Contents

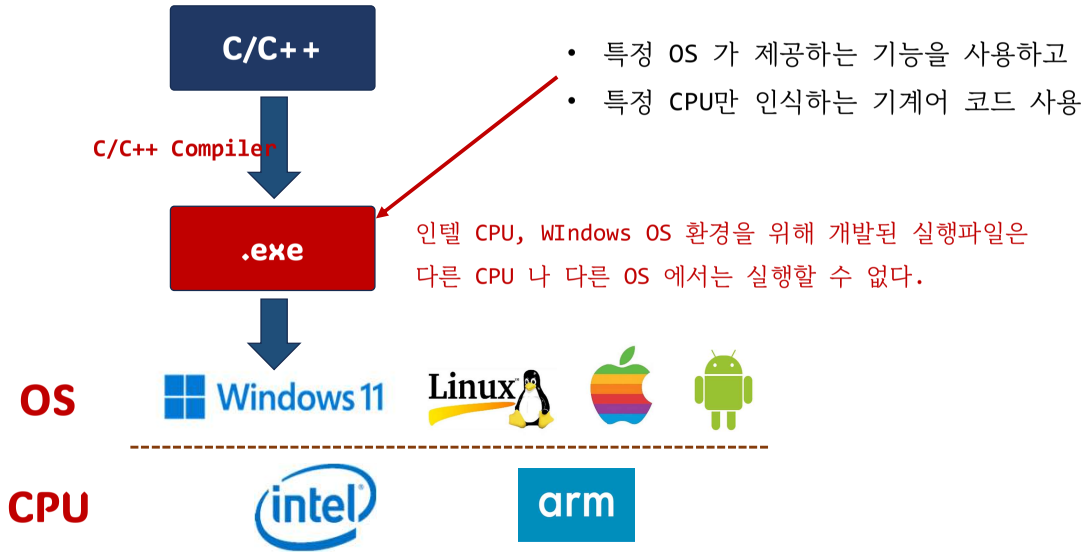
.net framework

3

4

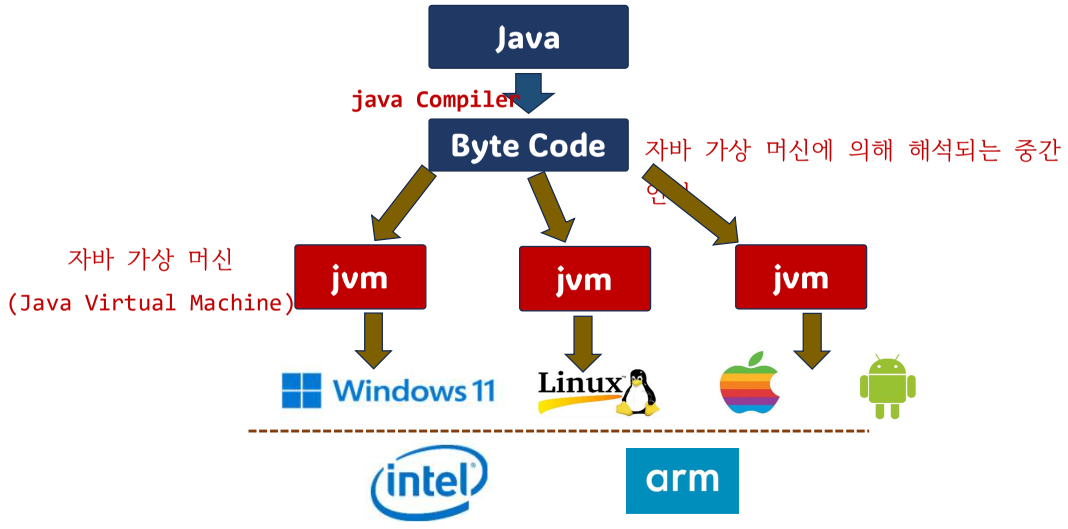
5

.Net 개념



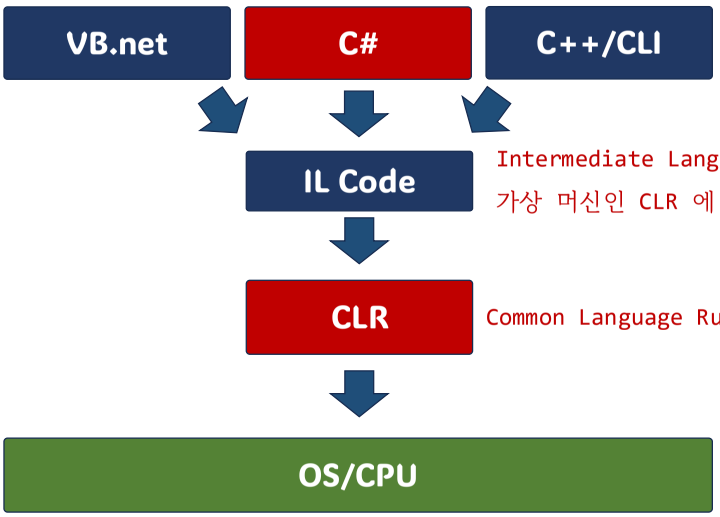
6

.Net 개념



7

.Net 개념

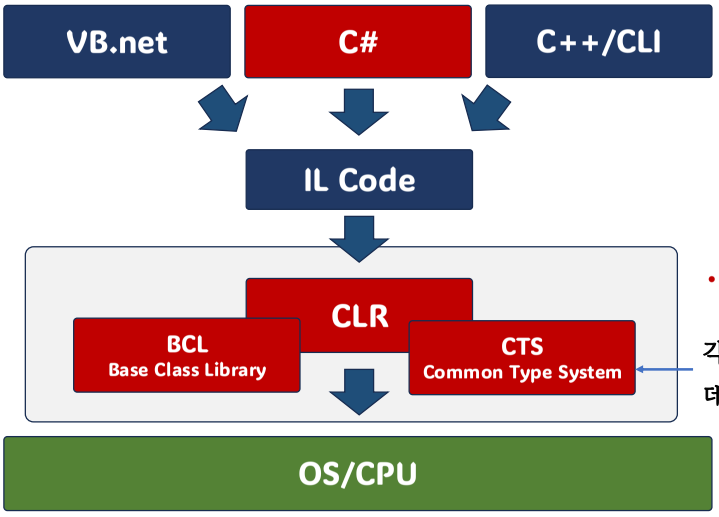


다양한 언어를 사용해서
IL CODE 생성하는 것이 가능

Intermediate Language
가상 머신인 CLR 에 의해 해석 되는 언어

Common Language Runtime (CLR)

8



.Net FrameWork

각각의 언어가 공통으로 사용할
데이타 타입이 필요

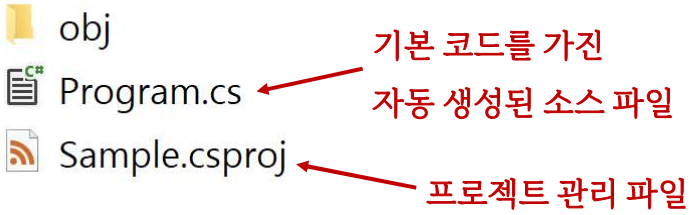
dotnet build

9

핵심 정리

- 1. 예제 소스를 저장할 폴더 생성
  ⇨ 강의에서는 "**C:\\Sample**" 로 생성

- 2. 명령 프롬프트를 실행 후 해당 폴더로 이동해서
  ⇨ "**dotnet new console**" 명령 실행



- 3. Program.cs 파일을 편집기에서 열고 코드 작성

- 4. 명령 프롬프트에서 아래 명령으로 빌드

dotnet build	컴파일만 수행
dotnet run	컴파일 + 실행 까지 수행

10

핵심 정리

- 강의 에서는 생성된 프로젝트 폴더(C:\\Sample)을
  ⇨ **“Visual studio code”** 로 열어서 작업

- **새로운 예제를 작성하려면**
  ① 새로운 폴더를 생성해서 작업하거나
  ② 기존 폴더에 소스 파일만 추가(확장자 .cs)해서 예제 작성

- dotnet build 또는 dotnet run 을 수행하면
  ⇨ **프로젝트 폴더내의 모든 소스 코드를 빌드**

- 폴더내에 소스 파일 중 **“한개의 소스 파일만 빌드”** 하려면
  ⇨ **.csproj 파일 수정**

11

핵심 정리

- **Sample.csproj**
  ⇨ 프로젝트에 대한 다양한 빌드 설정값을 가진 파일

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

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

    <NoWarn>0219, 0168, 0414</NoWarn>

  </PropertyGroup>

    <ItemGroup>
        <Compile Remove="*.cs"/>
        <Compile Include="test.cs"/>
    </ItemGroup>

</Project>
```

12

csc Compiler

13

핵심 정리

- **C# Compiler**
  - ⇨ csc.exe

- **csc.exe 컴파일러를 직접 사용하려면**
  - ① **.Net 환경 설치시 설치되는 컴파일러 사용**
    - ⇨ C# 언어 최신 문법을 지원하지 못함
    - ⇨ .Net 8.0 설치시 C# 12.0 버전 지원
      (컴파일러는 C# 5.0 까지만 지원)
  - ② **Visual Studio 정식 버전 설치**
  - ③ **github 에서 컴파일러 소스 코드를 다운 받아서 직접 빌드해서 사용**

- **강의에서는**
  - ⇨ 대부분 **"dotnet run"** 사용
  - ⇨ 일부 간단한 예제만 csc 컴파일러 사용

14

핵심 정리

- .Net 설치시 아래 경로에 csc.exe 가 있음.

```
C:\Windows\Microsoft.NET\Framework  \v4.0.30319
C:\Windows\Microsoft.NET\Framework64\v4.0.30319
```

버전은 다를 수 있음.

- 명령 프롬프트 (또는 Visual Studio Code) 에서 csc 컴파일러를 사용하려면

  ⇨ Path 환경변수 설정

- 소스 코드 작성 후 아래의 명령으로 컴파일

```
➢ csc test.cs        ← test.exe 파일 생성
➢ test.exe           ← 실행
```

15

Entry Point

16

핵심 정리

- **프로그램의 Entry Point**

  ⇨ 소스코드에서 프로그램이 처음 실행되는 지점.

C 언어	main 이라는 이름을 가지는 함수의 첫번째 문장부터 실행
Python	소스 파일의 첫번째 문장 부 터 실행

C# 언어는 2가지 방식을 모두 지원

17

핵심 정리

- **Main 메소드를 만드는 방법**

  ⇨ C# 1.0 부터 사용한 전통적인 방식

클래스 이름은 어떤 이름을 사용해도 상관 없음.

관례적으로 "Program" 이라는 이름을 많이 사용

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

18

핵심 정리

- **Top Level Program 방식**
  - ⇨ 별도의 Main 함수를 만들지 않고 **소스 파일이 1번째 문장 부터 실행되게 하는 방식**
  - ⇨ **C# 9.0 부터 지원**
  - ⇨ **간단한 예제 작성시 편리**

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

- **Top Level Program 방식 원리**
  - ⇨ 컴파일러가 클래스와 Main 메소드를 생성한후
  - ⇨ **사용자가 만든 코드는 Main 메소드 안에 넣고 실행하는 것**

```
class Program
{
    public static void Main()
    {
        System.Console.WriteLine("Hello, C#");
    }
}
```

19

핵심 정리

- **주의**
  - ⇨ C# 언어가 지원하는 특정 문법이
  - ⇨ **Top Level Program 에서는 에러가 발생할 수 있음.**

```
class Program
{
    public static void Main()
    {
        int Square(int x)       { return x * x; }
        double Square(double x) { return x * x; }
        Square(3);
        Square(3.3);
    }
}
```

Local Function

함수 안에 함수를 만드는 문법

- **Function overloading**
  - ⇨ C# 언어의 메소드(멤버함수)는 **"함수 오버로딩"** 문법을 지원하지만
  - ⇨ **Local Function 은 함수 오버로딩을 지원하지 않는다.**

20

핵심 정리

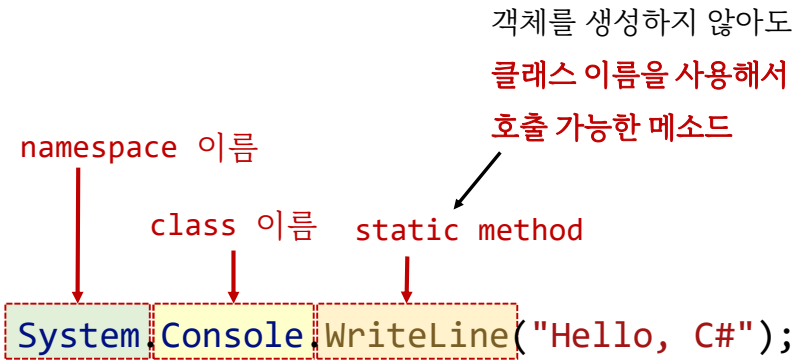
- 강의 에서는
  - ⇨ 간단한 예제는 **“Top Level Program”** 방식 사용
  - ⇨ **대부분의 예제는 Main 메소드를 만드는 방식 사용**

21

C# 표준 출력

22

핵심 정리




```
namespace System
{
    public static class Console
    {
        public static void WriteLine(int value);
        public static void WriteLine(string? value);
        // ......
    }
}
```

- 용어

C#, Java	메소드(**method**) 라는 용어 사용
C++	멤버 함수(**member function**) 용어 사용

23

핵심 정리

```
namespace System
{
    public static class Console
    {
        public static void WriteLine(string? value);
    }
}
```

- **namespace 안에 있는 요소에 접근 하는 방법**

C#, Java	namespace_name.class_name
C++	namespace_name::class_name

- **class 에 static member 에 접근하는 방법**

C#, Java	class_name.static_member_name
C++	class_name::static_member_name

24

핵심 정리

```
namespace System
{
    public static class Console
    {
        public static void WriteLine(int value);
        public static void WriteLine(string? value);
        // ......
    }
}
```

- **Console** 클래스
  - ⇨ 표준 "**입출력 관련 메소드를 제공**" 하는 "**static 클래스**"
  - ⇨ 모든 메소드가 "**static method**"
  - ⇨ 객체를 생성하지 않고 "**Console 이라는 클래스 이름**"으로 메소드를 호출

- 표준 출력을 위한 Console 클래스의 static method

Write()	출력 후 개행을 하지 않음.
WriteLine()	출력 후 개행.

25

핵심 정리

- **using System**
  - ⇨ System namespace 안에 있는 요소를 "**System 이름 없이 사용**" 가능.

- **using static System.Console**
  - ⇨ System.Console 클래스 안에 있는 static method 를 "**System.Console 없이 사용**" 가능.

- **using alias = namespace or type**
  - ⇨ namespace 이름이나 타입의 별명.

- **주의**
  - ⇨ using 지시어는 반드시 소스의 제일 앞부분에 놓아야 한다.

26

핵심 정리

- 강의 에서는 아래 2가지 표기법 사용

- 예제 소스가 간단한 경우

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

- 예제 소스가 복잡한 경우

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

27

변수 값을 출력하는 방법

- 변수 한 개 의 값만 출력

```
Console.WriteLine(n1);
```

- 서식에 맞추어 출력

```
Console.WriteLine( "n1 = {0}, n2 = {1}", n1, n2);
Console.WriteLine($"n1 = {n1}, n2 = {n2}");
```

- C# 문자열

Quoted string literals	"ABCD"
Verbatim string literals	@"ABCD"
Interpolated strings	$"ABCD"
Raw string literals	"""ABCD"""

"string" 강의 참고

28

C# 표준 입력

29

핵심 정리

- 사용자에게 입력을 받으려면
  - ⇨ Console 클래스의 “**Readxxx**” static method 사용

Console.ReadLine()	**입력 스트림**에서 **한 줄** 을 입력
Console.Read()	**입력 스트림**에서 **한 문자** 입력
Console.ReadKey()	스트림을 사용하지 않고 “**키 입력을 직접**” 읽을 때 사용.

- **Console.ReadLine() 메소드**
  - ⇨ 표준 입력 스트림 에서 “한 줄” 을 입력
  - ⇨ Reads the **next line of characters** from the **standard input stream.**

30

핵심 정리

```
string s = Console.ReadLine();
```

⇨ C# 7.0 까지는 빌드 문제 없음

⇨ C# 8.0 이후에는 빌드 환경에 따라 "**경고(CS8600)**" 발생할 수 있음

⇨ C# 8.0 에서 추가된 "**nullable reference**" 라는 문법 때문에

csc 컴파일러를 직접사용	컴파일러 옵션에 따라 nullable reference 문법 사용 여부 결정
dotnet build visual studio	**nullable reference 문법이 기본 적용됨**

31

핵심 정리

- **nullable reference 기능을 사용하지 않으려면**

  ⇨ 프로젝트 설정 파일(**.csproj**) 를 아래와 같이 변경

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

                disable
    <Nullable>~~enable~~</Nullable>


   <NoWarn>0219, 0168</NoWarn>
</PropertyGroup>
```

자세한 내용은 "**nullable reference**" 강의 참고

32

핵심 정리

- 문자열이 아닌 정수 또는 실수를 입력 받으려면

  ⇨ 사용자의 모든 입력은 결국 문자(열)를 입력한 것

10	"10" 이라는 문자열을 입력 한 것
3.4	"3.4" 라는 문자열을 입력 한 것

⇨ 문자열 형태로 입력 받은 후, "정수(또는 실수)로 변경" 해서 사용

⇨ 문자열 "10" ➔ 정수 10 으로 변경해서 사용.

핵심 정리

- 문자열을 정수(또는 실수)로 변경하는 방법

① System.Convert 클래스의 static method 사용

② int(double) 타입(struct)의 static method 사용

(C# 은 int, double 도 메소드를 가진다)

```
string s = "10";

int n1 = Convert.ToInt32(s);
int n2 = int.Parse(s);

int n3 = 0;
bool b = int.TryParse(s, out n3);
```

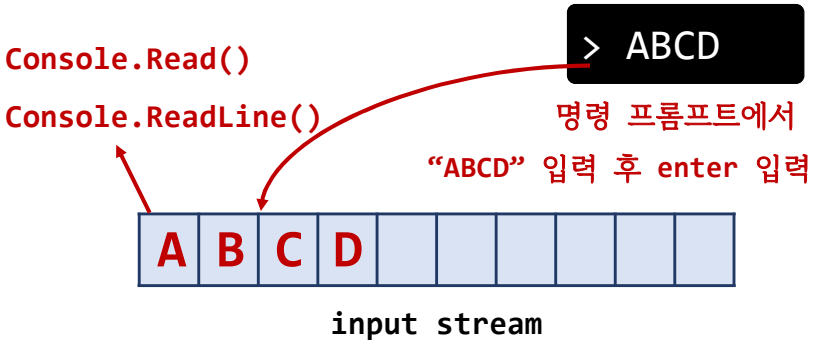
변환에 실패한 경우 예외 발생

반환 값은 성공/실패 여부 나타내는 bool 값

변환 결과는 인자로 전달한 n3에 담아 준다.

핵심 정리

- Console.Read(), Console.ReadLine()
  - ⇨ 입력 스트림으로 부터 입력.



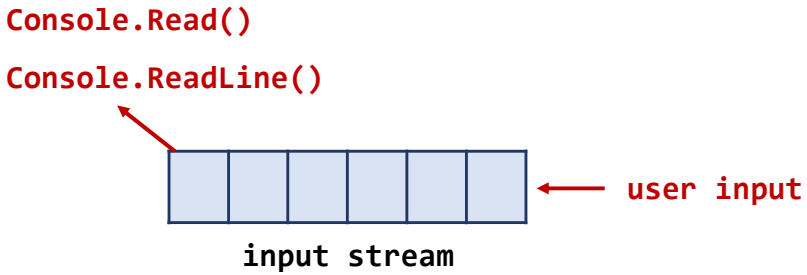
  - ⇨ 처음에는 입력 스트림이 비어 있다.
  - ⇨ 입력 스트림이 비어 있는 경우, 사용자로 부터 입력 스트림으로 입력 받는 내부 루틴이 실행.

Console.ReadLine()	입력 스트림에서 한 줄 을 입력
Console.Read()	입력 스트림에서 한 문자 입력

35

핵심 정리

- Console.Read(), Console.ReadLine()
  - ⇨ 입력 스트림으로 부터 입력.
  - ⇨ 입력 스트림이 비어 있는 경우, 사용자로 부터 입력 스트림으로 입력 받는 루틴이 실행.
  - ⇨ 사용자가 입력을 종료 하려면 Enter 키 입력 필요.



36

핵심 정리

- **Console.ReadKey()**
  - ⇨ 입력 스트림이 아닌 키보드 입력을 직접 읽을 때 사용.
  - ⇨ 입력 종료를 위해 enter 를 입력할 필요 없음.

`Console.ReadKey()` `Console.ReadKey(false)`	입력된 문자를 화면에 출력 **(echo)**
`Console.ReadKey(true)`	입력된 문자를 화면에 출력 안함 **(no-echo)**

Type & Variable #1

핵심 정리

- **C# Built-in Types**

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		

Built-in Types 외에도 .net Framework 에서

"다양한 분야의 클래스 라이브러리가 제공"

39

핵심 정리

- var
  - ⇨ **우변의 표현식으로 타입을 결정.**
  - ⇨ LINQPad 등의 유틸리티에서 타입 확인 가능.

40

핵심 정리

- 초기화 되지 않은 변수는 쓰기만 가능.

