복되는 코드 간략화

```
int a, b, c;
public MyClass()
{
    this.a = 777;
}

public MyClass(int b) : this()
{
    this.b = b;
}

public MyClass(int b, int c) : this(b)
{
    this.c = c;
}
```

접근 한정자

- public, protected, private는 Java와 동일.
- Java와 달리 접근 한정자 수식 안하면 **private**
- internal, protected internal은
  다른 어셈블리에 있는 코드에서는 private와 같은 수준의 접근성을 가지고, 같은 어셈블리(한 프로젝트에서 뽑아내는 결과물)에 있는 코드에 대해서만
  - internal : **private**로 접근 가능. 다른 어셈블리는 접근 불가.
  - protected internal : **protected**로 접근 가능. 상속 관계라면 다른 어셈블리도 접근 가능.

객체 상속 Type 검사

- is : bool 반환
- as : 성공하면 해당 객체를, 실패하면 null 반환. (당연하지만 Reference Type에만 사용 가능.)

```
Mammal mammal = new Dog();
Dog dog;

if(mammal is Dog)
{
    dog = (Dog) mammal;
}
```

Method Overriding

- Java와 달리 한정자를 붙여야 되는 추가작업 필요.
  - Overriding 될 부모 Class의 Method에 **virtual** 키워드 수식
  - Overriding 할 자식 Class의 Method에 **override** 키워드 수식

Method 숨기기

- 상속 관계에서 부모 클래스의 메소드를 감추고, 자식 클래스의 구현된 메소드만을 보여 주는 것.
- 자식 클래스에 **new** 키워드 수식

```
//다형성과 관계 있는 모습이 보인다.

//Base가 부모, Derived가 자식 관계일 때
//public new void MyMethod() {}
Derived d = new Derived();
d.MyMethod(); //자식

Base b = new Derived();
b.MyMethod();
```

Paritial Class (분할 클래스)

- 클래스 구현이 길어질 경우 사용.
- **클래스 이름**만 **동일**하게 구성 후 기술, **partitial** 키워드를 이용한다.

Extension Method (확장 메소드)

- C#에서 제공되는 기존 클래스의 기능을 확장하는 기법.
- static 클래스, static 메소드, 첫 번째 Parameter에 this를 지정하고, 다른 클래스에서 사용시 “using NAMESPACE”로 선언 후 접근한다.

```
public static class CLASSNAME
{
    public static TYPE METHODNAME(this TYPE PARAM_NAME /*확장할 Class or
```

```
Type*/, PARAM, ...)
    {
        //
    }
}
```

```
using System;
using MyExtension;

namespace MyExtension
{
    public static class IntegerExtension
    {
        public static int Square(this int myInt)
        {
            return myInt * myInt;
        }

        public static int Power(this int myInt, int exponent)
        {
            int result = myInt;
            for (int i = 0; i < exponent; i++)
            {
                result *= myInt;
            }

            return result;
        }
    }
}

namespace ExtensionMethod
{
    class MainApp
    {
        public static void Main(string[] args)
        {
            Console.WriteLine(3.Square());
            Console.WriteLine(3.Power(3));

        }
    }
}
```

Struct (구조체)

- 데이터를 담기 위한 자료구조.

- 클래스와 달리 **값 형식**으로 **깊은 복사**가 된다.
- 선언만으로도 인스턴스 생성됨.
- struct 키워드 이용.

Interface 활용

- “파생(자식) 클래스가 어떤 메소드를 구현해야하는가?”를 정의.
- C#에서 네이밍 규칙으로 “I”로 시작한다.
- Method, **Event**, **Indexer**, **Property**만을 가질 수 있다.
- “**:**“를 Interface나 Class 이름 옆에 붙여 사용.

Interface를 상속하는 Interface

- 아래의 경우 이용한다.
  1. 상속하려는 Interface가 소스 코드가 아닌 **Assembly로만 제공**되는 경우
  2. 상속하려는 Interface가 소스 코드를 갖고 있어도 **이미 Interface를 상속하는 Class들이 존재**하는 경우

Abstract Class (추상 클래스)

- Interface와 Class 사이의 개념
- 구현을 갖되 **Instance**를 갖지 못한다.

