목 차

SECTION 1.

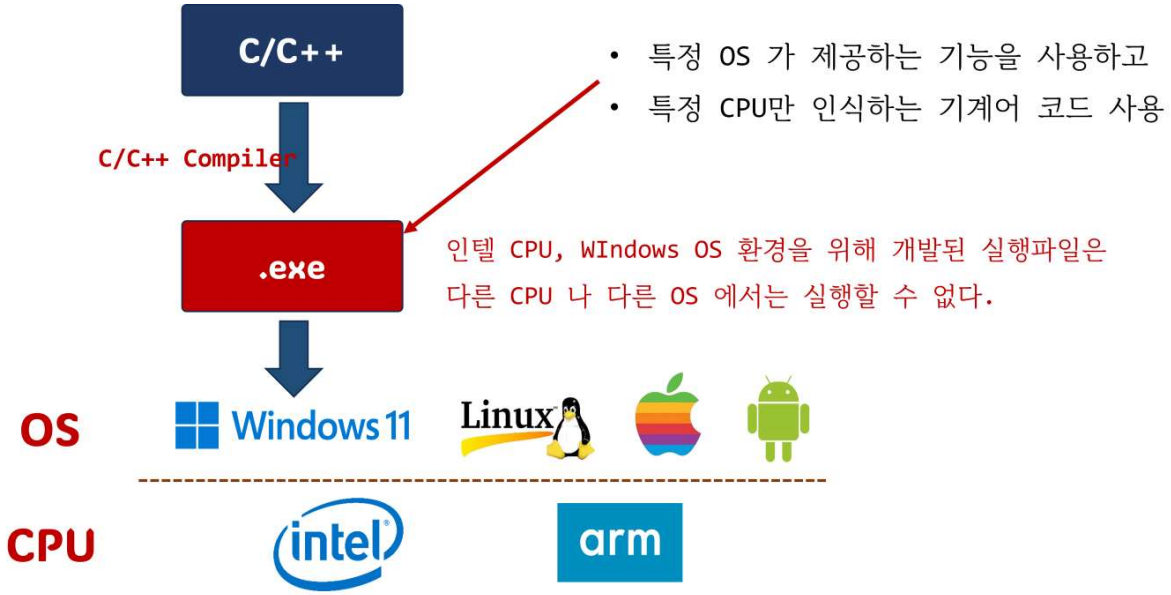
Intro

❑ 주요 학습 내용

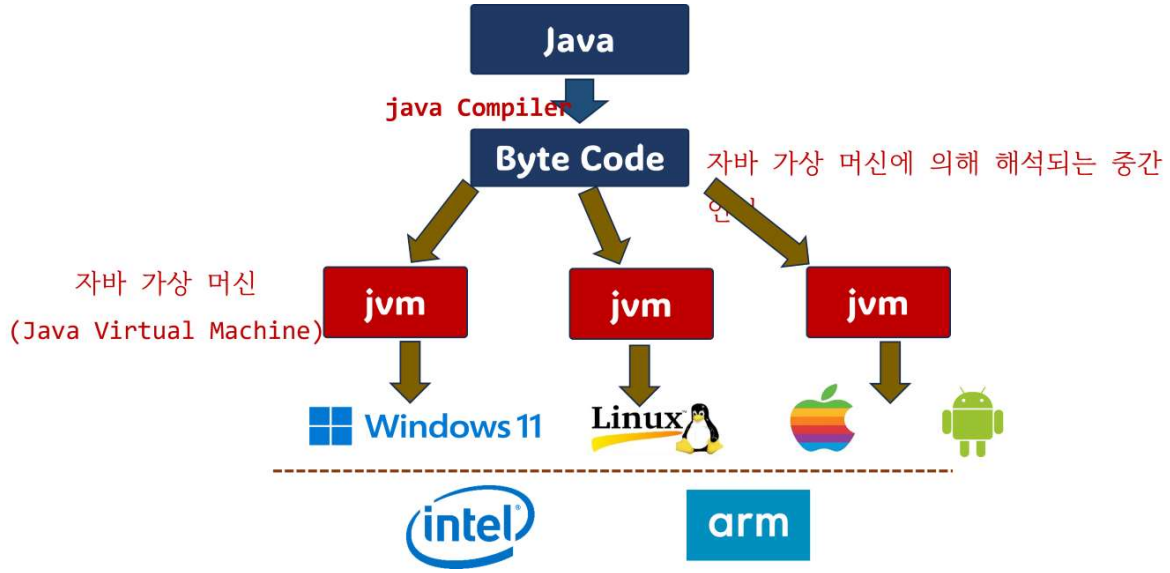
> .net Framework
>
> First C# Programming
>
> Program Entry

.net 개념

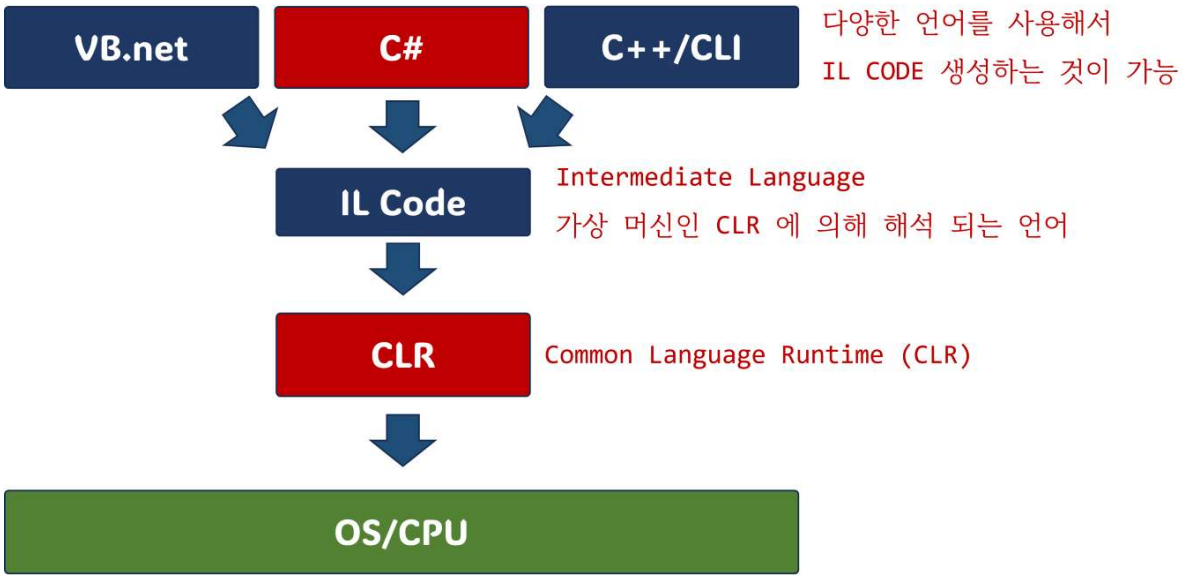
C/C++ 의 컴파일 과정



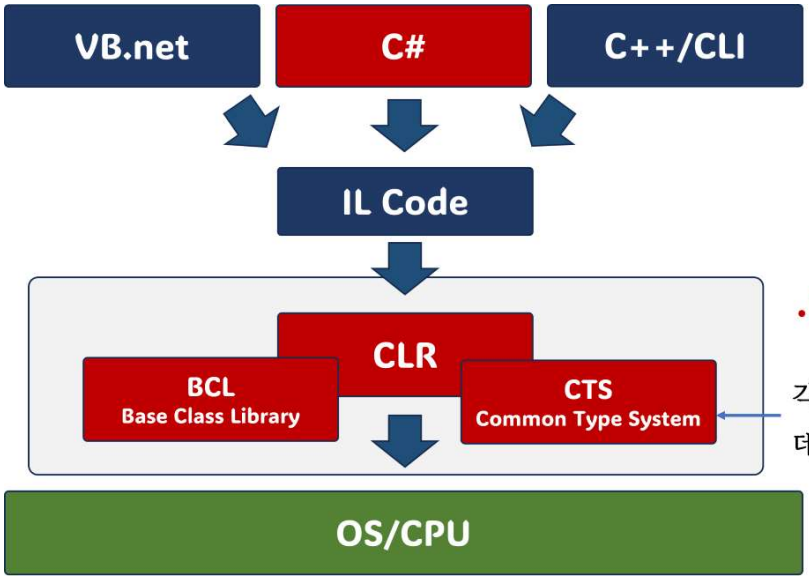
Java 의 컴파일 과정



.Net(C#) 의 컴파일 과정



CLR, BCL, CTS



.Net FrameWork

각각의 언어가 공통으로 사용할 데이타 타입이 필요

entry point

프로그램 실행시 소스코드에서 처음 실행되는 지점을 **"entry point"** 라고 합니다.

전통적으로는 아래의 2가지 방식이 있습니다.

1. 약속된 함수 부터 실행되는 경우

=> C/C++ 언어는 main 이라는 이름을 가지는 함수의 1번째 문장 부터 실행됨

2. 소스 파일의 첫번째 문장 부터 실행되는 경우

=> Python, Swift 언어

C# 언어는 위의 2가지 방식을 모두 지원 합니다.

방식	C# 버전
약속된 메소드(Main)를 만드는 방식	C# 1.0 부터 사용되던 방식
Top-Level Programming 방식	C# 9.0 부터 사용 가능 Main 메소드

방법 #1. Main 메소드를 만드는 방식

아래 코드는 C#1.0 시절 부터 사용되던 전통적인 방식으로 만들어진 C# 프로그램 입니다.

```
class Program
{
    public static void Main()
    {
        // 이곳부터 (`Main 메소드의 1번째 문장`) 프로그램이 실행 됩니다.
        System.Console.WriteLine("Hello, C#");
    }
}
```

방법 #2. Top-Level Programming 방식

C# 9.0 부터는 별도의 Main method 를 만들지 않고도, 아래와 같이 프로그램을 작성할 수 있습니다.

```
// 별도의 Main 메소드 없이 소스 코드의 1번째 문장 부터 실행됩니다.
System.Console.WriteLine("Hello, C#");
```

Top-Level Programming 방식을 사용하면

간단한 예제 작성시 별도의 Main 메소드를 만들 필요가 없으므로 편리합니다.

Top-Level Programming 방식의 원리

Top-Level Programming 방식으로 코드를 작성하면

- 컴파일러가 내부적으로 임의의 클래스와 Main 메소드를 만든 후에
- 사용자가 만든 코드를 Main 메소드 안에 넣고 실행하게 됩니다.

사용자가 만든 코드	컴파일러가 변경한 코드

`System.Console.WriteLine("Hello");`	`class CompilerGeneratedName`
	`{`
	`  public static void Main()`
	`  {`
	`  System.Console.WriteLine("Hello");`
	`  }`
	`}`

SECTION 2.

표준 입출력

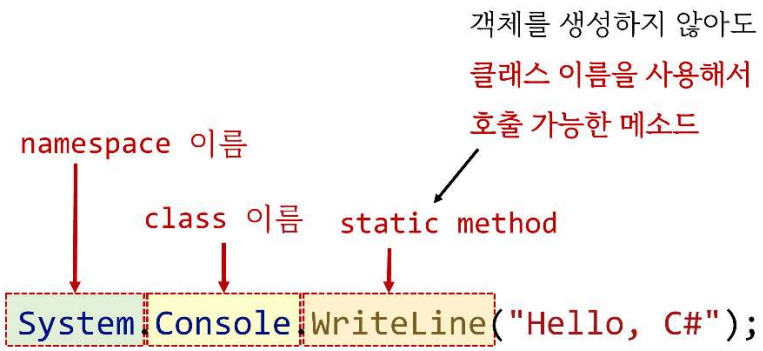
❑ 주요 학습 내용

> **표준 출력**
>
> using directive
>
> **표준 입력**

표준 출력

C# 에서 화면 출력을 위해서는 다음과 같은 코드를 사용합니다

```
System.Console.WriteLine("Hello, C#");
```



WriteLine() vs Write()

WriteLine() 메소드는 출력후 자동으로 개행을 하게 됩니다.
개행을 하지 않으려면 Write() 메소드를 사용하면 됩니다.

```
// WriteLine 메소드는 출력후 개행됩니다.
System.Console.WriteLine("Hello, ");    // Hello,
System.Console.WriteLine("C#");         // C#

// Write 메소드는 출력후 개행하지 않습니다.
System.Console.Write("Hello, ");
System.Console.Write("C#");             // Hello, C#
```

변수값 출력

화면에 변수의 값을 출력하려면 아래 처럼 하면 됩니다.

```
int n1 = 10;
int n2 = 20;

// #1. 변수 한개를 출력 하려면 Write 또는 WriteLine 의 인자로 바로 전달하면 됩니다.
System.Console.WriteLine(n1);

// #2. 문자열과 함께 출력하려면 아래 처럼 합니다.
System.Console.WriteLine("n1 = {0}, n2 = {1}, {0}", n1, n2);

// #3. $ 문자열(interpolated string) 을 사용하면 {}안에 변수 이름을 직접 표기할수 있습니다.
System.Console.WriteLine($"n1 = {n1}, n2 = {n2}, {n1 + n2}");
```

using

화면 출력시 사용하면 Console 클래스는 "System" 이라는 namespace 에 있습니다.
따라서, Console 클래스를 사용 할 때는 반드시 System 이름을 같이 적어야 합니다.

```
System.Console.WriteLine("hello, C#");  // A. ok
Console.WriteLine("hello, C#");         // B. error
```

C# 10.0 이상 버전을 사용하고, visual studio 나 dotnet build 로 빌드하는 경우
위 코드에서 B 부분이 에러가 나지 않을수 있습니다.
에러가 나지 않은 이유에 대해서는 뒷부분에서 설명 됩니다.
C# 10.0 이상이라도 csc 컴파일러 직접 사용하면 B 부분은 에러 입니다.

using directive

using 지시어(directive) 사용하면 Console 클래스 사용시 System 이라는 namespace 이름을 생략하고 사용할 수 있습니다.

```
// A. using directive(지시어) 사용
//    "using namespace 이름" 형태로 사용합니다.
using System;

// B. 이제 System namespace 안에 있는 모든 요소는
//    System 이라는 이름 없이 사용가능합니다.
Console.Write("Hello, ");
Console.WriteLine("C#");
```

using static directive

C# 6.0 에서 추가된 using static 지시어를 사용하면 namespace 이름 뿐 아니라 클래스 이름도 생략 가능합니다.

```
// using static directive
// using static "namespace_name.type_name" 으로 사용합니다.
// 타입이 가진 static 메소드를 타입이름 없이 사용가능합니다.
using static System.Console;

Write("Hello, "); // System.Console.Write("Hello, ") 와 동일
WriteLine("C#");
```

using alias

using alias 를 사용하면 namespace 또는 class 의 별칭을 만들 수도 있습니다.

```
using Sys = System;         // A. System 대신 Sys 사용 (namespace 별칭)
using Std = System.Console; // B. System.Console 대신 Std 사용(type 별칭)

Sys.Console.WriteLine("Hello");
Std.WriteLine("Hello");
```

위 코드에서 **A 는 namespace 에 대한 별칭(alias)** 이고

B 는 type 에 대한 별칭(alias) 입니다.

global using directive

C# 10.0 부터 global using 지시어가 추가되었습니다.

global using 을 사용하면 여러개의 파일로 나누어서 소스 코드를 작성할 때

- 한개의 파일에만 global using 구문을 작성 하면
- 프로젝트내의 모든 소스 파일에서 namespace 이름(또는 타입이름)을 생략할수 있습니다.

a.cs	b.cs

// 한 개의 파일에만 아래 처럼 만들면
global using System;
global using static System.Console;
 	
Console.WriteLine("hello, C#")
WriteLine("hello, C#");
	// 프로젝트 내의 모든 소스 파일에서
	// 아래와 같이 사용가능 합니다.
	Console.WriteLine("hello, C#")
	WriteLine("hello, C#");

implicit using

C# 10.0 이후 visual studio 나 dotnet tools 를 사용해서 빌드 하면 아래 코드가 에러가 나지 않습니다.

```
Console.WriteLine("hello");   // using System 지시어를 사용하지 않았지만
                              // 에러가 아닙니다.
```

에러가 나지 않은 이유는

1. 프로젝트 생성시 자동 생성되는 **"project 파일(.csproj)"** 에 아래와 같은 "ImplicitUsings" 항목이 "enable" 로 설정되게 됩니다.

```
<Project Sdk="Microsoft.NET.Sdk">
    <PropertyGroup>
        <OutputType>Exe</OutputType>
        <TargetFramework>net9.0</TargetFramework>
          <ImplicitUsings>enable</ImplicitUsings>
        <Nullable>enable</Nullable>
    </PropertyGroup>

   ......
</Project>
```

2. "ImplicitUsings" 항목이 "enable" 로 되어 있는 경우 아래의 경로에서 다음과 같은 파일이 자동 생성 되어있습니다.

project_name/obj/x64(또는x86)/Debug/net9.0/***project_name***.GlobalUsings.g.cs

이 파일의 내부에 보면 아래와 같은 코드가 있습니다.

```
// <auto-generated/>
global using global::System;
global using global::System.Collections.Generic;
global using global::System.IO;
global using global::System.Linq;
global using global::System.Net.Http;
global using global::System.Threading;
global using global::System.Threading.Tasks;
```

위와 같이 프로젝트 폴더내에 자동생성된 소스 코드가 있고, 이 안에서 global using 지시문을 가지고 있기 때문에 사용자 코드에서 using System 없어도 System 이름을 생략할수 있습니다.

이경우 csc 컴파일러를 직접 사용 해서 빌드 (**"csc 소스이름.cs"**) 하면 using System 이 없기 때문에 에러가 발생하게 됩니다.

표준 입력

문자열 입력

사용자에게 문자열을 입력 받으려면 Console 클래스의 **"ReadLine()"** 이라는 static method 를 사용하면 됩니다.

```
using System;

Console.Write("input your name >> ");

string s = Console.ReadLine();

Console.WriteLine(s);
```

정수, 실수 입력

문자열이 아닌 정수 또는 실수를 입력 받으려면

- 문자열 형태로 먼저 입력을 받은 후에
- 입력된 문자열을 정수 또는 실수로 변환해서 사용해야 합니다.

```
using System;

Console.Write("input yout age >> ");

// #1. 먼저 문자열 형태로 입력을 받습니다.
string s = Console.ReadLine(); // "10" 입력

// #2. 입력된 문자열을 정수(또는 실수)로 변경합니다.
//     아래의 2 가지 방법이 가능합니다
int n1 = Convert.ToInt32(s);
int n2 = int.Parse(s);

Console.WriteLine($"{n1}, {n2}");
```

int.TryParse()

모든 문자열을 정수 또는 실수로 변경할 수 있는 것은 아닙니다.

- "10" 이라는 문자열을 정수로 변경할 수 있습니다. 10 이 됩니다.
- "hello" 라는 문자열은 정수로 변경할 수 없습니다.

문자열을 정수(또는 실수)로 변경하는데 실패 하면 runtime error(예외 발생) 가 발생하게 됩니다.

에러대신 성공/실패 여부를 반환 값으로 얻고 싶다면 **"int.TryParse()"** static method 를 사용하면 됩니다.

```
using System;

string s = "hello";

// #1. s 는 문자열로 변경될수 없습니다.
```

```
//     아래 코드는 실행시 오류(예외 발생)가 발생합니다
int n1 = int.Parse(s);

// #2. int.TryParse 를 사용하면
// 성공 실패 여부 : 반환값(bool) 로 전달
// 성공시 결과 값 : TryParse 의 2 번째 인자로 전달된 변수에 저장.

int n2 = 0;
bool b = int.TryParse(s, out n2);
             // out 의 의미는 결과를 담아 오겠다는 의미인데
              // "method" 를 배울때 자세히 배우게 됩니다.
```

SECTION 3.

Type & Variable

❑ 주요 학습 내용

> Type & Variable
>
> Array
>
> CTS (Common Type System)
>
> Method, Property
>
> Value Type & Reference Type
>
> Mutable, Immutable, string
>
> tuple

Built-in Types

프로그래밍 언어에서 **"언어 자체가 제공하는 타입"** 을 **"Built-in types"** 또는 **"primitive type"** 이라고 합니다.

C# 언어는 다음과 같은 "Built-in types" 을 제공합니다.

TYPE	SIZE	DESC
byte	1	Signed 8-bit value
sbyte	1	Unsigned 8-bit value
short	2	Signed 16-bit value
ushort	2	Unsigned 16-bit value
int	4	Signed 32-bit value
uint	4	Unsigned 32-bit value
long	8	Signed 64-bit value
ulong	8	Unsigned 64-bit value
nint	**4/8**	**Signed 32-bit or 64-bit integer**
nuint	**4/8**	**Unsigned 32-bit or 64-bit integer**
float	4	IEEE 32-bit floating point value
double	8	IEEE 64-bit floating point value
decimal	**16**	**A 128-bit high-precision floating-point value.**
bool	1	A true/false value
char	2	16-bit Unicode character
string		An array of characters
object		**Base type of all types**
dynamic		

using variable

Declare variable

C# 언어도 C/C++/Java 와 동일한 방식으로 변수를 선언합니다.

```
int    n = 0;
double d = 3.4;
char   c = 'A';
string s = "hello";
```

literal

"10, 3.4" 같이 소스 코드에서 직접 사용하는 값 을 **"리터럴(literal)"** 이라고 합니다.

```
int a1 = 10;        // 10 진수
int a2 = 0x10;      // 16 진수
int a3 = 0b10;      // 2 진수

int a4 = 1000000;   // 큰 값을 사용하는 경우
int a5 = 1_000_000; // 자릿수 표기법 을 사용하면 편리
```

var keyword

var 키워드를 사용하면 변수 선언시 타입을 생략할수 있습니다.

```
// 우변의 초기값을 보고 좌변 변수의 타입을 결정합니다.
var v1 = 10;        // int v1 = 10
var v2 = 3.4;       // double v2 = 3.4
```

```
var v3 = "hello";   // string v3 = "hello"
```

초기값이 없는 경우는 var 키워드를 사용할수 없습니다

```
int n; // ok
var v; // error
```

casting

C/C++ 언어는 int 형 변수에 double 값을 넣는 것을 허용합니다.(컴파일시 경고 발생)
C# 은 타입이 엄격하기 때문에 double 값을 int 타입 변수에 넣을수 없습니다.(double => int 로의 암시적 변환 허용안됨)
하지만, 명시적 캐스팅을 사용하면 double 값을 int 에 넣을수도 있습니다.
C# 의 명시적 캐스팅은 C/C++ 과 유사하게 () 를 사용합니다.

```
using static System.Console;

// casting
double d = 3.7;

//int n1 = d;       // error
int n2 = (int)d;    // ok. 3.7 이 3 으로 변경되어 n2 에 들어 갑니다.

WriteLine(n2);      // 3
```

nameof

nameof 키워드를 사용하면 "변수이름", "속성 이름" 등을 문자열로 변경할수 있습니다.

```
int color = 100;
string s = "abcd";

WriteLine($"{nameof(color)} : {color}");    // nameof(color) => "color"
               // "color"  : 100

WriteLine($"{nameof(s.Length)} : {s.Length}");
                                       // nameof(s.Length) => "Length"
                                       // 결과 : "Length" : 4
                        // 속성(property) 개념은 나중에 배우게 됩니다.
```

변수의 초기화

초기화 되지 않은 변수는 쓰기만 가능 합니다.

```
using static System.Console;

// 초기화되지 않은 변수는 쓰기만 가능.
int n;
int a = 0;

a = n;          // error. 초기화되지 않은 변수를 읽을수 없습니다..
WriteLine(n);   // error .역시 읽는 코드 입니다.

n = a; // ok. 초기화되지 않아도 쓰기는 가능합니다
       //    이 코드 때문에 n 은 초기값을 가지게 됩니다.

a = n; // ok. 위에서 n 에 값을 넣었으므로 이제 읽 을 수 있습니다.
```

변수 초기값을 지정하는 방법

C# 에서는 변수에 초기값을 넣는 다양한 방법이 존재 합니다.
아래 코드는 모두 0으로 초기화된 변수를 만드는 코드입니다.(에러인 n5 제외)

```
int n1 = 0;
int n2 = default(int); // () 안에 있는 타입(int)의 default 값으로 초기화
int n3 = default;      // 좌변의 타입(int)의 default 값으로 초기화
var n4 = default(int); // (int) 가 있으므로 var 사용가능
var n5 = default;      // error. 타입이 추론될수 없다.
int n6 = new int();     // int n6 = 0 과 동일
```

Default value

C# 언어는 각 타입에 디폴트 값이라는 개념이 있습니다.

Type	Default value
reference	null
integral	**0 (zero)**
floating point	**0 (zero)**
bool	FALSE
char	'\0' (U+0000)
enum	The value produced by the expression (E)0 , where E is the enum identifier.
struct	The value produced by setting all value-type fields to their default values and all reference-type fields to null .
nullable value type	An instance for which the HasValue property is false and the Value property is undefined. That default value is also known as the null value of a nullable value type.

위 테이블에서 볼수 있듯이 int 타입의 default 값은 0 입니다.
따라서 아래 코드는 모두 0으로 초기화 된 변수 입니다.

```
int n2 = default(int); // int n2 = 0;
int n3 = default;      // int n3 = 0;
```

그냥 **"=0"** 으로 초기화 하면 되는데 왜 default 를 사용할까 ? 라는 궁금증이 있을수 있습니다. 일반적으로는 그냥 0 으로 초기화 하면 됩니다.

후반부에서 다루게 되는 Generic 문법등을 사용하면 **"=0"** 을 사용할수 없을때가 있습니다.지금은 그냥, **"default 라는 것이 있다"** 정도만 알아두면 되고, Generic 을 배울때 더 자세히 다루게 되게 됩니다.

new int

C# 언어는 각 타입에 디폴트 값이라는 개념이 있습니다

C# 에서 모든 변수는 기본적으로 new 를 사용해서 만들어야 합니다.
하지만 "Built-in types" 에 대해서는 new 를 사용하지 않아도 되는 **"편의 표기법"** 을 제공합니다.

```
int n1 = new int(); // A. C# 에서 모든 변수는 new 로 만들어야 하지만
int n2 = 0;         // B. 이렇게 해도 됩니다. 위와 동일한 IL 코드 생성
```

위 코드와 같이 int 타입인 경우 위 A, B 는 동일한 IL 코드를 생성하게 됩니다.
하지만 string 타입 같은 경우는 new 를 사용하는 것과 사용하지 않은 것은 약간의 차이가 있고, 사용하지 않은 것이 효율적인 IL 코드를 생성합니다. 자세한 내용은 string 주제를 배울때 설명 됩니다.

```
// 아래 코드는 약간의 차이점이 있습니다.
string s1 = new string("ABCD"); // 이 코드 보다
string s2 = "ABCD"              // 이 코드가 효율적인 코드 입니다.
```

정리하면 C# 에는 표준 타입(Built-in types) 외에 class(struct) 문법으로 만들어져서 라이브러리 형태로 제공되는 다양한 타입이 존재 합니다.

