**PPT.3**

======[C#의 시작 첫걸음]========================

using System;

->

using : 네임스페이스를 사용하겠다고 선언하는 키워드

System : 숫자, 텍스트와 같은 데이터를 다룰 수 있는 기본적인 데이터 처리 클래스를 비롯하여

C#코드가 기본적으로 필요로하는 클래스를 담고 있는 네임스페이스

세미콜론(;) :컴파일러에세 문장의 끝을 알리는 기호

namespace는 성격이나 하는 일이 비슷한 클래스나 구초체형식 등을 하나의 이름아래 묶는 역할을 수행

클래스는 C# 프로그램을 구성하는 기본단위이며 "데이터 +메소드"로 이루어짐

static void Main( string[] args ) { } 프로그램의 시작점 (ex. start)

static void Main ( string[] args ) { }

한정자 반환형식 메소드이름 매개 변수 코드블록 쓰는곳

-----------------------------------------------------------------------------------------

C#데이터는 기존의 CPP데이터 형식과 동일하다

데이터 형식2개

기본 데이터 형식 : int, char......

복합 데이터 형식 : string, Socket, Image.....

형식

값 형식 : 변수의 데이터의 값을 담는 형식

참조 형식 : 변수에 데이터의 위치를 담는 형식 [배열, CLASS(object)]

변수 : 데이터를 담는 일정 크기의 공간(여기에서 "일정 크기"는 데이터형식에 따라 결정)

EX.) int( <-데이터 형식)\_\_\_\_\_ <-변수의 이름

데이터할당

\_\_\_<-변수의 이름 = <--대입연산자 100<-데이터 ;====> X=100;

======값 형식=====

int X// 변수선언

X = 100; //데이터 할당

======> 한번에 int X = 100; 변수 선언과 데이터할당(초기화)을 한번에 해주었다.

=== 참조 형식=====

[object, Class 중 한개] = 위치(변수의 주소)

EX.) odject a = 10 //object a는 10이라는 위치(힙)에 있는 값이다.

===데이터의 형식=====

int 정수형

float 실수형

char 문자데이터 ('가', 'a')

string 문자열 ("안녕하세요", "hello")

bool 논리형 참(TURE, 0이 아닌값)과 거칫(FALSE, 0인값)을 판단함

======\*\*\*\*object가 참조형식이면서 값형식도 가능한 이유\*\*\*\*\*\*====

Boxing 과 Unboxing이 가능해서 object는 참조형식이면서 값형식도 가능하다.

Boxing : 값형식의 데이터를 힙(변수의 주소)에 넣고 글 힙의 위치를 object형식 변수가 가리키도록해서

Unboxing : 힙 안에 있는 값형식의 데이터를 상자를 풀어서 꺼내는 것

==============데이터의 형변환===================

[숫자 형변환 정수 -> 실수]

int a = 500;

Console.WriteLine(a);

uint b = (uint)a; //int a를 uint a로 형변환

Console.WriteLine(b);

출력되는 값은 500 , 500 , -30 , 4294967266

int x = -30;

Console.WriteLine(x);

uint y = (uint)x;

Console.WriteLine(y);

참고 \*\*\*\*[실수형을 정수로 형변환을 하면 소숫점아래를 버린다.]\*\*\*\*\*\*\*\*\*\*

[문자형와 숫자의 형변환]

숫자 -> 문자열 : ToString() 메소드 이용

예) int a = 3;

string b = a.ToString();

문자열 -> 숫자 : Parse() 메소드 이용

예) int a = int.Parse(“12345”);

================상수(Constant)=======================

상수

변수와는 달리 그안에 담긴 데이터를 절대 바꿀 수 없는 메모리 공간

예) public const int a = 4; 라고 선언하면 절대 바꿀 수 없다.

사용하는 이유는 개발자는 완벽한 존재가 아니여서 바꿀 수 없는 값으로 기정해 놓은것

const의 형식을 지정안하면 자동으로 static이 된다.

상수는 기본형과 열거형이 있다.