Generic Programming (일반화 프로그래밍)

- Logic이 같은 데, **Type**만 다를 때 이용.

Parameter의 Type 제약시키기

```
GENERIC_METHOD/CLASS where T : CONSTRAINT

/* CONSTRAINT */
/*
  struct : 값 형식
  class : 참조 형식
  new() : Parameter가 없는 생성자
 기반_클래스_이름 : 기반 클래스의 파생 클래스
 인터페이스_이름 : 인터페이스 구현 명시. (여러개 명시 가능)
  U : 또 다른 Parameter "U"로부터 상속받은 클래스
*/
```

공통 기능이지만 다른 문법

"Hello, World!"로 살펴보는 전체적 구조

- C#

```
namespace HelloWorld {
    class MainApp()    {
        public static void Main() {
            Console.WriteLine("Hello World!");
        }
    }
}
```

- Java

```
class MainApp() {
    public static void main() {
        System.out.println("Hello World!");
    }
}
```

C#의 소스는 Class가 최상위 개념이 아니다!

- C#은 Class 상위에 이들을 묶는 **namespace**라는 개념이 있다! (= namespace는 class의 집합체)
  = C#의 기본 프로그램 단위는 Class가 아닌 **namespace**!
  = Class(혹은 namespace) 이름과 Source file 이름이 같을 필요가 없다!

C#의 모든 Method 첫자는 "대문자"로 시작한다!

- Main() 쓸 때 헤깔린다.
- 특히 자동 완성(Intelligence) 이용할 때, 대소문자 구별안하고 타이핑하면 작동이 안된다! ~~(이럴 때는 Eclipse가 그립다…)~~

배열 사용법

- C#은 "[]"을 "Type"에만 붙이는 것을 허용!
- 다차원 구분은 ",로 구분한다.

```
//int arr[] = new int[10]; /* Error! Java는 가능!*/
int[] arr = new int[10]; /* 이것만 가능!*/
```

```
//int[][] arr = new int[2][3]; /* Java */
int[,] = new int[2,3];
```

가변 배열 (Jagged Array)

- 배열이 요소인 배열

```
int[][] arr1 = new int[3][];
arr1[0] = new int[]{ 1, 2, 3, 4, 5 };
arr1[1] = new int[]{ 10, 20, 30 };
arr1[3] = new int[]{ 400, 500 };

int[][] arr2 = new int[3][] { new int[]{ 1, 2, 3, 4, 5 }, new int[]{ 10, 20,
30 }, new int[]{ 400, 500 } };
```

foreach

```
int[] arr = {1,2,3};
for(int n : arr)
    System.out.println(n);
```

```
int[] arr = {1,2,3};
foreach(int n in arr)
    Console.WriteLine(n);
```

Random

- C#은 seed 필요!

Lambda

- Java는 **jdk1.8** 부터 사용 가능하다.
- C#은 **3.0**부터 사용 가능하다.

C#

- Anonymous Method(익명 메소드)를 만들기 위해 사용한다.(Delegate를 이용할 때보다 훨씬 간단해진다!)
- 람다식으로 만들어진 익명 메소드는 **Anonymous Function(무명 함수)**라고 한다.