- 변수에 초기값을 지정하는 방법

```
int n1;
int n2 = 0;
int n3 = new int();
int n4 = default(int);
int n5 = default;
```

← 모두 0 으로 초기화
동일한 IL 코드 생성
LINQPad 등에서 확인가능

- "int n1 = 0" vs "int n2 = new int()"
  - ⇨ C++ 에서는 stack/heap 의 차이
  - ⇨ C# 에서는 동일한 IL 코드 생성
  - ⇨ "int n1 = 0" 권장.
  - ⇨ int 가 아닌 다른 타입인 경우 IL 코드에 차이가 있을 수 있음

"struct, class" 강의 참고

41

핵심 정리

- C#언어의 모든 타입은 "default value" 이라는 개념을 가지고 있다.

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.

- int n4 = default(int);
  - ( ) 안에 있는 타입의 디폴트 값으로 초기화
  - ( ) 생략시, 좌변의 타입의 디폴트 값으로 초기화

42

핵심 정리

- Array
  - ⇨ “Array” 강의 참고

- 실수 ➔ 정수 의 암시적 변환은 허용되지 않음.
  - ⇨ ( ) 를 사용한 명시적 캐스팅 필요
  - ⇨ “casting” 강의 참고

- nameof
  - ⇨ 변수, 형식(Type), 멤버의 이름을 문자열 상수로 생성.
  - ⇨ nameof(color) ➔ “color”

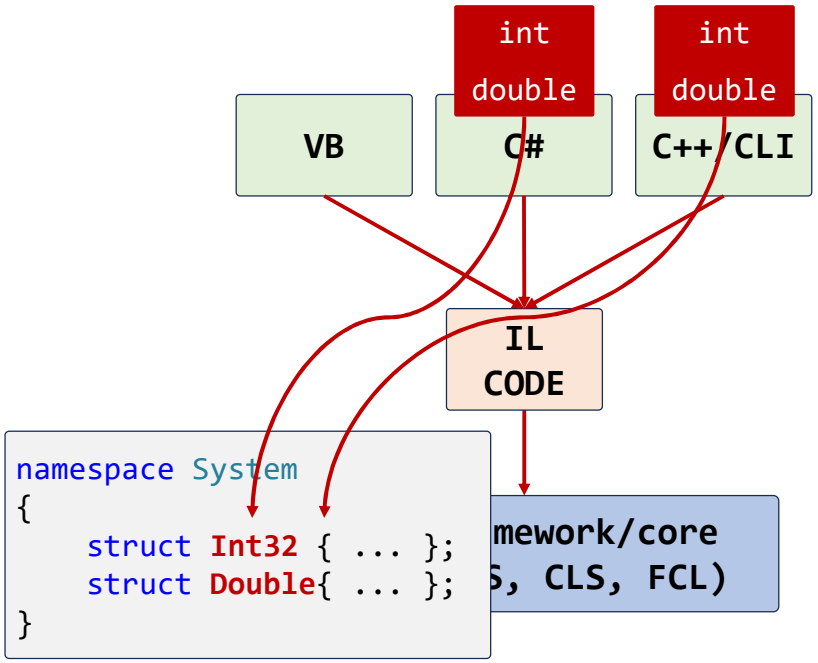
43

Type & Variable #2

44

핵심 정리

- 핵심 1. CTS (Common Type System)
  - ⇨ 모든 타입에 대한 정의는 CTS(Common Type System) 에서 정의 한다.
  - ⇨ C#, VB, C++/CLI 에서 사용하는 타입(키워드)은 "CTS 에서 정의한 타입에 대한 별명(alias)"
  - ⇨ C# 에서는 CTS 에서 정의한 타입 이름도 사용 가능.



45

핵심 정리

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

46

핵심 정리

- 핵심 2. 모든 것은 "객체(object)" 이다.

- C++, Java

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

47

핵심 정리

- C#
  - ⇨ C#이 처음 발표 될 때 "**everything is object**" 라는 슬로건 사용.
  - ⇨ 모든 타입은 class 또는 struct 로 만들어진 타입.
  - ⇨ int(System.Int32) 도 struct 로 만들어진 타입.
  - ⇨ **따라서, 모든 타입은 메소드를 가지고 있다.**
    (모든 객체(변수) 는 메소드가 있고,
    literal 도 메소드를 가진다.)

C# : int, double

```
namespace System
{
    struct Int32 { ... };
    struct Double{ ... };
}
```

CTS
(Common Type System)

48

핵심 정리

- 핵심 3. 대부분의 타입은 "**object(System.Object) 로부터 파생**" 된다.

```
class Car
{
}
```

```
class Car : System.Object
{
}
```

- object(System.Object) 에 대한 자세한 설명은
  ⇨ **"System.Object" 강의 참고**

- 객체지향 언어의 가장 큰 특징
  ⇨ **기반 클래스 타입의 참조(reference)로 파생 클래스 객체를 가리킬 수 있다.**

49

핵심 정리

- 핵심 4. Methods 와 Property

```
class TypeName
{
    Fields  ( member-data )
    Methods ( member-function )

    Constructor
    Finalizers

    Property
    Indexers

    Events
    Operators

    Constants
    Nested Types
}
```

private Field 에 접근하기 위한 문법 getter/setter 개념

"Property" 강의 참고

s.Contains('b')	Method 호출.	() 가 있음
s.Length	Property.	() 가 없음

50

Value type

Reference Type

51

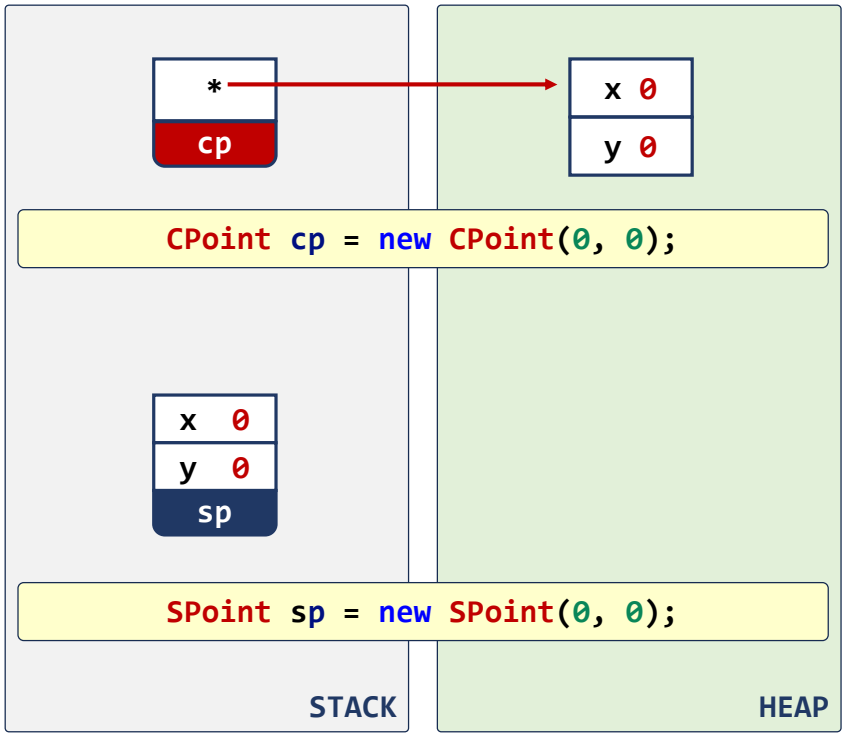
핵심 정리

- class, struct 관련 문법
  - ⇨ “struct, class 강의” 참고
  - ⇨ 이번 강의에서는 “value type” 과 “reference type” 주제만 설명

52

핵심 정리

CPoint	class	Reference type
SPoint	struct	Value type



53

핵심 정리

- C/C++
  - ⇨ 타입 설계자가 아닌 "타입 사용자가 메모리 위치를 결정"
  - ⇨ 동일한 타입이라도 "객체를 생성하는 방법에 따라 메모리 위치 결정"

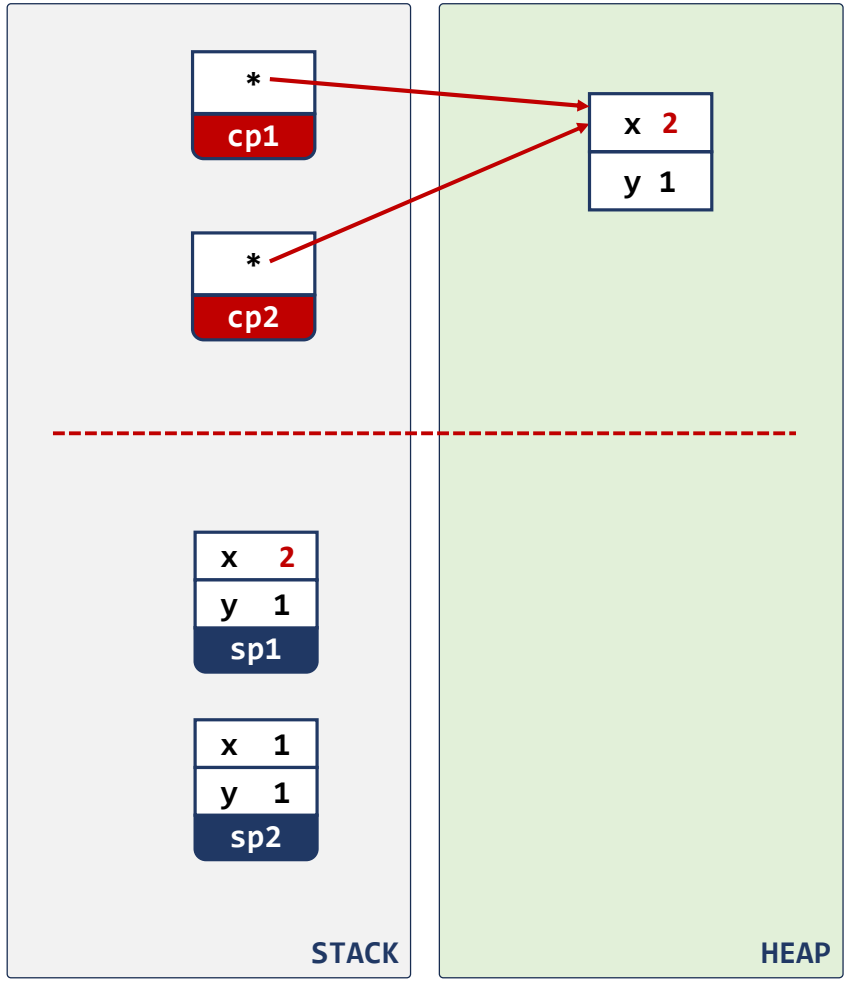
```
CPoint  cp1(0,0);  // stack 에 객체 생성
CPoint* cp2 = new CPoint(0, 0);
                   // Heap  에 객체 생성
```

- C#
  - ⇨ 타입 사용자가 아닌 "타입 설계자" 가 메모리 위치 결정

class	힙에 객체 생성.	Reference Type
struct	스택에 객체 생성.	Value Type

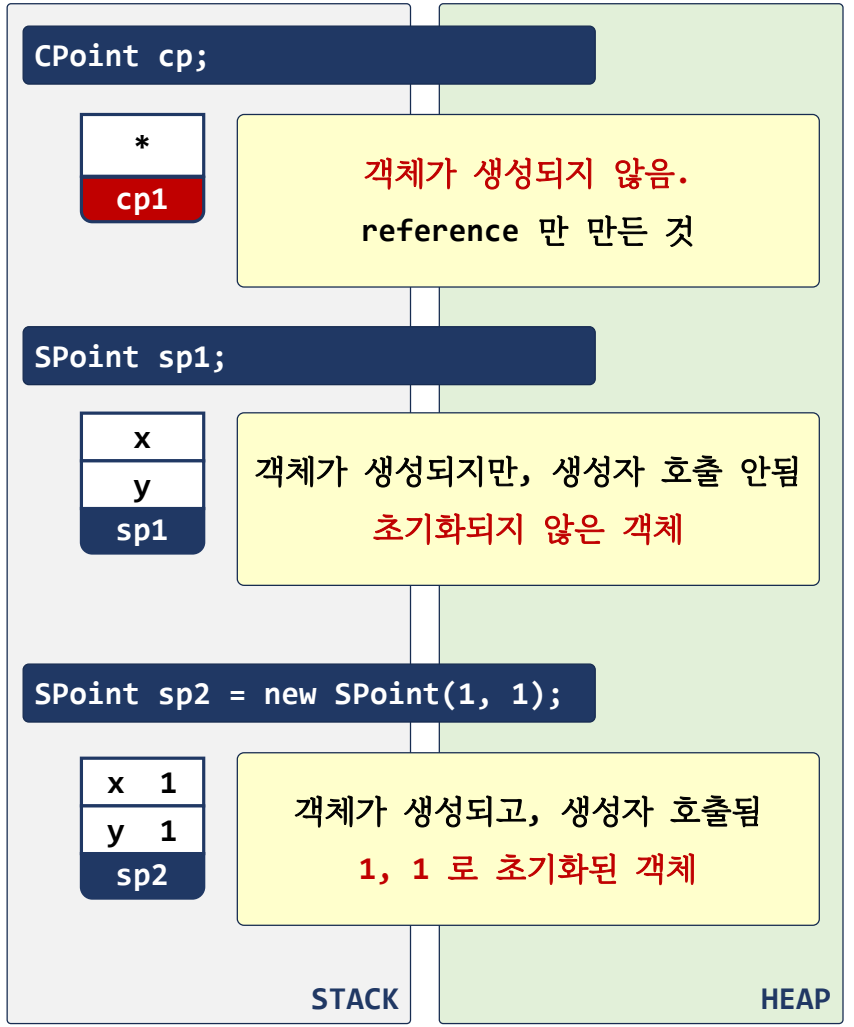
54

핵심 정리



55

핵심 정리



56

핵심 정리

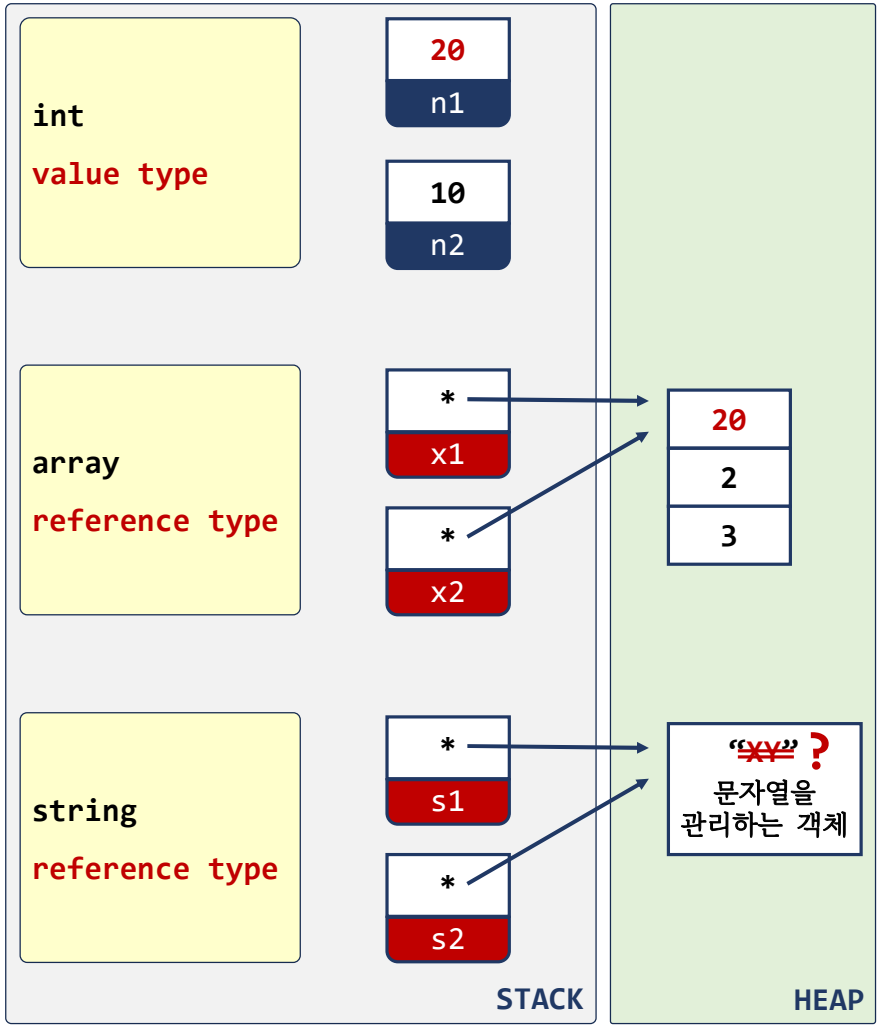
- Value type vs Reference Type

	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

"각각의 주제별 강의" 참고

57

핵심 정리



58

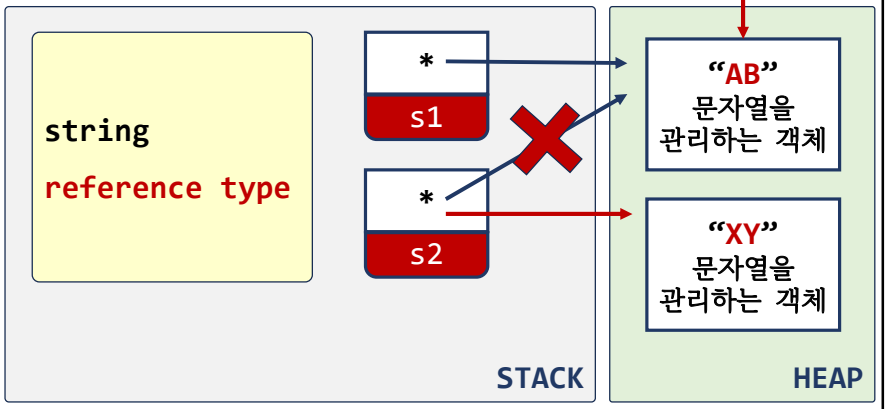
string &

mutable, immutable

59

핵심 정리

string 객체는 “immutable”
문자열(“AB”)을 변경할 수 없다.

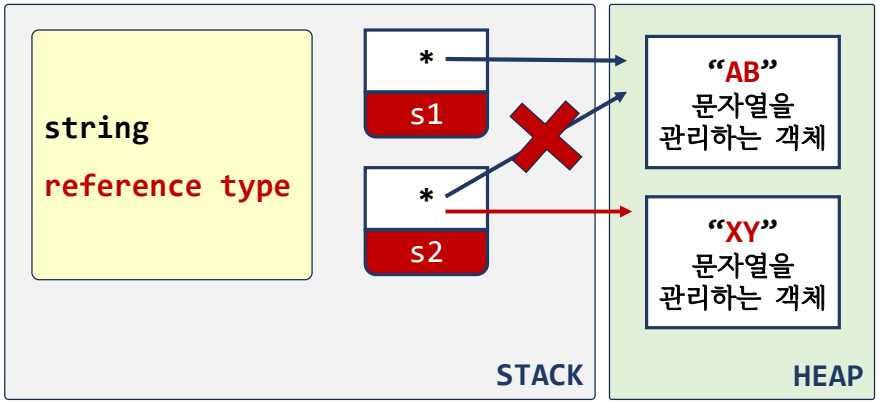


- string 객체
  - ⇨ immutable.

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

60

핵심 정리



- **object.ReferenceEquals(r1, r2)**
  - ⇨ System.Object class 가 제공하는 static method
  - ⇨ reference r1, r2 가 "가리키는 객체가 동일한 객체" 인지 조사
  - ⇨ 인자로 전달되는 r1, r2 는 반드시 reference type 이어야 한다. value type 인 경우 잘못된 결과.

"Equality" 강의 참고

61

핵심 정리

- C# 에서 객체(변수)를 생성하는 방법
  - ⇨ Value Type 과 Reference Type 모두 "new" 를 사용해서 생성하는 것이 원칙.
  - ⇨ 대부분의 표준 타입은 간단한 코드로 변수 생성을 할 수 있도록 하기 위해 "편의(단축) 표기법(syntax sugar)" 제공
  - ⇨ "new" 를 사용하는 경우 과 사용하지 않은 경우가 동일할 수도 있고, 차이점이 있을 수도 있다.

```
int    n1 = new int();
int    n2 = 0;

string s1 = new string("ABCD");
string s2 = "ABCD";
```

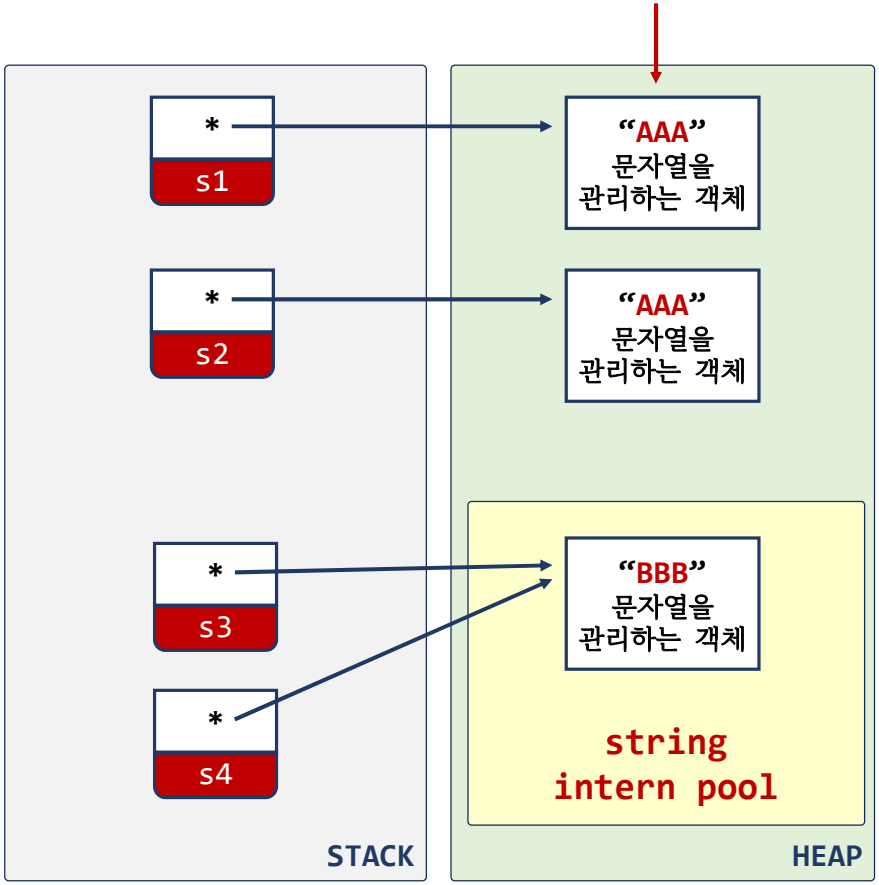
동일한 코드

다르게 동작

62

핵심 정리

"변경할 수 없는 동일한 문자열"을
여러 개 생성하는 것은 비효율적이다.