1. C# 에서 모든 변수(객체)의 생성은 new 를 사용하는 것이 원칙 입니다.
2. 하지만, 표준 타입(Built-in types) 은 new 없이 변수를 만들수 있는 편의 표기법을 제공하고 이 방법을 사용하는 것을 권장 합니다.
3. new 를 사용한 것과 사용하지 않은 것은 타입에 따라 차이가 없을수도 있고 있을수도 있습니다.(타입별로 다름)

array

C# 언어도 다른 언어와 유사하게 배열을 제공합니다.

배열(array) 란 ?
동일 타입의 값을 연속적인 메모리에 여러개 보관하고, [] 연산자를 통해서 접근할수 있게 하는 타입으로 요소의 갯수가 N개인 배열은 0~N-1 의 인덱스를 사용해서 접근해야 합니다.

C# 에서 int 타입의 1차 배열의 타입은 "int[]" 입니다.

```
int[] arr = { 1, 2, 3, 4, 5, 6}; // 6 개의 요소를 가지는 배열을 생성합니다.
arr[0] = 0; // 배열의 각 요소 접근은 [] 연산자를 사용합니다.
arr[6] = 0; // runtime-error(예외 발생),
            // 크기가 6 인 배열은 0~5 까지의 index 를 사용합니다.
arr[^1];  // ^1 도 가능, 마지막 요소(arr[5]) 접근
arr[^2];  // arr[4]
```

배열의 초기화

C# 의 배열은 다양한 방법으로 초기화 될 수 있습니다.

```
int[] x1; // A. 배열을 만든것이 아니라 배열을 가리키는 reference 를 만든 것
          //    reference 를 배울때 자세히 설명
// B. C# 에서 모든 객체(변수)는 new 를 사용해서 만드는 것이 원칙 입니다.
int[] x2 = new int[6];  // 모든 요소가 0 으로 초기화 됩니다.
int[] x3 = new int[6]{1,2,3,4,5,6};
int[] x4 = new int[ ]{1,2,3,4,5,6};
// C. 배열도 new 없이 만들수 있는 "편의 표기법"을 제공합니다.
int[] x5 = {1,2,3,4,5,6};
```

```
int[] x6 = [1,2,3,4,5,6];   // C# 12 부터는 {} 대신 [] 도 사용 가능 합니다.

// D. 타입에는 크기를 표기하지 않습니다.
int[6] x7 = {1,2,3,4,5,6};    // error. 일차 배열의 타입은 "int[]" 입니다.
                              // C/C++ 언어와는 달리 타입에 크기를
                              // 넣을수 없습니다.
int[] x8 = new int[6];        // ok. new 표현식 사용시 우변에는
                              // 크기를 넣을수 있습니다.
```

배열은 위와 같은 다양한 방법으로 초기화 될수 있지만 이어지는 예제 에서는 대부분 아래의 방식을 사용합니다.

```
int[] arr = {1,2,3,4,5,6};
```

배열과 함수 인자

메소드(함수) 인자가 배열일때 **"{1,2,3}"** 로 바로 전달할수는 없고 new 표현식을 사용해야 합니다.
C#12.0 부터 지원되는 "[] 초기화(Collection expression)"* 은 사용할수 있습니다.

```
// A. 메소드(함수)가 인자로 배열을 요구 합니다.
void foo(int[] arr) {}
foo({1,2,3});          // error. {} 초기화로 전달 할수 없습니다.
foo( new int[]{1,2,3}); // ok.    new 를 사용해서 만들어야 합니다.
foo([1,2,3]); // ok. C# 12.0 부터 지원되는 [] 초기화 로는 가능합니다.
              // ([] 초기화를 "Collection Expression" 이라고 합니다.
```

다차원 배열과 가변 배열

1차 배열 뿐 아니라 다차원 배열(2차, 3차)을 만들수 있고, 가변 배열(jagged array)도 만들수 있습니다.

```
using static System.Console;

// #1. 다차원 배열
int[]   arr1 = {1, 2, 3, 4, 5, 6};
int[,]  arr2 = {{1, 2}, {3, 4}, {4, 5}};
int[,,] arr3 = {{{1, 2}, {3, 4}},{{5, 6}, {7, 8}}};

WriteLine(arr1[2]);      // 3
WriteLine(arr2[1,0]);    // 3
WriteLine(arr3[1,1, 0]); // 7

// #2. 가변 배열(jagged array)
int[][] arr = new int[3][];
arr[0] = new int[]{1,2};
arr[1] = new int[]{3,4,5};
arr[2] = new int[]{6,7,8,9};

WriteLine(arr[1][2]); // 5
```

다차원 배열과, 가변배열, 그리고 배열의 대한 보다 많은 다양한 기법은 별도의 "Array 섹션" 에서 자세히 배우게 됩니다.

CTS (Common Type System)

.net 언어들(C#, C++/CLI, VB등)이 사용하는 모든 타입에 대한 정의는 각각의 개별 언어가 아닌 .net 내부에서 정의 합니다.

이런 것을 **"공통의 타입 시스템(CTS, Common Type System)"** 이라고 합니다.

각각의 언어에서 사용되는 데이타 타입은 결국 .net 내부에서 정의한 타입의 별명일 뿐 입니다.int 타입의 정확한 타입명은 System.Int32 입니다.

아래 표는 .net 내부에서 정의한 타입명과 C# 언어에서 사용하는 별칭 나타냅니다.

C# Type	CTS Type
byte	System.Byte
sbyte	System.SByte
short	System.Int16
ushort	System.UInt16
int	System.Int32
uint	System.UInt32
long	System.Int64
ulong	System.UInt64
nint	System.IntPtr
nuint	System.UIntPtr
float	System.Single
double	System.Double
decimal	System.Decimal
bool	System.Boolean
char	System.Char
string	System.String
object	System.Object
dynamic	System.Object

Using CTS Type

C# 언어에서는 CTS 타입의 이름도 사용할수 있습니다.

```
// 아래 코드에서 n1, n2, n3 는 모두 동일합니다.
int n1 = 0;
Int32 n2 = 0;
System.Int32 n3 = 0;
//System.int   n4 = 0;  // error. int 는 C# 언어의 키워드 이므로
                        //        System 을 붙이지 않습니다.
double d1 = 0;
Double d2 = 0;
System.Double d3 = 0;

string s1 = "A";
String s2 = "A";
System.String s3 = "A";
```

CTS 의 타입 명칭과 C# 언어의 타입이름은 아래와 같은 특징이 있습니다.

C# 타입 이름	소문자로 되어 있습니다. C# 키워드 입니다. System 을 붙이지 않습니다.
CTS 타입 이름	첫글자가 대문자 입니다. System namespace 안에 class 또는 struct 로 만들어져 있습니다. 사용할때 System 을 붙여야 합니다. using 이 있다면 생략 가능.

가끔 C#을 처음 배우시는 분들이 **"string(첫글자 소문자)" 과 "String(첫글자 대문자)", "object" 와 "Object"** 등의 차이점을 궁금해 하시는 분들이 있는데, 결국 같은 타입 입니다.

method & property

C++/Java 같은 언어는 primitive type 과 user define type 이 명확히 구분됩니다

primitive type	int, double 등 언어자체가 지원하는 타입. 타입 이름 자체가 키워드 이다. **메소드(멤버 함수)가 없다.**
user define type	class(struct) 문법으로 만든 타입. 타입 이름은 키워드가 아닌 class(struct) 이름. **메소드(멤버 함수)등을 가질 수 있다.**

따라서 언어 자체가 키워드 형태로 제공 되는 타입은 멤소드(멤버 함수)를 가질수 없습니다. 아래 코드는 Java 언어의 일부 코드 입니다

```
// Java 언어 코드 입니다.
int n = 10;
String s = "ab";
n.toString(); // error. int 타입은 메소드가 없습니다
s.toString(); // ok.    String 타입은 메소드가 있습니다
```

하지만 C# 언어는 모든 타입이 메소드를 가지고 있습니다.

Everything is Object !

C#이 처음 발표 될 때 **"everything is object"** 라는 슬로건 사용했습니다.

C# 언어의 모든 타입은 class 또는 struct 로 만들어진 타입입니다.

int(System.Int32) 도 struct 로 만들어진 타입입니다. **따라서, 모든 타입은 메소드를 가지고 있습니다. (모든 객체(변수) 는 메소드가 있고, literal 도 메소드를 가지고 있습니다)**

```
int n1 = 10;
// n1 이라는 int 형 변수도 메소드가 있습니다.
string s1 = n1.ToString();

// 10 이라는 literal 도 메소드가 있습니다
string s2 = 10.ToString();
```

Property

C# 언어에는 속성(Property) 라는 개념이 있습니다.

- 메소드는 사용할 때 “객체.메소드 이름()” 로 사용하지만
- 속성(Property) 은 “객체.속성이름” 으로 사용합니다. ()가 없습니다.

속성(Property) 은 마치 필드(멤버데이타) 를 접근하는 것과 유사하지만, 멤버 데이터와는 다른 개념입니다.

```
string s = "hello";
var ret1 = s.ToUpper(); // () 가 있으므로 메소드 입니다.
                        // s 가 가진 문자열을 대문자로 변경한
                        // 새로운 문자열 반환 합니다.
var ret2 = s.Length;    // () 가 없으므로 속성(Property) 입니다.
                        // s 의 문자열 갯수(5)를 반환 합니다.
```

읽기만 가능한 속성도 있고, 읽기/쓰기 가 모두 가능한 속성도 있습니다.

속성(Property) 에 대한 자세한 설명은 class 문법을 배울 때 설명됩니다.

static method vs instance method

대부분의 객체지향 프로그래밍 언어에는 static method 와 instance method 라는 개념이 있습니다.

instance method	객체(변수)를 생성하고 **"객체(변수)이름.메소드이름()"** 으로 호출. 메소드가 하는 일은 객체가 가진 데이터와 관련되어 있습니다. Ex) n.ToString() : n 이 가진 데이터를 문자열로 변경
static method	객체(변수)의 이름이 아닌 **"타입이름.메소드이름()"** 으로 호출. 메소드가 하는 일은 특정 객체가 가지 데이터가 아닌, 타입 자체와 관련되어 있습니다. Ex) int.Parse("10") : "10" 이라는 문자열을 int 타입의 값으로 변경

```
int n = 20;

string s = n.ToString(); // instance method
                         // n 이 가진 데이타 20 을 문자열로 반환
                         // s = "20" 이 됩니다.

int v = int.Parse("20"); // static method
                         // 인자로 전달된 "20" 이라는 문자열을 정수로 반환.
                         // v = 20 이 됩니다.
```

static method 에 대한 보다 자세한 설명은 class 문법을 배울 때 보다 자세히 설명 됩니다.

System.Object

C# 에 있는 대부분의 타입은 System.Object class 로부터 상속 받게 됩니다.

System.Object 는 CTS 타입 이름 입니다. C# 타입은 object 입니다.

따라서, C# 언어의 대부분의 타입은

- System.Object 가 가진 다양한 멤버를 물려 받게 됩니다.
- 대부분의 객체는 기반 클래스인 System.Object(object) 로 가리킬 수 있습니다.

```
Car c = new Car();

// #2. Car 객체인 c 안에는 System.Object 로 부터 상속 받은 멤버가 있습니다
string s = c.ToString();
Type t = c.GetType();

// #1. Car 클래스는 어떠한 멤버도 없습니다.
//     하지만 자동으로 System.Object 로 부터 상속받게 됩니다.
//     따라서 Car 안에는 몇개의 멤버가 있습니다.
class Car { }
```

System.Object 가 가진 멤버에 대해서는 별도의 항목에서 자세하 다루게 됩니다.

```
int    n = 0;
string s = "A";

// 대부분의 객체는 System.Object 로 가리킬수 있습니다.
System.Object o1 = n;
System.Object o2 = s;

// System.Object 의 C# 타입 별칭은 object 입니다.
object o3 = n;
object o4 = s;
```

Value Type vs Reference Type #1

C# 에는 Value Type 과 Reference Type 이라는 개념이 있습니다.(Java, Swift 등의 언어에 있는 개념과 유사, C++ 언어에는 없는 개념)

Value Type	**struct 문법으로 만들어진 타입.** 객체(변수)를 생성하면 stack 에 만들어지게 됩니다.
Reference Type	**class 문법으로 만들어진 타입.** 객체(변수)를 생성하면 heap 에 생성되고, stack 에는 참조 변수가 만들어 지게 됩니다.

class 및 struct 문법은 뒤쪽에서 자세히 배우게 됩니다. 지금은 Value Type 과 Reference Type 의 개념만 이야기 합니다.

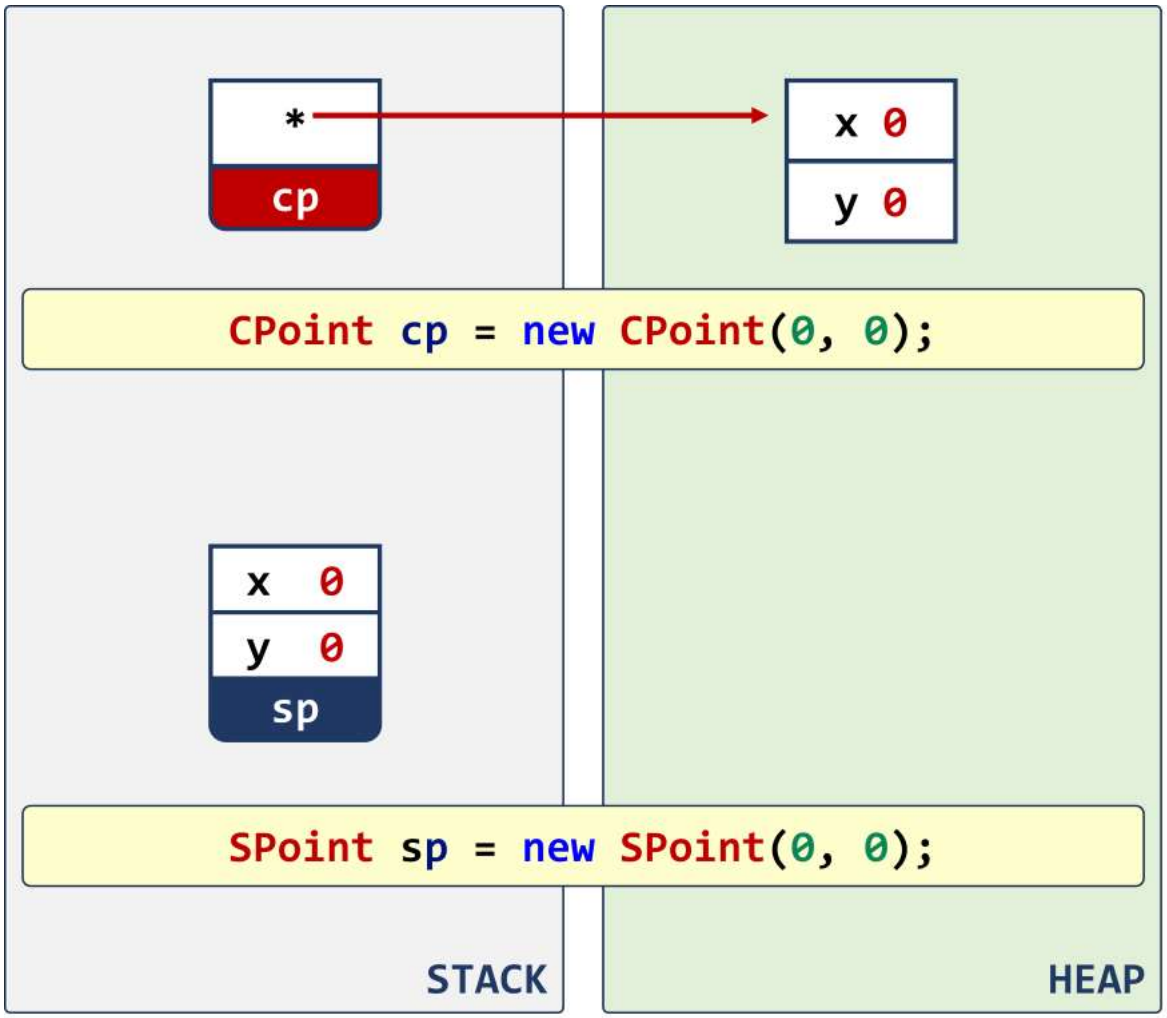
이번 항목에는 아래의 2개의 타입을 가지고 이야기 합니다

```
// CPoint : class 문법으로 만들어진 타입입니다.
class CPoint
{
    public int x;
    public int y;
    public CPoint(int a, int b) { x = a; y = b;}
}
// SPoint : struct 문법으로 만들어진 타입입니다.
struct SPoint
{
    public int x;
    public int y;
    public SPoint(int a, int b) { x = a; y = b;}
}
```

Memory layout

```
CPoint cp = new CPoint(0, 0); // Reference Type 의 객체 생성
SPoint sp = new SPoint(0, 0); // Reference Type 의 객체 생성
```

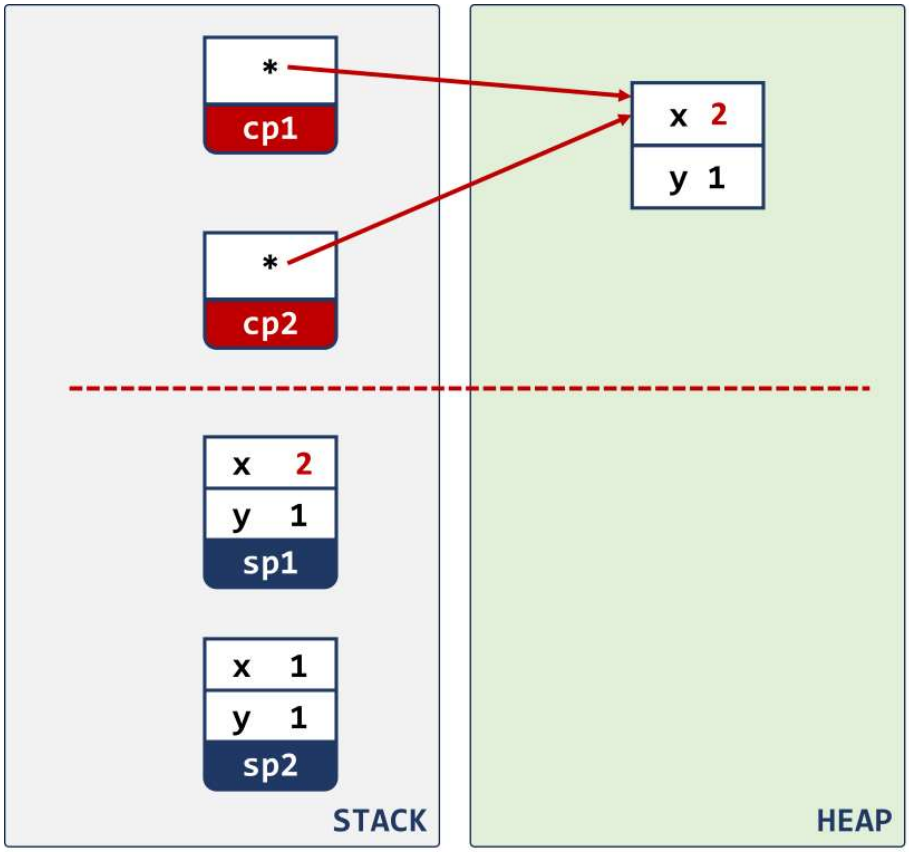
위 소스가 실행되었을 때 메모리 모양은 아래와 같습니다.



객체의 복사

```
CPoint cp1 = new CPoint(1, 1);
CPoint cp2 = cp1;

SPoint sp1 = new SPoint(1, 1);
SPoint sp2 = sp1;

// 메모리 그림을 보고 아래 코드의 결과를 예측해 보세요
cp1.x = 2;
sp1.x = 2;

WriteLine($"{cp1.x} {cp2.x}"); // 2  2
WriteLine($"{sp1.x} {sp2.x}"); // 2  1
```

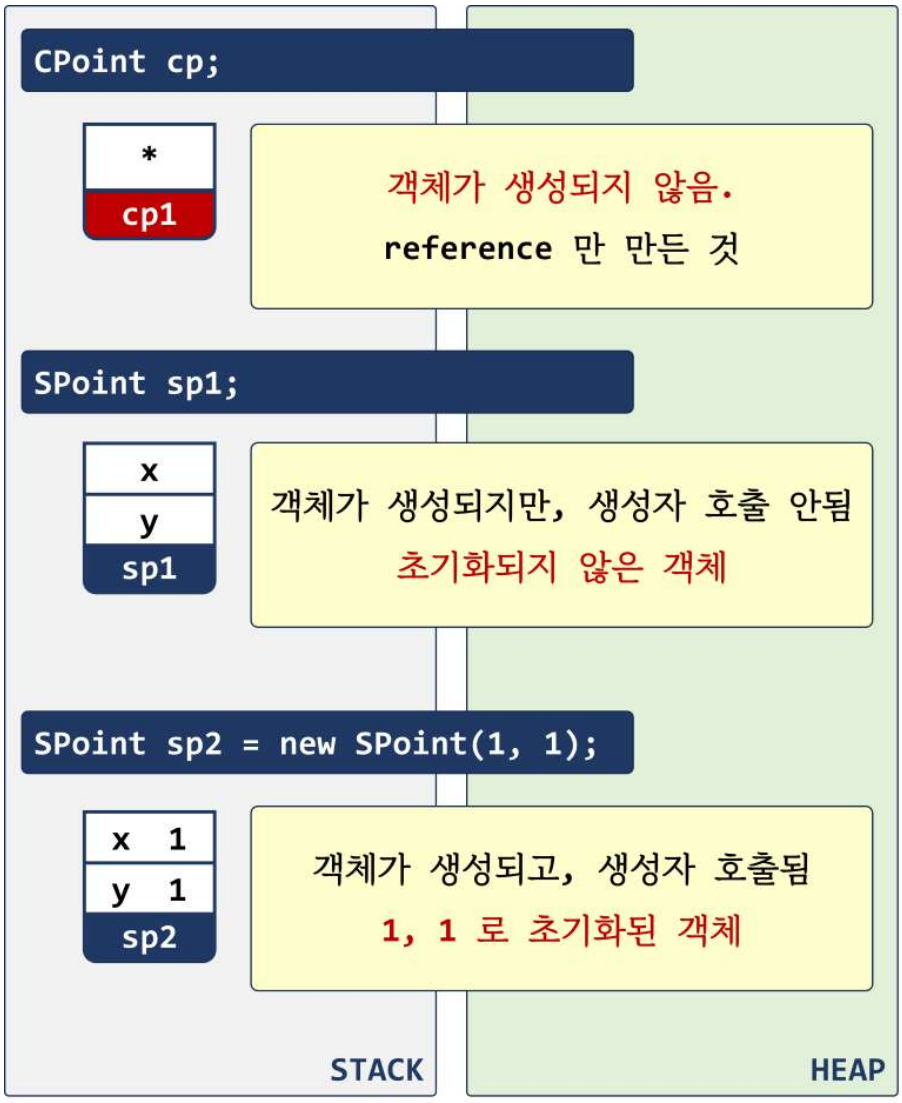
위 소스가 실행되었을 때 메모리 모양은 아래와 같습니다.



객체의 생성

```
CPoint cp1;
SPoint sp1;
SPoint sp2 = new SPoint(1, 1);
```

위 3줄이 실행되었을 때 의 메모리는 다름과 같습니다.



Value Type vs Reference Type #2

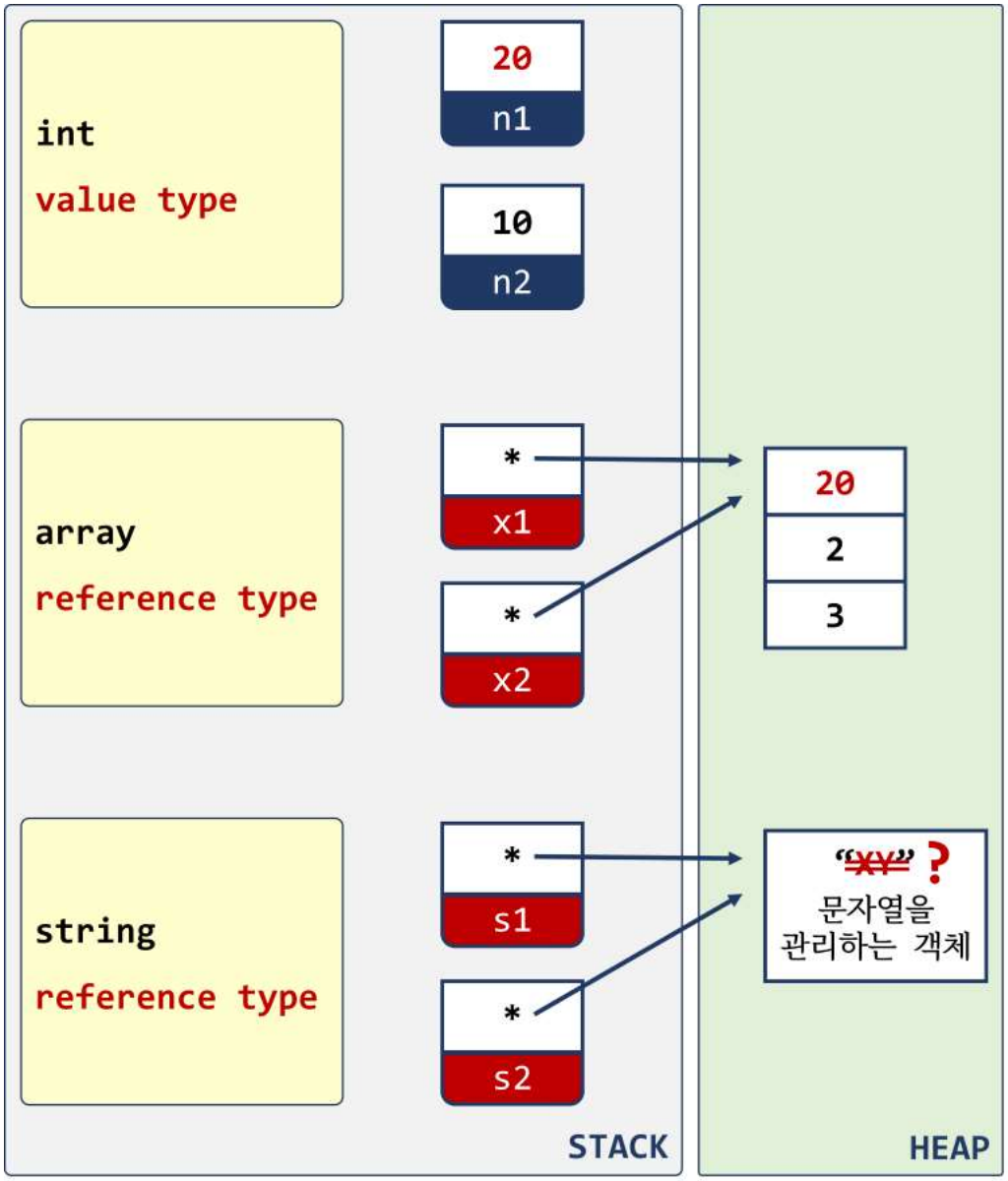
C# 의 표준 타입 중에 int, double 은 value type 입니다.

string 과 배열(array) 는 reference 타입 입니다.