Expression(식) Lambda

```
(PARAMETERS...) => EXPRESSION;
```

```
using System;

namespace ConsoleApplication1
{
    class Program
    {
        delegate int Calc(int a, int b);

        static void Main(string[] args)
        {
            Calc calc = (a, b) => a + b; //자동으로 Parameter Types 판단.
            /* delegate를 이용하면 아래와 같은 표현이다.*/
            /*
            Calc calc = delegate(int a, int b)
                        {
                            return a + b;
                        };
            */

            Console.WriteLine(calc(5,6));
        }
    }
}
```

Statement(문) Lambda

```
(PARAMETERS...) => {
            STATEMENTS;
            ...;
           };
```

```
using System;

namespace ConsoleApplication1
{
    class Program
    {
```

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

        static void Main(string[] args)
        {
            BigValue val =
                (a, b) =>
                {
                    return a > b ? a : b;
                };

            Console.WriteLine(val(10,20));
        }
    }
}
```

Func와 Action을 이용한 더 간편한 Anonymous Function 만들기

- **별개의 Delegate를 생성하지 않고,** Anonymous Method/Function를 만들 수 있다.

Func Delegate

- **반환값이 있는 경우** 사용한다.
- 기본적으로 1개의 Generic Type으로 Return Type을 갖고, 여러개일 경우 맨 마지막에 표현된다.

```
using System;

namespace ConsoleApplication1
using System;

namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
            /* Parameter 없이 반환값만 있는 경우 */
            Func<string> func1 = () => "Hello World";
            Console.WriteLine(func1());


            /* Parameter 및 반환값이 있는 경우 */
            Func<int, int, int> func2 =
                (a, b) =>
                {
                    int result = a > b? a : b;
                    return result;
```

```
                };
            Console.WriteLine(func2(10,20));
        }
    }
}
```

Action Delegate

- **반환값이 없는** 경우 사용한다.
- 기본적으로 어떤 Generic Type도 갖지 않는다. (반환값이 없기 때문.)

```
using System;

namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
            /* Parameter 없는 경우 */
            Action act1 = () => Console.WriteLine("Hello World");
            act1();


            /* Parameter가 있는 경우 */
            Action<int, int> act2 =
                (a, b) =>
                {
                    int result = a > b? a : b;
                    Console.WriteLine(result+"가 더 크다!");
                };
            act2(10,20);
        }
    }
}
```

Expression Tree (식 트리)

- Code를 **Data**로 보관 가능.
  - 파일에 저장
  - Network를 통해 다른 Process에 전달
  - Database Server에 보내어 실행 ⇒ LINQ

Data Structure (자료구조)

ArrayList	LinkedList, List
HashTable	**Dictionary**
Type 명시 불필요 (= object) (System.Collections)	Type 명시 필요 (System.Collections.Generic)

C#에만 있는 문법

Property

- 은닉성을 지키면서 getter/setter를 축약적으로 편리하게 이용할 수 있는 기능.

```
class 클래스이름
{
    TYPE FIELD이름;
  접근한정자 TYPE PROPERTY이름
    {
        get
        {
            return FIELD이름;
        }

        /* set 접근자를 구현하지 않으면 읽기 전용 */
        set
        {
            FIELD이름 = value; //value는 암묵적 매개 변수
        }
    }
}
```

자동 구현 Property

- get, set 구현 없이 그냥 “get; set;”으로 대체하면 된다. ⇒ 단순 반복 노동 감소!
- 3.0부터 사용 가능.

```
using System;

namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
            MyName mn = new MyName();
            mn.Name1 = "James";
            Console.WriteLine(mn.Name1);
```

```
            mn.Name2 = "Luke";
            Console.WriteLine(mn.Name2);
        }
    }

    class MyName
    {
   //기본 Property 사용
        private string _name;
        public string Name1
        {
            get
            {
                return _name;
            }

            set
            {
                _name = value;
            }
        }


   //자동 구현 Property 사용
        public string Name2
        {
            get;
            set;
        }
    }
}
```

Peroperty & Constructor (프로퍼티와 생성자)

- 프로퍼티의 생성자를 이용하여 Default 값 지정.