63

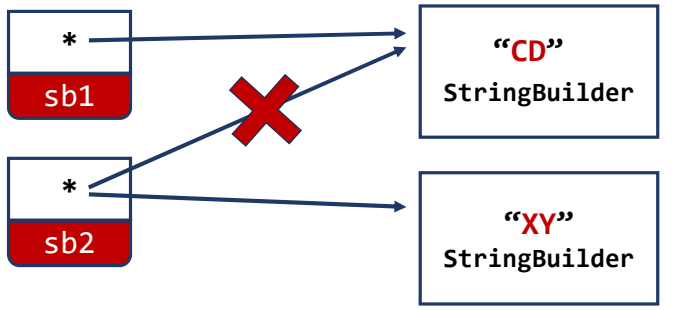
핵심 정리

- C# 의 문자열 타입

string	immutable
StringBuilder	mutable

- StringBuilder
  - ⇨ System.Text.StringBuilder
  - ⇨ "new" 를 사용해서 객체 생성.
  - ⇨ 다양한 메소드와 속성을 활용해서 문자열을 변경(조작) 가능

"상태(문자열) 변경 가능"



64

Tuple

65

핵심 정리

- Array 와 Tuple

Array	“동일 타입” 의 객체를 여러 개 보관
Tuple	“다른 타입” 의 객체를 여러 개 보관

- C# 언어는 2개의 Tuple 타입을 제공

Tuple	Reference Type	immutable	.Net 4.0
ValueTuple	Value Type	mutable	.Net 4.7

기능도 많고, 가볍고 편리해서 널리 사용

66

핵심 정리

- System.Tuple
  - ⇨ **Reference Type**
  - ⇨ “**t1.Item1, t.Item2 ...**” 으로 요소 접근
  - ⇨ 각 Item 은 읽기만 가능
  - ⇨ 최대 8개 까지만 가능
  - ⇨ 8개 이상 사용하려면 “**nested tuple**” 기법 사용

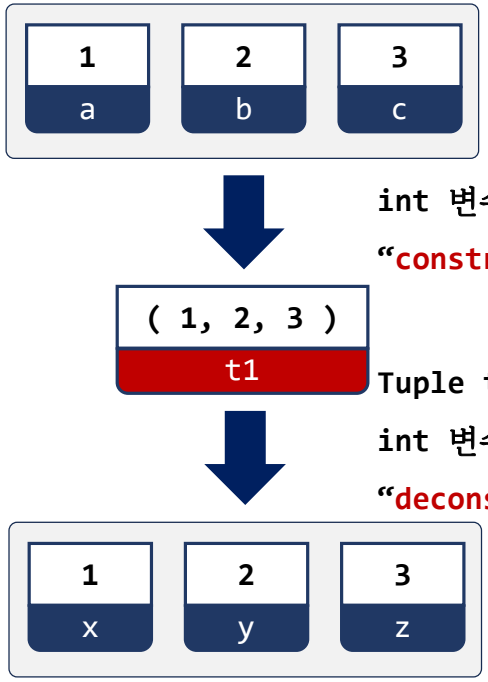
67

핵심 정리

Tuple	**Reference Type**	**immutable**	**.Net 4.0**
ValueTuple	**Value Type**	**mutable**	**.Net 4.7 C# 7.0**

68

핵심 정리



int 변수 3개로 Tuple 생성

"construction"

Tuple t1의 값을

int 변수 3개로 분리

"deconstruction"

69

핵심 정리

tuple 객체 t2 생성

초기값은 (1, 2, 3)

a1, a2, a3 는 멤버의 이름

객체의 이름

```
(int a1, int a2, int a3) t2 = (1, 2, 3);
(int b1, int b2, int b3)    = (4, 5, 6);
```

destruction

tuple (4, 5, 6) 의 값을

변수 b1, b2, b3 에 분리.

b1, b2, b3 은 변수를 선언 한 것

70

Nullable Type

71

핵심 정리

- 프로그래밍 언어에서 "null" 의 일반적인 의미
  - ⇨ "데이타(값) 없음" 을 표현.

- C# 언어와 null

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

⇨ C# 8.0 부터 추가된 개념.

⇨ 8.0 이전에 사용되던 개념을 먼저 설명하고

⇨ 8.0 이후에 추가된 개념을 설명

72

핵심 정리

- reference type 은 null 이 될 수 있다.

```
string s2 = null;
```

이 코드가 컴파일 시에 경고(CS8600)가 발생하는 이유는
C# 8.0 에서 추가된 nullable reference 문법 때문에

73

핵심 정리

- 예제 빌드시 C# 8.0 에서 추가된
  nullable reference 기능을 사용하지 않으려면
  ⇨ 프로젝트 설정 파일 변경

```
<PropertyGroup>
  <OutputType>Exe</OutputType>
  <TargetFramework>net8.0</TargetFramework>
  <ImplicitUsings>enable</ImplicitUsings>
  <Nullable>enable</Nullable>
  <Nowarn>0219, 0168, 8600</Nowarn>
</PropertyGroup>
```

이 부분을 disable 로 변경

또는 이 부분을 추가

- 또는 csc.exe 를 직접 사용

csc 소스.cs -nullable	nullable reference 사용
csc 소스.cs -nullable+	nullable reference 사용
csc 소스.cs -nullable-	nullable reference 사용 안함
csc 소스.cs	컴파일러 버전에 따라 다름.

74

핵심 정리

- **value type 은 null 이 될 수 없다.**
  - ⇨ "데이타(값) 없음" 을 표현할 수 없다.

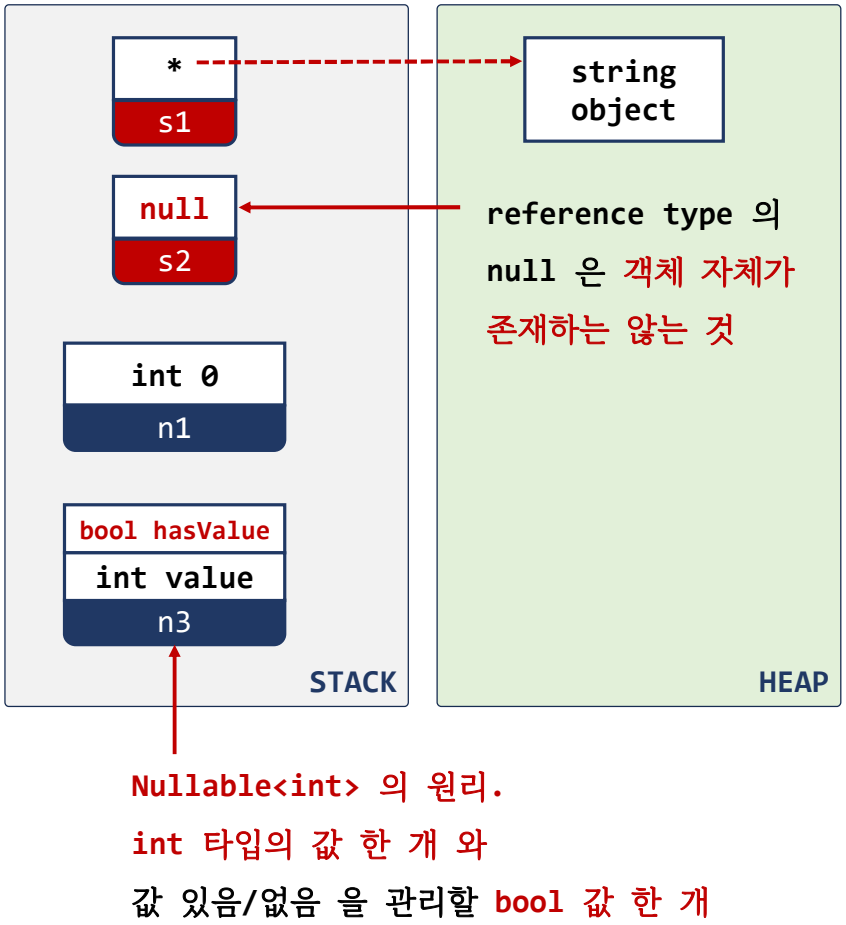
- **Nullable<T>**
  - ⇨ Nullable Value Type (C# 2.0 ~)
  - ⇨ "값 타입 T 가 null 을 가질 수 있는 있도록" 하는 타입.
  - ⇨ T 는 반드시 value type 이어야 한다.
  - ⇨ **Nullable<T> 타입 자체도 value type**

- **int?**
  - ⇨ **Nullable<int> 이 축약 표기법.**
  - ⇨ 이어지는 예제 에서는 "int?" 로 사용.
  - ⇨ "int?" 표기법이 "Nullable<int>" 와 동일하다고 반드시 기억.

75

핵심 정리

- **Nullable<T> 의 원리**



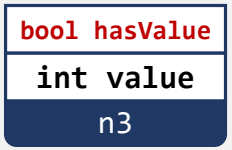
76

핵심 정리

- **Nullable<T> 의 원리**

```
public partial
struct Nullable<T> where T : struct
{
    private readonly bool hasValue;
    internal T value;

    // ......
}
```



77

핵심 정리

- 변수가 null 을 확인하는 방법

	reference type	Nullable<T>
v == null	O	O
v is null	O	O
v.HasValue	X	O

HasValue 는

Nullable<T> struct 가 제공하는 속성

- null 관련 다양한 기법이

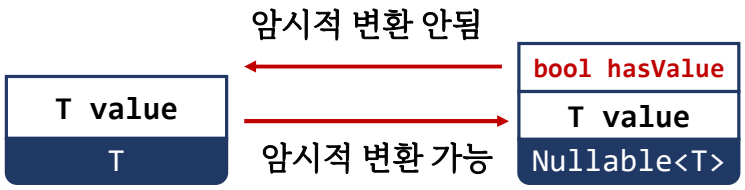
  ⇨ **reference type 과 Nullable<T> 에 공통으로 적용되는지 ?**

  ⇨ 아니면 **"Nullable<T> 에만 적용"** 되는지 잘 구별할 것

78

핵심 정리

- T ⇔ Nullable<T> 사이의 변환



- T ← Nullable<T> 에 담는 방법

`n = (int)ni;`	ni 가 null 인 경우 예외 발생
`n = ni.Value;`	
`n = ni.GetValueOrDefault();`	null 이경우 int 타입 디폴트 값인 0 반환
`n = ni.GetValueOrDefault(3);`	null 인 경우 3을 반환

79

Null & operator

80

핵심 정리

- null 병합연산자(**null-coalescing operator**)

  ⇨ **??, ??=**

- `int n3 = n1 ?? 3;`

n3 != null	**int n3 = n1;**
n3 == null	**int n3 = 3;**

- **n4 ??= 10**

n4 != null	**no operation**
n4 == null	**n4 = 10**

81

핵심 정리

- null 조건부 연산자

  (**Null-conditional operators operator**)

  ⇨ **?. ?[]**

- `c?.Go()`

```
if ( c != null )
{
   c.Go();
}
```

- **int**? ret = c**?.**GetSpeed()

```
int? ret;
if ( c!= null )
   ret = c.GetSpeed();
else
   ret = null;
```

82

핵심 정리

- **is** operator

  o1 이 int 타입으로 변환될 수 있다면 true

```
if ( o1 is int )
{
    int n2 = (int)o1;
}

if ( o1 is int n3)
{
}
```

- **as** operator

```
Type ret = o1 as Type;
```

  - **o1 을 Type 으로 변환 할 수 있다면 변환**
  - **변환 할 수 없다면 null 반환**

  Type 은 반드시 null 가능한 타입 이어야 한다.

83

nullable reference

84

핵심 정리

- null 상태를 가진 객체를 사용하는 것은 위험 하다.
  ⇨ **runtime error**

- 되도록이면 모든 객체는 null 이 아닌 유효한 상태 가지도록 하는 것이 안전한다.
  ⇨ **null을 가질 수 없는 타입을 사용하는 것이 안전.**

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

- Value Type
  ⇨ **null 가질 수 없는 타입과 null 을 가질 수 있는 타입을 개발자가 선택할 수 있다.**

int	**null 을 가질 수 없는 타입**
int?	**null 을 가질 수 있는 타입**

85

핵심 정리

- Reference Type 도 사용자가 선택할 수 있게 할 수 없을까 ?
  ⇨ **C# 9.0 부터 "nullable reference" 문법 도입**

~ C# 8.0	**reference type 은 "항상 null 을 가질 수 있다."** `string s = null;`

C# 9.0 ~	**reference type 도 ? 타입이 존재**
	`string  s1 = null; // warning`
	`string? s2 = null; // ok`

이 기능을 사용하려면 "nullable" 옵션이 필요

86

핵심 정리

- nullable option을 지정하는 방법

- csc 컴파일러를 직접 사용시

  ⇨ **-nullable 옵션 전달**

- 소스 코드에 표기

  ⇨ **#nullable enable**

- .csproj 파일에 표기

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

87

Control Statement & Expression Bodied

88

핵심 정리

- C# 언어의 제어문

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

대부분의 프로그래밍 언어가
지원하는 공통적인 특징

89

핵심 정리

- if ~ else statement
  - ⇨ 조건만족시 실행할 문장이 한줄이면 {} 는 생략가능
    하지만, {}를 사용하는 것을 권장.
  - ⇨ 조건식은 "반드시 bool 타입"이어야 한다.

- switch statement

  - ⇨ 값 뿐 아니라 타입도 매칭 가능

  - ⇨ "switch expression" 개념 지원
    "switch expression" 강의 참고

  - ⇨ 다양한 패턴 매칭 기술 제공
    "pattern matching" 강의 참고

90

핵심 정리

2개의 조건문	if, switch
4개의 반복문	while, do~while, for, foreach (break, continue 사용가능)

x 의 모든 요소를 차례대로 e 에 복사

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

- foreach
  - ⇨ 배열 뿐 아니라 다양한 컬렉션의 요소를 열거 가능
  - ⇨ "enumerator" 를 가지는 모든 타입을 열거 가능

91

핵심 정리

- expression bodied
  - ⇨ 메소드의 구현이 한 문장으로 되어 있는 경우
  - ⇨ 중괄호({}) 대신 "=> 연산자" 를 사용해서 "한개의 표현식으로 메소드 몸체" 를 만들 수 있는 문법

```
void SayHello()
{
    WriteLine("Hello");
}
```

```
void SayHello() => WriteLine("Hello");
```

  - ⇨ 값은 반환하는 경우는 "return 은 표기하지 않고" 반환 값만 표기.
  - ⇨ 메소드 뿐 아니라 "생성자", "property", "indexer" 등 다양한 C# 문법에서 사용가능
  - ⇨ 강의 예제에서도 많이 사용

92

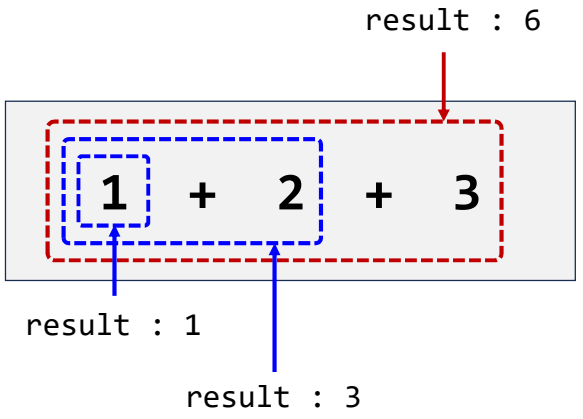
switch expression

93

핵심 정리

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

- **expression**
  - ⇨ “**표현식**” 이라고 번역
  - ⇨ “**하나의 값(result)**” 을 만들어내는 코드 집합
    (**combination of literals, variables, operators, operand, and function(method) invoke**)



94

핵심 정리

- **expression**
  - ⇨ "하나의 값(result)" 을 만들어내는 코드 집합

    a sequence of *operators* and their *operands*, that specifies a computation
  - ⇨ **expression 은 세미콜론(;) 을 포함하지 않는다.**

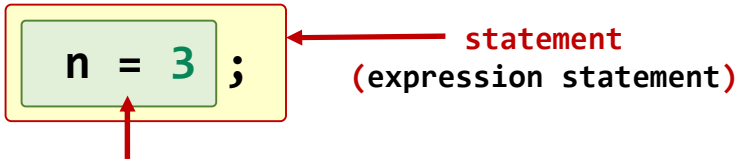
- **statement**
  - ⇨ 프로그램이 실행되는 단위

    Statements are fragments of the C# program that are executed in sequence
  - ⇨ 세미 콜론(;) 으로 끝나는 코드

95

핵심 정리

- **expression 자체는 컴파일/실행될 수 없다.**
  - ⇨ **expression 을 컴파일/실행 하려면 statement 로 만들어야 한다.**
  - ⇨ **expression 을 statement 로 만들려면 끝에 ; 을 붙이면 된다.**



n 에 3을 넣고 최종 결과로 나오는 값은 n

결국 한개의 값으로 결정됨

96

핵심 정리

- **function(method)는 statement 의 집합(sequence)**
  - ⇨ The body of any function(method) is a sequence of statements
  - ⇨ C#에는 다양한 종류의 statement 가 있음.

97

핵심 정리

- **statement 의 종류**
  - ⇨ declaration statement;
  - ⇨ **expression statement;**
  - ⇨ selection statement; **if, switch**
  - ⇨ iteration statement;
    while, do-while, for, foreach
  - ⇨ jump statement;
    break, continue, return, goto, yield
  - ⇨ exception handling statement;
    throw, try-catch, try-finally, try-catch-finally
  - ⇨ checked/unchecked statement;
  - ⇨ await statement;
  - ⇨ yield return statement;
  - ⇨ fixed statement;
  - ⇨ lock statement;
  - ⇨ labeled statement;
  - ⇨ empty statement;

98

핵심 정리

statement2.cs

```
1  int Foo()
2  {
3      int n;                    ← declaration statement
4
5      n = 3;                    ← expression statement
6
7      if ( n == 3 )
8      {
9          int a = 0;            ← declaration statement
10         a = 10;               ← expression statement
11     }                         selection(if) statement
12
13
14     return 0;                 ← jump(return) statement
15 }
16
```

99

핵심 정리

switch statement

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

    default : s1 = "unknown"; break;
}
```

생략 가능

switch expression

```
string s2 = dayofweek switch
{
    0 => "sun",
    1 => "mon",
    2 => "tue",

    _ => "unknown"
};
```

반드시 필요. 생략 할 수 없다

최종적으로 한개의 값으로 결정

100

핵심 정리

- **"if expression" 도 지원 될까 ?**

C++	if, switch 모두 statement expression 지원 하지 않음.
C#	C# 8.0 부터 **switch expression** 지원
Rust **Haskell**	if **expression**, switch **expression**

- **C# 에는 switch, if 등을 사용한 강력한 패턴 매칭 기술 제공**
  ⇨ "패턴 매칭" 강의 참고

101

C# Method

- Named arguments
- Optional arguments
- Params

102

핵심 정리

- 메소드 호출 코드 만으로는 인자 값의 의미가 명확하지 않은 경우가 있다.