열거형은 enum 키워드를 사용

예) enum DialogResult { YES, NO, CANCEL, CONFIRM, OK }

=====================Nullable=============================

null(비어있는) 값이 필요할 때 사용 ==>정해지지 않은값

데이터형식 뒤에 "?"만 붙여주면 된다.

예) int? a = null;

float? b = null;

double? c = null;

===================VAR(자동형식지정)==========

int 와 string은 많이 쓰므로 명시적 형식말고 vard을 사용하면 자동으로 해당 변수의 병식을 지정

예) var a = 3; // a는 int 형식

var b = "Hello"; // b는 string 형식

**PPT.4**

===================산술연산자================

+, -, \*, /(나누기), %(나누기 나머지까지)

===================증가 연산자와 감소 연산자================

++ : 증가 연산자

-- : 감소 연산자

===================문자열 결합 연산자================

int result = 123+ 456;

result 는 579 덧셈

string result = "123"+"456";

result 는 "123456"

===================관계 연산자================

비교연산자

>,<,<= 등...

!= -->거짓연산자 X(fals) 참 , O(true) 거짓 [거짓말쟁이 비교연산자]

=================== 논리 연산자================

&& 둘다 참이면 참

|| 둘중 하나가 참이면 참

! ----> 반대 참이면 거짓, 거짓이면 참

=================조건연산 자=========================

조건연산자는 조건에 따라 두 값 중 하나의 값을 반환

[사용형식]

조건식 ? 참일\_때의\_값 : 거짓일\_때의\_값

[사용 예]

int a = 30;

string result = a == 30 ? “삼십” : “삼십아님” ; // result는 “삼십”

=================== 비트 연산자 ================

ppt4 13번 슬라이드

**PPT.5**

=================== 분 기 문 ================

if, else, else if

if {

}

else if{

} //세 갈래 이상일때 사용

else{

}

---switch----

switch (조건식)

{

case 상수1:

// 실행할 코드

break;

case 상수2:

// 실행할 코드

break;

default:

break;

}

=================== 반 복 문 ================

C#은 다음 네 가지의 반복문을 제공 [ppt5 10번 슬라이드]

[[[[while]]]]

while ( 조건식 )

반복실행할\_코드

[[[[do while]]]]]

do

{

반복실행할\_코드\_블록

}

while( 조건식 ) ;

[[[[for]]]]

for( 초기화식; 조건식; 반복식; )

반복실행할\_코드;

[[[[[foreach (배열)]]]]

int[] arr = new int[] {0, 1, 2, 3, 4}; // 배열 선언

foreach (int a in arr)

{

Console.WriteLine(a);

}

=================== 점 프 문 ================

[[[Break]]]

int i = 0; // i를 초기화하는 코드가 실행되고

while ( i >= 0) // 반복이 실행되다가

{

if ( i == 10 )

break; // i가 10이 되면 while 문에서 탈출

Console.WriteLine( i++ );

}

// 프로그램의 실행 위치는 while 블록 다음으로 이동

Console.WriteLine("Prison Break");

[[[continue]]]

for ( int i=0; i<5; i++ )

{

if ( i == 3 )

continue;

Console.WriteLine( i );

}

[[[continue]]]

goto 레이블;

레이블 :

// 이어지는 코드

-------- 사용 예제-------

{

Console.WriteLine( " 1 " );

goto JUMP;

Console.WriteLine( " 2 " );

Console.WriteLine( " 3 " );

JUMP:

Console.WriteLine( " 4" );

}

----- 출력 --> 1 4

**PPT.6**

===============메소드================

메소드란 일련의 코드를 하나의 이름 아래 묶는 것 ( Class )

메소드 선언

class 클래스의\_이름

{

한정자 반환\_형식 메소드의\_이름( 매개\_변수\_목록 )

{

// 실행하고자 하는 코드 1

// …

// 실행하고자 하는 코드 n

return 메소드의\_결과;

}

}

메소드 호출

int 이름 = 클래스이름.메소드이름(값1,값2);

예시}--------------------------------------------------

메소드 선언

class Calculator

{

public static int Plus( int a, int b )

{

Console.WriteLine(“Input : {0}, {1}”, a, b);

int result = a + b;

return result;

}

}

메소드 호출

int x = Calculator.Plus( 3, 4 ); //x는 7

int y = Calculator.Plus( 5, 6 ); //y는 11

=============리턴(return)===============

점프문의 한 종류

return; 그냥 호출자에게 돌림

return (인자or 식); 인자를 호출자에게 돌림(종료) or 식을 계산해서 호출자에게 돌림(종료)

============= 매개 변수 ========================

메소드를 호출할 때 넘기는 매개 변수는 그대로 메소드안으로 넘겨지지 않는다.