```
using System;

namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
            MyProfile mp = new MyProfile()
            {
                Name = "James",
                Age = 77
```

```
            };

            Console.WriteLine("{0} {1}", mp.Name, mp.Age);
        }
    }

    class MyProfile
    {
        public string Name
        {
            get;
            set;
        }

        public int Age
        {
            get;
            set;
        }
    }
}
```

Anonymous Type (무명 형식)

- 한 번 사용하고 다시는 사용하지 않을 인스턴스가 필요할 때 이용.
- 선언과 동시에 인스턴스 할당하고, 프로퍼티에 접근한다.
- **읽기 전용** (변경 불가)

```
using System;

namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
            var x = new { Subject = "Math", Scores = new int[] {77,30,94,50}
};

            Console.WriteLine(x.Subject);
            foreach (var scores in x.Scores)
            {
                Console.Write(scores+" ");
            }
        }
    }
}
```

Property of Interface (인터페이스의 프로퍼티)

- 자동 구현 프로퍼티와 동일한 형태
- 당연한 얘기지만 이를 상속한 클래스는 반드시 모든 프로퍼티를 구현해야 한다.
    - 자동 구현 프로퍼티를 이용하는 것도 가능하다.

Property of Abstract Class (추상 클래스의 프로퍼티)

```
using System;

namespace ConsoleApplication1
{
    abstract class Product
    {
        private static int serial = 0;
        public string SerialID
        {
            get
            {
                return String.Format("{0:d5}", serial++);
            }
        }

        abstract public DateTime ProductDate
        {
            get;
            set;
        }
    }

    class MyProduct : Product
    {

        public override DateTime ProductDate
        {
            get;
            set;
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            Product p = new MyProduct()
            {
                ProductDate = DateTime.Now
```

```
            };

            Console.WriteLine(p.SerialID + "/" + p.ProductDate);
            Console.WriteLine(p.SerialID + "/" + p.ProductDate);
        }
    }
}
```

Source 비교

Property 이용X	Property 이용O

using System;
 	
namespace ConsoleApplication1
{
class Program
{
static void Main(string[] args)
{
MyProfile mp = new MyProfile();
mp.SetName("James");
mp.SetAge(77);
 	
Console.WriteLine(mp.GetName()+"/"+mp.GetAge());
}
}
 	
class MyProfile
{
private string name;
private int age;
 	
public string GetName()
{
return name;
}
 	
public void SetName(string value)
{
name = value;
}
 	
public int GetAge()
{
return age;
}
 	
public void SetAge(int value)
{
age = value;
}
}
}
	using System;
	
	namespace ConsoleApplication1
	{
	class Program
	{
	static void Main(string[] args)
	{
	MyProfile mp = new MyProfile
	{
	Name = "James",
	Age = 77
	};
	
	Console.WriteLine(mp.Name+"/"+mp.Age);
	}
	}
	
	class MyProfile
	{
	public string Name
	{
	get;
	set;
	}
	
	public int Age
	{
	get;
	set;
	}
	}
	}

Indexer

- Property처럼 객체 내의 Data에 접근 할 수 있도록 하는 통로.

다른점이라면 **Index**를 이용한다.

- 즉, **배열** Data의 getter/setter가 필요할 때 사용!

```
using System;

namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
            MyList list = new MyList();

            for (int i = 0; i < 5; i++)
                list[i] = i;

            for (int i = 0; i < list.Length; i++ )
                Console.WriteLine(i);

            Console.WriteLine("index 3 : "+list[3]);
        }
    }

    class MyList
    {
        private int[] arr;

        public MyList()
        {
            arr = new int[3];
        }

        public int this[int idx]
        {
            get
            {
                return arr[idx];
            }

            set
            {
                //Resize
                if (idx >= arr.Length)
                {
                    Array.Resize<int>(ref arr, idx + 1);
                    Console.WriteLine("Resized : " + arr.Length);
                }

                arr[idx] = value;
```

```
            }
        }

        public int Length
        {
            get
            {
                return arr.Length;
            }
        }
    }
}
```

foreach가 가능한 객체 만들기

- 기본 예제로는 foreach가 사용 불가능!
- **IEnumerable**과 **Ienumerator**를 상속해야 한다!