	만드는 방법	built in types
value type	struct enum	bool, char 모든 정수/실수 타입
reference type	class interface delegate record	object string array dynamic

```
using static System.Console;

int n1 = 10;
int n2 = n1;

n1 = 20;
WriteLine($"{n1} {n2}");    // 20  10

int[] x1 = {1, 2, 3};
int[] x2 = x1;

x1[0] = 20;
WriteLine($"{x1[0]} {x2[0]}");  // 20 20

string s1 = "AB";
string s2 = s1;

s1 = "XY";
WriteLine($"{s1} {s2}");  // 결과를 예상해 보세요
```

위 코드가 실행되었을 때 메모리 구조가 다음과 같습니다.



int 변수와 배열은 예상과 다르지 않지만 문자열의 경우는 약간 까다롭습니다.

string 항목에서 자세히 살펴 보겠습니다.

mutable, immutable

대부분의 프로그래밍 언어에는 mutable 과 immutable 의 개념이 있습니다.

mutable	**객체의 상태를 변경할 수 있는 것**
immutable	**객체의 상태를 변경할 수 없는 것**

C# 에서 int 타입은 mutable 하지만 string 타입은 immutable 합니다.

```
// int 타입의 변수는 생성한 상태(데이타)를 변경할수 있습니다.(mutable)
int n = 0;

n = 10; // ok.

// string 타입의 변수는 생성후에 상태(데이타)를 변경할수 없습니다.(immutable)
string s = "AB";

char c = s[0];  // ok. [] 연산자로 상태(데이타)를 읽을수는 있습니다.
s[0] = 'X';     // error. 하지만 변경할수는 없습니다.
```

string

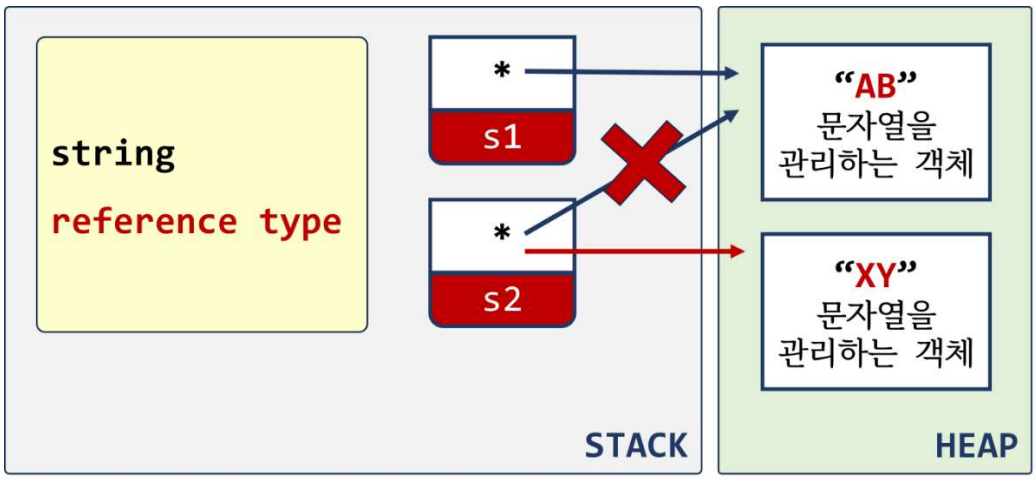
C# string 타입은 immutable 합니다.

즉, string 변수를 생성한 후에는 데이터를 변경할 수 없습니다.

= 연산자로 새로운 문자열을 넣는 것은 기존 데이터의 변경이 아닌, 새로운 문자열 객체를 생성한 것 입니다.

아래 코드와 메모리 그림을 잘 생각해보세요.

```
string s1 = "AB";
string s2 = s1;

s2 = "XY";  // ok. s2 가 가리키는 객체의 상태(데이타)를 변경한것이 아니라
            //     새로운 객체를 만들고 s2 가 가리키게 되는 것.
            // s2 = new string("XY") 와 유사한 의미

WriteLine($"{s1} {s2}"); // "AB"  "XY"
```

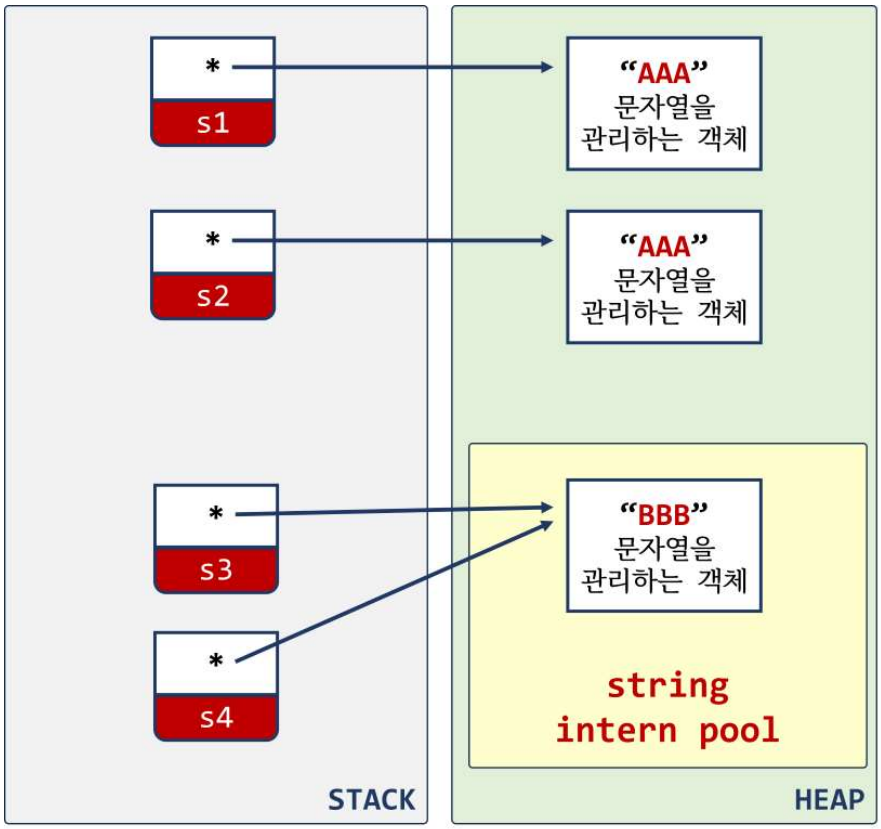


string Intern pool

string 타입은 immutable 하기 때문에 동일한 문자열을 가리키는 string 변수는 공유 됩니다. 공유 되는 문자열은 “intern pool” 이라는 곳에서 관리 됩니다

하지만 new 로 string 변수를 만들면 동일한 문자열이라도 공유 되지 않습니다

```
using static System.Console;

string s1 = new string("AAA");
string s2 = new string("AAA");

string s3 = "BBB";
string s4 = "BBB";

WriteLine($"{object.ReferenceEquals(s1, s2)}"); // false
WriteLine($"{object.ReferenceEquals(s3, s4)}"); // true
```



변경 할 수 없는 동일한 문자열을 메모리에 만드는 것은 비효율적이므로 string 변수는 new 로 만드는 것은 권장하지 않습니다

또한, 2개의 string 변수의 상등을 조사 할 때

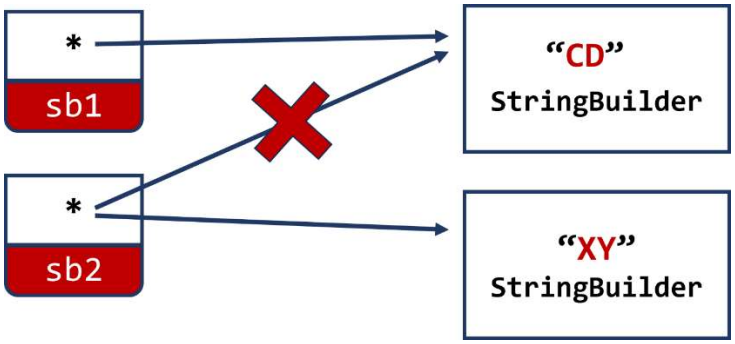
`s1 == s2`	s1, s2 가 가진 문자열이 동일한지 조사합니다. (상태가 동일 한가 ?)
`object.ReferenceEquals(s1, s2);`	**s1, s2 가 같은 객체를 가리키는지 조사합니다.** (객체가 동일 한가?)

동일성 조사(Equality)
뒤쪽에서 별도의 섹션에서 자세히 다루고 있습니다.

StringBuilder

string 이 아닌 StringBuilder 라는 타입을 사용하면 변경가능(mutable) 한 문자열을 만들수 있습니다.

```
using System.Text;
using static System.Console;

//StringBuilder sb1 = "AB"; // error. StringBuilder 객체는
                            // 반드시 new 로 만들어야 합니다.

StringBuilder sb1 = new StringBuilder("AB");
StringBuilder sb2 = sb1;    // sb1, sb2 는 같은 객체를 가리킵니다.

sb1[0] = 'X';   // 변경 가능합니다.

WriteLine($"{sb1} {sb2}"); // XB  XB

//sb1 = "CD";  // error. 이렇게 변경은 안되고 아래 처럼해야 합니다.
sb1.Clear();
sb1.Append("CD");

WriteLine($"{sb1} {sb2}");  // CD   CD
WriteLine($"{object.ReferenceEquals(sb1, sb2)}"); // true

sb2 = new StringBuilder("XY");
WriteLine($"{object.ReferenceEquals(sb1, sb2)}"); // false
```



tuple

배열은 같은 타입의 객체를 여러개 보관할수 있습니다.
반면, 튜플(tuple) 을 사용하면 서로 다른 타입의 객체를 여러개 보관할수 있습니다.

```
int[] arr = { 1, 2, 3 };    // 배열. 같은 타입 3 개를 보관하는 코드
var   tp  = (1, 3.5, "AB"); // 튜플. 다른 타입의 값 3 개를 보관
```

C# 언어는 2개의 tuple 타입을 제공합니다

Tuple	Reference Type	.Net 4.0 이상 지원
ValueTuple	Value Type	.Net 4.7 이상 지원

이중 ValueTuple 이 기능도 많고, 가볍고, 사용하기 편리해서 널리 사용됩니다.

ValueTuple 의 타입

ValueType 의 정확한 타입은 **"ValueType<타입A, 타입B, 타입C, ...>"** 입니다.

하지만, 튜플의 단축 표기법인 () 를 사용하면 **"(타입A, 타입B, 타입C)"** 처럼 간단히 표기할수 있습니다.

```
// 아래 2 줄의 코드는 동일한 코드 입니다.
ValueTuple<int, double, string> t1 =
                    new ValueTuple<int, double, string>(1, 3.4, "abc");

(int, double, string) t2 = (1, 3.4, "abc"); // () 사용한 단축 표기법
```

ValueTuple 의 생성

ValueTuple 은 아래와 같이 다양한 방법으로 만들수 있습니다

```
// #1. ValueTuple 생성
ValueTuple<int, int, int> vt1 = new ValueTuple<int, int, int>(1, 2, 3);
ValueTuple<int, int, int> vt2 = ValueTuple.Create(1,2,3);
ValueTuple<int, int, int> vt3 = (1, 2, 3);

(int, int, int) vt4 = (1, 2, 3);    // 이 코드 또는
var vt5 = (1, 2, 3);                // 이 코드를 권장

// #2. 주의
// var 사용시 () 안에 요소가 한 개만 있을 때는 튜플이 아닌 int 타입이 됩니다.
var a = (1);        // int a = 1
var b = (1, 2);     // (int, int) b = (1, 2)
```

ValueTuple 의 요소 접근

ValueTuple 의 요소에 접근할때는 **"Item1, Item2, Item3 ..."** 의 속성을 사용하면 됩니다.(0 이 아닌 1부터 시작)

```
using static System.Console;

var t = (1, 3.4, "abc");

WriteLine($"{t.Item1}");    // 1
WriteLine($"{t.Item2}");    // 3.4
WriteLine($"{t.Item3}");    // abc
```

Named Member

ValueTuple 을 생성할때 각 멤버에 이름을 지정할수 있습니다.

```
using static System.Console;

(int,     int,     int)      t1 = (1, 2, 3); // unnamed member
(int one, int too, int three) t2 = (1, 2, 3); // named member

WriteLine($"{t1.Item1}");  // unnamed 는 "Item1, Item2, ..." 를 사용합니다
WriteLine($"{t2.one}");    // named   는 "one, too, ..." 등의 이름을
                           // 사용할수 있습니다
WriteLine($"{t2.Item1}");  // "Item1, Item2, ..." 도 계속 사용가능합니다.

// named member 를 만드는 2 가지 방법
(int one, int too, int three) t3 = (1, 2, 3); // 좌변(타입)에 이름을 지정
var t4 = (one: 1, too: 2, three: 3);     // 우변(초기값)에 이름을 지정
```

Tuple destruction

튜플이 가진 요소의 값을 각각 별도의 변수에 저장하려면 아래 2가지 방법이 있습니다.

- Item1, Item2, ... 등의 속성을 사용하는 방법
- tuple deconstruction 기술을 사용하는 방법

```
var t = (1, 2, 3);

// A. Item1, Item2, ... 속성을 사용
int a1 = t.Item1;
int a2 = t.Item2;
int a3 = t.Item3;

// B. tuple deconstruction 문법 사용
(int b1, int b2, int b3) = t;// int b1 = t.Item1;
                             // int b2 = t.Item2;
                             // int b3 = t.Item3;
```

tuple deconstruction 문법을 사용하면 여러개 변수를 복사(대입)할때 간결하게 할수 있습니다.

```
void foo(int a, int b)
{
    // 인자로 전달된 a, b 의 값을 다른 변수에 복사하는 방법
    // 방법 #1. tuple 을 사용하지 않는 경우
    int x1 = a;
    int y1 = b;

    // 방법 #2. tuple deconstruction 을 사용하는 경우
    (int x2, int y2) = (a, b);  // 변수 선언과 동시에 복사

    (x1, y1) = (a, b);          // 선언되어 있는 변수에 대입
}
```

Named Member 와 tuple deconstruction 의 코드는 유사하므로 잘 구별해야 합니다.

핵심은 아래 코드에서 **"tp 같은 변수의 이름이 있느냐 없느냐"** 에 따라서 구문의 의미가 달라 집니다.

```
(int a1, int a2, int a3) tp = (1, 2, 3); // tp 라는 튜플을 생성하는 코드
(int b1, int b2, int b3)    = (4, 5, 6); // 이 문장에는 위 문장에 있는
                                         // tp 라는 변수 이름이 없습니다.
                                         // tuple deconstruction 입니다.
                                         // int b1 = 4;
                                         // int b2 = 5;
                                         // int b3 = 6; 의 의미 입니다.

// tp 는 튜플 이므로 아래와 같이 사용 가능 합니다.
int n1 = tp.Item1;
int n2 = tp.a1;
```

튜플과 메소드 반환 타입

일반적으로 메소드는 한개의 값만 반환 할수 있습니다.

하지만, 튜플을 사용하면 메소드로 부터 여러 개의 값을 반환 할수 있습니다.

```
string Get1()       // 반환값이 한개 입니다
{
    return "john";
}
(string, int) Get2() // 튜플을 사용해서 2 개 반환 합니다.
{
    return ("john", 30);
}
(string name, int age) Get3() // Named Member 를 사용합니다.
{
    return ("john", 30);
}

var ret1 = Get1();  // ret1 은 string
var ret2 = Get2();
var ret3 = Get3();
```

SECTION 4.

Nullable type

❑ 주요 학습 내용

> Nullable<T>
>
> ?, ??, ??=

null

Reference Type 은 **"값 없음"** 의미하는 null 을 사용할 수 있지만,

Value Type 은 **"null"** 값을 가질 수 없습니다.

```
// Reference Type
string s1 = "hello";    // 값을 가질 수도 있고
string s2 = null;       // 값이 없음을 나타내는 null 로 초기화 될 수 도
                        // 있습니다.
// Value Type
int n1 = 0;         // 반드시 임의의 값을 가져야 합니다.
int n2 = null;      // error. null 로 초기화 될 수 없습니다
```

C# 8.0 이후 환경에서 위 코드를 빌드하면 "string s2 = null" 부분에서 에러는 아니지만 경고가 발생할 수 있습니다.
이 경우 아래의 한 줄을 소스코드의 제일 위에 추가 하면 경고가 발생하지 않습니다.

```
#nullable disable
```

자세한 설명은 "nullable reference" 에서 배우게 됩니다.

하지만 Value Type 도 **"Nullable<T>"** 를 사용하면 null 로 초기화 될 수 있습니다

	null 이 될 수 있는가 ?
reference type	O
value type	X
nullable value type	O
nullable reference type	O

Nullable reference 개념은 C# 8.0 에서 추가된 개념입니다. C# 8.0 이전에 사용되던 개념을 먼저 이야기 하고 그 이후에 살펴 보도록 하겠습니다.

Nullable<T>

Nullable<T> 타입을 사용하면 Value Type 도 null 을 가질수 있습니다.

Nullable<T> 의 단축 표기법인 **"타입이름?"** 를 사용해도 동일한 효과를 볼 수 있습니다.

```
int n1 = null;             // error. value type 은 null 이 될 수 없습니다.
Nullable<int> n2 = null;   // ok. Nullable<int> 는 null 이 될 수 있습니다.
int? n3 = null;            // ok. int? 는 Nullable<int> 에 대한
                           // 단축 표기법입니다.
```

int? 를 사용하면 Value Type 도 null 이 될 수 있으므로 메소드를 만들 때 실패의 전달을 보다 명확하게 할 수 있습니다.

```
int? some_work(int data)
{
    // 이 메소드에서 특정 작업을 수행하다 실패했을때
    // 호출자에게 실패 했다는 것을 알려야 합니다.
    // 방법 #1. 예외를 사용해서 실패 했다는 사실 전달
    // (예외 문법은 뒤에서 배우게 됩니다. )
    // 방법 #2. 실패로 약속된 값(주로 -1 또는 0 사용) 반환
    // (C 언어 등에서 사용하는 방식)
    // 방법 #3. null 을 반환 할수도 있습니다.
    // (방법 #2 보다 훨씬 더 명확합니다.)
    return null;
}

var ret = some_work(30);    // ret 는 int? 타입입니다.

if ( ret == null )
{
    // ......
}
```

Reference Type 은 * **Nullable<T> 의 타입 인자로 전달될 수는 없습니다.**

하지만 **"타입이름?"** 를 사용할 수 있습니다. (C# 8.0 부터 지원)

```
Nullable<string> s1 = null;   // error. Reference Type 은 Nullable 의
                              // 타입인자로 사용될수 없습니다.
string s2? = null;            // ok. ReferenceType? 는 C# 8.0 ~ 부터
                              // 지원 합니다.
string s3 = null;             // ok. ReferenceType 은 ? 가 없어도 null 로
                              // 초기화 가능합니다.
```

그런데, Reference Type 은 ? 가 없어도 s3 과 같이 null 로 초기화 가능합니다.
그렇다면 string 과 string? 의 차이점은 뭘까요 ?
뒤에 나오는 "nullable reference" 라는 항목에서 설명 됩니다.
그전에는 "Nullable Value Type" 에 대해서만 살펴 보겠습니다.

Nullable<T> 의 원리

Nullable<int> 타입, 즉, int? 타입의 원리는 내부적으로

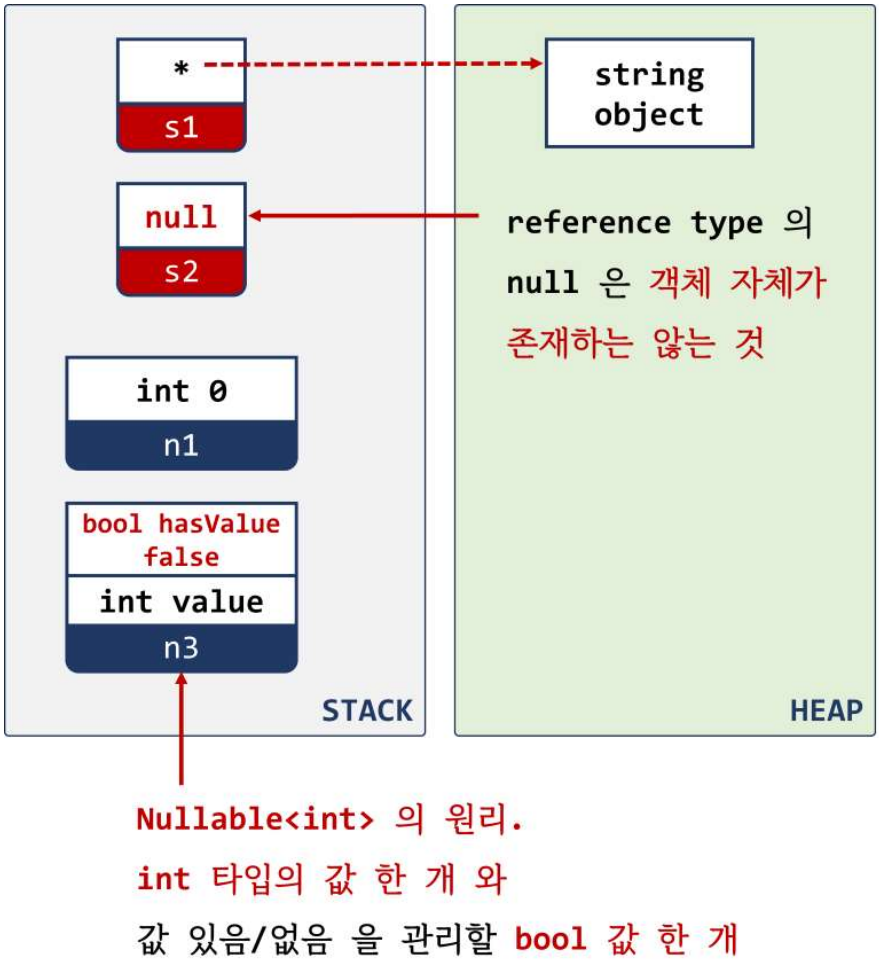
- **int 값 한개를 보관하기 위한 멤버가 있고**
- **값이 있음/없음을 보관하기 위한 bool 타입의 멤버**로 되어 있습니다.

아래 코드는 Nullable<T> 타입의 내부 소스 코드 입니다.

```
public partial struct Nullable<T> where T : struct
{
    private readonly bool hasValue;  // 값이 있음/없음 을 관리하는
                                     // bool 타입 멤버 hasValue
    internal T value;   // 한개의 값을 보관하는 T 타입 멤버 value
                        // Nullable<int> 로 사용했다면 int 값 한개 보관
    // ......
}
```

```
string s1 = "hello";
string s2 = null;   // ok. string type 은 null 이 될수 있습니다.

int n1 = 0;
//int n2 = null;    // error. int 타입은 null 이 될수 없습니다.

Nullable<int> n3 = null;    // ok
```

위 코드가 실행될 때 메모리 모양은 아래와 같습니다.



hasValue, Value Property

int? 타입을 사용하면서 null 로 초기화 하면 내부 멤버인 hasValue 가 false 로 설정 됩니다. int? 타입의 속성(Property, 읽기 전용)인 "HasValue" 를 통해서 확인할 수 있습니다.

```
using static System.Console;

int? n1 = 10;   // hasValue = True, value = 10
int? n2 = null; // hasValue = False

// hasValue 값은 "HasValue"
// value    값은 "Value" 라는 속성(property)로 접근 가능합니다.
WriteLine($"{n1.HasValue}");  // True
WriteLine($"{n2.HasValue}");  // False
WriteLine($"{n1.Value}");     // 10
WriteLine($"{n2.Value}");     // runtime error(exception),
                              // n2 는 값을 가지고 있지 않습니다.
```

GetValueOrDefault() Method

Nullable<T> 타입의 변수가 값이 없을때 "Value 속성" 에 접근하면 실행시간 오류(예외)가 발생합니다. 이 경우 **"GetValueOrDefault()"** 메소드를 사용하면 예외가 아닌 디폴트 값을 전달 받을수 있습니다.

```
int? n = null;  // 현재 n 은 값이 없습니다.
int v1 = n.Value; // 이렇게 사용하면 runtime-error(예외 발생) 입니다.
int v2 = n.GetValueOrDefault();  // 값이 있으면 값을 반환하고 값이 없으면
                                 // int 의 default 값인 0 반환
int v3 = n.GetValueOrDefault(3); // 값이 있으면 값을 반환하고 값이
                                 // 없으면 3 반환
```

Nullable<T> 핵심 정리

Nullable<T>(또는 "타입?") 타입을 사용할때 아래 3개의 멤버는 반드시 기억하고 있어야 합니다.

HasValue	읽기 전용 속성(read-only property)
Value	읽기 전용 속성(read-only property)
GetValueOrDefault()	메소드(method)

```
int? n = 10;
n.Value = 20;   // error. 읽기 전용입니다. 이렇게 사용할 수 없습니다.
n = 20;         // ok.    다른 값을 넣으려면 이렇게(= 연산자) 하면 됩니다.
```

주의

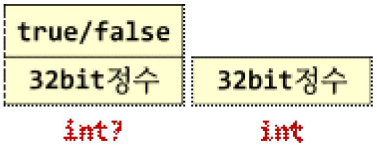
"Value, HasValue, GetValueOrDefault()" 멤버는 Nullable<T> 타입에만 존재하고, ReferenceType 에는 존재 하지 않습니다.

```
string  s1 = null;  // string 은 Nullable<T> 타입이 아니지만
                    // Reference Type 이므로 null 로 초기화 가능합니다.

// 하지만 아래 3 개의 멤버는 Nullable<T> 가 제공하므로 string 에는 없습니다.
s1.Value;               // error.
s1.HasValue;            // error.
s1.GetValueOrDefault();// error.
```

int vs int?