```
SetRect( 10, 10, 30, 30 );
```

width, height ?
x2, y2 ?

⇨Positional arguments

- 메소드 "인자에 이름을 사용" 할 수 있다면 보다 명확한 코드를 작성할 수 있다.

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

⇨Named arguments

C# 뿐 아니라 Objective-C, Swift 등의
몇몇 언어에서도 사용하는 기술.

103

핵심 정리

- #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.

104

핵심 정리

- **Optional parameter**
  - ⇨ 메소드 호출시 값을 전달하지 않으면 미리 지정된 값을 사용
  - ⇨ C++등의 언어 에서는 "**default parameter**" 라고 불리는 개념.
  - ⇨ **named argument 와 같이 사용하면 보다 편리하게 인자를 전달할 수 있다.**

- 용어

```
void M1( int a, int b = 0, int c = 0 )
```

Required Parameter — **Optional Parameter**

- 주의!
  - ⇨ Optional Parameter 뒤에 required parameter 가 올 수 없다.
  - ⇨ 마지막 인자 부터 차례대로만 디폴트 값 지정가능.

105

핵심 정리

- **메소드에 인자의 갯수를 상황에 따라 다르게 보내고 싶다면**
  - ⇨ 배열을 인자로 사용하면 된다.

- **메소드 인자로 배열을 사용하는 2가지 방법**

```
M1(int[] ar)
```

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

```
M2(params int[] ar)
```

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

C# 컴파일러가 각각

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

`M2(new int[] {});`

로 변경하는 것

106

핵심 정리

- 주의 사항
  - ⇨ `params` 는 메소드의 마지막 인자에만 사용할 수 있다.

107

Parameter Modifier #1

108

핵심 정리

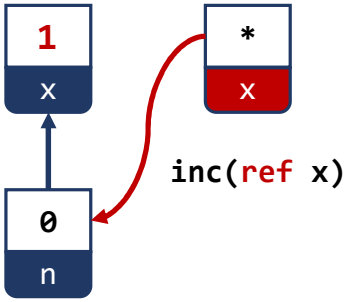
- 메소드에게 인자 전달하는 것은
  - ⇨ 메소드 안에서 사용할 새로운 지역변수를 만들고 인자가 가진 값으로 초기화 하는 것

int 는 value type 이므로
n 의 복사해서 x 변수를 생성

```
void Inc(int x)
{
   ++x;
}

int n = 0;
Inc( n );
```

int x = n;



- ref parameter modifier
  - ⇨ 메소드의 인자를 받을 때 복사본을 만들지 말고 reference 로 해달라는 의미
  - ⇨ "메소드를 만들 때와 호출할 때 모두 표기"해야 한다.

109

핵심 정리

- 메소드 인자가 아닌 "지역변수를 만들 때도 ref 를 사용" 할 수 있을까 ?
  - ⇨ C# 초기 버전에서는 메소드 인자로만 ref 사용가능
  - ⇨ C# 7.0 부터는 메소드의 지역변수와 반환 타입으로도 ref 사용가능
  - ⇨ "ref local", "ref return" 문법

110

핵심 정리

- **AddSub** 메소드
  - ⇨ 2개 정수의 덧셈 결과는 **"반환 값"**으로 알려 주고
  - ⇨ 2개 정수의 뺄셈 결과는 **"마지막 인자로 전달된 변수에 담아"** 주는 메소드

111

핵심 정리

```
void no_modifier_parameter(int x)
{
   1. 복사본 생성
   2. 인자에 대해서 R/W 작업 모두 가능
}

void out_parameter(out int x)
{
   1. 복사본 생성 안됨
   2. 인자 x 에는 Write 작업만 가능하다.
   3. x 에 Write 코드가 없다면 error.
}

void ref_parameter(ref int x)
{
   1. 복사본 생성 안됨
   2. 인자 x 에 대해서 R/W 작업이 모두 가능
   3. x 를 사용하지 않아도 에러 아님.
}
out_parameter(out 초기화되지_않은_변수도_사용가능);
ref_parameter(ref 초기화된_변수만_사용가능);
```

112

핵심 정리

- **Swap 메소드**
  - ⇨ 인자로 전달 받은 2개의 변수의 값을 교환 하는 메소드
  - ① **인자의 복사본을 생성하면 안된다.**
  - ② **Swap 내부적으로 전달받은 인자 값을 R/W 모두 사용하게 된다.** - out 이 아닌 ref 사용.

113

핵심 정리

- 문자열을 정수(int)로 변환
  - ⇨ int.Parse() 또는 int.TryParse() 사용

```
int n1 = int.Parse(“10");     // ok
int n2 = int.Parse("Hello"); // exception
```

  - ⇨ 변환에 성공하면 결과(정수) 반환
  - ⇨ **실패시 예외 발생**

```
bool b = int.TryParse("Hello", out int n);
```

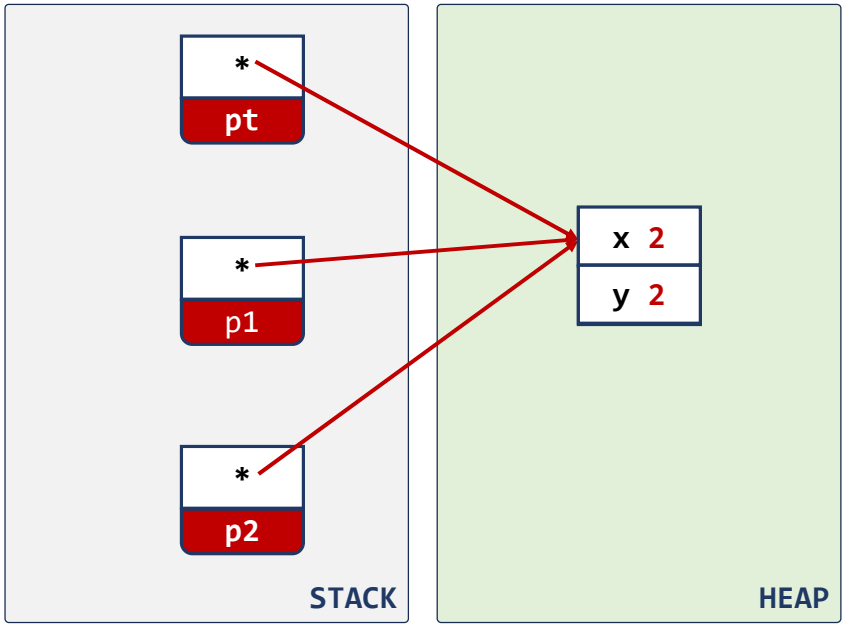
  - ⇨ 성공/실패 여부를 반환 값으로 알려주고(bool)
  - ⇨ **변환 결과는 out 파라미터로 전달.**
  - ⇨ **실패시 out 파라미터에는 0으로 채워줌.**

114

Parameter Modifier #2

115

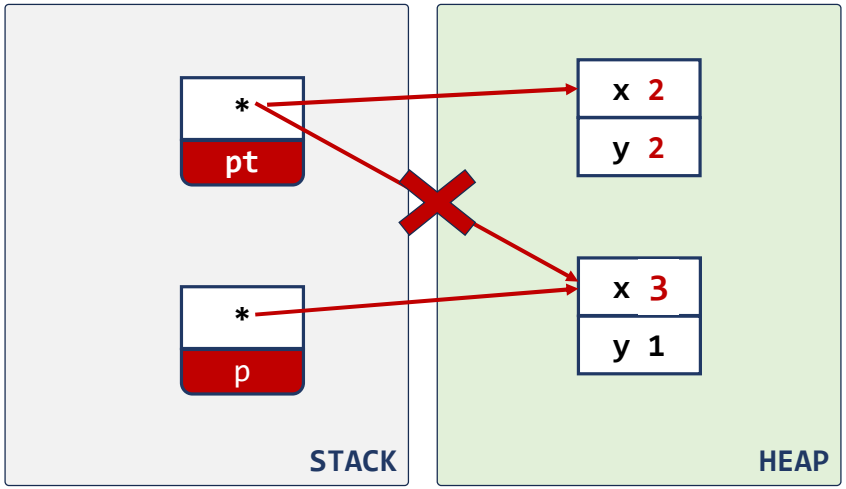
핵심 정리



- reference type 은
  - ⇨ ref, out modifier 를 사용하지 않아도
  - ⇨ "객체의 복사본을 만들지 않고" "reference 로 전달" 된다.

116

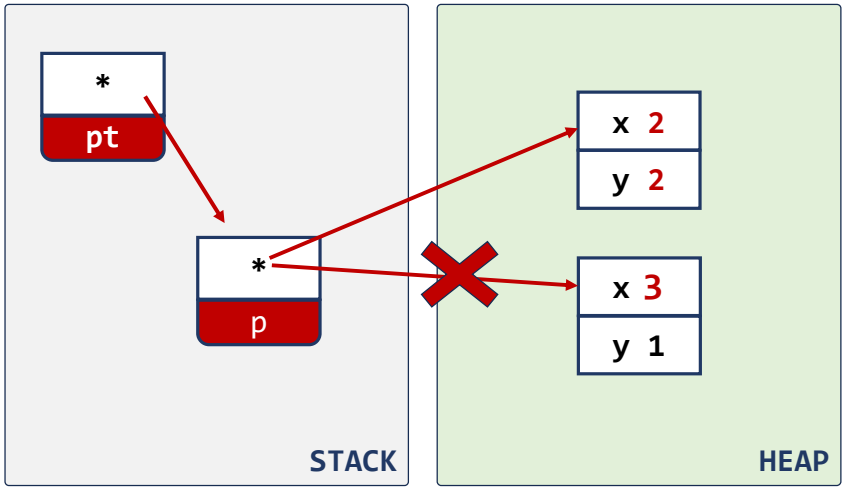
핵심 정리



- reference type 을 ref, out 을 사용하지 않고 전달하면
  - ⇨ 객체를 변경 할 수는 인지만
  - ⇨ "새로운 객체를 생성" 해서 담아올 수는 없다.

117

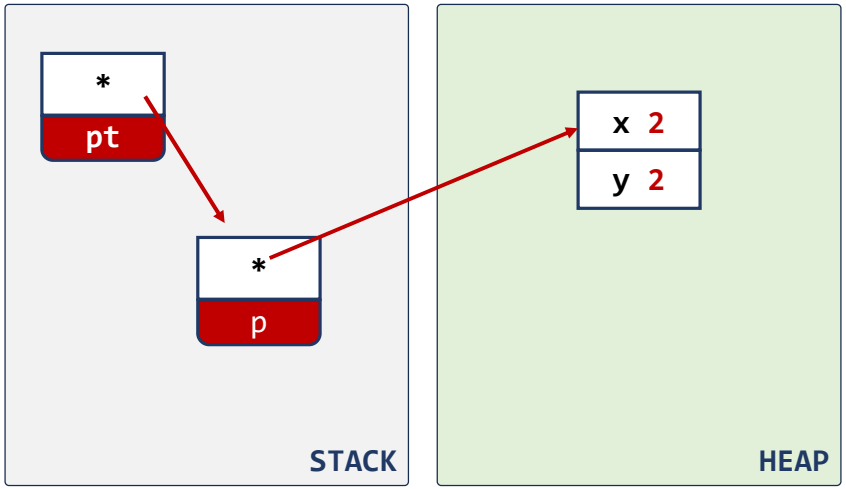
핵심 정리



- reference type 을 ref 사용해서 전달하면
  - ⇨ "객체를 변경 할 수"도 있고
  - ⇨ "새로운 객체를 생성" 해서 담아올 수도 있다.

118

핵심 정리



- **reference type 을 out 사용해서 전달하면**
  - ⇨ 초기화 되지 않은 reference 변수(객체가 없는)도 전달 가능.
  - ⇨ **"객체 자체에는 접근할 수 없고(R/W 모두)"**
  - ⇨ **"반드시 out parameter 에 객체를 할당"** 해야 한다.

119

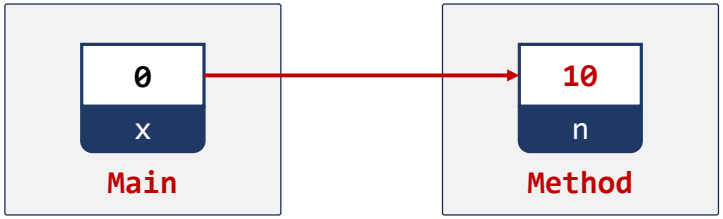
Parameter Modifier #3

in, ref readonly

120

핵심 정리

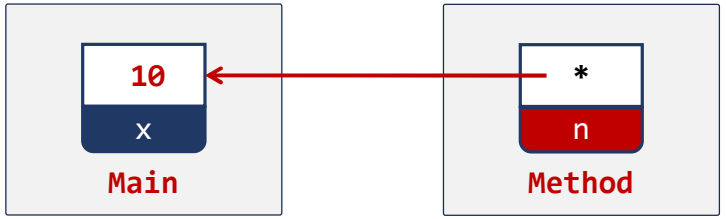
- value type 을 메소드에 인자로 전달하면
  - ⇨ "복사본이 생성" 된다.
  - ⇨ 메소드 안에서 인자를 수정하는 경우 "복사본이 변경" 되므로, 메소드 호출 시 전달한 "원본 객체는 변경되지 않는다.(안전하다)"
  - ⇨ value type 중 "타입의 크기가 매우 큰 경우" "복사본이 생성이 성능에 많은 영향을 준다."



121

핵심 정리

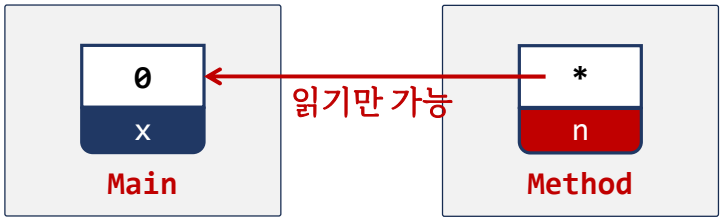
- value type 을 ref 로 전달하면
  - ⇨ reference 로 전달되므로 "복사본이 생성되지 않는다."
  - ⇨ 하지만 "메소드 안에서 객체의 상태를 변경하면 원본 객체도 변경" 된다.
  - ⇨ 의도 하지 않은 실수로 원본 객체의 상태가 변경되는 버그가 발생할 수 있다.



122

핵심 정리

- parameter modifier “**in**”
  - ⇨ C#7.2 에서 추가된 문법
  - ⇨ **value 타입을 reference 로 전달하지만 메소드 내부에서는 읽기만 가능(read only).**
  - ⇨ **크기가 큰 value type 을 메소드 인자로 전달할 때 사용하기 위해 추가된 문법.**
  - ⇨ 메소드를 호출 할 때는 **“in” 을 표기해도 되고, 표기하지 않아도 된다.**



123

핵심 정리

- reference type 에 in 을 사용하면
  - ⇨ 객체의 **“상태를 변경할 수 있지만”**
  - ⇨ **“새로운 객체를 생성”**해서 참조에 담을 수는 없다.

124

핵심 정리

- parameter modifier “**ref readonly**”
  - ⇨ **객체를 reference 로 전달하지만 읽기만 가능하다.**
  - ⇨ “**in**” 의 기능과 유사 (약간의 차이점이 있다)
  - ⇨ **C# 12.0 에서 추가된 문법**

125

핵심 정리

- C# 버전과 ref, ref readonly, in

ref		in		ref readonly	
param	local	param	local	param	local
C#1.0					
	C#7.0				
		C#7.2			C#7.2
				C#12.0	

- **C# 7.0**
  - ⇨ 메소드 인자에만 사용할 수 있었던 ref 를 local 변수를 만들 때도 사용할 수 있게 된다.

- **C# 7.2**
  - ⇨ **ref readonly local 문법 도입.**
  - ⇨ 하지만 “**메소드 인자로는 ref readonly 대신 in 을 사용하는 문법**” 추가.
  - ⇨ **ref readonly parameter 는 C# 12.0 에서 추가됨**

126

핵심 정리

- **ref, ref readonly**
  - ⇨ 기존 객체에 대한 참조로 받는 것이므로 "**이름이 있는 객체(lvalue) 만**" 받을 수 있다
  - ⇨ **rvalue(literal 같이 이름이 없고 값만 있는)을 받을 수 없다.**

- **in**
  - ⇨ "**rvalue 도 받을 수 있다.**"
  - ⇨ literal 과 같은 rvalue 전달시, "**임시로 객체가 생성되어서 메소드 내부에서 사용**"됨.
  - ⇨ 또한 암시적 변환에 의해 생성된 객체도 받을 수 있다.

127

핵심 정리

- **메소드 인자와 lvalue, rvalue, 암시적 변환**

M1(int n)	**복사본 생성 lvalue, rvalue 모두 전달 가능**
M2(ref int n)	**lvalue 만 전달 가능**
M3(ref readonly int n)	**lvalue 만 전달 가능**
M4(in int n)	**lvalue, rvalue 모두 전달 가능. 암시적 변환에 의한 전달도 가능**

128

ref local, ref return

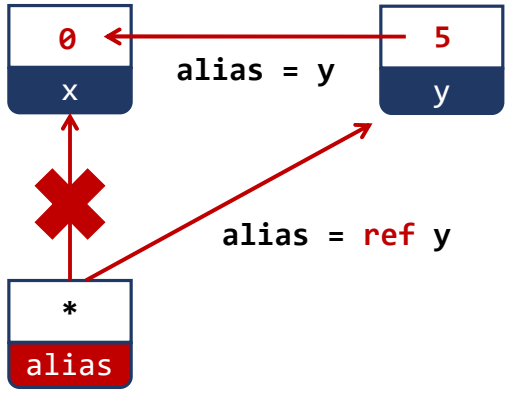
129

핵심 정리

- **ref local**
  - ⇨ 메소드 인자 전달에 사용하던 ref 키워드를 C# 7.0 부터 지역변수에도 사용가능
  - ⇨ 기존 지역변수에 대한 "**alias(reference)**" 를 만드는 문법

```
ref int alias = ref x;
```

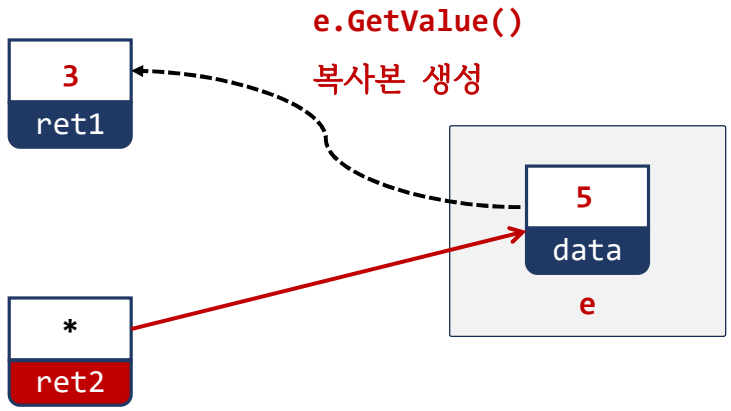
지역변수 x 에 대한 "**reference variable**"



130

핵심 정리

- **ref return**
  - ⇨ C# 7.0 부터 ref 키워드를 반환 타입에도 사용가능.
  - ⇨ ref return 를 사용하면 method 호출을 등호에 왼쪽에 놓을 수 도 있다.



131

static constructor

static class

132

핵심 정리

- **생성자(constructor)**
  - ⇨ 객체를 생성할 때 마다 호출됨.
  - ⇨ 객체(필드)를 초기화 하기 위해 사용.

- **정적 생성자(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.**

133

핵심 정리

- **첫 번째 열차가 출발한 시간을 모든 열차 객체가 공유하고 싶다.**
  - ⇨ 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 의 초기화**
  - ⇨ 생성자에서 초기화 하는 경우 "**객체를 생성할 때 마다 계속 초기값이 지정**" 된다.
  - ⇨ static 생성자에서 초기화 하면 "**최초 객체가 생성될 때 한번만 초기화**" 할 수 있다.

134

핵심 정리

- **class, struct**
  - ⇨ 프로그래밍 세계에서 필요한 다양한 **"타입을 설계"** 하는 도구
  - ⇨ 상태를 나타내는 **"필드"** 와 동작을 나타내는 **"메소드"** 로 구현

- Max, Min등 단순한 연산을 수행하는 메소드가 필요하다.
  - ⇨ C# 에서는 모든 것은 클래스의 메소드로 만들어야 한다. C++ 언어 처럼 **"클래스의 메소드가 아닌 일반 함수는 만들 수 없다."**
  - ⇨ "top level programming" 방식도 결국은 C# 컴파일러가 클래스 코드를 생성하는 것
  - ⇨ 상태(필드)가 필요 없이 단순히 연산만 수행한다면 **"static method"** 로 만들면 된다.

135

핵심 정리

- **static class**
  - ⇨ 모든 멤버가(field, method) 가 static 인 클래스.
  - ⇨ **객체를 생성할 수 없다.**

- C# 의 대표적인 static class
  - ⇨ **System.Console**
    입출력 관련 다양한 static method 제공
  - ⇨ **System.Math**
    수학 관련 다양한 static method 제공

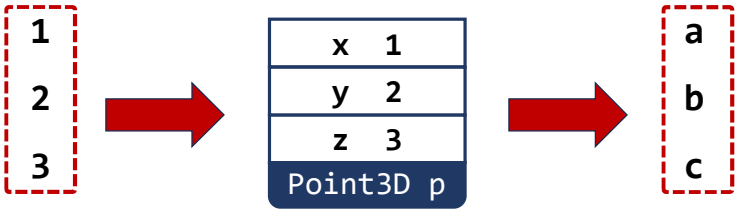
136

Deconstructor &

user define conversion

137

핵심 정리



생성자(constructor)

int 값 3개를 가지고 객체를 생성

Deconstructor

객체의 상태를 int 변수 3개로 분리

```
(int a, int b, int c) = p;
```

```
p.Deconstruct(out int a, out int b, out int c);
```

- **Deconstructor**
  - ⇨ 객체의 상태를 tuple 형태로 얻어올 때 호출되는 약속된 메소드
  - ⇨ "Deconstruct" 라는 약속된 이름 사용

138

핵심 정리

- **Deconstructor**
  - ⇨ **2개 이상 만들 수 있다.**

- **인자가 한 개인 Deconstructor**
  - ⇨ 만들 수는 있지만 "**p.Deconstruct(out int n)**" 형태로 사용.
  - ⇨ 객체를 한개의 다른 타입변수에 넣는 것은 "**객체의 분해(deconstruct)가 아닌 객체의 변환(conversion)**"
  - ⇨ "**user define conversion operator**" 사용

139

핵심 정리

```
Point3D p = new Point3D(1,2,3);
int x = p;
```

Point3D.operator int(p)