```
using System;
using System.Collections;

namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
            MyList list = new MyList();

            for (int i = 0; i < 5; i++)
                list[i] = i;

            foreach (int e in list)
                Console.WriteLine(e);

            Console.WriteLine("index 3 : "+list[3]);
        }
    }

    class MyList : IEnumerable, IEnumerator
    {
        private int[] arr;
        int position = -1; //MoveNext() 때문!

        public MyList()
        {
            arr = new int[3];
        }
```

```
        public int this[int idx]
        {
            get
            {
                return arr[idx];
            }

            set
            {
                //Resize
                if (idx >= arr.Length)
                {
                    Array.Resize<int>(ref arr, idx + 1);
                    Console.WriteLine("Resized : " + arr.Length);
                }

                arr[idx] = value;
            }
        }

        public int Length
        {
            get
            {
                return arr.Length;
            }
        }


    /* 아래 3개의 Method는 IEnumerator의 Member */
        public object Current
        {
            get
            {
                return arr[position];
            }
        }

        public bool MoveNext()
        {
            if (position == arr.Length - 1)
            {
                Reset();
                return false;
            }

            position++;
            return position < arr.Length;
        }

        public void Reset()
```

```
        {
            position = -1;
        }


    /* 아래 Method는 IEnumerable의 Member */
         public IEnumerator GetEnumerator()
         {
             /* "yield return"은 arr[i]를 반환하고, 그 중단된 위치에서 반복을 이어간다. */
             for (int i = 0; i < arr.Length; i++ )
                 yield return arr[i];
         }
     }
}
```

Delegate

- Method 자체를 Parameter로 넘기고 싶을 때 사용.
  - Method에 의한 참조 ⇒ Delegate에 Method 주소 할당 후 이를 호출하면 그 Method 호출 됨. (= Method 자체가 Parameter)
  - Parameter에는 Method **이름**만이 들어간다.
- Delegate는 **Type**이다.
  = Method 참조를 만들려면 Delegate의 Instance를 생성해야 한다.
- Callback을 구현하기 위해 사용. (중간의 대리/비서 역할)

```
ACCESS_MODIFIER delegate TYPE NAME(Param, ...);
```

예제

```
/* 결과를 출력한다는 공통된 기능을 갖고, 연산만 달라진다는 시나리오. */

using System;

namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
            Calculator calc = new Calculator();
            del_calc calc_plus = new del_calc(calc.Plus);
            del_calc calc_minus = new del_calc(calc.Minus);

            Print(calc_plus, 1, 2);
            Print(calc_minus, 10, 20);
```

```
        }


        delegate int del_calc(int a, int b);

        static void Print(del_calc c, int a, int b)
        {
            int result = c(a, b);
            Console.WriteLine(result);
        }
    }

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

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

```
using System;

namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
     //오름차순 버블 정렬
            int[] arr1 = { 3, 5, 1, 0, 8 };
            BubbleSort(arr1, new Compare(AscCompare));
            foreach (int i in arr1)
                Console.Write(i + " ");

            Console.WriteLine();

     //내림차순 버블 정렬
            int[] arr2 = { 4, 99, 32, 45, 0 };
            BubbleSort(arr2, new Compare(DescCompare));
            foreach (int i in arr2)
                Console.Write(i + " ");
        }
```

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

        static int AscCompare(int a, int b)
        {
            if (a > b)
                return 1;
            else if (a == b)
                return 0;
            else
                return -1;
        }

        static int DescCompare(int a, int b)
        {
            if (a < b)
                return 1;
            else if (a == b)
                return 0;
            else
                return -1;
        }

        static void BubbleSort(int[] arr, Compare cpr)
        {
            int i = 0;
            int j = 0;
            int temp = 0;

            for (i = 0; i < arr.Length - 1; i++ )
            {
                for (j = 0; j < arr.Length - (i + 1); j++)
                {
                    if (cpr(arr[j], arr[j + 1]) > 0)
                    {
                        temp = arr[j + 1];
                        arr[j + 1] = arr[j];
                        arr[j] = temp;
                    }
                }
            }
        }
    }
}
```

Delegate Chain

- 하나의 Delegate로 여러개의 Method를 연달아 호출해야 할 때 사용.
- Chain 생성 : "**+=**" 연산자를 이용하거나 "**Delegate.Combine()**"을 이용한다.
- Chain 제거 : "**-=**" 연산자를 이용하거나 "**Delegate.Remove()**"을 이용한다.

예제