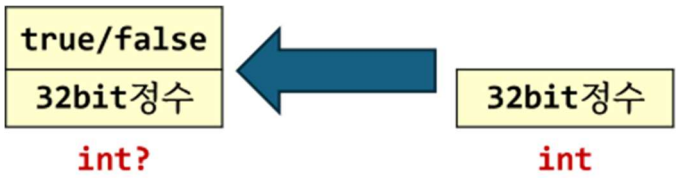
int 타입은 32bit 정수값 한개 보관 하지만,

int? 타입은 32bit 정수값 한개 와 bool 값 한개를 보관 합니다.

int	32bit 정수값 한개 보관	
int?	32bit 정수값 한개 + bool 값 한개 보관	

int? = int

int 변수를 int? 변수에 넣는 것은 결국 아래 그림과 같습니다.



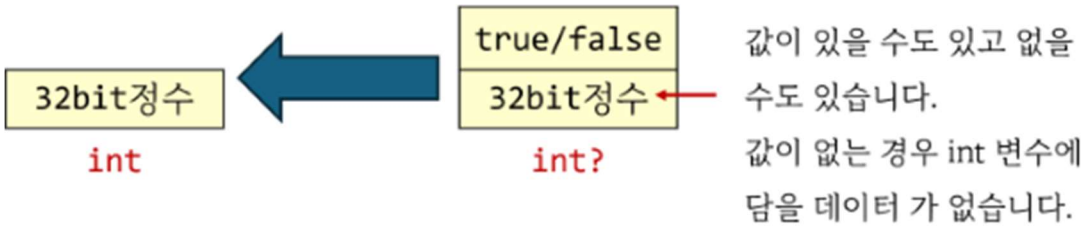
int? 타입은 int 안에 있는 모든 데이타를 담을 만큼 충분한 크기를 가지고 있으므로

int 를 int? 에 넣는 것은 항상 성공합니다.

```
int? n1 = null; // n1.hasValue = false
int? n2 = 10;   // n1.hasValue = true, n1.value = 10 으로 설정됩니다.
int v = 20;  // v 는 int? 가 아닌 int 입니다.
int? n3 = v; // 항상 성공합니다.

// 초기화가 아닌 대입도 항상 성공합니다.
n1 = v;   // n1.Value = 20 로 설정되고
          // n1.hasValue 가 false => true 로 변경됩니다.
n2 = v;   // n2.Value = 20 으로 변경
```

int = int?

int? 타입에는 값이 있을수도 있고 없을수도 있습니다.



따라서, int 변수를 초기화(대입)할때 우변에 int? 변수를 그냥 사용하는 것은 에러 입니다

```
int? n1 = 10;   // n1 은 값을 가지고 있습니다
int? n2 = null; // n2 는 값을 가지고 있지 않습니다.

// int? 타입의 변수가 값을 가지고 있던, 가지고 있지 않던
// int 변수에 바로 담을 수는 없습니다.
int v1 = n1;    // error.
int v2 = n2;    // error.
```

int? 타입의 변수에 있던 값을 int 변수에 담으려면 아래와 같은 방법들을 사용해야 합니다.

```
int? n1 = 10;
int? n2 = null;

// 방법 #1. 캐스팅, 이 방식은 int? 변수가 null 이면 실행시간 예외발생
int v1 = (int)n1; // ok
int v2 = (int)n2; // runtime error(예외) 발생

// 방법 #2. int? 타입의 Value 속성 사용
int v3 = n1.Value;
int v4 = n2.Value; // runtime error(예외) 발생

// 방법 #3. HasValue 속성을 조사후 얻기 ( if 또는 ?: 연산자 )
int v5 = n2.HasValue ? n2.Value : 0;
```

```
// 방법 #4. GetValueOrDefault() 메소드 사용
int v6 = n1.GetValueOrDefault();    // n1 이 null 인 경우 0 반환
int v7 = n2.GetValueOrDefault(3);   // n2 이 null 인 경우 3 반환

// 방법 #5. ?? 연산자 사용
int v8 = n2 ?? 3;   // 위 코드와 동일 합니다. n2 != null ? n2 : 3
                    // 별도의 항목에서 좀더 자세히 이야기 합니다.
```

다양한 방법이 있는데, 안전한 코드를 작성하기 위해서는 **"방법 #3 ~ #5"** 를 사용하는 것이 좋습니다.

방법 #5 의 ?? 연산자는 이어지는 항목에서 좀더 자세히 살펴 보겠습니다.

null check, ??, ??=, ?

null check

변수(객체)가 null 인지 조사하려면

- == 연산자 로 조사
- HasValue 속성으로 조사

그런데, HasValue 속성은 Nullable 에서만 사용가능하고, Reference Type 에서는 사용할수 없습니다.

```
int? n1 = null;
string s1 = null;

// 방법 #1. == 연산자로 조사.
if (n1 == null) { } // ok
if (s1 == null) { } // ok

// 방법 #2. HasValue 속성 사용.
if (n1.HasValue) { } // ok
if (s1.HasValue) { } // error. HasValue 속성은 Nullable<T> 타입만 사용가능
```

Null 병합연산자(null-coalescing operator, ??, ??=)

null 가능 타입(Reference Type 또는 Nullable<T>)에서

객체가 null 일때 default 값을 사용하려면

GetValueOrDefault() 메소드 사용	Nullable<T> 타입만 사용 가능하고, Reference Type 은 사용할수 없습니다.
?? 연산자 사용	Nullable<T> 타입 및 Reference Type 모두 사용가능 합니다.

```
int? n1 = null;
string s1 = null;

// A. GetValueOrDefault() 메소드는 Nullable<T> 타입만 사용가능합니다.
int    n2 = n1.GetValueOrDefault(3);        // ok
string s2 = s1.GetValueOrDefault("abcd");  // error. Nullable<T> 타입만
                                            // 사용 가능 합니다.
// B. ?? 연산자는 Nullable<T> 및 Reference Type 모두 사용가능 합니다.
int    n3 = n1 ?? 3;
string s3 = s1 ?? "abcd";

// C. ?? 뿐 아니라 ??= 연산자도 제공됩니다.
s1 ??= "abcd";  // s2 = s2 ??
n1 ??= 3;

// D. ??= 는 Reference Type 이나 Nullable<T> 타입만 가능합니다.
n2 ??= 3;   // error. n2 는 int? 가 아닌 int 입니다.
```

Null 조건부 연산자(null-conditional operator, ?, ?[])

null 인 객체에 대해서 메소스를 호출하는 것은 runtime error(예외) 가 발생합니다.

```
string s1 = "hello";
string s2 = null;

var ret1 = s1.ToString(); // ok
var ret2 = s2.ToString(); // runtime error(예외) 발생
```

객체를 안전하게 사용하려면 null 인지 확인하고 사용하는 것이 좋습니다.

?, ?[] 연산자가 null 이 아닌 경우만 객체에 접근하기 위한 연산자 입니다.

```
string s1 = null;
int arr[] = null;

// ? 와 ?[] 연산자
// 오른쪽 주석을 잘 생각해 보세요
string r1 = s1?.ToString();   // s1 != null ? s1.ToString() : null
int? r2 = arr?[0];            // arr != null ? arr[0] : null
```

nullable reference

null 상태를 가진 객체를 사용하는 것은 위험 합니다. **(runtime error 발생)**

되도록이면 모든 객체는 null 이 아닌 유효한 상태 가지도록 하는 것이 안전합니다.
따라서, null을 가질 수 없는 타입을 사용하는 것이 안전합니다.

하지만 가끔은 객체 없음을 뜻하는 **"null 이 필요한 경우도 있습니다."**

C# 8.0 이전

Value Type	null 가능 여부를 사용자가 선택할 수 있습니다. int : null 상태가 필요 없을 때 사용 int? : null 상태가 필요 할 때 사용
Reference Type	항상 null 이 가능합니다.

즉, reference type 의 경우, null 이 필요 없는 변수인 경우에도 null 을 허용하기 때문에 실수가 발생할 수도 있습니다.

C# 8.0 부터는 reference type 도 null 가능/불가능 타입으로 분리할 수 있도록 하기 위해 nullable reference type 이라는 개념이 도입되었습니다.

nullable reference

C# 8.0 이후에 visual studio 나 dotnet new console 명령으로 프로젝트를 생성하면 프로젝트 파일에 아래와 같이 <Nullable> 항목이 추가됩니다.

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net8.0</TargetFramework>
    <ImplicitUsings>enable</ImplicitUsings>
     <Nullable>enable</Nullable>
  </PropertyGroup>

</Project>
```

Nullable 항목이 enable 로 되어 있으면

- string 은 null 불가능 타입이 되고
- string? 은 null 가능 타입이 됩니다.

즉, Nullable 항목의 enable/disable 여부에 따라 컴파일 결과가 달라집니다.

	enable	disable
string s1 = null;	에러는 아니지만, 경고 발생 Null 불가능 타입	string, string? 둘 다 모두 null 가능 타입
string? s2 = null;	Ok. 아무 문제 없음. Null 가능 타입	

Nullable 항목이 enable 로 되면 string 타입은 null 불가능 타입이 됩니다.

하지만 “string s = null” 코드가 컴파일 에러가 발생하는 것은 아니고 경고가 발생합니다.

#nullable enable

파일 단위로 nullable reference 의 사용여부도 결정할 수 있습니다

소스 파일의 제일위에 **“#nullable enable”** 또는 **“#nullable disable”** 을 표기하면 됩니다.

```
#nullable disable

// #nullable disable 을 사용했으므로
// 아래 2 줄의 모두 경고 없이 잘 컴파일 됩니다.
string s1 = null;
string? s2 = null;
```

SECTION 5.

Control Statement & operator

❑ 주요 학습 내용

> control statement
> foreach
> switch expression
> expression bodied
> is, as

Control statement

C# 언어의 제어문

2개의 조건문	if, switch
4개의 반복문	while, do ~ while, for, foreach

C# 제어문 특징

실행할 문장이 한줄이면 {} 는 생략가능 하지만, {}를 사용하는 것을 권장.

조건식은 **"반드시 bool 타입"**이어야 한다.

foreach

foreach 를 사용하면 열거 가능한(열거자, enumerator 를 가진 타입) 타입의 객체가 가진 요소를 순차적으로 열거할 수 있습니다.

```
using static System.Console;

int[] x = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};

foreach( int e in x )
{
    Write($"{e}, ");
}
WriteLine();

string s = "ABCDEFGHIJ";

foreach( char e in s )
{
    Write($"{e}, ");
}
```

switch expression

C# 8.0 부터는 “switch statement(문장)” 뿐 아니라 **“switch expression(표현식)”** 개념 도 제공 합니다.

```
int dayofweek = 1;

string s1 = "";

switch(dayofweek)
{
    case 0: s1 = "sun"; break;
    case 1: s1 = "mon"; break;
    case 2: s1 = "tue"; break;
    default : s1 = "unknown"; break;
}

// 위 코드는 아래 처럼 만들수 있습니다
string s2 = dayofweek switch
            {
                0 => "sun",
                1 => "mon",
                2 => "tue",
                _ => "unknown"
            };
```

expression bodied

메소드의 구현이 한 문장으로 되어 있는 경우

중괄호({}) 대신 “=> 연산자” 를 사용해서 “한개의 표현식으로 메소드 몸체”

로 구현할 수 있습니다.

일반적인 메소드 구현	void SayHello()
	{
	    WriteLine("Hello");
	}

Expression bodied 구현	void SayHello() => WriteLine("Hello");

```
class Point
{
    private int x = 0;
    private int y = 0;

    // expression bodied 기법으로 만들어진 메소드 입니다.
    public Point(int a, int b) => ( x, y) = (a, b);
    public int  GetY() =>  y;
    public void SetY(int value) => y = value;
}

class Program
{
    public static void Main()
    {
        Point p = new Point(1,2);
        p.SetY(3);
        var n = p.GetY();
    }
}
```

is, as operator

is 연산자	객체가 가리키는 것이 특정 타입이 맞는지 조사
as 연산자	객체를 다른 타입으로 캐스팅, 실패시 null 반환

```
using static System.Console;

string s = "hello";
object obj = s;

// 방법 #1. 캐스팅
string s1 = (string)obj;   // obj 가 가리키는 것이
                           // string 이 아니라면 예외 발생

// 방법 #2. is 연산자로 조사후 캐스팅
if ( obj is string)
{
    string s2 = (string)obj;
}

// 방법 #3. 가장 편리한 방법. 권장
if (obj is string s3)      // obj 가 string 인지 조사 후,
                           // 맞다면 string 으로 캐스팅해서 s3 에
{
    // s3 사용
}

// 방법 #4. as 연산자 사용
string s4 = obj as string; // obj 가 string 이 아닌경우 null 반환
```

SECTION 6.

Method

❑ 주요 학습 내용

> Named argument
>
> Optional parameter
>
> Params
>
> Parameter modifier - ref, out, in, ref readonly

Named argument

메소드 호출 코드 만으로는 인자 값의 의미가 명확하지 않은 경우가 있습니다.

```
// 아래 코드의 인자의 의미를 생각해 보세요
SetRect(10, 20, 30, 40);

// 10, 20 은 x, y 로 추측 할수 있습니다.
// 30, 40 은 x2, y2 일까요 ? width, height 일까요 ?
```

메소드의 **"인자에 이름을 사용"** 할 수 있다면 보다 명확한 코드를 작성할 수 있습니다.

```
SetRect(x:10, y:20, width:30, height:30);
```

C# 은 메소드 인자를 전달에 2가지 방법을 사용할 수 있습니다.

```
// 방법 #1. positional argument
//         전통적으로 사용하던 방식 입니다.
SetRect(10, 20, 30, 40);

// 방법 #2. named argument
SetRect(x:10, y:20, width:30, height:30);
```

규칙

Positional argument 와 Named argument를 사용할때 다음과 같은 규칙이 있습니다.

- #1. Named arguments 사용시 인자의 순서를 변경할 수 있다.
- #2. Named arguments 와 Positional arguments 를 섞어서 사용할 수 있다.
- **#3. Named arguments 의 순서를 변경한 경우는 이어지는 인자로는 Positional arguments 를 사용할 수 없다.**

```
// #1. positional argument 와 named argument 혼합해서 사용하는 경우
//    인자의 순서를 변경하지 않은 이상 자유롭게 사용가능.
SetRect(10, 20, width:30, height:30);

SetRect(10, y:20, 30, height:30);
SetRect(x:10, 20, width:30, 30);

// #2. named argument 를 사용하는 경우 인자의 순서 변경 가능
SetRect(width:30, height:30, x:10, y:20);

// #3. 인자의 순서를 변경한 경우
//    named argument 뒤에 positional argument 가 올수 없습니다.
//    positional 은 반드시 named argument 앞에만 있어야 합니다.
SetRect(10, 20, height:30, width:30);   // ok
SetRect(10, height:30, y:20, width:30); // ok

SetRect(y:10, x:20, 30, 30);            // error
SetRect(y:10, x:20, 30, height: 30);    // error
```

Optional Parameter

메소드 호출시 값을 전달하지 않으면 미리 지정된 값을 사용하도록 할 수 있습니다.

C++등의 언어 에서는 **"default parameter"** 라고 불리는 개념입니다.

named argument 와 같이 사용하면 보다 편리하게 인자를 전달할 수 있습니다.

```
using static System.Console;

class Program
{
    // 1 번째 인자 a : required parameter, 반드시 전달해야 합니다
    // b, c        : optional parameter, 생략 가능합니다.
    public static void M1(int a, int b = 0, int c = 0)
              => WriteLine($"{a}, {b}, {c}");

    public static void Main()
    {
        M1(10, 20, 30); // 10, 20, 30
        M1(10, 20);     // 10, 20, 0
        M1(10);         // 10, 0,  0

        // named argument 문법과 섞어서 사용하면
        // b 를 생략하고 c 만 전달 가능합니다.
        M1(10, c:20);   // 10, 0, 20
    }
    // 주의!
    // Optional parameter 뒤에 Require parameter 를 만들수 없습니다.
    public static void M2(int a, int b = 0, int c = 0) { }  // ok
    public static void M3(int a = 0, int b, int c = 0) { }  // error
    public static void M4(int a, int b = 0, int c) { }      // error
}
```

주의 할점 은, 마지막 인자 부터 차례대로만 디폴트 값 지정가능 합니다.

params

메소드에 인자의 갯수를 상황에 따라 다르게 보내고 싶다면 **"배열을 인자로 사용"** 하면 됩니다.

이때 C# 에서는 메소드 인자로 배열을 사용하는 방법을 2가지 형태로 제공합니다.

params 를 사용하지 않은 경우	params 를 사용하는 경우
M1(int[] ar){}	**M2(params int[] ar){}**

M2(new int[]{1,2,3});
M2(new[]{1,2,3});
M2([1,2,3]);
 	
M2(1,2,3); // error
M2(); // error
	M2(new int[]{1,2,3});
	M2(new[]{1,2,3});
	M2([1, 2, 3]);
	
	M2(1,2,3); // ok.
	M2(); // ok.

params 를 사용하는 경우, "M2(1,2,3)" 형태로 사용 가능 하다는 것이 핵심 입니다.

원리는 아래와 같습니다.

```
M2(1, 2, 3);// ok. 컴파일러가 M2(new int[]{1,2,3}) 로 변경
M2();       // ok. 컴파일러가 M2(new int[]{}) 로 변경
```

주의할 점은 params 를 사용한 인자는 메소드의 마지막 인자로만 사용할 수 있습니다.

```
void M3(int[] ar, int n) { }          // ok     일반 배열은
                                      //        1 번째 인자로 사용가능
void M4(params int[] ar, int n) { } // error  params 를 사용한 배열은
                                      //        마지막인자 만 가능
```

params 를 사용한 예제 입니다

```
class Program
{
    public static void M1(int[] ar) { }
    public static void M2(params int[] ar) { }

    public static void Main()
    {
        M1(new int[] { 1, 2 });
        M1(new int[] { 1, 2, 3 });

        M1(new int[] { 1, 2, 3 });
        M1(new[] { 1, 2, 3 });
        M1([1, 2, 3]);
        M1(1,2,3);  // error
        M1();       // error

        M2(new int[] { 1, 2, 3 });
        M2(new[] { 1, 2, 3 });
        M2([1, 2, 3]);
        M2(1,2,3);  // ok
        M2();       // ok
    }
}
```

Parameter modifier #1 - ref

메소드에게 인자 전달하는 것은 **"메소드 안에서 사용할 새로운 지역변수를 만들고 인자가 가진 값으로 초기화 하는 것"** 입니다.

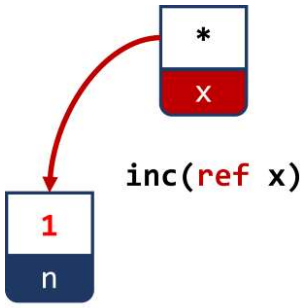
```
class Program
{
    public static void Inc(int x) { ++x; } // 복사본 x 가 증가 합니다.
    public static void Main()
    {
        int n = 0;
        Inc(n);  // 전달된 인자를 x 로 받는 것은
                 // int x = n 의 의미 입니다.
        WriteLine(n); // 0  원본 n 은 변화가 없습니다
    }
}
```

ref

"ref" parameter modifier 를 사용하면

- 메소드의 인자를 받을 때 복사본을 만들지 말고 reference 로 해달라는 의미
- **"메소드를 만들 때와 호출할 때 모두 표기"**해야 합니다.

```
class Program
{
    public static void Inc(ref int x) { ++x; }
    public static void Main()
    {
        int n = 0;
        Inc(ref n);
        WriteLine(n); // 1
    }
}
```



ref local

C# 초기 버전에서는 메소드 인자로만 ref 사용가능 했습니다.

C# 7.0 부터는 메소드의 지역변수와 반환 타입으로도 ref 사용가능 합니다.

“ref local”, “ref return” 이라는 문법 입니다.

```
public static void Main()
{
    int n = 0;

    ref int r = ref n; // r 은 n 을 가리키는 reference 입니다.

    r = 20;

    WriteLine(n); // 20
}
```

Parameter modifier #2 - out

ref modifier 사용하면 메소드 안에서 R/W 가 모두 가능합니다. 따라서 인자를 전달할 때 반드시 초기화된 인자를 전달해야 합니다.

```
class Program
{
    public static void Inc(ref int x)
    {
         ++x;
         // 위 코드는 "x = x + 1" 의 의미 이므로
         // x 에 대해서 "Read/Write" 를 모두 수행하는 코드입니다.
         // 반드시 초기화된 변수를 보내야 합니다.
    }
    public static void Main()
    {
        int n;

        Inc(ref n); // error. ref 는 초기화 되지 않은 변수는
                    //        인자로 사용할 수 없습니다.
    }
}
```

out

“out” parameter modifier 를 사용하면

- 메소드 안에서는 쓰기만 가능합니다.
- 따라서, 메소드에 인자를 전달할 때 초기화 하지 않은 변수도 가능합니다.

```
class Program
{
    // 아래 메소드는
    // a + b 의 결과는 반환 값으로 알려주고
    // a - b 의 결과는 out parameter 에 담아 주고 있습니다.
    public static int Add(int a, int b, out int ret)
    {
        ret = a - b;
        return a + b;
    }
    public static void Main()
    {
        int ret1;       // 초기화 되지 않았습니다.
        int ret2 = AddSub(5, 3, out ret1);  // ok
        int ret3 = AddSub(5, 3, out int ret4); // 이 코드도 가능합니다.
                                               // 이 위치에서 선언
    }
}
```

out vs ref

다음 예제의 코드와 주석을 읽고 정확히 이해해 보세요

```
class Program
{
    public static void no_modifier_parameter(int x)
    {
        // 1. 인자(x)에 대한 복사본 생성
        // 2. 인자(x)에 대해서 R/W 모두 가능
        int n = x;  // ok. Read
        x = 0;      // ok. Write
    }
    public static void out_parameter(out int x)
    {
        // 1. 복사본 생성 안됨
        // 2. 인자(x)에 대해서 Write 만 가능
        // 3. 인자(x)에 대해서 Write 가 없다면 에러
//      int n = x;  // error
        x = 0;      // 이 코드가 없으면 error
    }
    public static void ref_parameter(ref int x)
    {
        // 1. 복사본 생성 안됨
        // 2. 인자(x)에 대해서 R/W 작업 모두 가능
        // 3. 인자(x)를 사용하지 않아도 에러 아님
//      int n = x;  // ok
//      x = 0;      // ok
    }
    public static void Main()
    {
        // out_parameter(out 초기화 되지 않은 변수도 사용가능 );
        // ret_parameter(out 초기화 된 변수만 사용가능 );

        int n1;
        int n2 = 0;
        out_parameter(out n1); // ok
        out_parameter(out n2); // ok
        ref_parameter(ref n1); // error
        ref_parameter(ref n2); // ok

        out_parameter(out int n3); // ok
        // 여기서 부터 n3 사용가능.
    }
}
```

example

swap method

swap 은 인자로 전달 받은 2개의 변수의 값을 교환 하는 메소드입니다.

swap 메소드를 만들려면

- **인자의 복사본을 생성하면 안되고**
- **Swap 내부적으로 전달받은 인자 값을 R/W 모두 사용하게 됩니다.**
- 따라서, out 이 아닌 ref 사용해야 합니다.

```
using static System.Console;

class Program
{
    public static void Swap(ref int a, ref int b)
    {
        int t = a;
        a = b;
        b = t;
    }

    public static void Main()
    {
        int x = 1;
        int y = 2;

        Swap(ref x, ref y);

        WriteLine($"{x}, {y}");
                //   2,   1
    }
}
```

int.TryParse

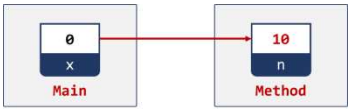
문자열을 정수(int)로 변환하려면 int.Parse() 또는 int.TryParse() 사용합니다.

2개 메소드의 차이점은

```
using static System.Console;

class Program
{
    public static void Main()
    {
        // 1. 변환에 성공하면 결과인 정수를 반환 하고
        // 2. 실패하면 예외가 발생됩니다.
        // 아래 코드는 예외 입니다.
        int n = int.Parse("Hello");

        // 1. 성공/실패 여부는 반환 타입으로 알려 줍니다.(bool)
        // 2. 변환 결과는 out 파라미터에 담아 줍니다.
        // 3. 실패시 out 파라미터는 0 으로 채워 줍니다.
        bool b = int.TryParse("Hello", out int n);

        WriteLine($"{b}, {n}");
                    // False, 0
    }
}
```

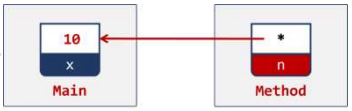
Parameter modifier #3 – in, ref readonly

value type 을 메소드에 인자로 전달하면



- **"복사본이 생성"** 됩니다.
- 메소드 안에서 인자를 수정하는 경우 **"복사본이 변경"** 되므로, 메소드 호출 시 전달한 **"원본 객체는 변경되지 않습니다.(안전하다)"**
- 하지만, value type 중 **"타입의 크기가 매우 큰 경우" "복사본이 생성이 성능에 많은 영향을 준다."**

value type 을 ref 로 전달하면



- reference 로 전달되므로 **"복사본이 생성되지 않는다."**
- 하지만 **"원본 객체도 변경"** 된다.
- **의도 하지 않은 실수로 원본 객체의 상태가 변경되는 버그가 발생할 수 있다.**

즉, 크기가 큰 Value Type 의 객체를 메소드 인자로 보낼 때

- ref 를 사용하지 않으면 "복사본" 에 대한 오버헤드가 있고
- ref 를 사용하면 원본이 수정될 위험(안정성)이 있습니다.

제일 좋은 방법은 **"읽기 전용 ref"** 로 전달하는 것입니다.

in, ref readonly

메소드 인자를 "읽기 전용 ref" 로 전달하려면 아래 2가지 방법이 있습니다.

in	C# 7.2 에서 추가, rvalue 도 전달 가능
ref read only	C# 12.0 에서 추가, rvalue 는 전달할 수 없다.

```
class Program
{
    public static void M1(in int n)
    {
        int a = n;   // ok
//      n = 10;      // error. in 사용시에는 읽기만 가능
    }

    public static void M2(ref readonly int n)
    {
        int a = n;   // ok
//      n = 10;      // error. ref readonly 사용시에도 읽기만 가능
    }

    public static void Main()
    {
        int x = 0;
        M1(in x);
        M1(x);      // ok. in 생략가능
        M1(3);      // ok. rvalue 도 가능.