```
p = x;
```

Point3D.operator Point3D(x)

- **변환의 2가지 종류**

implicit conversion	`int x = p;` 캐스팅 없이 변환 가능한 것
explicit conversion	`int x = (int)p;` 캐스팅 연산자를 사용해서 변환.

- **user define conversion operator**
  - ⇨ **변환이 필요 할 때 호출되는 약속된 형태의 메소드**
  - ⇨ "**메소드 이름 자체에 반환 타입이 포함**"

140

핵심 정리

- **주의 !**
  - ⇨ 변환을 많은 사용하는 것은 **"버그의 원인"** 이 될 수 있다
  - ⇨ 되도록 사용하지 않은 것이 좋고, 사용한다면 **"암시적 변환(implicit) 보다는 명시적 변환(explicit)"** 를 사용하는 것을 권장.

141

method override

142

핵심 정리

- **method override 란 ?**
  - ⇨ 기반 클래스가 가진 메소드를 "파생 클래스에서 다시 만드는 것"
  - ⇨ 대부분의 객체지향 프로그래밍 언어에서 지원하는 공통적인 기능

- **"new" 키워드**
  - ⇨ 실수가 아니라 의도적으로 override 하는 것이 라고 알려 주는 것.
  - ⇨ new 키워드를 표기하지 않으면 "에러는 아니지만 경고" 발생

143

핵심 정리

객체를 가리키는 reference 는 Animal 타입

객체 자체는 Dog 타입

- ad.Cry() 가 "어떤 함수를 호출하는 것이 논리적으로 맞을까 ?"

java, swift, python C++/C# virtual function	Dog Cry
C++/C#	Animal Cry

144

핵심 정리

- 메소드(함수) 바인딩(Method(Function) Binding)
  - ⇨ "ad.cry()" 메소드 호출 표현식을 "어느 메소드와 연결"할지를 결정하는 과정.

핵심 정리

- static binding(early 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(late binding)
  - ⇨ 컴파일 시간에는 ad가 가리키는 곳을 조사하는 기계어 코드를 생성
  - ⇨ "실행 시간에 ad가 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 언어가사용
  - ⇨ C++/C# 의 virtual method(function)

핵심 정리

- 메소드를 "**dynamic binding**" 되도록 하려면
  - ① 기반 클래스에서 메소드를 만들 때 "**virtual**" 표시
  - ② 파생 클래스에서 **override** 할 때
    "**new 가 아닌 override**" 표시

```
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() { }
}
```

static binding

147

핵심 정리

SetColor()	파생 클래스에서 override 할 이유가 없다 **non-virtual method**
Draw()	도형 마다 그리는 방법이 다르므로 파생 클래스에서 override 하게 된다. "**Shape s = new Rect()**" 일 때 "**s.Draw()**" 는 Rect 를 그려야 한다. **virtual method**

148

핵심 정리

- 메소드가 virtual 이 아닌 경우
  ⇨ 파생 클래스에서는 “new” 만 사용가능

- 메소드가 virtual 인 경우
  ⇨ 파생 클래스에서는 “new” 와 “override” 모두 사용 가능

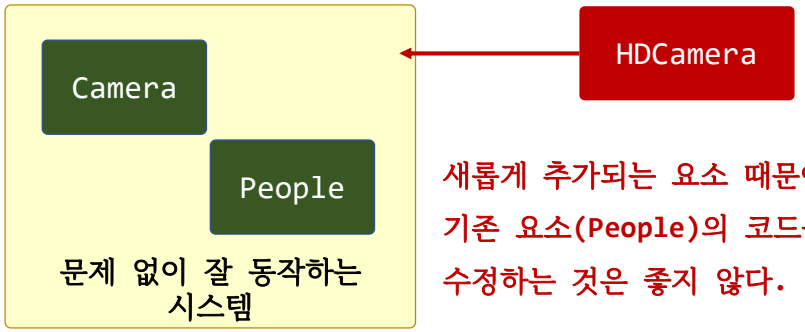
new	기반 클래스 메소드 와는 완전히 다른 함수라고 알려주는 것. “bd.M3()” 는 Base 의 M3() 메소드 호출
override	기반 클래스 메소드를 override. “bd.M4()” 는 Derived 의 M4() 메소드 호출. dynamic binding

149

interface

150

핵심 정리

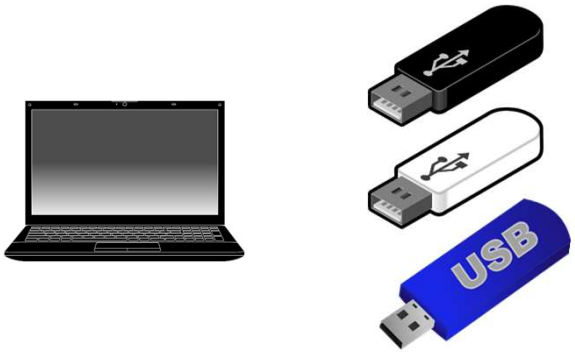


새롭게 추가되는 요소 때문에 기존 요소(People)의 코드를 수정하는 것은 좋지 않다.

- **개방-폐쇄 원칙(OCP, Open-Closed Principle)**
  - ⇨ 소프트웨어 개체(클래스, 모듈, 함수등)는 "확장에 대해 열려(Open) 있어야" 하고, "수정에 대해서는 닫혀(Close) 있어야 한다."는 프로그래밍 원칙(Principle)
  - ⇨ 새로운 요소가 추가 되어도 기존 요소가 변경되지 않도록 설계해야 한다는 원칙

151

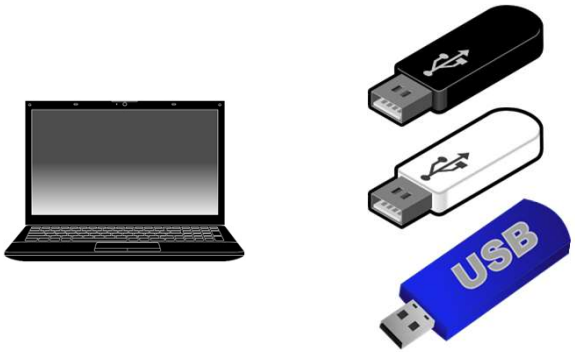
핵심 정리



- 우리가 가진 노트북에는
  - ⇨ 다양한 회사에서 만든 USB 메모리를 연결 할 수 있다.
  - ⇨ 현재의 USB 메모리뿐만 아니라, "미래에 만들어지는 USB 메모리도 노트북 자체를 수정하지 않고 연결" 할 수 있다.
  - ⇨ 새로운 요소(USB 메모리)가 추가되어도 기존 시스템(노트북)을 변경되지 않는다.

152

핵심 정리



- 이것이 가능한 이유는 뭘까 ?
  - ⇨ 노트북 제작자와 USB 메모리 제작자가 지켜야 하는 "규칙이 이미 정해져 있다."
  - ⇨ 각각의 제품을 만드는 사람은 "미리 정해진 규칙" 대로 만들기만 하면 된다.
  - ⇨ S/W 도 규칙을 미리 설계 하면 어떨까 ?

153

핵심 정리

- 카메라 사용자 와 제작자 사이에 지켜야 하는 규칙설계
  - ⇨ interface 문법 사용
- 카메라 사용자(People)
  - ⇨ 특정 제품이 아닌 "규칙의 이름(interface)만 사용."

154

핵심 정리

- 다양한 카메라 제품
  - ⇨ 반드시 규칙을 지켜야 한다.
  - ⇨ 규칙을 담은 "**interface 를 구현**" 되어야 한다.

상속과 동일한 표기법 사용

```
class Camera : ICamera
{
    // 반드시 Take 메소드를 만들어야 한다.
}
```

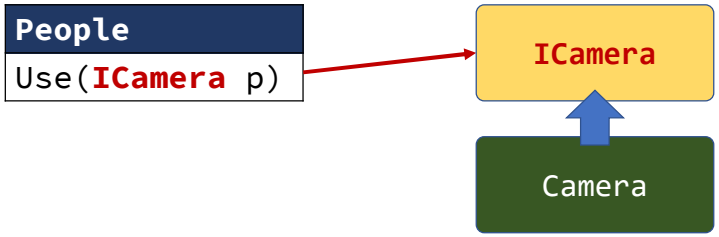
~~모든 카메라는 ICamera 로 부터 파생 되어야 한다.~~

모든 카메라는 **ICamera 인터페이스를 구현해야 한다.**

155

핵심 정리

- 강한 결합 (tightly coupling)
  - ⇨ 객체가 다른 객체와 강하게 결합되어 있는 것.
  - ⇨ **교체가 불가능 하고 확장성 없는 경직된 디자인**



- 약한 결합 (loosely coupling)
  - ⇨ 객체가 다른 객체와 약하게 결합되어 있는 것.
    (인터페이스를 사용해서 통신)
  - ⇨ **교체가 가능하고 확장성 있는 유연한 디자인**

156

interface #2

157

핵심 정리

- 객체의 크기를 비교하고 싶다.
  - ⇨ 비교 연산자(<, >) 또는
  - ⇨ "**CompareTo()**" 메소드 사용

- **a.CompareTo(b)**
  - ⇨ a 와 b 의 크기를 비교하는 메소드

a < b	less than zero (-1)
a == b	zero
a > b	greater than zero (1)

- 크기 비교가 가능한 대부분의 타입에는 **CompareTo** 메소드가 있다.
  - ⇨ "동일한 메소드 이름을 사용하도록 규칙(interface)이 먼저 설계" 되어 있다.

158

핵심 정리

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

```
struct Int32 : IComparable, IConvertible, ...
{
}
class String : IComparable, IConvertible, ...
{
}
```

- **C# 언어의 핵심**
  - ⇨ 어떤 메소드가 몇개의 타입에서 동일한 이름으로 제공된다면
  - ⇨ **"인터페이스를 먼저 설계"**하고 타입을 설계

159

핵심 정리

- **interface 기반 설계의 장점**
  - ⇨ 메소드의 이름이 동일함을 보장하고
  - ⇨ 해당 메소드를 구현한 타입만 인자로 가지는 메소드 등을 설계 할 수 도 있다.

160

핵심 정리

- C# 언어는 "다중 상속이 지원되지 않는다."

```
class A {}
class B {}

class C : A, B Error
{
}
```

- 하지만, "인터페이스를 여러개 구현" 하는 것을 허용.

```
interface IFoo { void Foo(); }
interface IGoo { void Goo(); }

class C : IFoo, IGoo OK
{
   void Foo() {}
   void Foo() {}
}
```

161

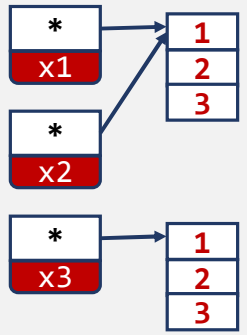
핵심 정리

- **Clone() 메소드**

  ⇨ 객체의 복사본을 반환하는 메소드.

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

int[] x3 = (int[])x1.Clone();
```



162

핵심 정리

```
public interface ICloneable
{
    object Clone();
}
```

- **대부분의 메소드가 인터페이스를 먼저 설계했기 때문에**
  - ⇨ 대부분 메소드의 **"인자와 반환타입은 object"**
  - ⇨ 메소드 구현/호출 시 **"캐스팅하는 코드"**가 많아지고
  - ⇨ **"Boxing/Unboxing 현상"**이 발생 할 수 있다.
  - ⇨ 해결을 위해 C# 2.0 부터 **"Generic interface"** 제공

```
public interface ICloneable<T>
{
    T Clone();
}
```

"Boxing/Unboxing" 강의 참고

163

핵심 정리

- **explicit interface implementation**
  - ⇨ 2개 이상의 인터페이스 안에
    "동일한 signature 를 가지는 메소드가 있을 때"
  - ⇨ 각각 다른 구현을 제공할 수 있는 문법
  - ⇨ 접근지정자는 표기하지 않는다.

164

property

165

핵심 정리

사람(Person) 클래스를 만드는 데,
Person 객체의 "나이(age) 를 변경" 하고 싶다

- **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.**

- **다른 방법은 없을까 ?**
  - ⇨ C# 에서는 **property** 라는 방법이 제공됨.

166

핵심 정리

- Property 코드의 모양

  ⇨ **field 와 method 를 혼합한 형태.**

이 부분은 field 와 유사

```
public  int Age
{
    get { return age; }
    set { if ( value > 0 ) age = value; }
}
```

이 부분은 method 구현과 유사

```
p1.Age = 10;
int n1 = p1.Age;
```

167

핵심 정리

- **Property 의미**

  ⇨ **field 와 method 의 장점을 결합한 문법**

  ⇨ **"사용할 때" 는 field 처럼 편리하고 가독성이 좋다.**

  ⇨ **"구현할 때" 는 메소드 처럼 값의 유효성 을 조사할 수 있다.**

  ⇨ 객체의 private 필드에 접근하기 위해 "메소드 처럼 구현하고 필드 처럼 사용" 하는 문법

- **Property 코딩 관례**

  ⇨ field 는 private 에 만들고,

  ⇨ public property 를 만들어서 field 에 접근.

  ⇨ property 이름은 "대문자로 시작" 하는 것이 관례.

168

핵심 정리

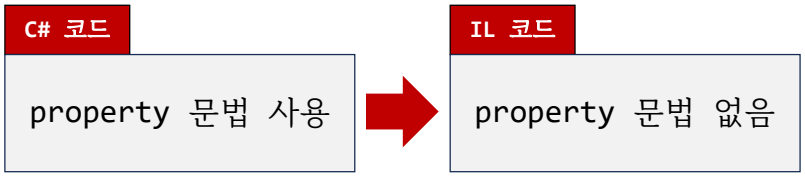
- set{} 안에서의 **value**
  - ⇨ 사용자가 전달한 값을 가진 키워드.
  - ⇨ property 의 setter 와 같이 **특정한 문맥 에서만 사용가능한 키워드.**
  - **(선택적 키워드(Contextual keyword))**

`int if = 0;`	**error.** **키워드 if 를 변수명으로 사용할 수 없다.**
`int value = 0;`	**ok.** **value 는 특정 문맥 안에서만 키워드로 사용(contextual keyword).** **변수명으로 사용가능**

169

핵심 정리

- Property 원리
  - ⇨ property 문법은 C# 언어가 제공하는 문법
  - ⇨ **IL(Intermediate Language) 에는 property 문법이 없음**



170

핵심 정리

```
public int Age
{
  set { age = value; }
  get { return age;  }
}
```

사용자가 작성한 코드를 C# 컴파일러가 변경

```
public void set_Age(int value)
{
    age = value;
}
public void get_Age()
{
    return age;
}
```

```
p.Age = 10;
int n = p.Age;
```

사용자가 작성한 코드를 C# 컴파일러가 변경

```
p.set_Age(10);
int n = p.get_Age();
```

"LINQPad" 등 다양한 도구로 확인가능

171

property #2

172

핵심 정리

- #1. property 의 set, get
  - ⇨ “accessor” 라는 용어 사용
  - ⇨ set accessor
  - ⇨ get accessor
- #2. accessor 구현시
  - ⇨ “식-본문(expression bodied)” 형태로 구현 가능
- #3. “get, set 중에 한 개만 제공” 가능
  - ⇨ read-only, write-only
- #4. 접근 지정자 사용가능
  - ⇨ private set : 멤버 내에서만 사용가능

173

핵심 정리

- 초기화 전용(init only) property
  - ⇨ set accessor 대신 init accessor 사용

init	생성자 또는 Object Initializer 에서만 사용 가능한 property. 그 외에서는 사용할 수 없음.
set	모든 곳에서 사용 가능한 property.

- required property
  - ⇨ 반드시 “Object Initializer” 에서 초기화 되어야 한다.

174

핵심 정리

A property is a member that
"provides a flexible mechanism to **read, write, or compute** the value of a **private field(backing field)**."

- **calculate property**
  - ⇨ swift 등의 언어 에서 사용하는 용어.
  - ⇨ field 에 직접 Read/Write 하는 것이 아니라 **연산을 수행 위해서 사용하는 property.**

175

auto-implemented property

176

핵심 정리

- **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 있다.

177

핵심 정리

- **자동 구현(auto-implemented) property**
  - ⇨ **C# 컴파일러가 자동으로 구현해 주는 property**

```
public int Data2 { set; get; } = 0;
```

C# 컴파일러가
생성한 코드

```
private int <Data2>k__BackingField = 0;

public int Data2
{
    set => <Data2>k__BackingField  = value;
    get => <Data2>k__BackingField;
}
```

178

핵심 정리

- 추가적인 logic 없이 단순하게 Read/Write 만 한다면
  - ⇨ property 를 만들지 말고,
  - ⇨ **public field 를 사용하면 되지 않을까 ?**

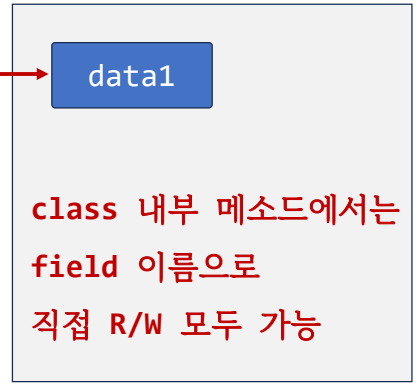
- public field 대신 Property 를 사용하면
  - ① 유연성과 확장성이 뛰어나고
  - ② **Property 만들어야만 사용 할 수 있는 기술이 있다.**
  - ③ **public field 는 Read/Write 이 모두 가능하지만Property 를 사용하면 Read 만 가능하게 할 수 있다**

179

핵심 정리

- **user-implemented property 의 set-only**
  - ⇨ getter 가 없으므로 외부에서는 Read 할 수 없다.

외부에서는 set_Data1() 메소드로 Write 만 가능



- **auto-implemented property 의 set-only**
  - ⇨ field 의 이름을 알 수 없으므로 class 내부에서도 data 를 읽을 수 없게 된다.
  - ⇨ **따라서, set-only property 는 만들 수 없다.**

180

핵심 정리

- user-implemented vs auto-implemented

	user-implemented	auto-implemented
set-only	O	X
get-only	O	O

class 의 모든 메소드 안에서는 private field 를 직접 접근해서 변경(초기화 및 대입) 가능.

단, 생성자에서는 Write 가능

181

핵심 정리

- Name, Address 를
  - ⇨ 생성자에서 모두 초기화 하고
  - ⇨ Name 은 immutable(변경 불가능, read-only), Address 는 mutable(변경 가능, read/write) 하게 하고 싶다.
- public field 을 사용하면
  - ⇨ Name, Address 모두 mutable(변경 가능).
  - ⇨ 언제라도 R/W 가능
- (auto-implemented) property를 사용하면
  - ⇨ 다양한 정책을 사용할 수 있다.

182

핵심 정리

```
public string Name{ get;}
public string Address{ get; set; }
```

⇨ 생성자에서는 Name, Address 모두 초기화 가능

⇨ 객체 생성 이후에는 Name 은 Read 만 가능

```
public string Name{ get; private set;}
public string Address{ get; set; }
```

⇨ Name 은 클래스 내부 메소드 에서만 변경 가능

⇨ Address 는 외부에서도 변경 가능.

```
public required string Name{ get; init; }
public required string Address{ get; set; }
```

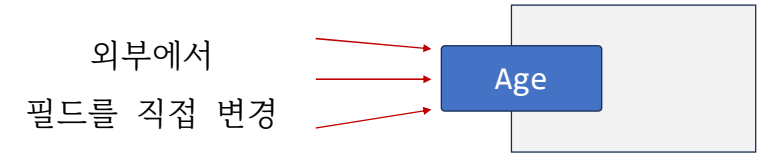
⇨ Name, Address 모두 Object Initializer 에서 반드시 초기화 되어야 함.

⇨ Name 은 초기화 이후에는 변경할 수 없음.

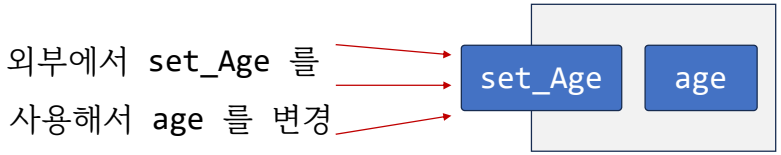
183

핵심 정리

요구 조건의 추가 : 나이를 변경할 때 값의 유효성 여부를 조사하는 코드를 추가하고 싶다.



필드에 접근하는 모든 외부 코드를 수정.



Property 의 set 만 수정하면 된다.
외부의 사용자 코드가 수정될 필요는 없다.

public field 를 사용하지 말고,
property 를 사용 하는 것이 좋다.

184

Indexer

185

핵심 정리

- **Indexer 란 ?**
  - ⇨ 객체를 배열처럼 사용할 수 있도록 하는 문법
  - ⇨ "**[] 연산자를 사용**"해서 Read/Write 가능하게

- **Sentence 클래스 예제**
  - ⇨ 생성자로 전달된 문장을
    단어 별로 분리해서 배열에 보관하고 있다가
  - ⇨ **[] 연산자를 사용해서 각 단어에 개별적으로 접근 할 수 있고,**
  - ⇨ "**Text 라는 이름의 Property**" 로 전체 문장을 얻을 수 있다.

186

핵심 정리

- **Indexer 구현**

  ⇨ Property 의 구현과 유사.

```
public string Text
{
    get { ..... } // get => .....
    set { ..... } // set => .....
}
```

Property

```
public string this[ int idx ]
{
    get { ..... } // get => .....
    set { ..... } // set => .....
}
```

indexer

187

핵심 정리

- **Property vs Indexer**

  ⇨ Property 는 Parameter 를 가질 수 없지만

  ⇨ Indexer 는 Parameter 를 가질 수 있다.

  ⇨ **Parameter 는 반드시 int 일 필요는 없고,**

  ⇨ **2개 이상을 가지도록 만들 수 도 있다.**

188

partial class

189

핵심 정리

- **partial class 란 ?**
  - ⇨ 하나의 클래스를 “여러 개의 파일로 나누어서 작성” 할 수 있게 하는 문법
  - ⇨ class 앞에 “**partial**” 키워드를 붙여야 한다.

Example_a.cs

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

Example_b.cs

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

컴파일 하면 Example 클래스는 2개의 메소드(M1, M2)를 가지게 된다.