```
using System;

namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
            MyDelegate dele1 = null;
            dele1 += func1;
            dele1 += func2;
            dele1 += func3;
            dele1 -= func2;

            dele1();


            Console.WriteLine();


            MyDelegate dele2_1 = new MyDelegate(func1);
            MyDelegate dele2_2 = new MyDelegate(func2);
            MyDelegate dele2_3 = new MyDelegate(func3);

            MyDelegate dele2_all = null;
            dele2_all = (MyDelegate)Delegate.Combine(dele2_1, dele2_2,
dele2_3);
            dele2_all = (MyDelegate)Delegate.Remove(dele2_all, dele2_2);

            dele2_all();
        }

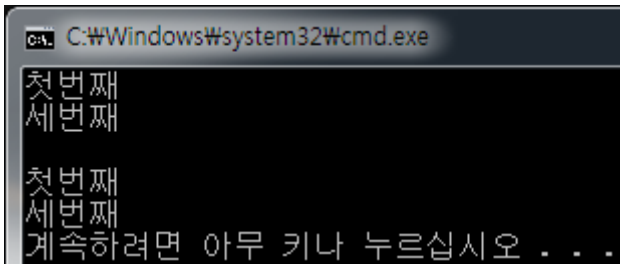

        delegate void MyDelegate();

        public static void func1()
        {
            Console.WriteLine("첫번째");
        }

        public static void func2()
        {
            Console.WriteLine("두번째");
        }

        public static void func3()
```

```
        {
            Console.WriteLine("세번째");
        }
    }
}
```




```
using System;

namespace ConsoleApplication1
{
    class Program
    {
        Notifier notifier = new Notifier();
        EventListener listener1 = new EventListener("listener1");
        EventListener listener2 = new EventListener("listener2");
        EventListener listener3 = new EventListener("listener3");

        static void Main(string[] args)
        {
            Program program = new Program();

            program.Example1();
            program.Example2();
            program.Example3();
        }

        public void Example1()
        {
            Console.WriteLine("*** Example1 ***");

            notifier.EventOccured += listener1.SomethingHappend;
            notifier.EventOccured += listener2.SomethingHappend;
            notifier.EventOccured += listener3.SomethingHappend;
            notifier.EventOccured("You've got mail.");

            Console.WriteLine();

    //끊기
            notifier.EventOccured -= listener2.SomethingHappend;
```

```
            notifier.EventOccured("Download Complete.");

            Console.WriteLine();
        }

        public void Example2()
        {
            Console.WriteLine("*** Example2 ***");

            notifier.EventOccured = new
NotifyDelgate(listener2.SomethingHappend) + new
NotifyDelgate(listener3.SomethingHappend);
            notifier.EventOccured("Nuclear launch detected.");

            Console.WriteLine();
        }

        public void Example3()
        {
            Console.WriteLine("*** Example3 ***");

            NotifyDelgate noti1 = new
NotifyDelgate(listener1.SomethingHappend);
            NotifyDelgate noti2 = new
NotifyDelgate(listener2.SomethingHappend);

            notifier.EventOccured = (NotifyDelgate) Delegate.Combine(noti1,
noti2);
            notifier.EventOccured("Fire!");

            Console.WriteLine();

    //끊기
            notifier.EventOccured = (NotifyDelgate)
Delegate.Remove(notifier.EventOccured, noti2); //Delegate source, Delegate
value
            notifier.EventOccured("OK");
        }
    }