        M2(ref x);
        M2(in x);   // ok, ref 대신 in 도 사용가능.
        M2(x);      // ref 생략시, 경고 발생
        M2(ref 3);  // error. rvalue 안됨
    }
}
```

SECTION 7.

class

❑ 주요 학습 내용

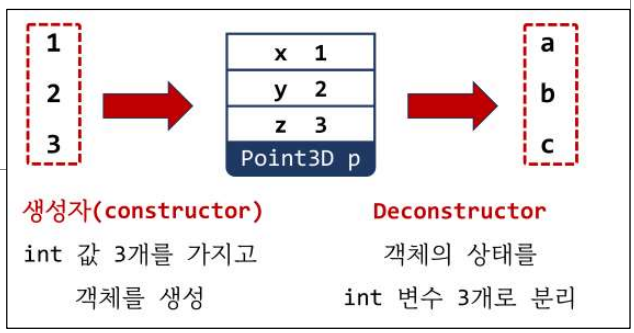
> deconstructor
>
> static field, method, class, constructor

deconstructor

객체를 초기화 하기 위해 “생성자(constructor)” 를 사용합니다.

반대 개념으로 객체의 모든 상태를 각각의 변수에 담기 위해서는 “deconstructor” 라는 메소드를 사용합니다.

```
using static System.Console;

class Point3D
{
    private int x, y, z;

    public Point3D(int a, int b, int c) => (x, y, z) = (a, b, c);

    public void Deconstruct(out int a, out int b, out int c)
        => (a, b, c) = (x, y, z);

    public void Deconstruct(out int a, out int b) => (a, b) = (x, y);
    public void Deconstruct(out int a)  => a = x;
}
class Program
{
    public static void Main()
    {
        Point3D p = new Point3D(1,2,3); // constructor

        (int a1, int a2, int a3) = p;   // deconstructor

        (int b1, int b2) = p;

//      (int c1) = p;   // error
//      int c1 = p;     // error
        p.Deconstruct(out int c);
    }
}
```



static field, static method

static field	객체당 한 개가 아닌 모든 객체가 공유하는 필드 객체를 여러 개 만들어도 오직 한 개만 메모리에 존재 객체의 속성이 아닌 클래스의 특징을 관리
static method	객체이름이 아닌 클래스 이름으로 호출하는 메소드 객체의 상태와 관계없이 클래스에 관련된 기능을 수행하는 메소드

아래 예제는 static field 와 static method 를 사용해서 생성된 Car 객체의 개수를 알아내는 코드 입니다.

```
using static System.Console;

class Car
{
    private static int cnt = 0;
    public Car() { ++cnt; }

    public static int getCount() { return cnt; }
}

class Program
{
    public static void Main()
    {
        WriteLine(Car.getCount()); // 0

        Car c1 = new Car();
        Car c2 = new Car();

        WriteLine(Car.getCount());  // 2
    }
}
```

static class

모든 멤버가 static 이라면 객체를 만들 필요 없이 클래스 이름으로만 사용하면 됩니다. 이런 경우 클래스 자체를 static 으로 만들면 객체를 생성할수 없게 됩니다.

```
class Ex1
{
    public static void M1() { }
    public static void M2() { }
}
static class Ex2
{
    public static void M1() { }
    public static void M2() { }
}
class Program
{
    public static void Main()
    {
       // Ex1, Ex2 의 모든 멤버는 static 이므로 객체를
       // 생성할필요는 없습니다.
       Ex1 ex1 = new Ex1();   // ok. Ex1 는 static 클래스는 아니므로
                              // 객체를 생성할 수 있습니다.

        Ex2 ex2 = new Ex2();  // error. static 클래스는 객체를
                              // 만들 수 없습니다.
    }
}
```

대표적인 C# 표준 라이브러리 중에 대표적인 static class 는 System.Math 와 System.Console 이 있습니다.

```
var v1 = new System.Math();      // error. static class
var v2 = new System.Console();   // error. static class
```

static constructor

constructor	객체를 생성할 때 마다 호출됨. 객체(필드)를 초기화 하기 위해 사용.
static constructor	생성자 앞에 static 을 붙이는 문법. **"객체를 처음 생성할 때 한번 호출"**됨. 이후 동일 타입의 객체를 추가로 생성해도 호출되지는 않음. 즉, 객체 마다 초기화가 아닌 **"클래스당 한번의 초기화"**를 위해 사용. **객체를 한 개도 생성하지 않으면 호출되지 않음.** **접근 지정자를 사용할 수 없고, 인자도 받을 수 없음**

```
using static System.Console;

class Train
{
    public Train() { WriteLine("Train()"); }
    static Train() { WriteLine("static Train()"); }
}

class Program
{
    public static void Main()
    {
        Train t1 = new Train();   // static Train
                                  // Train
        Train t2 = new Train();   // Train
        Train t3 = new Train();   // Train
    }
}
```

example

아래 예제는, 1번째 열차가 출발한 시간을 모든 열차 객체가 공유할 수 있게 하는 예제 입니다.

⇨ 열차 객체가 생성되면 바로 출발한다고 가정하면, 객체의 생성시간이 출발 시간.

⇨ 1번째 열차의 출발 시간은 모든 객체가 공유 하므로 **"static field 에 보관"**

⇨ 초기화 후에는 읽기만 하므로 **"readonly"** 사용.

⇨ **static field 는** 생성자에서 초기화 하는 경우 **"객체를 생성할 때 마다 계속 초기값이 지정"** 되므로 static 생성자에서 해야 합니다.

```
using System;
using static System.Console;

class Train
{
    private static readonly DateTime startTime;

    public  static string FirstTrainStartTime()
        => startTime.ToLongTimeString();

    public Train() { }
    static Train() { startTime = DateTime.Now;}
}
class Program
{
    public static void Main()
    {
        Train t1 = new Train();
        WriteLine($"{Train.FirstTrainStartTime()}");

        Train t2 = new Train();
        WriteLine($"{Train.FirstTrainStartTime()}");
    }
}
```

SECTION 8.

Property

❑ 주요 학습 내용

> property
>
> auto-implemented property

Getter/Setter

사람(Person) 클래스를 만드는 데, Person 객체의 나이를 변경하는 것이 대해서 생각해 봅시다.

- **방법 #1. “age를 public field” 로 작성**

  ⇨ 편리하고, 가독성이 좋다.

  ⇨ **안전성 부족.**

  ⇨ **잘못된 값에 대한 방어를 할 수 없다.**

- **방법 #2. “SetAge 메소드(setter)” 만들어서 사용**

  ⇨ **잘못된 값에 대한 오류처리를 할 수 있다.**

  ⇨ **방법 #1 보다 가독성은 부족.**

```
class Person1
{
    public int age;
}
class Person2
{
    private int age;
    public int  GetAge(int a) => age;
    public void SetAge(int value)
    {
        if ( value > 0 )
            age = value;
    }
}
class Program
{
    public static void Main()
```

```
    {
        Person1 p1 = new Person1();
        Person2 p2 = new Person2();

        // #1. public field
        //     사용하기 쉽고 가독성도 좋지만 안전하지 않습니다.
        p1.age = 10;
//      p1.age = -5;    // 잘못된 데이타에 대해서 방어를 할수 없습니다.
        int n1 = p1.age;

        // #2. Getter/Setter
        //     안전하지만 public field 보다는 가독성이 좋지 않습니다.
        p2.SetAge(10);
        int n2 = p2.GetAge();
        p2.SetAge(-5);  // 잘못된 데이타를 SetAge 내부에서 조사하고 있습니다.
    }
}
```

- **다른 방법은 없을까 ?**

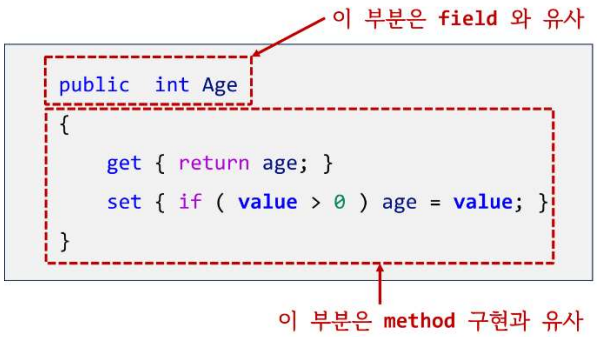
    ⇨ C# 에서는 **property** 라는 방법이 제공됨.

Property(속성)

Property(속성)

- Getter/Setter 를 자동으로 만드는 문법으로
- method 와 유사하게 만들지만 사용할 때는 field 처럼 사용합니다.

```
class Person
{
    private int age = 0;

    public  int Age
    {
        get { return age; }
        set { if ( value > 0 ) age = value; }
    }
}
class Program
{
    public static void Main()
    {
        Person p1 = new Person();

        p1.Age = 10;        // set {} 호출
        int n1 = p1.Age;    // get {} 호출
    }
}
```



Property example

Property 에 대한 다양한 예제 입니다. 주석을 읽어 보세요

```
class Example1
{
    private int data1 = 0;
    private int data2 = 0;
    private int data3 = 0;
    private int data4 = 0;
    private int data5 = 0;

    // #1. get/set 모두 지원
    public int Data1
    {
        get { return data1; }
        set { data1 = value; }
    }

    // #2. get/set 모두 지원
    //     expression bodied 구현
    public int Data2
    {
        get => data2;
        set => data2 = value;
    }

    // #3. get 만 가능, set 안됨
    public int Data3
    {
        get => data3;           // get_Data3()
    }

    // #4. set 만 가능. get 안됨
    public int Data4
    {
        set => data4 = value;   // set_Data4()
    }

    // #5. get 은 public, set 은 private
    public int Data5
```

```
    {
        get => data5;
        private set => data5 = value;
    }

    public Example1()
    {
//      data5 = 10;
        Data5 = 10; // ok
    }
}

class Program
{
    public static void Main()
    {
        Example1 e1 = new();
//      e1.Data5 = 10;  // error
    }
}
```

auto-implemented property

어떤 Property 가 어떠한 추가적인 로직 없이 **단순히 private field 를 Read/Write 만 한다면 "auto-implemented property"** 를 사용하면 편리하게 property 를 만들 수 있습니다.

```
class Example
{
    // #1. field 먼저 만들고, Property 를 직접 구현한 코드
    private int data1 = 0;

    public int Data1
    {
        set => data1 = value;
        get => data1;
    }

    // #2. auto-implemented property 를 사용한 코드
    public int Data2 { set; get; } = 0;
}
```

auto-implemented property 의 원리

사용자 코드	public int Data2 { set; get; } = 0
컴파일러가 생성한 코드	**private int <Data2>k__BackingField = 0;** **public int Data2** **{** **set => <Data2>k__BackingField = value;** **get => <Data2>k__BackingField;** **}**

indexer

인덱서(Indexer) 를 사용하면 객체를 배열 처럼 사용할 수 있습니다.

```
using static System.Console;

class Sentence
{
    private string[] words;

    public Sentence(string s) { words = s.Split();  }

    public string Text { get => string.Join(" ", words); }

    public string this[int idx]
    {
        get => words[idx];
        set => words[idx] = value;
    }
}
class Program
{
    public static void Main()
    {
        Sentence s = new Sentence("Dog is Animal");

        WriteLine( s[2] );  // Animal

        s[0] = "Cat";

        WriteLine(s.Text);  // Cat is Animal
    }
}
```

SECTION 9.

Inheritance

❑ 주요 학습 내용

> inheritance
> method override
> virtual method
> interface
> partial class
> extension method

inheritance

상속문법을 사용하면 기존 타입(클래스)를 확장해서 새로운 클래스를 만들 수 있습니다.

```
class Person
{
    private string name;
    private int age;
}
class Student : public Person
{
    private int id;
}
```

method override

method override 란 ?

- **기반 클래스가 가진 메소드를 “파생 클래스에서 다시 만드는 것”**
- 대부분의 객체지향 프로그래밍 언어에서 지원하는 공통적인 기능
- **실수가 아니라 의도적으로 override 하는 것이 라 고 알려 주기 위해 “new” 를 표기 합니다.**
- **new 키워드를 표기하지 않으면 “에러는 아니지만 경고” 발생**

```
using static System.Console;

class Animal
{
    public void Cry() { WriteLine("1. Animal Cry");}
}
class Dog : Animal
{
    // A. method override, new 가 없어도 에러는 아니지만 경고가 발생합니다.
//  public     void Cry() { WriteLine("2. Dog Cry");}
    public new void Cry() { WriteLine("2. Dog Cry");}
}
class Program
{
    public static void Main()
    {
        // B. Animal 객체를 생성해서 Cry() 를 호출하면 Animal Cry() 가 호출
        Animal a = new Animal();
        a.Cry();    // "1. Animal Cry"

        // C. Dog 객체를 생성해서 Cry() 를 호출하면 Dog Cry() 가 호출
        Dog    d = new Dog();
        d.Cry();    // "2. Dog Cry"
    }
}
```

virtual method

Animal 클래스에 Cry() 메소드가 있는데, Dog 가 override 했다고 할 때, 아래 코드의 주석을 생각해 보세요

```
public static void Main()
{
    Animal ad = new Dog();
    ad.Cry();   // 이 코드는 어떤 메소드를 호출하는 것이 맞을까요 ?
                // 1. "Animal Cry"
                // 2. "Dog Cry"
}
```

Function binding

static binding	컴파일러가 “컴파일 시간에 호출”을 결정 컴파일러는 ad가 실제로 어느 객체를 가리키는지는 컴파일 시간에 알 수 없다. reference 인 “ad의 타입으로” 메소드 호출을 결정. “Animal Cry” 호출 빠르지만 논리적이지 않다. C++/C# 의 non virtual method (function)
dynamic binding	컴파일 시간에는 ad 가 가리키는 곳을 조사하는 기계어 코드를 생성“실행 시간에 ad 가 가리키는 곳을 조사 후 실제 메모리에 있는 객체에 따라 메소드 호출 결정” Dog 객체가 있었다면 “Dog cry” 호출 느리지만 논리적인 동작 java, python 등의 대부분의 객체지향 언어가 사용 C++/C# 의 virtual method(function)

메소드를 **"dynamic binding"** 되도록 하려면

⇨ **기반 클래스에서 메소드를 만들 때 "virtual" 표시**

⇨ **파생 클래스에서 override 할 때 "new 가 아닌 override" 표시**

static binding	class Animal
	{
	    public **virtual** void Cry() { }
	}
	
	class Dog : Animal
	{
	    public **override** void Cry() { }
	}

dynamic binding	class Animal
	{
	    public void Cry() { }
	}
	
	class Dog : Animal
	{
	    public **new** void Cry() { }
	}

```
using static System.Console;

class Animal
{
    public         void Cry1() { WriteLine("1. Animal Cry1");}
    public virtual void Cry2() { WriteLine("1. Animal Cry2");}
}
class Dog : Animal
{
    public new      void Cry1() { WriteLine("2. Dog Cry1");}
    public override void Cry2() { WriteLine("2. Dog Cry2");}
}
class Program
{
    public static void Main()
    {
        Animal ad = new Dog();
        ad.Cry1();  // static binding
```

```
        ad.Cry2();  // dynamic binding
    }
}
```

new, override

메소드가 virtual 이 아닌 경우

⇨ 파생 클래스에서는 **"new"** 만 사용가능

메소드가 virtual 인 경우

⇨ 파생 클래스에서는 **"new"** 와 **"override"** 모두 사용 가능

new	기반 클래스 메소드 와는 완전히 다른 함수라고 알려주는 것. "bd.M3()" 는 Base 의 M3() 메소드 호출
override	기반 클래스 메소드를 override. "bd.M4()" 는 Derived 의 M4() 메소드 호출. dynamic binding

```
using static System.Console;

class Base
{
    public void M1() {}
    public void M2() {}
    public virtual void M3() { WriteLine("Base M3");}
    public virtual void M4() { WriteLine("Base M4");}
}

class Derived : Base
{
    public new      void M1() {} // ok
//  public override void M2() {} // error
    public new      void M3() { WriteLine("Derived M3");}
    public override void M4() { WriteLine("Derived M4");}
}

class Program
```

```
{
    public static void Main()
    {
        Base bd = new Derived();
        bd.M3(); // Base M3
        bd.M4();
    }
}
```

interface

강한 결합 (tightly coupling)

아래 코드는 People 이 Camera 를 사용할 때 “Camera” 라는 클래스이름을 직접 사용합니다. 어떤 클래스 사용시 “클래스 이름을 직접 사용” 하는 것을 **“강한 결합(tightly coupling)”** 이라고 합니다.

```
class Camera
{
    public void Take() => WriteLine("Take a picture");
}
class People
{
    public void Use(Camera c) => c.Take();
}
```

만약 위 코드에 “HDCamera” 라는 새로운 클래스가 추가 되면 어떻게 될까요 ?

People 클래스가 새로운 HDCamera 를 사용할 수 있게 하려면 아래와 같이 수정(추가) 되어야 합니다

```
class People
{
    public void Use(Camera c) => c.Take();
    public void Use(HDCamera c) => c.Take(); // 새롭게 추가!
}
```

시스템에 새로운 요소가 추가되었을 때 기존에 있던 코드가 수정되는 것은 좋은 디자인은 아닙니다.

약한 결합 (loosely coupiling)

새로운 카메라가 추가되어도 카메라를 사용하는 코드가 변경되지 않게 하려면 카메라가 지켜야 하는 규칙을 먼저 설계하고,

- 카메라 사용자(People) 도 규칙 대로 사용하고
- 카메라 설계자 도 규칙대로 설계하면 됩니다.

C# 에서 이런 규칙을 설계하는 문법이 “interface” 입니다.

```
using static System.Console;

interface ICamera
{
    void Take();
}
class Camera : ICamera
{
    public void Take() => WriteLine("Take a picture");
}
class HDCamera : ICamera
{
    public void Take() => WriteLine("Take a HD picture");
}
class UHDCamera : ICamera
{
    public void Take() => WriteLine("Take a UHD picture");
}

class People
{
    // 다양한 종류의 카메라가 있지만(나중에 추가되어도)
    // 아래 코드는 수정될 필요 없습니다.
    public void Use(ICamera c) => c.Take();

}
class Program
{
    public static void Main()
    {
```

```
        People p = new People();
        Camera c = new Camera();
        p.Use(c);

        HDCamera hc = new HDCamera();
        p.Use(hc);

        UHDCamera uhc = new UHDCamera();
        p.Use(uhc);
    }
}
```

IComparable, ICloneable

C# 타입 중 크기 비교가 가능한 대부분의 타입에는 **"CompareTo()"** 메소드가 있습니다.

⇨ **"동일한 메소드 이름을 사용하도록 규칙(interface)이 먼저 설계"** 되어 있습니다.(IComparable)

객체의 복사본을 만들기 위해서는 **"Clone()"** 사용합니다.

⇨ 메소드 이름을 동일하게 하기 위해 인터페이스가 먼저 설계되어 있습니다.(ICloneable)

사용자가 만든 타입에 **"CompareTo, Clone" 메소드를 제공하고 싶다면,** 해당하는 인터페이스를 구현하면 됩니다.

```
class Label : IComparable,  ICloneable
{
    private string title;
    public Label(string s) => title = s;

    public int CompareTo(object? obj)
    {
        Label? lb = obj as Label;
        return title.CompareTo(lb?.title);
    }
    public object Clone() => new Label(title);
}
class Program
{
    public static void Main()
    {
        Label d1 = new Label("GOOD");
        Label d2 = d1;
        Label d3 = (Label)d1.Clone();
    }
}
```

Partial class

하나의 클래스를 **“여러 개의 파일로 나누어서 작성”** 할 수 있습니다

class 작성시 클래스 앞에 **“partial”** 키워드를 붙이면 됩니다.

a.cs

```
partial class Example
{
    public M1()
    {
    }
}
```

b.cs

```
partial class Example
{
    public M2()
    {
    }
}
```

하나의 클래스를 여러 개 파일로 나누어 작업하는 이유는

⇨ 하나의 클래스를 여러 명의 개별자가 분야별로 나누어 작성 할 때

⇨ **“코드 생성기가 자동 생성한 코드”** 와 **“인간 개발자가 만드는 코드를 분리”** 해서 작업 할 때(WPF 나 WinUI3 등의 GUI 라이브러리 사용에서 널리 사용)

partial method

partial method 문법을 사용하면

⇨ 메소드의 구현이 다른 partial class 에 있다면 **"정상적으로 메소드를 호출"**

⇨ 메소드의 구현이 제공되지 않으면 에러가 아니라 **"메소드를 호출하는 코드를 제거"**

a.cs

```
partial class Example
{
    // 호출하는 곳에도 partial 선언을
    // 제공해야 합니다.
    partial void Init();
    partial void Run();
    partial void Exit();

    public void Process()
    {
        Init();
        Run();
        Exit(); // A
    }
}
```

b.cs

```
partial class Example
{
    // 메소드 앞에 partial 을
    // 붙어야 합니다.
    partial void Init() { }
    partial void Run()  { }
}
```

위 코드에서 A 부분에서 **"Exit()"** 메소드를 호출하지만 **"Exit()"** 메소드의 구현은 없습니다. 이때 에러가 나오는 것이 아니라 A 부분의 호출하는 코드가 제거 됩니다.

extension method

extension method 란 ?

⇨ 기존에 이미 만들어져 있는 타입에

⇨ 기존 타입의 소스 코드를 변경(추가)하지 않고

⇨ 상속을 사용(파생 클래스를 생성)하지 않고

⇨ 다시 컴파일 하지도 않고

⇨ 새로운 메소드를 추가하는 문법

```
using static System.Console;

class Example
{
    public void Foo() => WriteLine("Example Foo");
}
// A. extension method 는
//    static class 의 static method 형태로 만들어야 합니다.
static class ExampleExtension
{
    // B. 인자 앞에 this 키워드를 적어야 합니다.
    public static void Goo( this Example e, int a)
        => WriteLine("Example Goo {0}", a);
}
class Program
{
    public static void Main()
    {
        Example e = new();
        e.Foo();
        e.Goo(3);   // extension method
        ExampleExtension.Goo(e); // C. 이렇게 호출해도 됩니다.
    }
}
```

example

아래 예제는 extension method 를 사용해서 “string” 클래스에 “WordCount” 라는 메소드를 추가하는 예제 입니다.

```
using static System.Console;

static class MyExtensions
{
    public static int WordCount(this string s)
    {
        return s.Split(new char[] { ' ', '.', '?' },
                    StringSplitOptions.RemoveEmptyEntries).Length;
    }
}
class Program
{
    public static void Main()
    {
        string s = "to be or not to be";

        int wc = s.WordCount(); // extension method 로 추가한 메소드
                                // MyExtensions.WordCount(s)
        WriteLine(wc);
    }
}
```

SECTION 10.

System.Object

❑ 주요 학습 내용

> System.Object
>
> ToString() method
>
> GetType() method

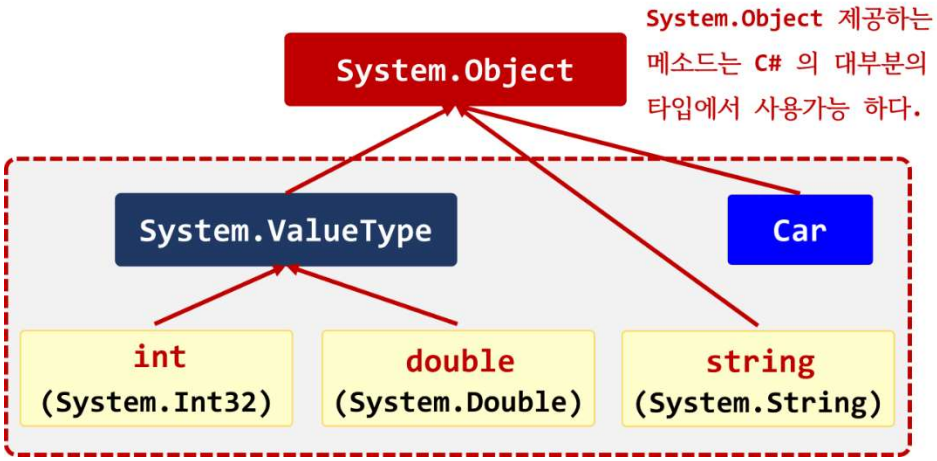
System.Object

C# 의 대부분의 타입은 “object(System.Object)”로 부터 상속 받습니다.

사용자가 class 나 struct 를 만들 때, 명시적으로 상속을 표기하지 않아도 자동으로 상속 받게 됩니다.

```
// 사용자가 만든 Car 클래스를 컴파일러가 오른쪽 주석 처럼 변경합니다
class Car  // class Car : object
{
    // 현재 Car 에는 어떠한 멤버도 없지만 object 로 부터 상속 받게 됩니다.
}
class Program
{
    public static void Main()
    {
        Car c = new Car();
        // ToString() 은 object 로 부터 상속 받은 메소드 입니다.
        var ret = c.ToString();
    }
}
```

따라서, object(System.Object) 가 제공하는 메소드는 C# 의 대부분의 타입에서 사용가능하고, C# 의 대부분의 객체는 object 타입으로 가리킬 수 있습니다



System.Object 멤버

System.Object 는 7개의 메소드를 제공합니다.

- **static method 2개**
- **instance method 5개(virtual method 3개)**

총 7개 중에 protected 가 1개, public 이 6개 입니다

```
public static bool Equals(object? oa, object? ob);
public static bool ReferenceEquals(object? oa,  object? ob);

public virtual bool Equals(object? obj);
public virtual int GetHashCode();
public virtual string? ToString();

public Type GetType();
protected object MemberwiseClone();
```

ToString() virtual method

ToString() 메소드는 객체의 상태를 문자열로 얻고 싶을 때 사용하는 메소드 입니다.

System.Object 의 기본 구현은 **"객체의 타입 이름(클래스 이름)"** 을 반환 합니다.

핵심은 virtual method 이므로 사용자가 override 해서 구현을 변경할 수 있습니다.

```
using static System.Console;

class Point
{
   private int x = 0;
   private int y = 0;
   public Point(int a, int b) => (x, y) = (a, b);

   // A. object 제공하는 ToString() 메소드를 override 하는 코드
   //    필드(x, y)의 상태를 문자열로 만들어서 반환 합니다.
   public override string? ToString()
   {
       return string.Format($"({x}, {y})");
   }
}

class Program
{
   public static void Main()
   {
       Point p = new Point(1, 2);

       string? s = p.ToString();
       WriteLine( s ); // A 의 ToString() 메소드가 없다면 "Point"
                       //                        있다면 "(1, 2)"
   }
}
```

GetType() instance method

객체의 타입정보를 가진 Type 객체를 얻을 때 사용합니다.

virtual method 가 아니므로 사용자가 override 하는 것은 아닙니다.

```
using System;
using static System.Console;

class Program
{
    // A. Main 메소드에서 string, int 를 전달했는데,
    //    받을때 object 로 받았습니다.
    public static void PrintType(object obj)
    {
        // obj 가 가리키는 타입을 알고 싶다면 Type 정보를 담은
        // 객체가 필요합니다.
        Type t = obj.GetType();

        // 이제 t 를 사용하면 obj 가 가리키는 곳에 있는 객체에 대한
        // 다양한 정보를 얻을수 있습니다.
        WriteLine($"{t.Name}, {t.FullName}");

        Type bt = t.BaseType;
        WriteLine($"{bt.Name}, {bt.FullName}");
    }

    public static void Main()
    {
        string s = "ABC";
        int    n = 10;
        PrintType(s);
        PrintType(n);
    }
}
```

typeof operator

타입의 정보를 담는 Type 객체는 아래의 2가지 방법을 얻을 수 있습니다.