이름만 같은 서로다른 매개변수이다.

ppt 6 9번 슬라이드

-------Swap()---------------

두 매개변수의 값을 교환하는 Swap()문

\*\*\*\*\*\*사용 방법 [메소드 선언과 호출 시 ref 키워드를 이용]\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

static void Swap( ref int a, ref int b)

{

int temp = b;

b = a;

a = temp;

}

static void Main(string[] args)

{

int x = 3;

int y = 4;

Swap( ref x, ref y);

}

x = 3 //// Y = 4 ====> x = 4 //// Y = 3

-----------출력 전용 매개 변수( out ) ------------

void Divide( int a, int b, out int quotient, out int remainder )

{

quotient = a / b;

remainder = a % b;

}

static void Main(string[] args)

{

int a = 20;

int b = 3;

int c;

int d;

Divide( a, b, out c, out d );

Console.WriteLine("Quotient : {0}, Remainder {1}", c, d );

}

// ref : 원본 변경이 선택 부분

// out : 원본 변경이 강제적 부분

================메소드 오버로딩=================

메소드 오버로딩 : 하나의 메소드 이름에 여러 개의 구현을 올림

int Plus(int a, int b)

{

return a + b;

}

double Plus(double a, double b)

{

return a + b;

}

int result1 = Plus( 1, 2 ); ------> int Plus 메소드 사용

double result2 = Plus( 3.1, 2.4 );------>double Plus 메소드 사용

================가변 길이 매개 변수 ==================

가변길이 매개변수는 params 키워드와 배열을 이용하여 선언

int Sum( params int[] args )

{

int sum = 0;

for(int i=0; i<args.Length; i++)

{

sum += args[i];

}

return sum;

}

========== 매소드 오버로딩 + 가변 길이 매개변수 ===============

(메소드 오버로딩)

메소드 오버로딩은 매개변수의 개수가 유한하게 정해져 있을때

(매개변수의 개수가 2개면 메소드 매개변수 개수도 2개)

매개변수의 각 형식이 다를 때 사용

(가변길이 매개 변수)

가변길이 매개 변수 형식은 같으나 매개 변수의 개수만 유연하게 달라질 수 있는 경우에 사용

======================명명된 매개 변수========================

명명된 매개 변수(Named Parameter)는 메소드를 호출할 때 매개 변수의 이름을 명시함으로써

순서에 관계없이 매개변수에 할당할 데이터를 바인드하는 기능

예))

static void PrintProfile( string name, string phone)

{

Console.WriteLine("Name:{0}, Phone:{1}", name, phone====~~

~===> index 0에 name 값 , index 1에 phone값 );

------> 뒤에 순서를 바꿀수 있다.

static void Main(string[] args)

{

PrintProfile( name : "박찬호", phone : "010-123-1234");

}

======================선택적 매개 변수========================

선택적 매개 변수는 메소드 선언시 매개 변수에 기본 값을 할당함으로 써, 해당 매개변수에 명시적으로

값을 할당할지/않을지를 선택가능하게 하는 기능

예)) 매개변수 a와 b의 값을 0으로 기본 할당했다.

void MyMethod( int a = 0, int b = 0 ) ----> b는 선택적 매개변수

{

Console.WriteLine( “{0}, {1}”, a, b );

}

MyMethod(3); // 매개변수 b 생략

MyMethod(3, 4);

**PPT.7**

==========기본지식=====

메소드는 일련의 코드를 하나로 묶는것 (Class)

클레스의 인스턴스(객체생성)는 객체생성이다.