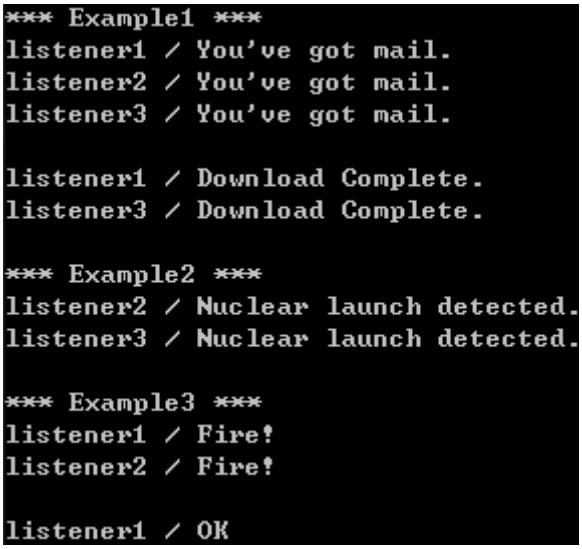

    delegate void NotifyDelgate(string message);

    class Notifier
    {
        public NotifyDelgate EventOccured;
    }

    class EventListener
    {
        private string name;
```

```
        public EventListener(string name)
        {
            this.name = name;
        }

        public void SomethingHappend(string message)
        {
            Console.WriteLine(name+" / "+message);
        }
    }
}
```



Anonymous Method (익명 메소드)

- **delegate** 키워드를 이용하여 선언.
- 어떤 메소드가 두 번 다시 사용할 일이 없다고 판단될 때 사용.

```
DELEGATE_INSTANCE = delegate(PARAMETER, ...)
            {
            //code
            }
```

```
using System;

namespace ConsoleApplication1
{
    class Program
    {
        delegate int del_calc(int a, int b);

        static void Main(string[] args)
```

```
        {
            Print(
                delegate(int a, int b)
                {
                    return a * b;
                }
                , 5, 7);
        }

        static void Print(del_calc d, int a, int b)
        {
            Console.WriteLine(d(a,b));
        }
    }
}
```

Event

- 객체에 일어난 사건 알리기. Delegate의 주 용도.

```
using System;

namespace ConsoleApplication2
{
    delegate void onClick(string what);

    class Program
    {
        //event EventType EventName
        public event onClick myClick;

        public static void Main(string[] args)
        {
            Program main = new Program();
            TestDelegate td = new TestDelegate();

            onClick deleMethod1 = new onClick(td.MouseClick);
            onClick deleMethod2 = new onClick(td.KeyBoardClick);

            //Event += Delegate
            main.myClick += deleMethod1;
            main.myClick += deleMethod2;

            main.myClick("Hello"); //두 Method에 전달됨.
        }
    }

    public class TestDelegate
```

```
    {
        public void MouseClick(string what)
        {
            Console.WriteLine("Mouse Click! " + what);
        }

        public void KeyBoardClick(string what)
        {
            Console.WriteLine("KeyBoard Click! " + what);
        }
    }
}
```

```
/* Ver.1 */
using System;

namespace ConsoleApplication2
{
    delegate void sendMessage(string message);

    class Program
    {
        public static event sendMessage sendMessageEvent;

        public static void Main(string[] args)
        {
            Program.sendMessageEvent += Program_sendMessageEvent;
            Program.sendMessageEvent("Hello, World!");
        }

        static void Program_sendMessageEvent(string message)
        {
            Console.WriteLine(message);
        }
    }

}
```

```
/* Ver.2 */
using System;

namespace ConsoleApplication2
{
    class Program
    {
        public static void Main(string[] args)
```

```
        {
            MyEventManager.sendMessageEvent +=
MyEventManager_sendMessageEvent;
            MyEventManager.SetOnSendMessage("Hello World!");
        }

        static void MyEventManager_sendMessageEvent(string message)
        {
            Console.WriteLine(message);
        }
    }

    class MyEventManager
    {
        public delegate void sendMessage(string message);
        public static event sendMessage sendMessageEvent;

        public static void SetOnSendMessage(string message)
        {
            if (sendMessageEvent != null)
                sendMessageEvent(message);
        }
    }
}
```

Event vs. Delegate

- Event는 public 수식일지라도 자신이 선언된 클래스 외부에서 호출 불가.
- Delegate는 Callback 용도로, Event는 객체의 상태 변화 or 사건 발생을 알리는 용도로 구분하여 사용하는 게 옳다.