```
Type t1 = 객체이름.GetType();
Type t2 = typeof(타입이름);
```

아래 코드는 객체가 int 타입인지 조사하는 코드 입니다.

```
using System;
using static System.Console;

class Program
{
    public static void Foo(object obj)
    {
       // obj 가 int 인지 확인하고 싶다.

       // 방법 #1. Type 객체를 얻어서 == 로 비교
       Type t1 = obj.GetType();
       Type t2 = typeof(int);
       WriteLine($"{t1 == t2}");

       // 방법 #2. is 연산자 사용
       WriteLine($"{obj is int}");
    }
    public static void Main()
    {
        int n = 10;
        string s = "ABC";

        Foo(n);
        Foo(s);
    }
}
```

Example

아래 코드는 특정 객체의 클래스 이름과 모든 기반 클래스의 이름(클래스 계층도)을 출력하는 코드 입니다.

```
using System;
using System.Reflection;
using static System.Console;

class Program
{
    public static void Main()
    {
        int n = 10;
        PrintHierachy(n);
    }
    public static void PrintHierachy(object obj)
    {
        if (obj == null)
        {
            Console.WriteLine("PrintHierachy : error. null reference");
            return;
        }
        Type? t = obj.GetType();
        Type? objType = typeof(object);

        while (true)
        {
            Console.Write($"{t?.FullName}");
            if (t == objType) break;
            Console.Write(" -> ");
            t = t?.BaseType;
        }
        Console.WriteLine();
    }
}
```

SECTION 11.

Equality

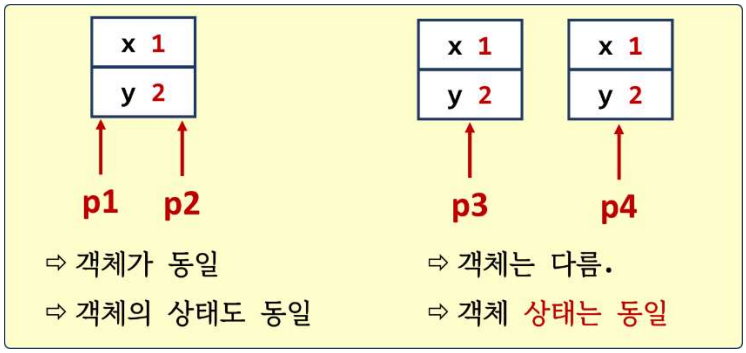
❑ 주요 학습 내용

> 동일성의 2가지 개념
>
> Reference type equality
>
> Value type equality

동일성의 2가지 개념

C# 에서 "2개의 객체가 동일한가 ?" 라는 개념에는 다음의 2가지 개념이 있습니다.

- **동일한 객체 인가?**
- **객체의 상태가 동일 한가 ?**



위그림에서 p1, p2 는 동일한 객체를 가리키고 있습니다.

p3, p4 는 서로 다른 객체를 가리키지만, 각 객체의 상태는 동일합니다.

동일성을 조사하는 4가지 방법

C# 에서는 아래의 4가지 방법으로 객체의 동일성을 조사할 수 있습니다

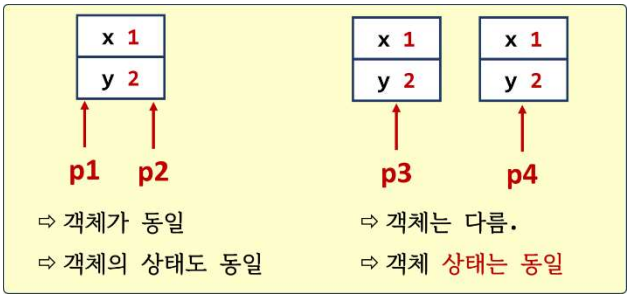
1	**a** == b	== 연산자 사용
2	a.**Equals**(b)	object 로 부터 상속된 instance method
3	object.**Equals**(a, b)	object 클래스의 static method
4	object.**ReferenceEquals**(a, b)	object 클래스의 static method

Reference Type 의 Equality

이번 항목에서는 reference type 에 대한 동일성에 대해서 살펴 봅니다.

아래와 같은 Point 클래스를 가지고 설명합니다.

```
 class Point
 {
    private int x = 0;
    private int y = 0;
    public Point(int a, int b)
          => (x, y) = (a, b);
}
```




```
class Program
{
    public static void Main()
    {
        // #1. p1, p2 는 동일한 객체를 가리킵니다.
        Point p1 = new Point(1,2);
        Point p2 = p1;

        // #2. p3, p4 는 동일한 객체는 아니지만 상태는 동일합니다.
        Point p3 = new Point(1,2);
        Point p4 = new Point(1,2);
    }
}
```

위 코드에서

- p1, p2 는 동일한 객체를 가리킵니다.
- p3, p4 는 동일한 객체는 아니지만, 객체의 상태는 동일합니다.

== 연산자 사용

객체가 reference type 일 때

■ == 연산자는 **"객체가 동일"** 한가를 조사 합니다.

```
WriteLine($"{p1 == p2}"); // true
WriteLine($"{p3 == p4}"); // false
```

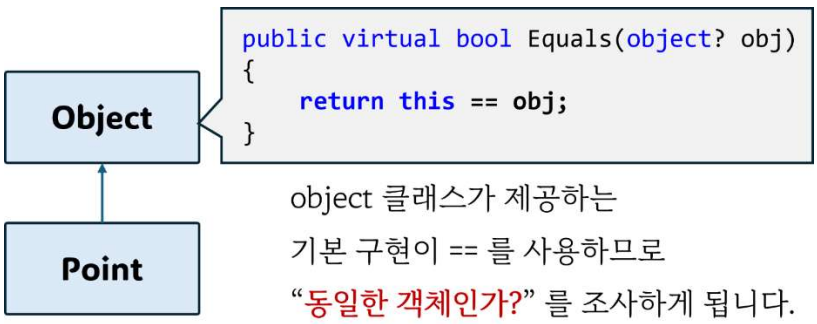
단, "연산자 재정의 문법을 통해서 == 연산자를 재정의" 할 수 있습니다.

이 경우 다른 의미를 가지도록 변경할 수 있습니다.
(object.ReferenceEquals() 배울때 자세히 설명)

p1.Equals(p2) 사용

object 클래스 안에는 Equals 라는 instance method 가 있습니다. 따라서, 대부분의 C# 타입은 Equals 라는 instance method 가 있습니다.

object 클래스 안에 있는 Equals instance method 는 다음과 같이 구현되어 있습니다.




```
WriteLine($"{p1.Equals(p2)}"); // true
WriteLine($"{p3.Equals(p4)}"); // false
```

결국 == 연산자를 사용한 것과 동일한 결과 입니다.

그런데, Equals() 는 가상 메소드 이므로 클래스 설계자가 자신의 의도에 맞도록 override 할 수 있습니다.

일반 적으로 클래스 설계자는 Equals() 가상 메소드를 override 해서 “상태가 동일한가?” 조사하도록 변경하는 경우가 많습니다.

```
using static System.Console;

class Point
{
    private int x = 0;
    private int y = 0;
    public Point(int a, int b) => (x, y) = (a, b);

    public override bool Equals(object? obj)
    {
        if ( obj == null || !(obj is Point) ) return false;
        Point p = (Point)obj;
        return x == p.x && y == p.y;
    }
    // Equals() 가상 메소드를 override 하는 경우
    // GetHashCode() 도 override 해야 합니다.
    public override int GetHashCode()
              =>  x.GetHashCode() + y.GetHashCode();
}
class Program
{
    public static void Main()
    {
        Point p1 = new Point(1,2);
        Point p2 = p1;
        Point p3 = new Point(1,2);
        Point p4 = new Point(1,2);

        WriteLine($"{p1 == p2}, {p3 == p4}"); // True, False
        WriteLine($"{p1.Equals(p2)}, {p3.Equals(p4)}"); // True, True
    }
}
```

object.Equals(p1, p2) static method

object 클래스에는 Equals() 라는 이름을 가지는 메소드가 2개 있습니다.

instance method 와 static method, 이번에는 **static method 버전**에 대해서 생각해 보겠습니다.

앞에서 배운 2가지 방법을 정리해 보면

p1 == p2	p1, p2 가 동일한 객체를 가리키는지 조사
p1.Equals(p2)	Object 의 기본 구현은 == 을 사용해서 조사 하지만, class 설계자가 override 해서 상태가 동일한가로 변경하는 경우가 많습니다.

그래서 상태가 동일한지를 조사하려면 == 가 아닌 **"p1.Equals(p2)"** 를 사용하면 됩니다.

그런데, 가장 좋은 방법은 무조건 "p1.Equals(p2)" 를 사용하지 말고

- 먼저 p1 == p2 를 조사해서 false 인 경우만
- 다시 p1.Equals(p2) 를 호출해서 조사

하는 것이 효율적입니다. == 연산자가 p1.Equals(p2) 보다 빠르게 동작하고, 동일한 객체라면 무조건 상태도 동일하기 때문입니다.

object.Equals() static method 가 아래와 같이 구현되어 있습니다.

```
public static bool Equals(object? objA, object? objB)
{
    if (objA == objB) return true;

    if (objA == null || objB == null) return false;

    return objA.Equals(objB);
}
```

결국 객체의 상태가 동일한가를 조사하려면 아래의 2가지 방법이 가능한데,

static method 버전을 사용하는 것이 좀더 효율적일수 있습니다.

p1.Equals(p2)	사용자의 구현에 따라 다를 수 있지만, 일반적으로는 모든 필드의 내용을 각각 비교
object.Equals(p1,p2)	1. p1 == p2 라면 true 2. p1, p2 중 한 개라면 null 이면 false 3. return p1.Equals(p2) 로 반환

object.ReferenceEquals(p1, p2) static method

== 연산자를 사용하면 “동일한 객체” 인지를 조사할 수 있습니다

그런데, == 연산자는 class 설계자가 “연산자 재정의 문법으로 다시 만들수 있습니다.”

아래 코드는 == 연산자를 override 해서 객체의 동질성이 아닌 상태의 동일성을 조사하도록 만든 코드입니다.

```
using static System.Console;

class Point
{
    private int x = 0;
    private int y = 0;
    public Point(int a, int b) => (x, y) = (a, b);

    public static bool operator==(Point p1, Point p2)
        => p1.x == p2.x && p1.y == p2.y;

    public static bool operator!=(Point p1, Point p2)
        => return !( p1 == p2 );
}
class Program
{
    public static void Main()
    {
```

```
        Point p1 = new Point(1,2);
        Point p2 = new Point(1,2);

        // p1, p2 는 다른 객체지만 상태가 동일하면 아래 결과는 True 입니다.
        WriteLine($"{p1 == p2}"); // True
    }
}
```

== 연산자가 override 되어 있는 상태에서 상태가 동일성 말고, 객체가 동일한지 조사하려면 p1, p2 객체를 object 타입으로 캐스팅해서 == 연산을 적용하면 됩니다.

```
Point p1 = new Point(1,2);
Point p2 = new Point(1,2);

// 사용자가 == 를 override 했지만,
// object 타입으로 캐스팅 한후 사용하므로
// 사용자가 만든 == 를 호출하는 것이 아니라,
// == 연산자의 기본 동작인 동일한 객체인지를 조사하게 됩니다.
WriteLine($"{(object)p1 == (object)p2}"); // False
```

object.ReferenceEquals() static method 는 다음과 같이 구현되어 있습니다.

```
// 인자를 object 로 받고 있으므로 어떠한 타입의 객체를 전달해도
// 항상 객체가 동일한지를 조사하게 됩니다.
public static bool ReferenceEquals(object? objA, object? objB)
{
    return objA == objB;
}
```

따라서 아래 처럼 사용하면 항상 객체가 동일한지를 조사하게 됩니다.

```
object.ReferenceEquals(p1, p2);
```

정리

p1 == p2	기본적으로 객체가 동일한지를 조사. 하지만 class 설계자가 == 연산자를 override 해서 정책 변경 가능
p1.Equals(p2)	기본 구현은 == 연산자 를 사용해서 조사 가상 메소드 이므로 class 설계자가 override 해서 변경가능. 보통 상태가 동일한가로 변경하는 경우가 많음.
object. Equals(p1, p2);	1. p1 == p2 라면 true 2. p1, p2 중 한 개라도 null 이며 false 3. 1, 2 모두 만족하지 않으면 return p1.Equals(p2)
object. ReferenceEquals(p1, p2);	항상 동일한 객체인지 조사

string equality

string 은 reference type 입니다. 그런데, 내부적으로 == 연산자를 override 상태가 동일한지를 조사하도록 변경해 놓았습니다.

따라서, s1, s2 가 string 일 때

s1 == s2	s1, s2 의 상태(문자열)이 동일한지 조사.
object. ReferenceEquals(s1, s2)	s1, s2 가 동일한 객체인지 조사.

```
using static System.Console;

// s1, s2 는 상태는 동일하지만 다른 객체 입니다
string s1 = new string("AAA");
string s2 = new string("AAA");

// s3, s4 는 동일한 객체를 가리킵니다.
string s3 = new string("AAA");
string s4 = s3;

// s5, s6 는 동일한 객체입니다(intern pool)
string s5 = "AAA";
string s6 = "AAA";

WriteLine($"{s1==s2}, {object.ReferenceEquals(s1, s2)}");// True, False
WriteLine($"{s3==s4}, {object.ReferenceEquals(s3, s4)}");// True, True
WriteLine($"{s5==s6}, {object.ReferenceEquals(s5, s6)}");// True, True
```

Value Type Equality

이번 항목에서는 Value type 에 대한 동일성에 대해서 살펴 봅니다.

아래와 같은 Point struct를 가지고 설명합니다. 핵심은 class 가 아닌 Point 라는 점입니다.

```
struct Point
{
    private int x;
    private int y;
    public Point(int a, int b) => (x, y) = (a, b);
}
class Program
{
    public static void Main()
    {
        // p1, p2 는 다른 객체 입니다. 상태는 동일합니다.
        Point p1 = new Point(1,2);
        Point p2 = new Point(1,2);
    }
}
```

== 연산자 사용

Value Type 은 기본적으로 == 를 지원 하지 않습니다. 따라서 아래 코드는 에러입니다.

```
WriteLine($"{p1 == p2}"); // true
```

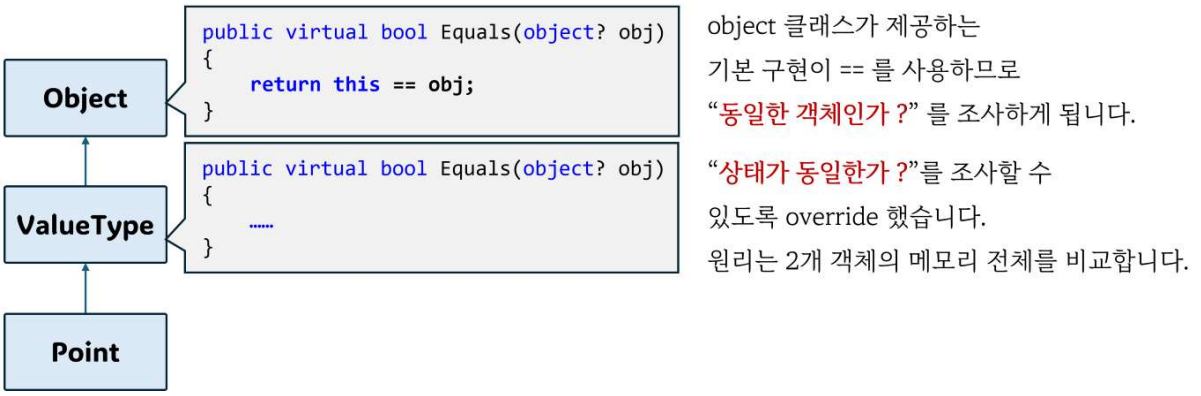
단, “연산자 재정의 문법을 통해서 == 연산자를 재정의” 할 수 있습니다.

p1.Equals(p2) 사용

모든 value type 은 ValueType 이라는 타입으로부터 상속 받게 됩니다.

그리고 ValueType 은 Equals() 가상 메소드를 override 하고 있습니다.

아래 그림을 생각해 보세요.



따라서, struct 은 Point 타입에 대해서 Equals() 가상 메소드를 2개 객체의 메모리를 통째로 비교해서 상태가 동일한지를 조사하게 됩니다.

```
Point p1 = new Point(1, 2);
Point p2 = new Point(1, 2);

// 아래 코드는 "상태의 동일성" 을 조사합니다.
WriteLine($"{p1.Equals(p2)}");  // True
```

"p1.Equals(p2)" 의 기본 구현 원리는 객체의 메모리 전체를 비교하는 것입니다.

그런데, Point 설계자가 비교하는 정책을 변경하고 싶다면 Equals() 가상 메소드를 override 하면 됩니다. (x, y 멤버중 x만 같으면 동일하게 하고 싶다 등..)

object.Equals(p1, p2), object.ReferenceEquals(p1, p2)

Value Type 의 객체 p1, p2 에 대해서

object.Equals(p1, p2)	결과는 맞게 나오지만 오버헤드가 있으므로 사용하지 않는 것이 좋습니다. **"Boxing/Unboxing"** 항목에서 자세히 설명
object. ReferenceEquals(p1, p2)	결과 자체가 잘못 나오게 됩니다. (항상 false) 절대 사용하면 안됩니다. **"Boxing/Unboxing"** 항목에서 자세히 설명

```
Point p1 = new Point(1, 2);

// p1, p1 이므로 동일한 객체를 보내지만 결과는 False 입니다.(잘못된 결과)
WriteLine($"{object.ReferenceEquals(p1, p1)}");  // False
```

위 코드의 결과라 False 인 이유는 **"Boxing/Unboxing"** 항목에서 자세히 설명 됩니다.

int type

Value Type 은 기본적으로 == 연산을 제공하지 않습니다.

int 타입은 Value Type 이지만 연산자 재정의 문법을 통해서 == 연산자를 제공합니다.
== 연산자는 상태의 동일성을 조사하도록 되어 있습니다.

```
int n1 = 10;
int n2 = 10;

WriteLine($"{n1 == n2}");  // True
```

SECTION 12.

Boxing, Unboxing

❑ 주요 학습 내용

> Boxing
>
> Unboxing
>
> Generic Interface

Boxing, Unboxing

C# 에는 다음과 같은 규칙이 있습니다.

1. value type 의 객체는 스택에 만들어 진다.

2. reference type 은 힙에 있는 객체를 가리 킨다.

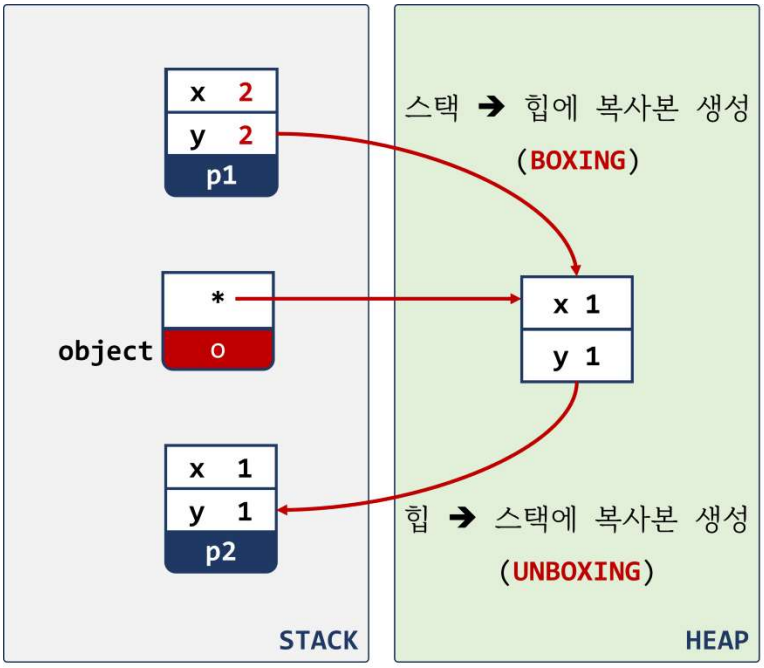
그런다면, value type 의 객체를 reference type 인 object 로 가리키면 어떻게 될까요 ?

```
using static System.Console;

struct Point
{
    public int X{set; get;} = 0;
    public int Y{set; get;} = 0;
    public Point(int a, int b) => (X, Y) = (a, b);
}
class Program
{
    public static void Main()
    {
        Point p1 = new Point(1, 1);  // A. p1 은 value type(struct) 이므로
                                     // 스택에 놓여 있습니다.
        object o = p1;  // B. 스택에 있는 객체 p1 을
                        // reference type 인 object 로 가리 키고 있습니다.
                        // 이 순간의 메모리 구조가 어떻게 될까요 ?
        Point p2 = (Point)o;    // C. 이순간의 메모리 구조는 어떻게 될까요 ?
        p1.X = 2;
        p1.Y = 2; // D. p1 객체의 상태를 변경하고 있습니다.
        WriteLine($"{p2.X}, {p2.Y}"); // E. 실행결과는 어떻게 될까요 ?
    }
}
```

C# 에서 value type 의 객체를 reference type 으로 가리키게 되면(위 코드에서 A 부분) **"힙에 복사본이 생성"**되게 됩니다. 이런 현상을 **"Boxing"** 이라고 합니다.

또한, B 와 같이 캐스팅하면 힙에 있는 객체를 복사한 p2가 스택에 만들어 지게 되는데 이런 현상을 **"Unboxing"** 이라고 합니다.

위 코드가 실행 되었을 때 의 메모리 그림이 다음과 같습니다. 코드와 그림을 잘 비교해 보세요.

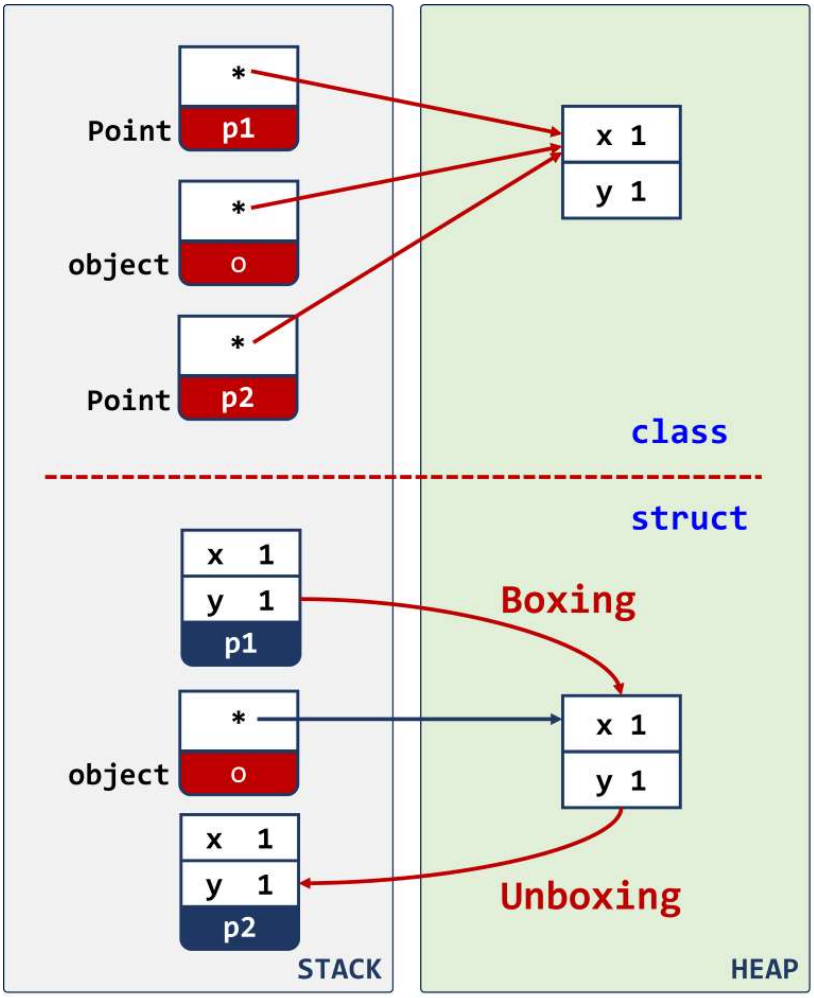


따라서, 위코드는 D 부분에서 p1 객체의 상태를 변경하지만 p2 는 별도의 복사본이므로 변경되지 않게 됩니다. 따라서 결과는 **"1, 1"** 이 나오게 됩니다.

class vs struct

만약 이전 예제에서 Point 가 struct 가 아닌 class 라면 객체는 힙에 놓이게 되므로 object 로 가리킬 때 Boxing 현상이 발생하지는 않습니다.

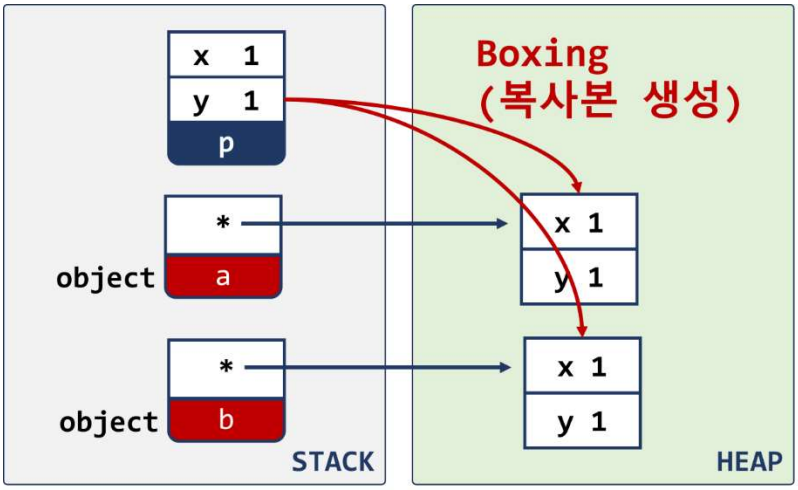
아래 그림을 참고해 보세요. Point 가 class 인 경우와 struct 일때의 메모리 구조 입니다.



object.ReferenceEquals() 와 Boxing

두개의 객체가 동일한 객체 인지 조사할 때는 “object.ReferenceEquals()” 를 사용하면 됩니다. 그런데, Value Type 에 대해서는 절대 사용하면 안됩니다. 항상 False 가 나오게 됩니다. 이유는 아래 코드를 생각해 보세요

```
// A. object.ReferenceEquals() 는 실제 아래 처럼 되어 있습니다.
//    인자 2 개가 모두 object 라는 것이 핵심 입니다.
bool MyReferenceEquals(object? a, object? b)
{
    return a == b;
}

// B. Point 가 struct 라면 아래 처럼 만들면 stack 에 생성됩니다.
Point p = new Point(1, 1);

// C. 동일한 객체를 전달하지만 object 로 받고 있으므로
//    힙에 복사 본이 생성됩니다.(Boxing)
//    이때 MyReferenceEquals 의 인자 a, b 가 각각
//    다른 복사본을 생성하게 됩니다. 아래 그림을 참고해 보세요
bool b = MyReferenceEquals(p, p);
```



IComparable vs IComparable<T>

C#에서 객체의 크기를 비교하려면

- **“CompareTo”** 메소드를 사용합니다.
- **“IComparable”** 인터페이스로 메소드 이름을 약속되어 있습니다.

IComparable 인터페이스는 다음과 같이 되어 있습니다.