190

핵심 정리

- 왜 하나의 클래스를 여러개 파일로 나누어 작업을 하는가 ?
  - ⇨ 하나의 클래스를 여러 명의 개별자가 분야별로 나누어 작성 할 때
  - ⇨ "코드 생성기가 자동 생성한 코드" 와 "인간 개발자가 만드는 코드를 분리" 해서 작업할때

191

핵심 정리

- WPF 와 WinUI3 등의 C# 기반 GUI 라이브러리
  - ⇨ UI 관련 코드는 visual studio 같은 도구가 자동으로 생성하고
  - ⇨ 인간 개발자는 event 를 처리하는 코드만 작성

```
partial class MainWindow
{
    // 코드 생성기가 자동 생성한 UI 를
    // 구축 하는 코드
}
```

MainWindow.xaml.cs

```
partial class MainWindow
{
    // UI 에서 발생한 이벤트를 처리하는 코드.
    // 개발자가 작성
}
```

MainWindow.cs

192

핵심 정리

- **#1. 상속을 사용하는 경우**

  ⇨ 모든 “partial class” 에 표기해도 되고

  “한 곳에만 표기” 해도 된다.

```
partial class Example : Base { }
partial class Example : Base { }
```

```
partial class Example { }
partial class Example : Base { }
```

- **#2. partial class 에서 서로 다른 인터페이스를 사용한 경우, 최종 타입에는 모두 합해진다.**

```
partial class Example : IFoo { }
partial class Example : IGoo { }
```

```
class Example : IFoo, IGoo
{
}
```

193

주의 사항

- **#1. partial class 로 작성하는 경우**

  ⇨ 모든 class 앞에 partial 을 붙여야 한다.

- **#2. 다음의 키워드는 모든 partial class 에서 서로 충돌 되게 작성될 수 없다.**

  ⇨ public, private, protected, internal

  ⇨ abstract, sealed

  ⇨ base class

  ⇨ new modifier(nested parts)

  ⇨ generic constraints

```
abstract partial class Example2{}
sealed   partial class Example2{}
```

```
abstract sealed class Example2
{
}       error. abstract 는 sealed 가 될 수 없다.
```

194

핵심 정리

class struct record interface	partial 로 작성 할 수 있다.
enum delegate	partial 로 작성 할 수 없다.

195

partial method

196

핵심 정리

- **partial method**
  - ⇨ 메소드의 구현이 다른 partial class 에 있다면
    "정상적으로 메소드를 호출"
  - ⇨ 메소드의 구현이 제공되지 않으면 에러가 아니라
    "메소드를 호출하는 코드를 제거"

```
partial class Example
{
    public void Process()
    {
        Init();

        Run();

        Exit();
    }
}
```

```
partial class Example
{
    void Init() {}
    void Exit() {}
}
```

197

partial method 를 사용하려면

- 메소드 구현 앞에 partial 을 붙인다

```
partial class Example
{
    partial void Init() { ..... }
    partial void Run()  { ..... }
    partial void Exit() { ..... }
}
```

- 메소드를 호출하는 partial class 안에도
  "**partial method 의 선언을 제공**" 해야 한다.

```
partial class Example
{
    partial void Init();
    partial void Run();
    partial void Exit();

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

198

핵심 정리

- partial method 은 언제 사용하는가 ?

```
partial class Example
{
    partial void Init();
    partial void Run();
    partial void Exit();

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

```
partial class Example
{
    void Init() {}
    void Run()  {}
}
```

[코드 생성기가 자동으로 생성]
프로그램의 전체적인 흐름(template)을 제공,
각 단계 별로 약속된 메소드 호출.
개발자가 메소드를 만드는 경우 사용하고,
메소드를 제공하지 않으면 사용하지 않음

핵심 정리

- partial method 를 만들 때 주의 사항

① 반환 타입은 반드시 void 이어야 한다.

② 메소드 인자로 out parameter 를 사용할 수 없다.

③ "접근한정자(public, private)" 를 표기하면
안된다. (표기 하지 않지만, 암시적으로 private)

④ "virtual, override, sealed, new, extern"
한정자(modifier) 를 사용할 수 없다.

⇨ static, generic method 도
partial method 로 만들 수 있다.

extension method

201

핵심 정리

- extension method 란 ?
  - ⇨ 기존에 이미 만들어져 있는 타입에
  - ① 기존 타입의 소스 코드를 변경(추가)하지 않고
  - ② 상속을 사용(파생 클래스를 생성)하지 않고
  - ③ 다시 컴파일 하지도 않고
  - ⇨ 새로운 메소드를 추가하는 문법

202

핵심 정리

- extension method 를 만들려면
  - ⇨ **static class 안에 static method 형태로 제공**

static class 이름은 어떤 이름도 상관 없음.

Example 타입에 추가할 것이면 1번째 인자로 Example 객체를 받음.

```
static class MyExtension
{
    public static void M2( Example obj ) {}
    public static void M3( this Example obj ) {}
}
```

- **M1(), M2() 는 MyExtension 의 static 메소드**

```
MyExtension.M2(e);
MyExtension.M3(e);
```

ok. static method 를 호출하는 일반적인 방법

```
e.M2();
e.M3();
```

이렇게 호출될 수 있게 하려면 인자의 타입 앞에 this 키워드필요

핵심 정리

- **기존의 string 타입에는**
  - ⇨ WordCount() 라는 메소드가 없다.
  - ⇨ 하지만 **extension method 를 사용하면 개발자가 추가**할 수 있다

- **string 클래스는 이미 컴파일 된 형태로 배포된다.**
  - ⇨ extension method 를 사용하면, **"string 타입을 다시 컴파일 하지 않고도 메소드를 추가"** 할 수 있다.

- **LINQ (Language INtegrated Query)**
  - ⇨ C# 언어의 핵심 기술 중 하나
  - ⇨ extension method 문법으로 만들어져 있다.

System.Object

205

핵심 정리

- C#의 대부분의 타입은 “**object(System.Object) class 로 부터 파생**” 된다.

```
class Car
{
}
```

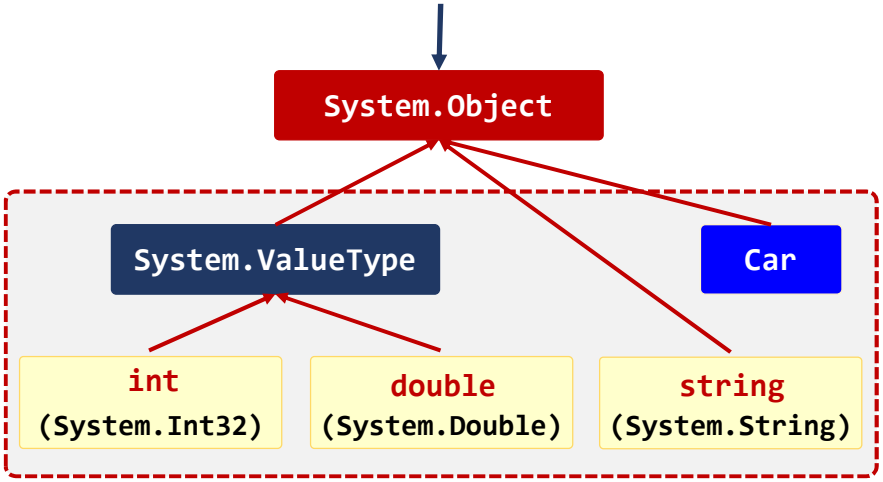
```
class Car : System.Object
{
}
```

⇨ 단, “**ref struct**” 는 System.Object 로 부터 파생되지 않은다.

206

핵심 정리

System.Object 제공하는 메소드는
C# 의 대부분의 타입에서 사용가능 하다.



⇨ C#의 대부분 타입은 System.Object 로 부터 **동일한 메소드(기능)을 상속** 받게 된다.

⇨ **C#의 대부분의 객체는 object 타입으로 가리킬 수 있다**

207

핵심 정리

- System.Object 는 **7개의 메소드 제공**

```
public virtual
bool Equals(object? obj);

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

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

protected object MemberwiseClone();

public virtual int GetHashCode();

public Type GetType();

public virtual string? ToString();
```

⇨ **static method 2개,**
instance method 5개(virtual method 2개)

⇨ protected 1개, public 6개

208

핵심 정리

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

⇨ 객체의 동일성(Equality) 를 조사하는 기능 제공.

⇨ "Equality" 강의 참고

```
object MemberwiseClone();
```

⇨ 객체의 얕은 복사본을 만드는 기능 제공.

⇨ "Object Copy" 강의 참고

```
virtual int GetHashCode();
```

⇨ 객체의 해쉬 코드값

⇨ "Collection" 강의 참고

```
Type GetType();
virtual string? ToString();
```

이어지는 예제에서 설명

209

핵심 정리

- method의 인자가 **object** 인 경우

⇨ **object 는 대부분의 타입의 기반 클래스 이므로**

⇨ **대부분의 객체를 인자로 받을 수(가리킬 수) 있다.**

⇨ 이때, object 타입의 reference 가 가리키는 **객체가 어떤 타입인지 알고 싶다면** 타입 정보를 담은 Type 객체를 꺼내면 된다.

```
Type t = obj.GetType();
```

⇨ obj reference 가 가리키는 객체에 대한 정보를 담은 "Type" 타입의 객체 반환

⇨ System.Object 에서 제공하므로, **대부분의 C# 객체는 GetType() 메소드가 있다.**

210

핵심 정리

- Type
  - ⇨ **타입에 대한 정보를 담는 타입**
  - ⇨ 타입에 대한 다양한 정보를 반환 하는 method 와 property 제공. (MS 도움말 참고)

- Type 객체를 얻는 방법
  - ⇨ **Type t = 객체.GetType();**
  - ⇨ **Type t = typeof(타입);**

- obj 객체가 int 타입인지 조사하려면

```
if ( obj is int ) {}
if ( obj.GetType() == typeof(int) ) {}
```

211

핵심 정리

- System.Object 클래스는
  "ToString() instance method" 를 제공
  - ⇨ **C#의 대부분의 객체는 ToString() 메소드를 가진다.**

- **ToString() 메소드**

  ① **객체의 상태를 문자열로 얻고 싶을 때 사용하는 메소드**

  ② System.Object 의 기본 구현은
  "객체의 타입 이름 반환"

  ③ **virtual method 이므로 사용자가 override 해서 원하는 방식으로 구현 가능**

```
public virtual string? ToString();
```

212

핵심 정리

- **System.Object**
  - ⇨ **7개의 method 를 제공**
  - ⇨ **C#의 대부분의 타입은 7개의 메소드를 가지게 된다**

ToString()	**객체의 상태를 문자열로 얻고 싶을 때 사용** virtual method 이므로 타입 설계자가 override 해서 원하는 방식으로 구현하는 것이 관례
GetType()	**객체의 타입 정보를 얻고 싶을 때 사용** virtual method 가 아니므로 override 하는 것은 아님.
Equals() Equals() ReferenceEquals() MemberwiseClone() GetHashCode()	각각의 주제를 배울 때 설명.

213

Equality

214

핵심 정리

- 객체의 "**동일성(Equality) 을 조사하는 4가지**" 방법

```
bool b1 = objA == objB;
bool b2 = objA.Equals(objB);
bool b3 = object.Equals(objA, objB);
bool b4 = object.ReferenceEquals(objA, objB);
```

핵심

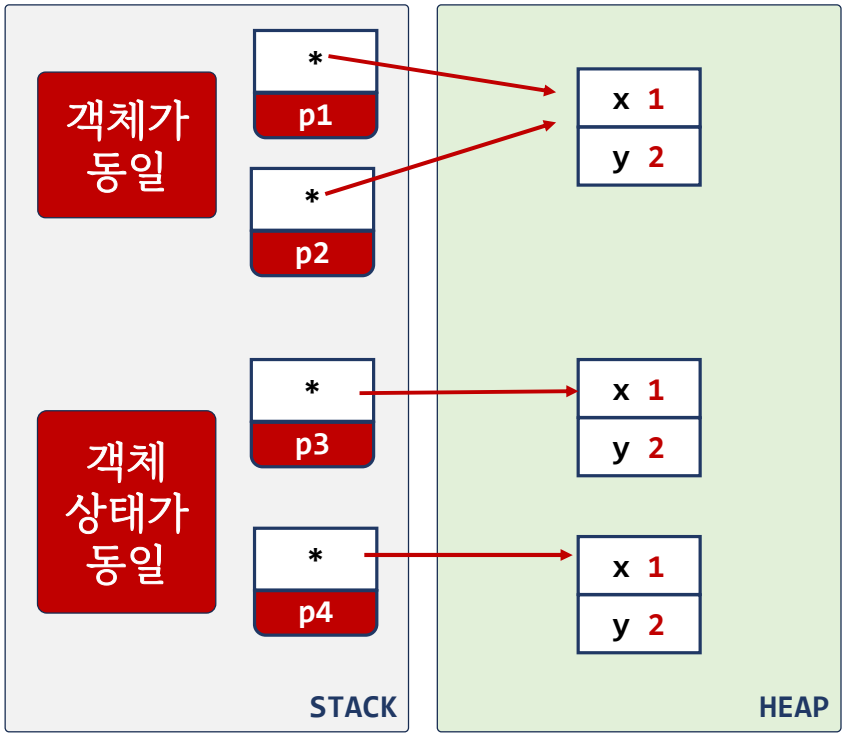
System.Object method

- reference type 과 value type 의 차이점
  - ⇨ **reference type 먼저 학습**

215

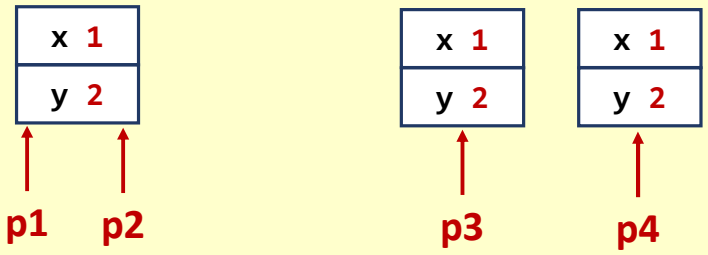
핵심 정리

- 동일성(Equality) 의 2가지 개념
  - ⇨ **객체(인스턴스)가 동일 한가?**
  - ⇨ **객체의 상태가 동일 한가 ?**



216

핵심 정리



⇨ 객체가 동일

⇨ 객체의 상태도 동일

⇨ 객체는 다름.

⇨ 객체 상태는 동일

- == 연산자
  - ⇨ 동일한 객체인지를 조사

- **Equals() virtual method**
  - ⇨ System.Object 가 제공하는 기본 구현은 "== 연산자를 사용" 해서 조사

```
public virtual bool Equals(object? obj)
{
    return this == obj;
}
```

⇨ **virtual method 이므로 override 해서 다른 구현을 제공 가능.**

핵심 정리

- 객체의 동일성이 아닌 "상태의 동일성" 을 조사하려면
  - ⇨ Equals() 가상 메소드를 override 해서 상태의 동일성을 조사하는 코드를 제공하면 된다.

  - ⇨ Equals() 메소드를 override 할 때 는 "**GetHashCode() 도 같이 override**" 하는 것이 관례이다.

```
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;
}

public override int GetHashCode()
{
   return x.GetHashCode() + y.GetHashCode();
}
```

핵심 정리

- == 와 Equals() method

a == b	동일한 객체인지 조사
a.Equals(b)	override 하지 않으면 “==” 와 동일. override 해서 다른 구현 제공 가능.

219

핵심 정리

- Equals(a, b) static method

a == b	동일한 객체인지 조사
a.Equals(b) instance method (virtual)	override 하지 않으면 “==” 와 동일. override 해서 다른 구현 제공 가능. 관례는 “상태의 동일성” 조사하는 코드 제공.
object.Equals(a, b) static method	1. a == b 로 조사 2. a.Equals(b) 로 조사

220

핵심 정리

- **object.Equals(a, b) static method**

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

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

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

⇨ ==로 먼저 조사후, false 인 경우만
a.Equals(b) 로 조사

⇨ virtual method 가 아니므로 사용자가 override 할 수 없다.

221

Equality #2

222

핵심 정리

- **"operator=="**
  - ⇨ 기본적으로 동일한 객체인지 조사
  - ⇨ **연산자 재정의 문법으로 "==" 연산자를 재정의 가능.**
  - ⇨ == 연산자 재정의시, != 연산자도 같이 재정의하는 것이 관례.

- **"operator==" 연산자를 재정의한 타입에서 동일한 객체인지 조사하려면**
  - ⇨ 객체를 **"object 타입으로 캐스팅해서 ==**" 를 사용하면 된다.

223

핵심 정리

- **a == b**
  - ⇨ 동일한 객체인지 조사
  - ⇨ 연산자 재정의로 다른 구현 제공 가능.(권장 안함)

- **a.Equals(b) instance method**
  - ⇨ 기본 구현은 "==" 사용
  - ⇨ **"상태의 동일성 조사 방식으로 override**" 하는 경우 많음.

- **object.Equals(a, b) static method**
  - ⇨ 내부적으로 a == b 와 a.Equals(b) 를 사용.
  - ⇨ 구현 변경 안됨

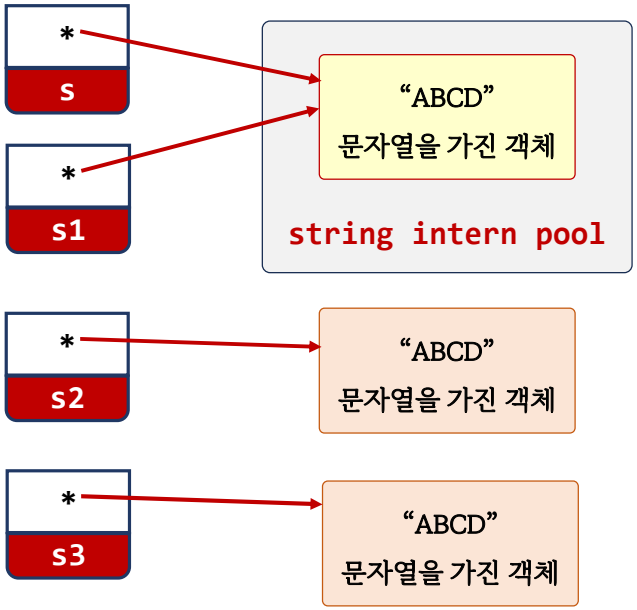
- **object.ReferenceEquals(a, b) static method**
  - ⇨ 동일한 객체인지 조사
  - ⇨ 구현 변경 안됨

동일한 객체인지 조사하는 가장 안전한 방법

224

핵심 정리

- **string intern pool**
  - ⇨ 메모리 사용량을 줄이기 위해
  - ⇨ **동일한 문자열을 사용하는 객체는 pool 에 공유.**



225

핵심 정리

- **"a == b" 에서 a, b 가 string 일 때**

  ⇨ 객체가 동일한지 조사하는 것이 아니라

  ⇨ **문자열(객체의 상태)이 동일한지 조사**

  ⇨ 객체가 동일한지 조사하려면

  "object.ReferenceEquals()" 사용

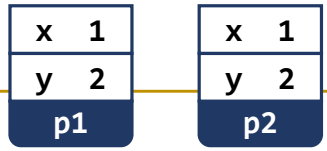
226

ValueType Equality

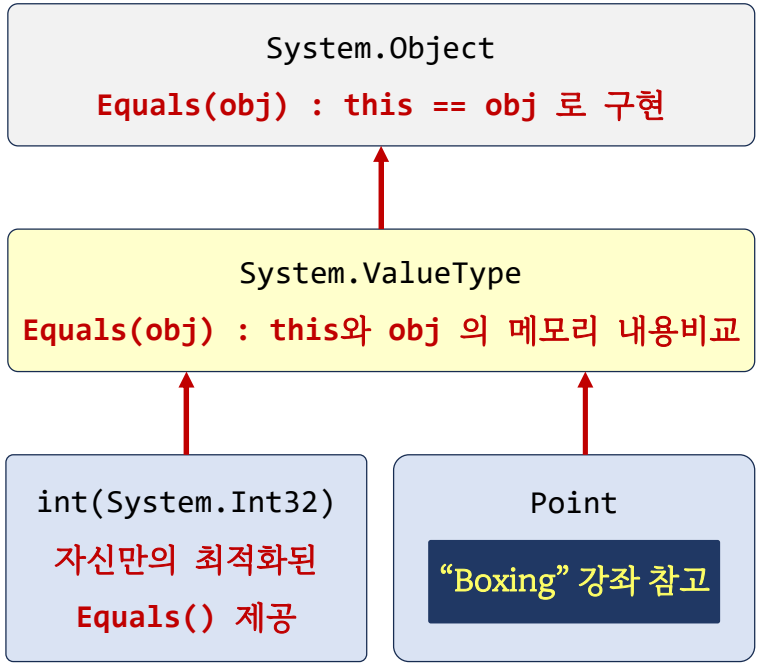
227

핵심 정리

- ValueType Equality



p1 == p2	error. 연산자 재정의 문법으로 제공 가능.
p1.Equals(p2)	p1, p2 의 메모리 내용을 비교



228

핵심 정리

- value type 에서 “object.Equals(a, b)”
  - ⇨ 사용가능 하지만
  - ⇨ 오버헤드가 있으므로 사용하지 않는 것이 좋다.
  - ⇨ “Boxing/Unboxing” 강좌 참고

- object.ReferenceEquals()
  - ⇨ value type 에 대해서는 사용하면 안된다.
  - ⇨ 잘못된 결과가 나오게 된다.
  - ⇨ “Boxing/Unboxing” 강좌 참고