========= 생성자 ========

class 클래스이름

{

한정자(public) 클래스이름 ( 매개변수목록 )

{

//

}

// 필드

// 메소드(예 - public 클래스이름(초기화에 필요한 파라미터 [string 이름])

}

========= 소멸자 =========

class 클래스이름

{

~클래스이름()

{

//

}

// 필드

// 메소드

}

((but)) 프로그래머가 명시적으로 소멸자를 선언하지 않아도 컴파일러가 암시적으로

기본소멸자를 제공해줌

========= 객체 복사하기: 얕은 복사와 깊은 복사 =========

<<<<얕은 복사>>>>

class MyClass

{

public int MyField1;

public int MyField2;

}

MyClass source = new MyClass();

source.MyField1 = 10;

source.MyField2 = 20;

MyClass target = source;

target.MyField2 = 30;

Console.WriteLine( "{0} {1}", source.MyField1, source.MyField2 );

Console.WriteLine( "{0} {1}", target.MyField1, target.MyField2 );

//아래의 코드에서 target은 source의 전체 필드 값을 복사하는 것이

아닌, source가 참조하고 있는 힙의 주소만 복사해왔기 때문임

|||

|||

▽

<<<<깊은 복사>>>>

class MyClass

{

public int MyField1;

public int MyField2;

public MyClass DeepCopy()

{

MyClass newCopy = new MyClass();

newCopy.MyField1 = this.MyField1;

newCopy.MyField2 = this.MyField2;

return newCopy;

}

}

// …

MyClass source = new MyClass();

source.MyField1 = 10;

source.MyField2 = 20;

MyClass target = source;.DeepCopy();

//깊은 복사를 위해서는 명시적으로 필드를 복사하는 코드가 필요

========================THIS키워드 ====================

\* 객체가 스스로를 가리키는 키워드

\* 객체 외부에서 객체의 필드나 메소드에 접근하기 위해

객체의 이름(변수 또는 식별자)를 사용하듯, 객체 내부에서는

자신의 필드나 메소드에 접근할 때 this 키워드 사용

class Employee

{

private string Name1;

public void SetName( string Name2 )

{

this.Name2 = Name1;

}

}

This() 생성자

자기 자신의 생성자를 나타냄

this() 생성자는 다른 코드에서는 사용불가하며, 오로지 생성자에서만 사용 가능

class MyClass

{

int a, b, c;

public MyClass1()

{

this.a = 5425;

}

public MyClass2(int b) : this() <----- MyClass1

{

this.b = b;

}

public MyClass(int b, int c) : this( b ) <----- MyClass2

{

this.c = c;

}

}

===================은닉성(Encapsulation)====================

최소한의 기능만을 노출하고 내부는 모두 감추는 것

상속성(Inheritance)과 다형성(Polymorphism)과 함께 OOP의 3대 특성

대체로 필드는 모두 감추고 메소드는 꼭 노출이 필요한 것만 공개

ppt07 16슬라이드 참고

class MyClass

{

private int MyField\_1; <------데이터 공개 (내외부에서 접근가능)

protected int MyField\_2; <------데이터 은닉 (내부에서 접근가능, 외부에서 접근불가능)

}

====================== 상속 ====================

한 클래스(자식 클래스)가 다른 클래스(부모 클래스)로부터 필드, 메소드, 프로퍼티 등을 물려 받는 것.

자식 클래스는 파생 클래스, 부모 클래스는 기반 클래스라고도 함

[[[[기본형]]]]

class 기반 클래스

{

// 멤버 선언

}

class 파생 클래스 : 기반 클래스

{

// 아무 멤버를 선언하지 않아도   
 // 기반 클래스의 모든 것을 물려받아 갖게 됩니다.

// 단, private으로 선언된 멤버는 제외입니다.