```
public interface IComparable
{
    int CompareTo(object? obj);
}
```

Struct(value type) 와 IComparable

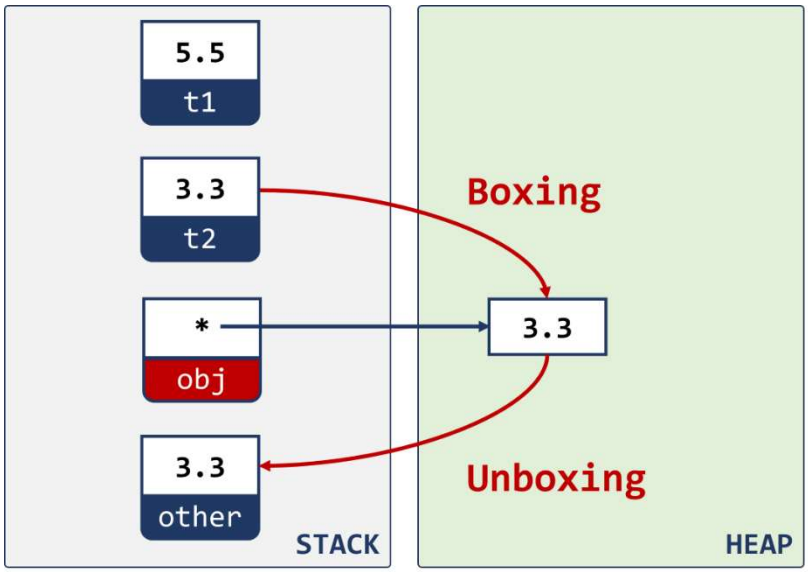
Temperature 라는 struct 를 설계 했는데, 크기를 비교하는 “ComparableTo” 메소드를 지원하고 싶다고 생각해 봅시다.

C# 의 규칙에 따라 IComparable 인터페이스를 구현하면 됩니다. 아래 완성된 코드 입니다. A 부분의 주석을 주의 깊게 읽어 보세요

```
struct Temperature : IComparable
{
    private double Value { set; get; } = 0;
    public Temperature(double val) => Value = val;

    // A. 아래 메소드의 인자가 object 라는 것이 핵심 입니다.
    //    Main 메소드의 B 에서 보낸 객체 t2 가 struct 이므로
    //    Boxing 현상이 발생합니다.
    //    성능 저하가 있습니다.
    public int CompareTo(object? obj)
    {
        if (obj == null) return 1;
```

```
        if ( obj is Temperature other)
            return Value.CompareTo(other.Value);
        else
           throw new ArgumentException("Object is not a Digit");
    }
}
class Program
{
    public static void Main()
    {
        Temperature t1 = new Temperature(5.5);
        Temperature t2 = new Temperature(3.3);

        WriteLine($"{t1.CompareTo(t2)}"); // B
    }
}
```

위 코드가 실행되었을 때 메모리 구조는 아래와 같습니다.



IComparable<T>

C# 1.0 에서 제공하는 인터페이스는 대부분 메소드 인자로 object 를 사용합니다.

```
public interface IComparable
{
    int CompareTo(object? obj);
}
```

단점은 value type(struct) 를 만들 때 이 인터페이스를 사용하면 Boxing, Unboxing 현상이 발생하게 됩니다.

C# 2.0 부터 Generic 문법이 도입되면서, 인터페이스도 Generic 인터페이스가 추가되었습니다.

기존에 있던 대부분의 인터페이스도 Generic 버전을 추가했습니다.

```
public interface IComparable<in T>
{
    int CompareTo(T other);
}
```

메소드 인자로 object 가 아닌 T 를 사용하므로 Boxing/Unboxing 현상이 발생하지 않습니다.

IComparable, IComparable<T>

Boxing/Unboxing 현상을 제거하기 위해 IComparable 보다는 IComparable<T> 를 구현하는 것이 좋습니다. 그런데, IComparable<T> 만 구현하는 경우 아래와 같은 경우에 에러가 발생합니다.

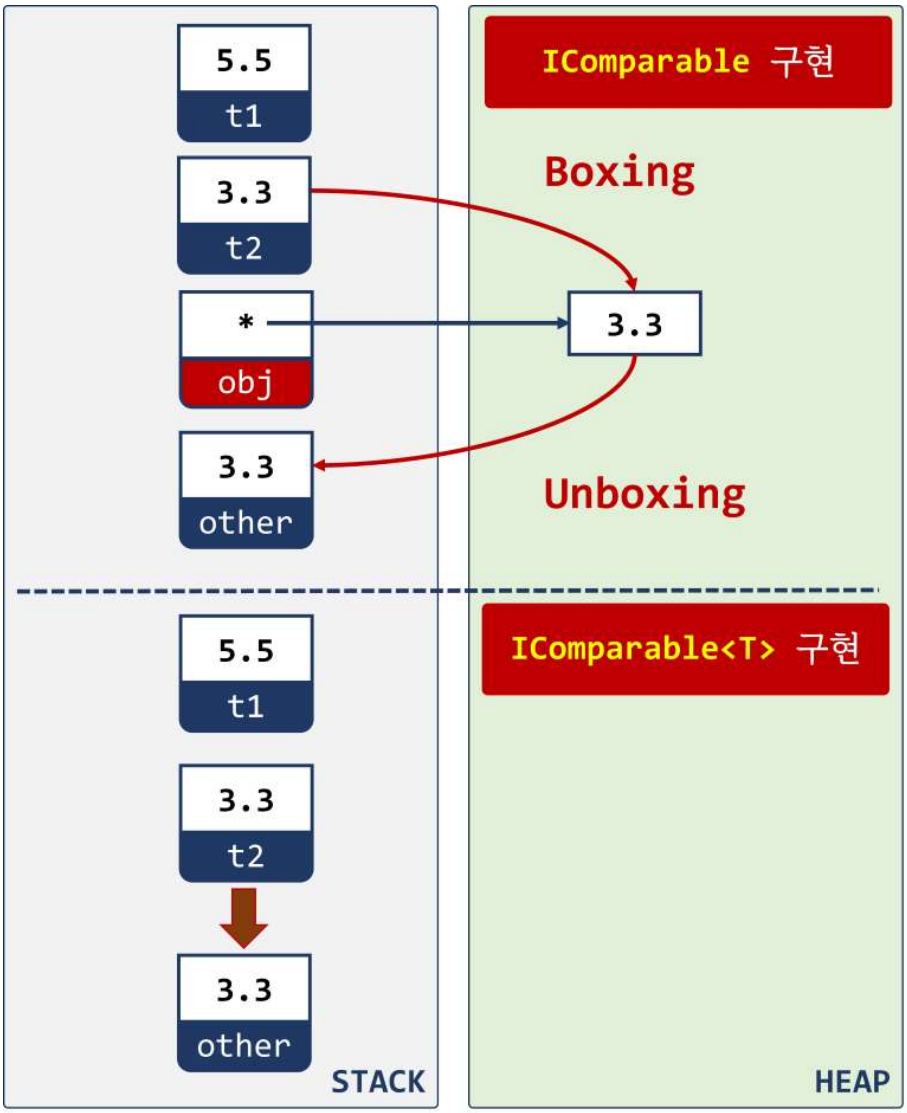
```
Temperature t = new Temperature(5.5);
object obj = new Temperature(3.3);

t1.CompareTo(o);    // IComparable<T> 만 구현한 경우
                    // ComparaTo(Temperature) 만 있으므로
                    // object 타입인 obj 를 받을수 없습니다.
                    // error.
```

따라서, 가장 좋은 코드는 IComparable 과 IComparable<T> 를 모두 구현하는 것이 좋습니다. 다음은 완성된 코드입니다.

```
using System;
using static System.Console;

struct Temperature : IComparable, IComparable<Temperature>
{
    private double Value { set; get; } = 0;
    public Temperature(double val) => Value = val;

    public int CompareTo(object? obj)
    {
        if (obj == null) return 1;

        if ( obj is Temperature other)
            return Value.CompareTo(other.Value);
        else
           throw new ArgumentException("Object is not a Digit");
    }
    public int CompareTo(Temperature other)
    {
        return Value.CompareTo(other.Value);
    }
}
class Program
{
    public static void Main()
    {
        Temperature t1 = new Temperature(5.5);
        Temperature t2 = new Temperature(3.3);

        Console.WriteLine($"{t1.CompareTo(t2)}");
```

```
        object o = t2;
        Console.WriteLine($"{t1.CompareTo(o)}");
    }
}
```

아래 그림은 IComparable 과 IComparable<T> 를 구현했을때의 각각의 메모리 그림입니다.



SECTION 13.

Generic

❑ 주요 학습 내용

> Generic Method
>
> Generic Class
>
> Generic Constraint

Generic Method

C# 은 메소드 오버로딩을 지원 합니다.

> **메소드 오버로딩(overloading) 이란 ?**
>
> **"동일한 이름의 메소드를 여러개 만들수 있는"** 문법.
> 단, 인자의 타입이나, 갯수를 다르게 해서 호출시 구별이 가능해야 합니다.

아래 코드는 변수값 2개를 서로 교환하는 swap 메소드를 int 버전과 double 버전으로 각각 만든 코드 입니다.

```
class Program
{
    public static void swap(ref int a, ref int b)
    {                               // A. int 변수 2 개의 값을 서로 교환
        int tmp = a;
        a = b;
        b = tmp;
    }
    public static void swap(ref double a, ref double b)
    {                               // B. double 변수 2 개의 값을 서로 교환
        double tmp = a;
        a = b;
        b = tmp;
    }
    public static void Main()
    {
        int n1 = 10,  n2 = 20;
        swap(ref n1, ref n2);

        double d1 = 1.1, d2 = 2.3;
        swap(ref d1, ref d2);
    }
}
```

위 코드의 단점은 A, B 2개의 메소드가 인자의 타입만 다르고 구현이 동일하다는 점입니다.

Generic Method

구현이 동일한 메소드가 여러개 필요한 경우 Generic 을 사용하면 편리 합니다.

아래 코드는 swap 메소드를 Generic 으로 작성한 코드 입니다.

```
class Program
{
    // A. Generic 으로 작성된 swap 메소드
    public static void swap<T>(ref T a, ref T b)
    {
        T tmp = a;
        a = b;
        b = tmp;
    }
    public static void Main()
    {
        int    n1 = 10,  n2 = 20;
        double d1 = 1.1, d2 = 2.3;

        swap(ref n1, ref n2);   // B. 인자로 int 타입 변수 전달
        swap(ref d1, ref d2);   // C. 인자로 double 타입 변수
    }
}
```

swap 메소드를 Generic 으로 만든 후에 B, C 처럼 각각 전달하면 컴파일러가 int 버전의 swap 과 double 버전의 swap 을 각각 생성하게 됩니다.

Using Generic Method

Generic Method 를 사용할 때 2가지 방법으로 사용 가능 합니다.

- 타입인자를 명시적을 전달하는 방법
- 타입인자를 전달하지 않은 방법. 이경우 함수 인자를 통해서 타입 인자를 컴파일러가 추론

```
// #1. 타입을 명시적으로 전달하는 경우
swap<int>(ref n1, ref n2);
swap<double>(ref d1, ref d2);

// #2. 타입 인자를 생략 하는 경우
swap(ref n1, ref n2);
swap(ref d1, ref d2);
```

Generic Class

Method 뿐 아니라 클래스도 Generic 으로 만들 수 있습니다

```
class Point<T>
{
//  private T x = 0; // error. 임의 타입 T 를 0 으로 초기할 수 없습니다.
//  private T y = 0; // error.

//  private T x = default;      // ok
//  private T y = default(T);   // ok

    private T x;
    private T y;

    public Point(T a, T b)
    {
        x = a;
        y = b;
    }
}

class Program
{
    public static void Main()
    {
        Point<int> p1 = new Point<int>(1, 2);
        var p2 = new Point<double>(1.1, 2.2);
    }
}
```

Generic Constraint

아래 코드는 2개의 변수중 큰 값을 구하는 Max 메소드를 Generic 으로 만든 코드 입니다.

하지만, 컴파일 되지 않고 A 부분에서 에러가 발생하게 됩니다.

```
using System;

class Program
{
    public static T Max<T>(T a, T b)
    {
        return a.CompareTo(b) < 0 ? b : a;  // A. 컴파일 에러
    }
    public static void Main()
    {
        var ret = Max(1, 2);
    }
}
```

에러가 발생하는 이유는 **"임의의 타입의 객체 a 안에 CompareTo() 메소드가 있다는 보장이 없기 때문"** 에 에러입니다. 아래 주석을 잘 읽어 보세요

```
public static T Max<T>(T a, T b)
{
    // 임의 타입의 객체 a 안에 CompareTo()가 있다는 보장이 없습니다.
    // 아래 코드는 에러 입니다.
    a.CompareTo(b); // error

    // 하지만, object 타입으로 부터 물려 받은 메소드 사용할 수 있습니다.
    // 아래 코드는 에러가 아닙니다.
    a.ToString();   // ok

    // 또한, = 연산등 object 타입으로 할수 있는 연산도 사용 할수 있습니다.
    a = b;          // ok
    ......
}
```

위 문제를 해결하는 방법은 아래 2 가지 방법이 가능합니다.

1. Generic constraint 사용 <<= 권장

2. T 타입의 객체 a 를 IComparable<T> 타입으로 캐스팅 해서 사용

Generic Constraint (제약)

Generic 메소드(클래스) 를 만들 때 제약(constraint) 를 표기 할 수 있습니다.

```
T Max<T>(T a, T b) where T : IComparable<T>
```

위 코드의 의미는

Generic 메소드인 Max 는 인자로 전달되는 객체가 IComarable<> 인터페이스를 구현한 타입만 사용가능 하다는 의미 입니다.

```
// a, b 의 타입이 IComparable<T> 인터페이스를 구현했다면 에러 아님
// a, b 의 타입이 IComparable<T> 인터페이스를 구현하지 않았다면 에러
Max(a, b);
```

아래 코드에서 A, B, C, D 의 주석을 차례대로 확인해 보세요

```
using System;
using static System.Console;

class Point
{
    public int X{set;get;} = 0;
    public int Y{set;get;} = 0;
    public Point(int x, int y) => (X, Y) = (x, y);
}

class Program
{
    // A. Max 는 IComparable<T> 인터페이스를 구현한 타입만 사용가능합니다.
```

```
    public static T Max<T>(T a, T b) where T : IComparable<T>
    {

        // B. a, b 는 IComparable<T> 를 구현한 타입의 객체이므로
        //    CompareTo() 가 있다고 보장 할수 있습니다.
        return a.CompareTo(b) < 0 ? b : a;
    }
    public static void Main()
    {
        // C. int 타입은 IComparable<T> 인터페이스를 구현했습니다.
        //    아래 코드는 에러가 아닙니다.
        var ret1 = Max(10, 20); // ok

        // D. Point 타입은 IComparable<T> 인터페이스를 구현하지 않았습니다.
        //    아래 코드는 에러 입니다.
        Point p1 = new Point(1,1);
        Point p2 = new Point(2,2);
        var ret2 = Max(p1, p2); // error.
    }
}
```

Generic Constraint 대신 casting 사용

Generic Constraint 대신 casting 을 사용해서도 Max 를 작성할 수 있습니다.

```
class Program
{
    // A. 아래 Max 는 Generic Constraint 문법을 사용하지 않습니다.
    public static T Max<T>(T a, T b)
    {
       // B. CompareTo 메소드를 사용하기 위해 a 를 IComparable<T> 로
       //    캐스팅 합니다
       //    a 가 IComparable<T> 인터페이스를 구현하지 않은 경우
       // ia 는 null 이 됩니다.
       IComparable<T> ia = a as IComparable<T>;

        if ( ia == null )
            throw new ArgumentException(
                        "argument isn't implement IComparable<T>");
```

```
        return ia.CompareTo(b) < 0 ? b : a;
    }
    public static void Main()
    {
        // C. int, double, string 은 모두 IComparable<T> 인터페이스를
        //    구현했으므로 아래 코드는 모두 문제 없이 실행됩니다.
        var ret1 = Max(10, 20);     // ok
        var ret2 = Max(3.4, 2.2);   // ok
        var ret3 = Max("AA", "BB"); // ok
    }
}
```

위 방식의 단점은 Max 인자로 IComparable 를 구현하지 않은 타입의 객체를 전달하면

컴파일 시간 에러가 아닌 실행시간 에러(예외 발생) 발생한다는 점입니다.

```
Point p1 = new Point(1,1);
Point p2 = new Point(2,2);

var ret = Max(p1, p2);// generic constraint 문법을 사용했다면 컴파일 에러
                      // casting 코드를 사용했다면 예외 발생
```

Generic Constraint 는 다양한 형태로 사용 가능 합니다.

```
class C1<T> where T : struct          {}
class C2<T> where T : class           {}
class C3<T> where T : class?          {}
class C4<T> where T : notnull         {}
class C5<T> where T : unmanaged       {}
class C6<T> where T : new()           {}
class C7<T> where T : base_class_name  {}
class C8<T> where T : base_class_name? {}
class C9<T> where T : interface_name   {}
class C10<T> where T : interface_name? {}
```

SECTION 14.

delegate

❑ 주요 학습 내용

> delegate
> multicast delegate
> event
> Func, Action
> Lambda expression

delegate

아래 코드를 보고 A, B, C, D 에 들어갈 타입을 생각해 봅시다. (var 말고 정확한 타입)

```
class Program
{
    public static void Foo(int arg)
    {
    }
    public static void Main()
    {
        (A) n = 10;
        (B) d = 3.4;
        (C) s = "abc";
        (D) f = Foo;   // Foo 는 메소드 입니다.
    }
}
```

- 10 은 정수 이므로 (A) 에 들어갈 타입은 **int** 입니다.
- 3.4 는 실수 이므로 (B) 에 들어갈 타입은 **double** 입니다.
- "abc" 는 문자열이므로 (C) 에 들어갈 타입은 **string** 입니다.
- Foo 는 메소드 이므로 (D) 에 들어갈 타입은 **"메소드를 담는 타입"** 이 필요합니다.

C# 에서 메소드를 담는 타입을 **"델리게이트(delegate)"** 라고 합니다.

정확히는 "메소드를 담는 것이 아니라 메소드에 대한 호출 정보" 를 저장합니다.

아래 코드는

- **Foo 메소드의 호출 정보를 담을수 있는 MyFunc라는 타입을 만들고 (코드에서 A부분)**
- **MyFunc 타입의 객체 f 에 Foo 메소등에 대한 정보를 담은 후 (코드에서 B부분)**
- **객체 f 를 사용해서 Foo 메소드를 호출하는 (코드에서 C부분) 코드 입니다.**

```
using static System.Console;

// A. 아래 코드는 MyFunc 라는 delegate 타입을 만드는 코드입니다.
//    ( 자세한 설명은 뒤에서 )
delegate void MyFunc(int arg);

class Program
{
    public static void Foo(int arg) => WriteLine($"Foo : {arg}");

    public static void Main()
    {
        MyFunc f = Foo; // B. MyFunc 타입의 객체 f 에 Foo 메소드의
                        //    호출정보를 보관합니다.
        // C. f 객체를 사용해서 "Foo 메소드" 를 호출 합니다.
        f(10);          // Foo(10) 와 동일합니다.
        f.Invoke(10);   // 이 코드도 역시 위와 동일합니다.
    }
}
```

delegate 를 만드는 법

메소드의 호출 정보를 담는 타입인 delegate 를 만들려면 다음과 같이 합니다.

1. 메소드의 선언과 동일한 모양을 만들고

2. 반환 타입앞에 delegate 를 추가 합니다.

3. 메소드 이름을 "원하는 데이타 타입 이름" 으로 변경하면 됩니다.

```
//      void Foo(int arg);       // Foo 메소드 모양입니다.
delegate void MyFunc(int arg);   // Foo 메소드 에 대한 호출 정보를 담을 수
                                 // 있는 MyFunc 라는 데이타 타입 입니다.
```

Delegate 원리

사용자가 delegate 문법을 사용하면 컴파일러는 아래와 같은 클래스를 만들게 됩니다.

사용자 코드	컴파일러가 생성한 코드
delegate void MyFunc(int arg);	**class MyFunc : System.MulticastDelegate** **{** **// Invoke(), BeginInvoke(),** **// EndInvoke()...** **}**

즉, delegate 문법으로 만든 MyFunc 는 class 의 이름이 됩니다.

Delegate 사용하기

MyFunc 델리게이트에는 Foo 뿐 아니라 signature 가 동일한 모든 메소드를 담을 수 있습니다.

메소드 시그니쳐(signature) 란 ?
메소드의 반환 타입과 인자의 타입을 나타내는 용어
Foo 의 메소드 시그니쳐는 반환 타입은 void, 인자는 int 타입 1개 가 됩니다.

```
delegate void MyFunc(int arg);

class Program
{
    // A. Foo 와 Goo 의 메소드 signature 는 동일합니다.
    // 반환 타입   : void
    // 메소드 인자 : int 한개
    // 인자의 이름(변수명)은 달라도 상관없습니다.
    public static void Foo(int arg)   {}
    public static void Goo(int param) {}

    public static void Main()
    {
```

```
        MyFunc f1 = Foo;    // B. MyFunc 타입에는 Foo 뿐 아니라
        MyFunc f2 = Goo;    //    Goo 의 호출정보도 담을수 있습니다.

        f1(10); // Foo(10)
        f2(10); // Goo(10)

        f1 = Goo;    // C. 이제 f1 은 Foo 가 아닌 Goo 의 호출정보를 담게
                     // 됩니다.
        f1(10);      // Goo(10)
    }
}
```

MyFunc 는 결국 class 로 만들어진 타입 이므로 new 를 사용해서 만들어도 됩니다.

```
MyFunc f1 = new MyFunc(Foo);  // A. 이렇게 만들어도 되고
MyFunc f2 = Foo;              // B. 이렇게 만들어도 됩니다.
f1(10); // Foo(10)
f2(10); // Foo(10)
```

위 코드에서 A, B 모두 사용 가능한 표기법 인데 약간의 차이가 있습니다.
string 타입과 유사한 특징이 있는데, 뒤에서 별도로 다루게 됩니다

delegate example

배열에서 첫번째로 나오는 “3의 배수” 를 찾으려면 다음과 같은 코드를 사용하면 됩니다.

```
bool Foo(int value)
{
    return value % 3 == 0;
}

int[] x = {1, 8, 6, 4, 3};

// 배열 x 에 있는 모든 요소를 차례대로 Foo 에 전달해서
// Foo 의 반환값이 True 를 반환 하는 요소를 찾습니다.
int ret1 = Array.FindIndex(x, Foo);
```

Array 클래스

배열에 관련된 다양한 연산을 수행하는 static method 제공.

값을 검색하려면 “Array.IndexOf” static method 사용합니다.

“Array 항목” 에서 자세히 다루게 됩니다.

Array.FindIndex 구현

Array.FindIndex 를 직접 구현해 보도록 하겠습니다.

Array.FindIndex 의 2번째 인자로 전달되는 메소드는 반드시 모양이

- bool 을 반환하고 인자가 한 개이어야 합니다.(단항 조건자 라고 합니다.)
- 메소드 인자로 메소드(함수)를 받아야 하므로 delegate 타입이 필요합니다.

```
using static System.Console;

// FindIndex 에서 사용하기 위한 delegate 타입 입니다.
delegate bool MyFunc(int value);

class Program
{
    public static int FindIndex(int[] arr, MyFunc match)
    {
        int sz = arr.Length;

        for( int i= 0; i < sz; i++)
        {
            if ( match( arr[i] ))
                return i;
        }
        return -1;
    }

    public static void Main()
    {
        int[] x = {1, 8, 6, 4, 3};

        int ret = FindIndex(x, Foo);
        WriteLine($"{ret}");
    }

    public static bool Foo(int value)
        => value % 3 == 0;
}
```

generic delegate

위에서 만든 MyFunc delegate 는 인자로 int 를 사용하고 있으므로 int 배열에만 사용할수 있습니다. 다양한 타입의 배열에서 사용하게 하려면 generic 으로 해야 합니다.

```
using static System.Console;

delegate bool MyPredicate<T>(T value);

class Program
{
    public static int FindIndex<T>(T[] arr,
                                   MyPredicate<T> match)
    {
        int sz = arr.Length;

        for( int i= 0; i < sz; i++)
        {
            if ( match( arr[i] ))
                return i;
        }
        return -1;
    }

    public static void Main()
    {
        double[] x = {1.1, 2.2, 5.2, 3.2, 4.7 };

        int ret = FindIndex(x, Foo);
        WriteLine($"{ret}");
    }

    public static bool Foo(double value)
        => value > 4.1;
}
```

delegate & method

메소드에는 2가지 종류가 있습니다.

static method	객체가 없어도 호출가능. 타입이름을 사용해서 호출 **"타입이름.static_method"** 로 호출
Instance method	객체가 있어야 호출 가능. 메소드를 호출하려면 먼저 객체 생성 후, **"객체이름.instance_method"** 로 호출

delegate 에 method 를(호출정보를) 저장하려면

static method	**"클래스이름.method이름" 으로 저장합니다.** **MyFunc f = 타입이름.static_method;**
instance Method	**"객체이름.method이름"** 으로 저장합니다. 타입 obj = new 타입(); **MyFunc f = obj.instance_method;**

```
using static System.Console;

delegate void MyFunc(int arg);

class Program
{
    public static void SMethod(int arg) => WriteLine("SMethod");
    public        void IMethod(int arg) => WriteLine("IMethod");
    public static void Main()
    {
        StaticMethod();
        Program p = new Program();
        p.InstanceMethod();
    }
    // #1. static method 안에서 delegate 사용
    public static void StaticMethod()
    {
        MyFunc f1 = Program.SMethod;  // ok
        MyFunc f2 = SMethod;          // ok
//      MyFunc f3 = IMethod;  // error

        Program p = new Program();
        MyFunc f4 = p.IMethod;  // ok
    }
    // #2. instance method 안에서 delegate 사용
    public void InstanceMethod()
    {
        MyFunc f1 = Program.SMethod;  // ok
        MyFunc f2 = SMethod;          // ok
        MyFunc f3 = this.IMethod;     // ok
        MyFunc f4 = IMethod;          // ok
    }
}
```

multicast delegate

하나의 delegate 에는 여러 개의 메소드를 등록할수 있습니다.

+=, -= 연산자를 사용합니다.