229

핵심 정리

	Reference Type	Value Type
a == b	O	X (연산자 재정의 문법으로 제공 가능)
a.Equals(b)	O	O (메모리의 내용을 비교)
Equals(a, b)	O	O (Boxing 현상 발생)
Reference Equal(a, b)	O	X (에러는 아니지만 경고 발생)

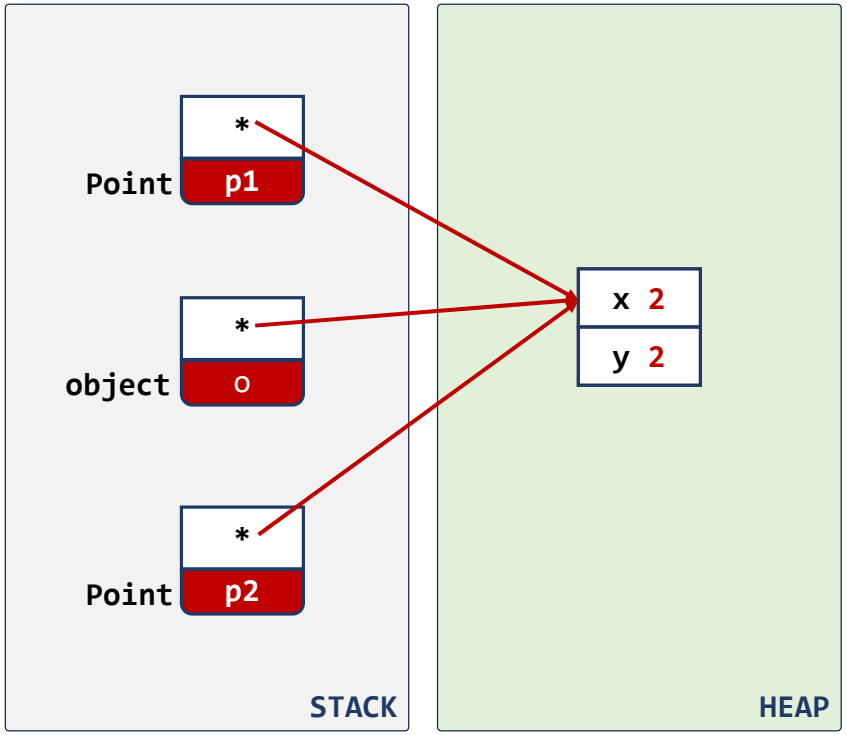
230

Boxing/Unboxing

231

핵심 정리

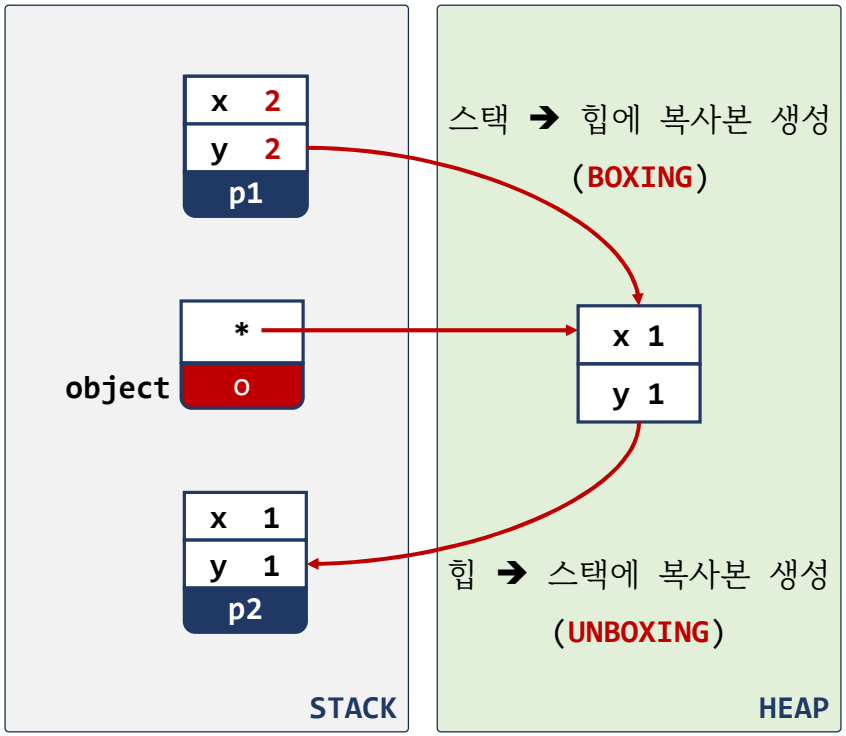
- Point 가 class 인 경우
  - ⇨ reference type



232

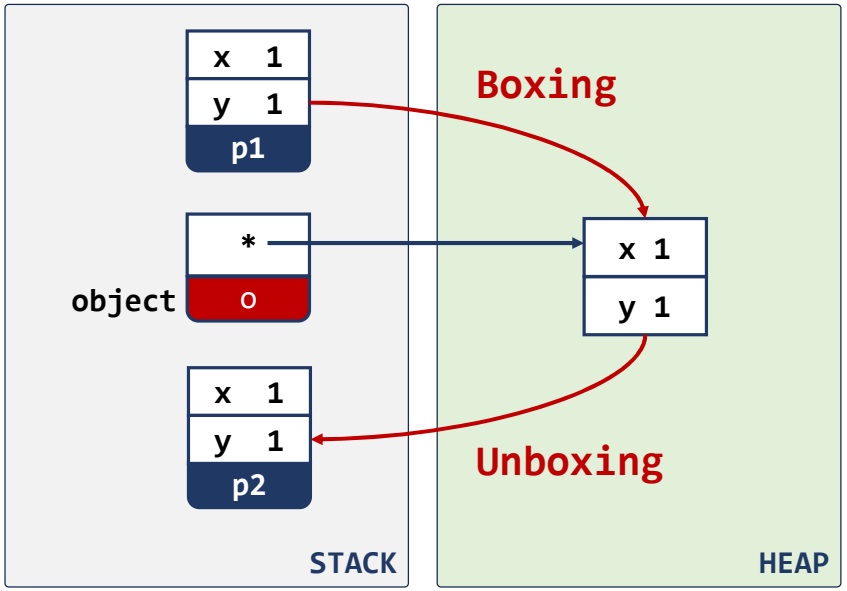
핵심 정리

- Point 가 struct 인 경우
  - ⇨ value type



233

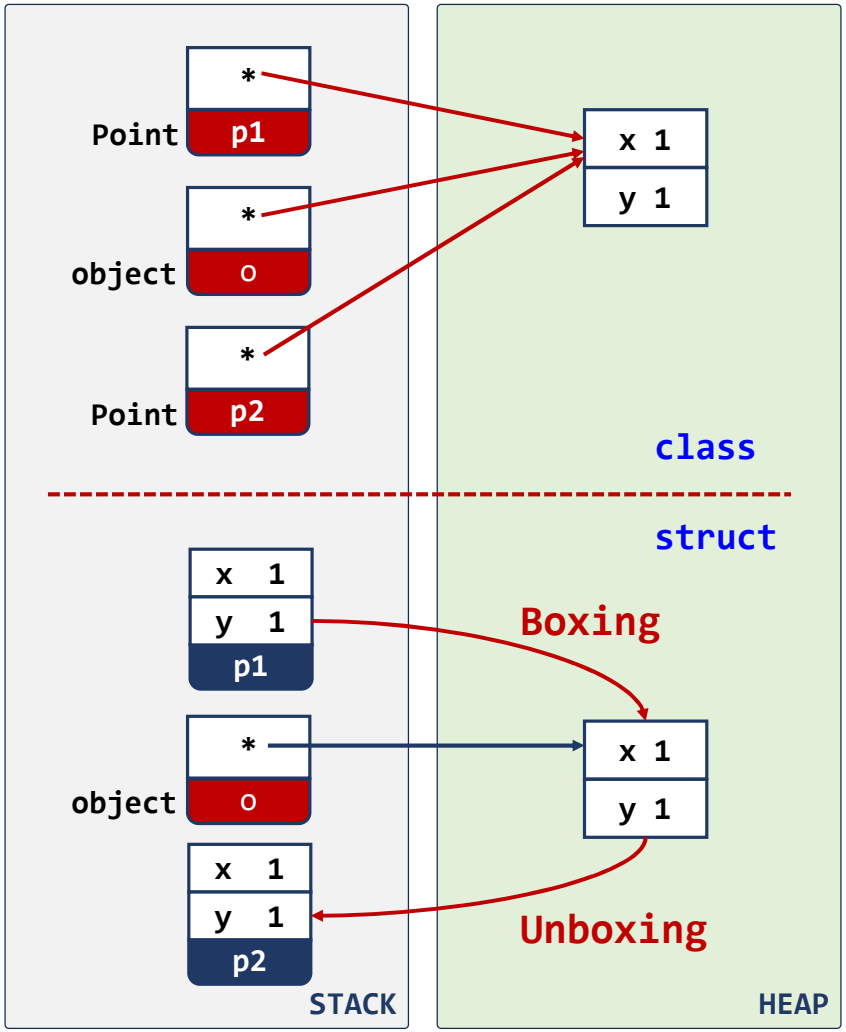
핵심 정리



- Boxing/Unboxing
  - ⇨ value type 의 객체를 reference type 의 참조(reference)로 가리킬 때 **힙에 복사본이 만들어지는 현상이 Boxing.**
  - ⇨ 복사본을 가리키는 참조를 다시 value_type 으로 캐스팅 할 때 **stack 에 복사본을 만드는 것을 Unboxing.**

234

핵심 정리



235

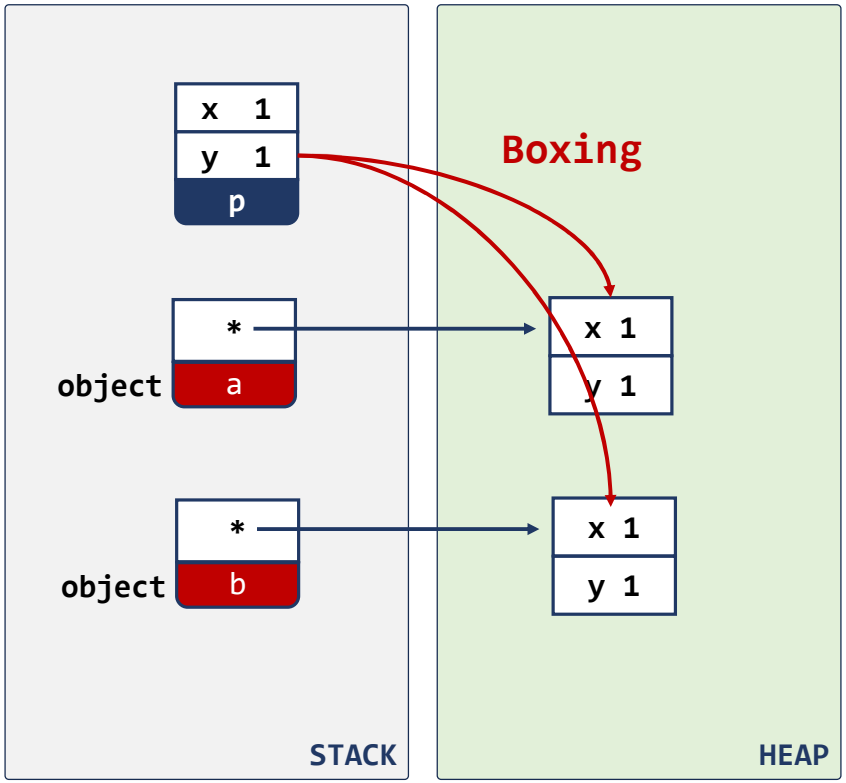
핵심 정리

- **Boxing/Unboxing**
  - ⇨ 복사본이 생성되므로 **성능 저하의 원인이** 될 수 있다.
  - ⇨ **되도록 Boxing/Unboxing 현상이 발생하지 않도록 코드를 작성하는 것이 좋다.**

236

핵심 정리

- **object.ReferenceEquals(a, b)**
  - ⇨ a, b 가 동일한 객체를 가리키는지 조사할 때 사용
  - ⇨ **단, value type 의 경우는 사용하면 안된다.**



237

Boxing/Unboxing #2

238

핵심 정리

- C#에서 객체의 크기를 비교하려면
  - ⇨ "**CompareTo**" 메소드를 사용하는 경우가 많다.
  - ⇨ "**IComparable**" 인터페이스로 메소드 이름을 약속

- **IComparable** 인터페이스

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

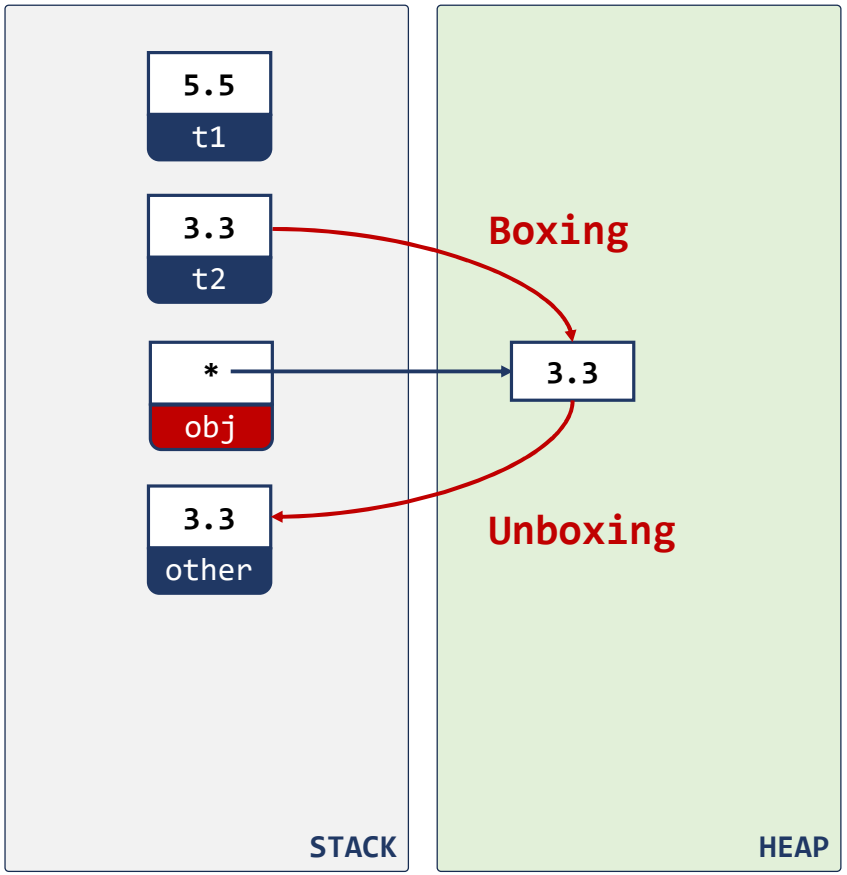
- 사용자 정의 타입 설계시
  객체의 크기를 비교할 수 있게 하려면
  - ⇨ "**IComparable 인터페이스를 구현**" 하면 된다

- a.CompareTo(b) 반환 값
  - ⇨ a > b 인 경우 1 반환
  - ⇨ a < b 인 경우 -1 반환
  - ⇨ a == b 인 경우 0 반환

239

핵심 정리

- IComparable 인터페이스를 구현하는 경우
  - ⇨ int CompareTo(object? obj)



240

핵심 정리

- C# 1.0 에서 제공하는 인터페이스
  - ⇨ 대부분의 메소드가 인자로 object 사용

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

  - ⇨ value type(struct) 만들 때 사용하는 경우
    Boxing/Unboxing 현상 발생.

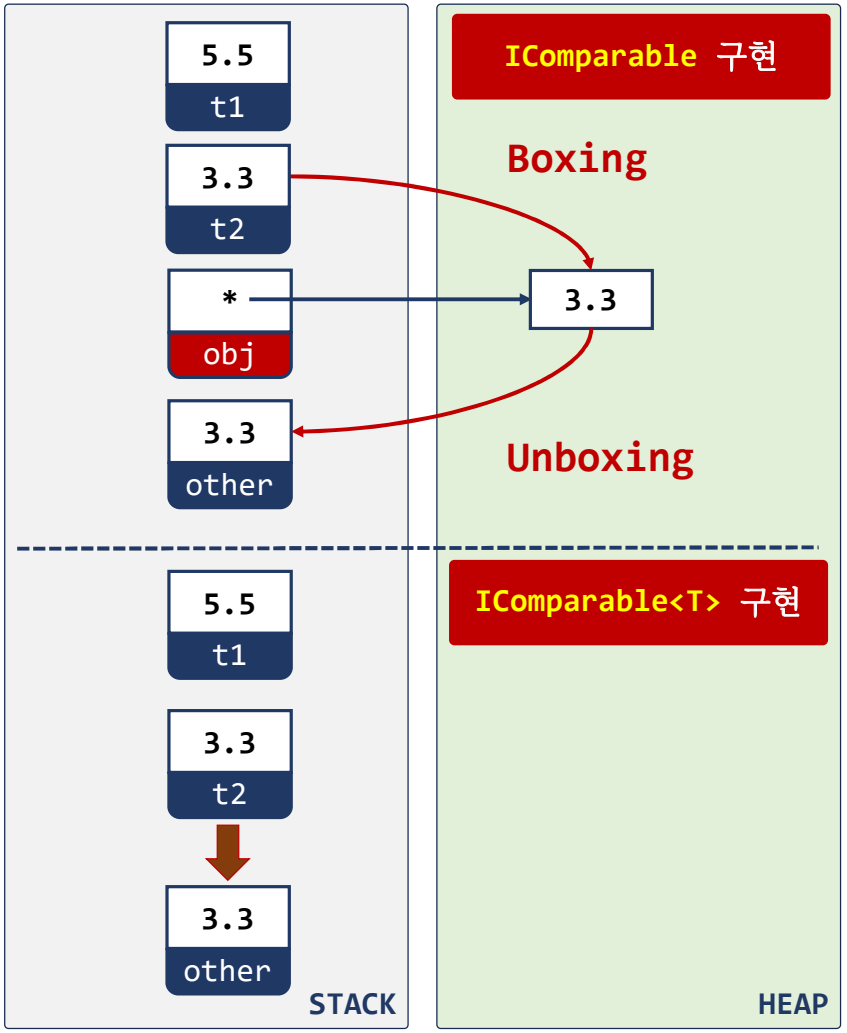
- C# 2.0 에서 generic 인터페이스 추가.

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

  - ⇨ 메소드의 인자로 "**object 가 아닌 T 를 사용**" 해서
    "**Boxing/Unboxing 현상이 발생되지 않는다.**"

241

핵심 정리



242

핵심 정리

- 코드 작성 관례
  - ⇨ **IComparable, IComparable<T>** 인터페이스를 모두 구현하는 것을 권장

243

핵심 정리

- **Temperature 객체 t1, t2 동일성을 확인하고 싶다.**
  - ⇨ System.Object 로 부터 "**Equals() 메소드를 상속**" 받게 되고,
  - ⇨ Value Type 의 경우 기본 구현은 "**메모리 비교**".
  - ⇨ 따라서, **Equals() 메소드를 사용하면 된다.**

- Temperature 객체 t1, t2 동일성 비교시
  - ⇨ 소수점 이하는 버림을 하고 비교 하고 싶다.
  - ⇨ "**Equals() 메소드를 override 해서 다시 구현을 제공**" 하면 된다.

244

핵심 정리

- **System.Object 의 Equals() 메소드**

  ⇨ 메소드 인자로 **object** 사용

```
public virtual bool Equals(object? obj)
{
    return this == obj;
}
```

  ⇨ struct 에서 override 하는 경우

  "Boxing/Unboxing" 현상 발생

- **IEquatable<T>** 인터페이스

  ⇨ object 가 아닌 **T** 를 인자로 사용

```
public interface IEquatable<T>
{
    bool Equals(T other);
}
```

245

핵심 정리

```
struct Temperature
{
   // Equals() 과 CompareTo() 를 제공하고 싶다.
}
```

- **CompareTo() 메소드**

  ① **"IComparable"** 인터페이스 구현

  ② **"IComparable<T>"** 인터페이스도 구현

  ("Boxing/Unboxing" 현상을 제거하기 위해)

- **Equals() 메소드**

  ① System.Object 의 가상 메소드 이므로

  "Equals() 메소드를 override" 해서 제공.

  ② **"IEquatable<T>"** 인터페이스도 구현

  ("Boxing/Unboxing" 현상을 제거하기 위해)

246

generic

247

핵심 정리

int 타입 뿐 아니라 double 등의
다른 타입에 대해서도 swap 을 사용하고 싶다.

- **방법 #1. method overloading 사용**
  - ⇨ 사용자 입장에서는 하나의 메소드 처럼 사용해서 편리하지만
  - ⇨ method 제작자 입장에서는 인자의 타입만 다를 뿐 "구현이 동일한 코드의 메소드를 여러 개 작성" 하는 불편함이 있다.

- **방법 #2. generic method 사용**
  - ⇨ method 가 아닌 method 를 만드는 틀.
  - ⇨ C++ 언어의 template.