```
using System;

namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
            MyNotifier noti = new MyNotifier();
            noti.Handler += new EventHandler(MyHandler);
            //noti.Handler("test");  /* Error! event는 객체 외부에서 호출 불가. */

            for (int i = 1; i < 30; i++)
                noti.DoSomething(i);
        }

        static public void MyHandler(string message)
```

```
        {
            Console.WriteLine(message);
        }
    }


    delegate void EventHandler(String message);

    class MyNotifier
    {
        public event EventHandler Handler;

        public void DoSomething(int num)
        {
            int temp = num % 10;
            if (temp != 0 && temp % 3 == 0)
                Handler(String.Format(num+" : 짝"));
        }
    }
}
```

LINQ (Language INtegrated Query)

- 선행 지식으로 Delegate & Lambda 필요.
- Collection(배열등 여러 자료구조)을 편리하게 다루기 위한 목적. (Read, Filtering, Sort, Merge, ...)
- from 절에서 **IEnumerable<T>** Type만 사용할 수 있다.

```
using System;
using System.Linq;

namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
            int[] numbers = { 99, 2, 55, 1, 3, 77, 30, 20};

            var result =
                from n in numbers //어떤 집단?
                where n % 2 == 0  //어떤 조건?
                orderby n
                select n;

            foreach(int n in result)
                Console.WriteLine(n);
        }
    }
```

```
}
```

명명법

Visual Studio 2013 사용 Tip

1. Eclipse처럼 Method 자동완성(Intelligence) 사용시 곧바로 자동 소괄호+매개 변수 설명을 보고싶다면 **Visual Assist X**를 사용하면 된다. ~~(그런데 유료)~~
2. Eclipse의 **syso, Ctrl+Space**를 치면 **System.out.println()**이 자동생성되는 것처럼 VS에도 같은 기능으로 **코드 조각**이라는 기능이 있다.
    - 예) cw, Ctrl+Space ⇒ Console.WriteLine();
    - 더 많은 예시는 https://msdn.microsoft.com/ko-kr/library/z41h7fat.aspx 참조.

Eclipse vs Visual Studio (IDE 단축키)

	Eclipse	**Visual Studio**
자동 Package 추가 (Import/using)	Ctrl+Shift+O	(대소문자 정확히 입력 후 맨 끝에서) Ctrl+. Alt+Shift+F10
자동 Interface & Abstract Class 요소 구현		(Header에 커서 두고) Alt+Shift+F10
Method의 Argument 보기	Ctrl+Space	Ctrl+Shift+Space
코드 자동 정렬	Ctrl+Shift+F	Ctrl+K,F
줄 단위 Block	Shift+↑/↓	
주석 (//)	설정/해제 : Ctrl+/	(코드 바로 앞에) 설정 : Ctrl+K+C, 해제 : Ctrl+K+U (줄 맨 앞에) 설정/해제 : /
주석 ()	설정 : Ctrl+Shift+/ 해제 : Ctrl+Shift+\	설정/해제 : Shift+8
줄 삭제	Ctrl+D	Ctrl+Shift+L
줄 이동	Alt+↑/↓	(Block 미지정 상태에서) Ctrl+X/L, Ctrl+V
줄 복사	Ctrl+Alt+↑/↓	(Block 미지정 상태에서) Ctrl+C, Ctrl+V

From:
https://wiki.ledyx.xyz/ - **Ledyx WIKI**

Permanent link:
https://wiki.ledyx.xyz/doku.php?id=c_sharp_vs_java&rev=1612667712

Last update: **2026/10/24 07:03**