```
using static System.Console;

delegate void MyFunc(int arg);

class Test
{
    public static void SMethod(int arg) => WriteLine("Test.SMethod");
    public        void IMethod(int arg) =>
WriteLine("Test_Object.IMethod");
}
class Program
{
    public static void SMethod(int arg) => WriteLine("Program.SMethod");

    public static void Main()
    {
        Test t = new Test();

        MyFunc f = Test.SMethod;

        // f 에 추가적으로 여러개의 메소드를 등록 합니다.
        f += Test.SMethod;
        f += t.IMethod;
        f += Program.SMethod;
        f += t.IMethod;

        f(10); // f 에 등록된 메소드가 순차적으로 호출됩니다.
    }
}
```

event

delegate 직접 사용	=, +=, -= 를 사용 해서 메소드 등록
event 사용	+=, -= 만 사용가능, = 를 사용할 수 없음.

```
using static System.Console;

class Button
{
    public delegate void ClickHandler(Button source);

//  public ClickHandler? Click = null;          // A
    public event ClickHandler? Click = null;    // B

    public void UserPressButton() { Click?.Invoke(this); }
}
class Program
{
    public static void Main()
    {
        Button btn = new Button();

//      btn.Click  = M1; // error.
        btn.Click += M2; // ok.
        btn.Click += M3;
//      btn.Click = M3;  // error

//      btn.Click?.Invoke(btn); // error

        btn.UserPressButton();
    }
    public static void M1(Button btn) => WriteLine("M1");
    public static void M2(Button btn) => WriteLine("M2");
    public static void M3(Button btn) => WriteLine("M3");
}
```

Action<T>, Func<T>

C# 에는 범용적으로 사용할 수 있는 delegate 타입인 미리 만들어져 있습니다.

Action<T> 와 Func<T> 입니다.

Action<T>

반환 타입이 void 인 메소드 위한 delegate 입니다.

```
/*
// 아래 와 같이 구현되어 있습니다.

delegate void Action();
delegate void Action<in T>(T arg);
delegate void Action<in T1, in T2>(T1 arg1, T2 arg2);

// 인자의 갯수가 16 개 까지 지원(버전에 따라 달라 질수 있음.)
*/

class Program
{
    public static void M1(){}
    public static void M2(int arg){}
    public static void M3(double arg){}
    public static void M4(int arg1, double arg2){}

    public static void Main()
    {
        Action f1 = M1;
        Action<int> f2 = M2;
        Action<double> f3 = M3;
        Action<int, double> f4 = M4;
    }
}
```

Func<T>

반환 타입이 void 가 아닌 메소드 위한 delegate 입니다.

```
/*
// 아래와 같이 구현되어 있습니다.

delegate R Func<R>();
delegate R Func<T, R>(T arg);
delegate R Func<T1, T2, R>(T1 arg1, T2 arg2);

// 인자가 15 개 까지 지원 (버전에 따라 달라 질수 있음)

*/
class Program
{
    public static int    M1() => 0;
    public static double M2(int arg) => 0;
    public static string M3(double arg) => "";
    public static object M4(int arg1, double arg2) => 0;

    public static void Main()
    {
        Func<int> f1 = M1;
        Func<int, double> f2 = M2;
        Func<double, string> f3 = M3;
        Func<int, double, object> f4 = M4;
    }
}
```

Lambda expression

람다 표현식(lambda expression) 은

- 익명의 함수(anonymous function) 을 만드는 문법으로
- C# 뿐 아니라 다양한 프로그래밍언어에서 제공되는 기능 입니다.

람다표현식을 사용하면

- 인자로 전달되는 메소드(함수)를 간결하게 만들수 있고
- 자신이 속한 문맥의 지역변수(outer variable)를 캡쳐 할 수 있는 기능이 있습니다.

다음 코드는 배열에서 1번째 나오는 짝수를 찾는 예제 입니다

주석을 잘 읽어 보세요

```
using static System.Console;

class Program
{
    static bool IsEven(int n) { return n % 2 == 0; }
    public static void Main()
    {
        int[] array = { 1, 3, 6, 4, 5 };

        // #1. FindIndex 의 2 번째 인자로 IsEven 이라는 메소드를 전달합니다.
        int idx1 = Array.FindIndex(array, IsEven);

        // #2. FindIndex 의 2 번째 인자로 람다표현식을 사용합니다.
        //     위 코드와 동일한 기능을 수행합니다
        int idx2 = Array.FindIndex(array, n => n % 2 == 0 );
      WriteLine($"{idx1, idx2}");
    }
}
```

Capture Outer Variable

람다 표현식을 사용하면 자식이 속한 문맥의 지역변수를 캡쳐 할 수 있습니다.

C# 에서는 “capture outer variable” 이라고 표현합니다.

아래 코드는 배열에서 짝수가 아닌 k 의 배수를 찾는 코드입니다. 그런데, k 는 Main 메소드에서 사용하는 지역변수 입니다.

주석을 잘 읽어 보세요

```
using static System.Console;

class Program
{
    // #1. k 의 배수를 찾으려면 아래 처럼 만들어야 하는데
    //     IsMultipleOf 에서는 Main 메소드의 지역변수에 접근할 수 없습니다.
//  static bool IsMultipleOf(int n) { return n % k == 0; }

    public static void Main()
    {
        int[] array = { 1, 3, 6, 4, 5 };

        int k = 3;  // 사용자가 입력한 값이라고 가정.

        // #2. IsMultipleOf 에서 k 를 사용할수 없기 때문에
        //     아래 처럼 사용할수가 없습니다.
//      int idx = Array.FindIndex(array, IsMultipleOf );

        // #3. 하지만 람다 표현식에서는 Main 메소드의 지역변수
        //     k 에 접근할수 있습니다.
        int idx = Array.FindIndex(array, n => n % k == 0);
    }
}
```

Lambda expression 활용

람다 표현식은

① Array.FindIndex 등의 다양한 **"메소드에 인자"**로 전달하거나

② Func, Action delegate 에 담아서 **"함수(메소드) 처럼 사용"** 할 수 있습니다.

```
class Program
{
    public static void Main()
    {
        int[] array = { 1, 3, 6, 4, 5 };

        // 람다표현식 활용 #1. FindIndex 메소드 인자로 전달
        int idx = Array.FindIndex(array, n => n % 2 == 0);

        // #2.Func delegate 에 담아서 함수 처럼 사용
        Func<int, int> square = n => n * n;

        int ret = square(3);
    }
}
```

Lambda expression 을 만드는 다양한 모양

```
Func<int, int> f1 = (int n) => { return n * n; };
Func<int, int> f2 = (int n) => n * n;
Func<int, int> f3 = (n) => n * n;
var            f4 = (int n) => n * n;
Func<int, int> f5 = n => n * n;

var f6 = object (bool b) => b ? 1 : "one"; // 반환 타입 표기
```

SECTION 15.

Array

❑ 주요 학습 내용

> Array
>
> System.Index
>
> System.Range
>
> Array static method

Array

배열의 종류

C# 에서는 3가지 종류의 배열을 제공합니다

- 1차 배열 (single-dimensional array)
- 다차원 배열 (multi-dimensional array)
- 가변 배열 (jagged array)

```
// 1 차 배열(single-dimensional array)
int[] arr1 = { 1, 2, 3 };

// 다차 배열(multi-dimensional array)
int[,] arr2 = { { 1, 2 }, { 3, 4 } };

// 가변 배열(jagged array)
int[][] arr3 = new int[3][];
arr3[0] = new int[2] { 1, 2 };
arr3[1] = new int[3] { 1, 2, 3 };
arr3[2] = new int[4] { 1, 2, 3, 4 };
```

배열의 생성

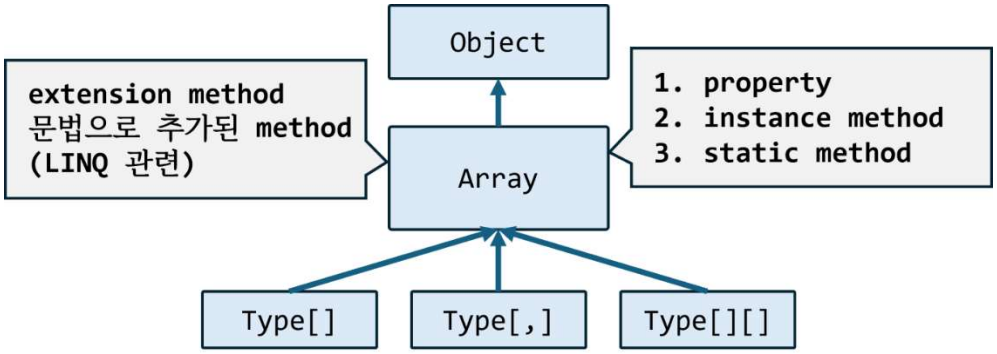
1차 배열은 아래와 같은 방법으로 생성할 수 있습니다

```
using static System.Console;

int[] arr1 = { 1, 2, 3, 4, 5 }; // 가장 널리 사용하는 방식
int[] arr2 = [1, 2, 3, 4, 5]; // C#12.0 이후 사용하는 방식

int[] arr3 = new int[5];
int[] arr4 = new int[5] { 1, 2, 3, 4, 5 };
int[] arr5 = new[] { 1, 2, 3, 4, 5 };

int[] arr6 = (int[])Array.CreateInstance(typeof(int), 5);
```

Array property, method

배열의 대해서 다양한 연산을 수행하거나, 정보를 얻으려면 아래 의 방법을 제공하면 됩니다.

1. 배열 객체의 속성(Property)
2. 배열 객체의 instance method
3. 배열 객체의 extension method – Linq
4. Array 클래스의 static method



배열의 속성

배열 객체에는 아래의 속성을 제공합니다.(모든 속성은 read only 입니다.)

property	Type	설명
Length	Int32	배열의 크기
LongLength	Int64	배열의 크기(64bit)
Rank	Int32	배열의 차수(1차, 2차, …)
SyncRoot	System.Object	동기화 하는데 사용할 수 있는 객체
IsReadOnly	Boolean	항상 False
IsFixedSize	Boolean	항상 True
IsSynchronized	Boolean	항상 False

```
using static System.Console;

int[] arr = { 1, 2, 3, 4, 5 };

WriteLine($"{arr.Length}");          // 5
WriteLine($"{arr.LongLength}");      // 5
WriteLine($"{arr.Rank}");            // 1
WriteLine($"{arr.IsReadOnly}");      // False
WriteLine($"{arr.IsFixedSize}");     // True
WriteLine($"{arr.IsSynchronized}"); // False
```

배열의 instance method

배열 객체는 다음과 같은 instance method 를 제공합니다.

arr.GetType();	Inherited from Object
arr.ToString();	
arr.GetHashCode();	
arr.Equals();	
arr.GetEnumerator();	Inherited from Array
arr.SetValue();	
arr.GetValue();	
arr.Initialize();	
arr.Clone();	
arr.CopyTo();	
arr.GetLength();	
arr.GetLongLength();	
arr.GetLowerBound();	
arr.GetUpperBound();	

Array class Static Method

Array 클래스는 모든 배열의 기반 클래스 입니다.

Array 클래스를 추상 클래스 이므로 객체를 생성할 수는 없지만, 배열에 대해 많은 연산 (검색, 정렬, …) 을 수행할 수 있는 많은 static method 를 제공합니다.

Array.Sort()

배열의 모든 요소를 정렬 하려면 “Array.Sort()” 를 사용하면 됩니다.

```
using static System.Console;

int[] arr = { 1, 3, 5, 7, 9, 2, 4, 6, 8, 10 };

Array.Sort(arr);

foreach( var e in arr )
{
    Write($"{e}, ");
}
```

“Array.Sort(arr)” 기본적으로 오름 차순으로 정렬합니다. 정책을 변경하려면 Array.Sort() 의 2 번째 인자를 “2 개 요소를 비교” 할 수 있는 비교 정책 함수를 전달해야 합니다.

다음의 2 가지 방법으로 사용할 수 있습니다.

1. Comparison<T> delegate 를 전달하는 방법 – 권장
2. ICompare<T> 인터페이스를 구현한 객체를 전달하는 방법

using Comparison<T> delegate

C# 표준에 다음과 같은 delegate 가 정의 되어 있고, Array.Sort() 의 2번째 인자로 사용됩니다.

```
delegate int Comparison<in T>(T x, T y);
```

아래 코드는 람다표현식을 사용해서 비교 정책을 전달하는 코드 입니다.

```
using static System.Console;

int[] arr = { 1, 3, 5, 7, 9, 2, 4, 6, 8, 10 };

Array.Sort(arr, (a, b)=>b.CompareTo(a) );

foreach( var e in arr )  Write($"{e}, ");
          // 10, 9, 8, 7, 6, 5, 4, 3, 2, 1
```

using ICompare<T> interface

C# 표준에 다음과 같은 interface 가 정의 되어 있고, Array.Sort() 의 2번째 인자로 사용됩니다.

```
public interface IComparer<in T>
{
    int Compare(T? x, T? y);
}
```

따라서, 사용자는 IComparer<T> 를 구현한 객체를 만들어서 Array.Sort() 의 2 번째 인자로 전달할 수 있습니다.

```
using static System.Console;

int[] arr = { 1, 3, 5, 7, 9, 2, 4, 6, 8, 10 };

// 람다 표현식이 아닌 객체 전달
Array.Sort(arr, new MyCompare() );

foreach( var e in arr )  Write($"{e}, ");
// 10, 9, 8, 7, 6, 5, 4, 3, 2, 1

// 비교 정책으로 사용할 객체
class MyCompare : IComparer<int>
{
    public int Compare(int x, int y)
    {
        return y.CompareTo(x);
    }
}
```

System.Index

배열의 요소는 **“arr[인덱스]”** 로 접근합니다.

“인덱스” 에는 int 타입 또는 System.Index 타입을 사용합니다.

int	Sequence 접근을 위한 값만 보관
System.Index	Sequence 접근을 위한 값과 방향 을 보관

```
using static System.Console;

int[] arr = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

// #1. int 는 값만 보관합니다.
int i = 1; //

// #2.
Index idx1 = 1;  // 앞에서 1 번째
Index idx2 = ^1; // 뒤에서 1 번째

WriteLine($"{arr[i]}");    // 2
WriteLine($"{arr[idx1]}"); // 2
WriteLine($"{arr[idx2]}"); // 10
```

주의 할 점은 뒤에서부터 접근시는 0이 아닌 1부터 시작합니다.

0	1	2	3	4	5	6	7	8	9
1	2	3	4	5	6	7	8	9	10
^10	^9	^8	^7	^6	^5	^4	^3	^2	^1

System.Index 객체는 다양한 방법으로 만들 수 있습니다.

```
// #1. new 사용
Index idx1 = new Index(3);
Index idx2 = new Index(3, fromEnd: true);

// #2. 정적 메소드 사용
Index idx3 = Index.FromStart(3);
Index idx4 = Index.FromEnd(3);

// #3. 단축 표기법 사용
Index idx5 = 3;
Index idx6 = ^3;
```

System.Range & Slicing

System.Range 는 2개의 System.Index 를 사용해서 “하나의 구간”을 나타내는 타입 입니다.

Range 객체를 사용해서 배열의 일부 요소를 복사한 새로운 배열을 만드는 것을 "**Slicing**" 이라고 합니다.

```
using static System.Console;
int[] arr = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };

Range r1 = new Range(2, 5);  // 2 ~ 4 의 구간 (5 는 포함안됨)
Range r2 = new Range(2, ^3); // 2 ~ ^4(6) 의 구간 (^3(7)는 포함안됨)

// Range 객체를 사용해서 배열의 일부 요소를 복사한 새로운 배열을 만드는 것을
// "Slicing" 이라고 합니다.
int[] x1 = arr[r1];
int[] x2 = arr[r2];
print_array(x1);  // 2, 3, 4
print_array(x2);  // 2, 3, 4, 5, 6

void print_array(int[] x)
{
    foreach (var e in x)
        Write($"{e}, ");
    WriteLine();
}
```

Range 객체를 생성하는 단축 표기법도 있습니다.

```
int[] arr = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
Range r1 = new Range(2, 5);
Range r2 = 2..5;       // 위와 동일합니다.
int[] x = arr[2..5];   // 2, 3, 4
```

SECTION 16.

Collection

❑ 주요 학습 내용

> Enumerator
>
> Collection

Generic vs Non Generic

C# Collection 에는 크게 2가지 종류가 있습니다.

Generic Collection	C# 2.0 부터 제공되는 Collection. 타입 안정성이 뛰어나고 Boxing/Unboxing 이 발생되지 않으므로 성능이 좋다. 권장
Non-Generic Collection	**C# 1.0 시절, Generic 문법이 없을 때 제공했던 Collection** 타입 안정성이 좋지 않고 Boxing/Unboxing 이 발생 해서 성능이 좋지 않다. 되도록 사용하지 말 것.

```
using System;
using System.Collections;
using System.Collections.Generic;
using static System.Console;

class Program
{
    public static void Main()
    {
        // ArrayList 는 Non Generic Collection
        ArrayList col1 = new ArrayList(); // object 보관

        col1.Add(10);
        //col1.Add("hello")   ;// 에러가 없다.
                              // 단점 #1. type 안정성이 떨어진다.

        col1.Add(30);         // 스택에 있던 30 이
                              // 힙에 복사본을 만들어서 놓인다. - Boxing
                              // 단점 #2. 성능저하.

        int n =(int)col1[0]; // 단점 #3. 꺼낼때는 반드시 캐스팅이 필요하다

        // C# 2.0 부터 Generic(template) 문법 지원
        // List 는 Generic Collection
        List<int> col2 = new List<int>();
```

```
        col2.Add(10);
        col2.Add("hello"); // 1.  error. 타입 안정성이 뛰어나다.

        int n2 = col2[0]; //  2. ok. 캐스팅이 필요 없다.

                          // 3. Boxing/Unboxing 이 없다.
                          //    성능이 좋다.

    }
}
```

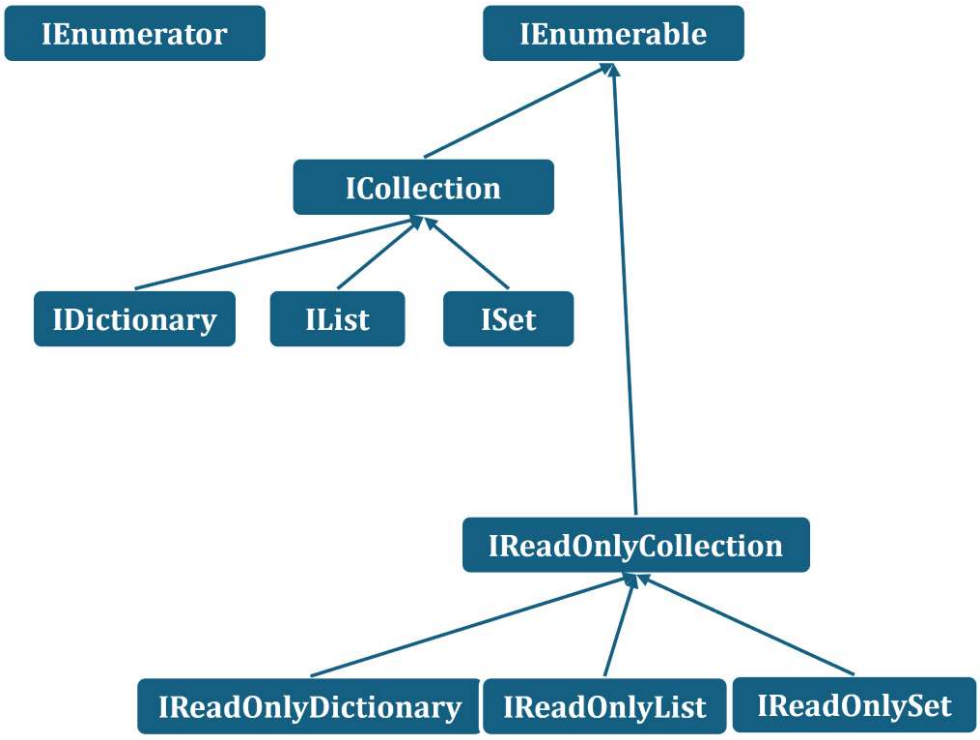
enumerator

Collection 의 내부구조가 달라도 “동일한 방법으로 모든 요소에 접근하기 위한 객체”.

```
using static System.Console;

var c1 = new List<int>(); // 모든 요소를 연속된 메모리에 보관
var c2 = new LinkedList<int>(); // 떨어진 메모리에 보관

c1.Add(1);     c1.Add(2);     c1.Add(3);
c2.AddLast(1); c2.AddLast(2); c2.AddLast(3);

// c1, c2 는 내부 구조는 다르지만
// 열거자(enumerator)를 사용하면 동일한 방식으로 열거 가능

// #1. 동일한 방법으로 열거자를 꺼내고
IEnumerator<int> e1 = c1.GetEnumerator();
IEnumerator<int> e2 = c2.GetEnumerator();

// #2. 동일한 방법으로 모든 요소에 접근
while (e1.MoveNext())
    Write($"{e1.Current}, ");

WriteLine();

while (e2.MoveNext())
    Write($"{e2.Current}, ");
```

Collection Interface

다양한 컬렉션들의 메소드 이름을 약속하기 위한 인터페이스 제공.



ArrayList, List<T>

List<T> 는 크기를 변경할 수 있는 배열로서, 연속된 메모리를 사용하고 배열과 달리 변경 가능 이 가능합니다.

검색이 빠르고, 인덱스로 접근 가능하지만 캐패시터 개념을 가지고 있습니다.

```
using System;
using System.Collections;
using System.Collections.Generic;
using static System.Console;

class Program
{
    public static void Main()
    {
        ArrayList arr1 = new ArrayList(); // C#1.0

        // C#2.0
        List<int> arr2 = new List<int>(){ 1,2,3,4,5,6,7,8,9,10 };
        WriteLine(arr2.Count); // 10 개

        arr2.Add(20);
        WriteLine(arr2.Count); // 11 개

        //-----------------------------------
        List<int> arr3 = new List<int>();

        arr3.Add(10);
        WriteLine(arr2.Count);    // 1
        WriteLine(arr2.Capacity); // 4

        arr3.Add(20); // 이미 캐패시터가 있다. 빠르다.!
        WriteLine(arr2.Count);    // 2
        WriteLine(arr2.Capacity); // 4

        arr3.Add(30);
        arr3.Add(40);
        arr3.Add(50);  // 메모리 재할당 필요. 느리다.
        WriteLine(arr2.Count);    // 5
        WriteLine(arr2.Capacity); // 8
```

```
        arr2.TrimExcess(); // 여분의 메모리 제거

        WriteLine(arr2.Count);    // 5
        WriteLine(arr2.Capacity); // 5
    }
}
```

LinkedList<T>

LinkedList 는 모든 요소가 떨어진 메모리로 관리됩니다.

인덱스를 사용해서 접근할 수 없으므로, 요소 접근은 enumerator 를 사용해야 합니다

임의 삽입이 빠르게 동작합니다.

```
using System;
using System.Collections;
using System.Collections.Generic;
using static System.Console;

class Program
{
    public static void Main()
    {
        LinkedList<int> st = new LinkedList<int>();
        st.AddFirst(10);
        st.AddLast(20);

        st[0] = 10; // error. IList<> 인터페이스 구현안함.
    }
}
```

Stack<T>, Queue<T>

Stack, Queue 는 내부적으로 동적 배열을 사용합니다.

```
using System;
using System.Collections;
using System.Collections.Generic;
using static System.Console;

class Program
{
    public static void Main()
    {
        Queue<int> Q = new Queue<int>();

        Q.Enqueue(10);
        Q.Enqueue(20);

        WriteLine(Q.Dequeue());   // 꺼내고 제거
        WriteLine(Q.Peek());      // 꺼내기만 제거안함.
                                  // 확인만할때 사용

        Stack<int> s = new Stack<int>();
        s.Push(10);
        s.Push(20);

        s.Pop();
        s.Peek();
    }
}
```

HashSet<T>, SortedSet<T>

Set 은 여러 개의 키값을 저장하는 Collection 입니다.

HashSet<T>	Hash Table 을 사용하는 Set, 정렬을 유지 하지 않음
SortedSet<T>	RB Tree 를 사용하는 Set, 정렬을 유지

```
using System;
using System.Collections;
using System.Collections.Generic;
using static System.Console;

class Program
{
    public static void Main()
    {
        HashSet<int> hs = new HashSet<int>();

        WriteLine(15.GetHashCode());
        WriteLine(15.GetHashCode());
        WriteLine(10.GetHashCode());

        hs.Add(15); // 이순간 15 를 해쉬함수로 전달합니다
                    // 반환된 bucket 번호 에 15 를 보관합니다.
        hs.Add(10);
        hs.Add(20);
        hs.Add(23);
        hs.Add(18);

        // 삽입된 순서도 유지하지 못하고, 크기로 정렬되지도 않는다.
        // 무작위순서
        foreach (var n in hs)
            WriteLine(n);

        WriteLine(hs.Contains(15)); // 검색합니다.
                        // 15 를 해쉬함수에 보내서 어디에 저장했는지
                        // 알아 냅니다.
        SortedSet<int> ss = new SortedSet<int>();
        ss.Add(15);
```

```
        ss.Add(10);
        ss.Add(20);
        ss.Add(23);
        ss.Add(18);

        // 크기순으로 정렬되어 있습니다.
        foreach (var n in ss)
            WriteLine(n);
    }
}
```

Dictionary<K, V>

Dictionary<K, V> 는 key 값을 가지고 value 를 보관하는 Collection 입니다.

```
using System;
using System.Collections;
using System.Collections.Generic;
using static System.Console;

class Program
{
    public static void Main()
    {
        // KeyValuePair 를 보관하는 hash table
        Dictionary<string, string> dic =
                            new Dictionary<string, string>();

        dic.Add("mon", "월요일");
                // "mon" 를 해쉬함수에 보내서(mon.GetHashCode()
                // bucket 에 keyValuePair("mon", "월요일") 보관


        // 값을 2 개를 보관하는 자료구조
        KeyValuePair<string, string> p =
            new KeyValuePair<string, string>("mon", "월요일");

        WriteLine(p.Key); // "mon"
        WriteLine(p.Value);// "월요일"
    }
}
```