248

핵심 정리

- generic method 기본 모양

```
public static void swap<T>(ref T a, ref T b)
{
   T tmp = a;
   a = b;
   b = tmp;
}
```

type parameter

- type parameter
  - ⇨ "T" 뿐 아니라 어떠한 이름도 사용가능.
  - ⇨ 관례적으로 "T" 를 많이 사용

249

핵심 정리

- generic method 를 사용하는 2가지 방법

- 방법 #1. type parameter 를 명시적으로 전달
  - ⇨ "swap<int>(ref n1, ref n2)"

- 방법 #2. type parameter 를 생략
  - ⇨ "swap(ref n1, ref n2)"
  - ⇨ 메소드 인자를 보고 컴파일러가 type 을 추론(inference)

```
swap<int>(ref n1, ref n2);
swap     (ref d1, ref d2);
```

250

핵심 정리

- **generic method 원리**
  - ⇨ generic method 자체는 메소드를 만드는 틀.
  - ⇨ type parameter 의 type 이 결정되면
  - ⇨ 실행시간에 해당 타입의 메소드를 생성.

```
void swap(ref int a, ref int b)
{
    int tmp = a;
    a = b;
    b = tmp;
}
```

```
void swap<T>(ref T a, ref T b)
{
    T tmp = a;
    a = b;
    b = tmp;
}
```

swap<int>(....)

swap<double>(....)

```
void swap(ref double a, ref double b)
{
    double tmp = a;
    a = b;
    b = tmp;
}
```


핵심 정리

- **type parameter**

generic method	**type parameter 를 생략 가능.** 생략시 메소드 인자를 통해서 타입 추론 swap(ref n1, ref n2)
generic class	type parameter 를 생략할 수 없다.

- **private T x = 0;**
  - ⇨ error.
  - ⇨ T 타입이 0 으로 초기화 될 수 있다는 보장이 없다.
- **private T x = default 또는 default(T) 사용**
  - ⇨ T 타입의 디폴트 값으로 초기화

Generic Constraint

253

핵심 정리

- **generic method 의 인자**
  - ⇨ System.Object 가 가진 메소드만 사용할 수 있다.
  - ⇨ System.Object 의 메소드 외에 **"다른 메소드(CompareTo) 를 호출"** 하려면 어떻게 해야 할까 ?

- **방법 #1. 캐스팅 사용**
- **방법 #2. generic constraint 문법 사용**

254

핵심 정리

- **방법 #1. 캐스팅 사용**
  - ⇨ CompareTo 메소드는 "**IComparable<T>**" 인터페이스가 제공
  - ⇨ generic method 의 인자를 "**IComparable<T> 인터페이스 로 캐스팅 후에 사용.**"

- **캐스팅 방식의 단점**
  - ⇨ IComparable<T> 인터페이스를 구현하지 않은 타입의 객체를 전달하는 경우,
  - ⇨ 컴파일 에러는 없지만 "**캐스팅 실패로 null 객체 사용**", "**runtime 오류(예외)가 발생**".
  - ⇨ runtime error 가 아닌 "**compile time error**" 가 발생하게 할 수 없을까 ?"

255

핵심 정리

- **generic constraint(제약)**
  - ⇨ generic 메소드(클래스)에 전달되는 타입이 지켜야 하는 제약조건(constraint) 을 표기 하는 문법

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

T 타입이 지켜야 하는 조건을 문법에 맞게 표기

- **장점 #1. 제약 조건을 만족하지 않은 타입은 generic 메소드(클래스)를 사용할 수 없다.**
  - ⇨ runtime error 가 아닌 **compile time error**

- **장점 #2. generic 메소드(클래스) 안에서 제약조건이 가진 기능(메소드 호출등)을 사용할 수 있다.**

256

핵심 정리

- generic method 뿐 아니라 "generic type" 에도 표기 가능.

- 다양한 형태의 constraints

```
class C<T> where T : struct           {}
class C<T> where T : class            {}
class C<T> where T : class?           {}
class C<T> where T : notnull          {}
class C<T> where T : unmanaged        {}
class C<T> where T : new()            {}
class C<T> where T : base_class_name  {}
class C<T> where T : base_class_name? {}
class C<T> where T : interface_name   {}
class C<T> where T : interface_name?  {}
```

257

핵심 정리

- multiple constraint

  ⇨ 한개의 타입에 여러 개 제약조건을 표기 가능

```
class C<T> where T : C1, C2, C3
```

- multiple generic parameter

```
class C2<T, U> where T : struct
               where U : class
{
}
```

258

delegate

259

핵심 정리

- **delegate 란 ?**
  - ⇨ 메소드에 대한 정보를 담는 타입
  - ⇨ **references to methods**
  - ⇨ **C/C++** 언어의 함수 포인터와 유사한 개념

260

핵심 정리

- **int, double, string 등의 타입은**
  - ⇨ C# 언어가 제공하는 기본 타입(built in types)
  - ⇨ **사용자가 만들 필요가 없다.**

- **delegate 타입**
  - ⇨ **"사용자가 직접 만들어서 사용하거나"**
  - ⇨ **"라이브러리 형태로 만들어져 있는 delegate(Func, Action)"** 를 사용
  - ⇨ 일반적인 타입을 만들 때는 class, struct 문법을 사용하지만
  - ⇨ delegate 는 **"delegate 를 만드는 별도의 문법 제공"**.

261

핵심 정리

- **delegate 타입을 만드는 방법.**
  - ① 저장할 메소드와 **"동일한 모양의 선언을 만들고"** (클래스 안 또는 밖 모두 가능)
  - ② 선언 앞에 **"delegate"** 를 표기
  - ③ 메소드 이름 자리에 **"원하는 타입이름"** 을 표기

MyFunc

```
delegate  void Foo(int arg);
```

- **MyFunc**
  - ⇨ **타입(Type)**
  - ⇨ **함수의 호출 정보를 보관 할 때 사용**
  - ⇨ MyFunc 타입의 객체는 **"() 연산자 또는 invoke() 메소드"**를 사용해서 저장된 메소드를 호출할 수 있다.

262

핵심 정리

- **delegate 란 ?**
  - ⇨ 메소드의 정보(주소, 객체 등)를 담는 타입
  - ⇨ 메소드에 대한 참조(reference to method)

  ⇨ A delegate is a **type that represents references to methods** with a particular parameter list and return type

  ⇨ **C/C++ 의 함수 포인터와 유사한 개념**

  ⇨ 메소드의 호출정보를 저장했다가 **"() 연산자 또는 invoke() 메소드"**를 사용해서 저장된 메소드를 호출할 수 있다.

[핵심]
- delegate 개념.
- delegate 를 만드는 방법.

핵심 정리

- **delegate 의 원리**

```
delegate void MyFunc (int arg);
```

위 코드를 참고해서 컴파일러가 아래와 같은 클래스를 생성.
delegate 이름이 클래스이름.

```
class MyFunc : System.MulticastDelegate
{
    // Invoke(), BeginInvoke(), EndInvoke()...
}
```

delegate 관련 기능을 제공하는 기반 클래스

클래스 이름

ILDasm.exe 같은 도구로 확인 가능

```
MyFunc m1 = new MyFunc(Foo);
MyFunc m2 = Foo;
```

위 코드와 유사한 단축 표기법

핵심 정리

- delegate 문법을 사용하지 말고 아래 처럼 사용자가 직접 클래스를 만들 수도 있지 않을까 ?

```
class MyFunc : System.MulticastDelegate
{
    // Invoke(), BeginInvoke(), EndInvoke()...
}
```

⇨ **error.**

⇨ 사용자가 직접 System.MulticastDelegate 에서 파생된 클래스를 만드는 것은 허용되지 않는다.

⇨ **반드시 delegate 문법을 사용해서 만들어야 한다.**

- **핵심**
  - ⇨ **delegate 개념**
  - ⇨ **delegate 만드는 방법**
  - ⇨ **delegate 의 내부 원리**

265

delegate example

266

핵심 정리

- 배열에서 1번째로 나오는 “3” 을 찾고 싶다
  - ⇨ Array 클래스 사용

- **Array 클래스**
  - ⇨ 배열에 관련된 다양한 연산을 수행하는 static method 제공
  - ⇨ 값을 검색하려면 “**Array.IndexOf**” static method 사용.

```
int ret1 = Array.IndexOf(x, 3);
```

267

핵심 정리

- 배열에서 1번째로 나오는 “**3의 배수**” 을 찾고 싶다
  - ⇨ **Array.FindIndex(x, 메소드 이름);**

- **Array.FindIndex(배열이름, 메소드 이름)**
  - ⇨ 배열의 모든 요소를 “**하나씩 차례대로**”
  - ⇨ “**메소드에 전달**” 해서
  - ⇨ “**메소드가 True 를 반환하는 요소**”의 index 를 반환

- Array.FindIndex 의 2번째 인자로 전달되는 메소드는
  - ⇨ “**반환 타입은 bool**” 이어야 하고
  - ⇨ 배열의 요소와 동일한 타입의 “**인자를 한 개**” 가져야 한다.

```
bool Foo(int value)
```

268

핵심 정리

- **FindIndex() 메소드**
  - ⇨ int 배열 뿐 아니라 모든 타입의 배열에 대해서 검색이 가능해야 한다.
  - ⇨ **"Generic"** 사용

- **"Predicate(조건자)"** 용어
  - ⇨ **"bool 을 반환 하는 메소드"** 로
  - ⇨ FindIndex 등에 전달하는 **"검색 조건을 담은 메소드"** 를 가리키는 용어

269

핵심 정리

- **Array.FindIndex()**

```
int FindIndex<T>(T[] array, Predicate<T> match);
```

C# 언어에서 미리 정의한 delegate 타입

```
delegate bool Predicate<in T>(T obj);
```

"covariant, contravariant" 강의 참고

270

핵심 정리

- 배열의 모든 요소를 정렬하고 싶다.
  - ⇨ "**Array.Sort()**" 사용

```
void Sort<T>(T[] array)
```

  - ⇨ "**오름 차순**"으로 정렬
  - ⇨ 정렬 기준을 변경하려면 정렬시 "**요소 2개의 크기 비교에 사용할 메소드**" 를 2번째 인자로 전달.

```
void Sort<T>( T[] array,
              Comparison<T> comparison)
```

```
delegate int Comparison<in T>(T x, T y);
```

271

핵심 정리

- **#1. "정책을 담은 메소드를 전달" 하기 위해**
  - ⇨ Sort(), FindIndex() 등의 메소드를 사용할 때
  - ⇨ "**비교 정책, 검색 조건**" 등을 담을 메소드를 전달할 때 사용.
  - ⇨ "**Predicate<T>, Comparison<T>**" 등 미리 만들어진 다양한 delegate 타입이 제공된다.
- **#2. 객체에서 발생하는 이벤트 등을 처리하는 함수를 등록하기 위해서 사용**
  - ⇨ "**event**" 강의 참고

272

delegate & method

273

핵심 정리

- **static method** vs **instance method**

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 저장하려면

static method	**MyFunc f = 타입이름.static_method;**
instance Method	타입 obj = new 타입(); **MyFunc f = obj.instance_method;**

274

핵심 정리

- **Signature 가 동일한 메소드만 등록 할 수 있다.**

- 하나의 delegate 에 여러 개의 method 를 등록할 수 있다
  - ⇨ **+=, -= 연산자 사용**
  - ⇨ **자세한 내용은 "delegate chain" 강의 참고**

- delegate 를 사용해서 메소드를 호출하려면
  - ⇨ **() 연산자 또는 Invoke() 메소드 사용**
  - ⇨ **? 연산자와 같이 사용할 때는 Invoke() 사용**

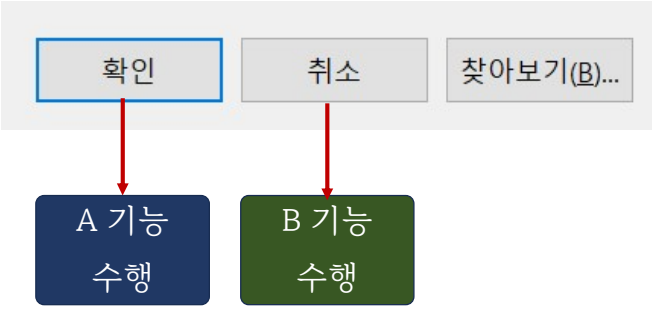
275

event

276

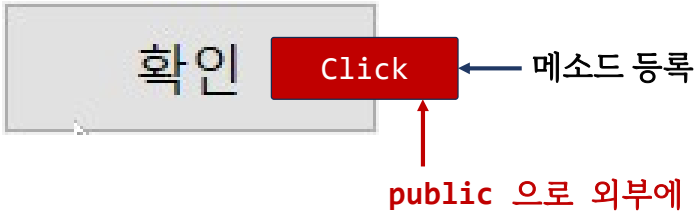
핵심 정리

- GUI 프로그램을 작성하는데, **"여러 개의 버튼을 생성"** 했다.
  - ⇨ 버튼을 누르면 **"특정 기능을 수행"** 해야 한다.
  - ⇨ 각각의 버튼을 **"다른 기능을 수행할 수 있어야"** 한다.



277

핵심 정리



- **"Click delegate 를 public 으로 외부에 노출"**하면
  - ⇨ 사용자의 잘못된 사용으로 버그가 발생할 수 있다.
  - ⇨ 실수로 **"+= 이 아닌 = 로 잘못 사용하면 이전에 등록된 메소드가 제거"** 된다.
  - ⇨ **"Click.Invoke()를 버튼 내부가 아닌 외부에서도 사용"**할 수 있고,

- **안전하게 사용하려면**
  - ⇨ Click delegate 를 Button 의 private 으로 만들고
  - ⇨ **"= 이 아닌 +=, -= 으로만 메소드를 등록"** 할 수 있도록 하면 된다.

278

핵심 정리

- **event 키워드의 원리**

```
class Button
{
   ......

   public event ClickHandler? Click = null;

   ......
}
```

delegate 타입의 객체를 클래스 필드로 추가 할 때

"**event 키워드를 적으면**" C# 컴파일러가

"**Button 클래스에 약속된 코드를 추가**"

279

Action<T>, Func<T>

280

핵심 정리

- **Action<T>** delegate
  - ⇨ "**반환 타입이 void 인 method**" 를 위한 delegate
  - ⇨ C# 표준에서 제공
  - ⇨ 인자가 0~16 개 까지의 Method 를 위한 버전 제공.

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

                    ......
                    ......

delegate void Action<in T1,in T2,in T3,in T4,in
T5,in T6,in T7,in T8,in T9,in T10,in T11,in
T12,in T13,in T14,in T15,in T16>(T1 arg1, T2
arg2, T3 arg3, T4 arg4, T5 arg5, T6 arg6, T7
arg7, T8 arg8, T9 arg9, T10 arg10, T11 arg11,
T12 arg12, T13 arg13, T14 arg14, T15 arg15, T16
arg16);
```

281

핵심 정리

- **Func<T>** delegate
  - ⇨ "**반환 타입이 void 가 아닌 method**" 를 위한 delegate
  - ⇨ 인자가 0~15 개 까지의 Method 를 위한 버전 제공.

```
delegate TResult Func<out TResult>();
delegate TResult Func<in T,out TResult>(T arg);
delegate TResult Func<in T1,in T2,
                out TResult>(T1 arg1, T2 arg2);

                    ......
                    ......

delegate TResult Func<in T1,in T2,in T3,in
T4,in T5,in T6,in T7,in T8,in T9,in T10,in
T11,in T12,in T13,in T14,in T15,out TResult>(T1
arg1, T2 arg2, T3 arg3, T4 arg4, T5 arg5, T6
arg6, T7 arg7, T8 arg8, T9 arg9, T10 arg10, T11
arg11, T12 arg12, T13 arg13, T14 arg14, T15
arg15);
```

282

핵심 정리

- 표준에서 제공하는 delegate

Action	반환 타입이 "**void**" 인 메소드를 위한 delegate (인자 최대 16개)
Func	반환 타입이 "**void 가 아닌**" 메소드를 위한 delegate (인자 최대 15개)

- **Action, Func 를 사용하면**
  ⇨ 인자 갯수 만 초과하지 않는 다면 모든 메소드를 위한 delegate 로 사용할 수 있다.

283

핵심 정리

- Action, Func 외에도 다양한 용도로 사용하기 위한 delegate 가 이미 정의 되어 있다.

- **Array.FindIndex()**의 인자로 사용하기 위한 delegate

```
int FindIndex<T>(T[] array, Predicate<T> match);
```

```
delegate bool Predicate<in T>(T obj);
```

Func<T, bool> 로도 사용 가능하지만
Predicate 라는 이름을 사용하는 것이
가독성이 좋다.

- **Array.Sort()의** 인자로 사용하기 위한 delegate

```
void Sort<T>( T[] array,
              Comparison<T> comparison)
```

```
delegate int Comparison<in T>(T x, T y);
```

284

Variant Generic

285

핵심 정리

- 상속을 사용하면
  - ⇨ Derived 클래스는 Base 클래스의 모든 기능을 물려받게 된다.
  - ⇨ “**Base 의 객체가 필요**” 할 때, “**Derived 객체를 대신 사용**” 할 수 있다.

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

286

핵심 정리

- `delegate R MyFunc<R>();`
  - ⇨ 인자가 없고 반환 타입이 R 인 메소드를 담을 수 있는 delegate

MyFunc<Derived> f1	Derived 객체를 반환하는 메소드를 저장할 수 있다. 즉, **f1을 사용하면 Derived 객체를 얻을 수 있다.**
MyFunc<Base> f2	**Base** 객체를 반환하는 메소드를 저장할 수 있다. 즉, **f2를 사용하면 Base 객체를 얻을 수 있다.**

```
MyFunc<Derived> f1 = CreateDerived;
MyFunc<Base>    f2 = f1;
```

Base 객체를 얻기 위한 delegate 를
Derived 객체를 얻을 수 있는 delegate 로 초기화.

핵심 정리

- **Covariant(공변성)**
  - ⇨ Generic 사용시 반환 타입 앞에 out 를 표기하는 문법.

```
delegate R MyFunc<out R>();
```

- ⇨ MyFunc<Derived> 타입의 delegate 를
- ⇨ MyFunc<Base> 타입의 delegate 에 저장할 수 있다.

```
MyFunc<Derived> f1 = CreateDerived;
MyFunc<Base>    f2 = f1;
```

핵심 정리

- `delegate void MyAction<T>(T obj);`

MyAction<Base> f1	f1 을 사용할 때는 반드시 인자로 Base 객체를 전달해야 한다. **f1("Base object")**
MyAction<Derived> f2	f2 를 사용할 때는 반드시 인자로 Derived 객체를 전달해야 한다. **f2("Derived object")**

인자로 "**Base 타입의 객체**" 를 요구

```
public static void UseBase(Base b) {...}

MyAction<Base>    f1 = UseBase;

MyAction<Derived> f2 = f1;

f2(new Derived());
```

f2를 사용할 때는 "**Derived 객체를 전달**".

289

핵심 정리

- **Contravariant(반변성)**
  - ⇨ Generic 사용시 인자타입 앞에 in 를 표기하는 문법.

```
delegate void MyAction<in T>(T obj);
```

- ⇨ MyAction<Base> 타입의 delegate 를
- ⇨ MyAction<Derived> 타입의 delegate 에 저장할 수 있다.

```
MyAction<Base>    f1 = UseBase;
MyAction<Derived> f2 = f1;
```

290

핵심 정리

- **C# 표준의 Action, Func**

```
delegate void Action();
delegate void Action<in T>(T obj);

delegate TResult Func<out TResult>();
delegate TResult Func<in T, out TResult>(T arg);
```

291

핵심 정리

- **Variant 개념**

covariant	반환 타입 앞에 **out** 표기
contravariant	인자 타입 앞에 **in** 표기

⇨ **"Generic Interface"** 와 **"Generic Delegate"** 를 만들 때 사용하는 개념

292

lambda expression

293

핵심 정리

- **배열에서 1번째 나오는 짝수를 찾고 싶다.**
  - ⇨ Array.FindIndex() static 메소드 사용

```
int FindIndex<T>(T[] array, Predicate<T> match);
```

```
delegate bool Predicate<in T>(T obj);
```

- **lambda expression**
  - ⇨ **"익명의 함수(anonymous function)"** 을 만드는 문법.
  - ⇨ C# 뿐 아니라 대부분의 언어에서 제공되는 기능.

- **lambda expression**
  - ⇨ 인자로 전달되는 **"메소드(함수)를 간결하게 만들 수 있고"**
  - ⇨ **"outer variable 을 캡쳐"** 할 수 있는 기능이 있다.

294

핵심 정리

- **짝수가 아닌 사용자가 입력한 값의 배수를 찾고 싶다.**
  - ⇨ 사용자가 입력한 값은 Main 함수의 지역변수 k 에 저장되어 있다.
  - ⇨ 람다 표현식이 아닌 IsMutipleOf 메소드 에서는 **"Main 함수에 있는 지역변수 k 를 사용할 수 없다."**
  - ⇨ Main 함수 안에서 만든 **"lambda expression"** 에서는 자신의 **"outer(Main) variable 를 사용"** 할 수 있다.

295

핵심 정리

- **lambda expression 활용**
  - ① Array.FindIndex 등의 다양한 **"메소드에 인자"**로 전달.
  - ② Func, Action delegate 에 담아서 **"함수(메소드) 처럼 사용"**.(이름이 없는 함수에 이름을 부여하는 의미)

296

핵심 정리

- lambda expression 의 다양한 형태

```
int Square(int n)=>{ return n * n; }
```

```
int Square(int n)=> n * n;
```

expression lambda : {} 가 없는 것

```
(input-parameters) => expression
```

```
(input-parameters) => { <sequence-of-statements> }
```

statement lambda : {} 가 있는 것

297

핵심 정리

- 타입 추론이 가능한 경우 인자 타입은 생략 가능.
- Func, Action 대신 var 도 사용가능.
- 인자가 한 개인 경우 () 도 생략 가능
- 반환 타입
  - ⇨ 표기하지 않아도 컴파일러가 추론.
  - ⇨ 컴파일러가 "추론할 수 없는 경우는 사용자가 직접 표기" 가능(C# 10.0 부터 표기 가능.)
  - ⇨ 반환 타입 표기할 때는 "인자의 () 가 반드시 필요"

298