}

[[[[예제]]]]

class Base

{

public void BaseMethod()

{

Console.WriteLine( "BaseMethod" )

};

}

class Derived : Base // 상속받은 자식 클래스(파생 클래스)

{

}

\*\*\*\*\*\*\*\*\*\* base 키워드 \*\*\*\*\*\*\*\*

this 키워드가 자기 자신을 가리키듯, base 키워드는 부모(기반) 클래스를 가리킴

[[[[예제]]]]

class Base

{

protected string Name;

public Base(string Name)

{ this.Name = Name;

}

}

class Derived : Base

{

public Derived(string Name) : base(Name)

{

Console.WriteLine("{0}.Derived()", this.Name);

}

}

\*\*\*\*\*\*\*\*\*\* 기반 클래스와 파생 클래스[ IS 와 AS ] 질문 \*\*\*\*\*\*\*\*

\*\*기반 클래스와 파생 클래스 사이에서는 족보를 오르내리는 형식 변환이 가능

\*\*자식 클래스의 객체는 부모 클래스의 객체로 간주할 수 있음

즉, 파생(자식) 클래스의 인스턴스는 기반(부모) 클래스의 인스턴스로써도 사용

[[[부모]]]

class Mammal

{

public void Nurse() { … }

}

class Dog : Mammal

{

public void Bark() { … }

}

class Cat : Mammal

{

public void Meow() { … }

}

[[[자식]]]

Mammal mammal = new Mammal();

mammal.Nurse();

mammal = new Dog();

mammal.Nurse();

Dog dog = (Dog)mammal;

dog.Nurse();

dog.Bark();

mammal = new Cat();

mammal.Nurse();

Cat cat = (Dog)mammal;

cat.Nurse();

cat.Meow();

<<Is>> 객체가 해당 형식에 해당하는지를 검사하여 그 결과를 bool 값으로 반환합니다.

<<As>> 형식 변환 연산자와 같은 역할을 합니다. 다만 형변환 연산자가 변환에 실패하는 경우

예외를 던지는 반면에 as 연산자는 객체 참조를 null로 만든다는 것이 다릅니다.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 오버라이딩과 다형성 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

<<다형성(Polymorphism)>>

\*OOP에서 다형성은 객체가 여러 형태를 가질 수 있음을 의미

\*자신으로부터 상속받아 만들어진 파생 클래스를 통해 다형성을 실현

\*다형성은 메소드 오버라이딩(Overriding)을 통해서 실현

<<오버라이딩(Overriding)>>

\*부모 클래스에서 선언된 메소드를 자식 클래스에서 재정의 하는 것

\*어떤 메소드가 오버라이딩이 가능하려면 부모 클래스에서 미리 virtual 한정자로 선언되어 있어야 함

\*자식 클래스는 부모 클래스에서 virtual로 선언되어 있는 메소드를 override 한정자를 이용하여 재선언(재정의)

<<<부모>>>

class ArmorSuite

{

public virtual void Initialize()

{

Console.WriteLine("Armored");

}

}

<<<자식>>>

class IronMan : ArmorSuite

{

public override void Initialize()

{

base.Initialize();

Console.WriteLine("Repulsor Rays Armed");

}

}

class WarMachine : ArmorSuite

{

public override void Initialize()

{

base.Initialize();

Console.WriteLine("Double-Barrel Cannons Armed");

Console.WriteLine("Micro-Rocket Launcher Armed");

}

}

==============

----->>>ArmorSuite 클래스는 IronMan, WarMachine등 다양한 모습으로 파생

=======================메소드숨기=======================

\*오버라이딩을 하기 위해서는 기반 클래스가 단단하게 설계되고 구현되어야 함.

-->파생클래스에서 override로 재정의 하려면 기반 클래스의 메소드가 virtual로 선언되어 있어야 함

\*메소드 숨기기는 기반 클래스의 메소드가 virtual로 선언되어 있지 않았을 때 이를 재정의할 수는 없지만 감추고 같은 이름으로 새 메소드를 선언하도록 하는 기능

\*오버라이딩의 경우에는 파생 클래스의 객체를 기반 클래스로 형변환해도 파생클래스 버전의 메소드가 호출되지만,

메소드 숨기기의 경우에는 같은 상황에서 기반 클래스의 메소드가 호출됨

\*new 한정자를 이용(객체를 할당할 때 사용하는 new 연산자가 아님)

class Base(부모)

{

public void MyMethod() <--------부모클래스에서 메소드를 virtual로 선언하지 않음

{

Console.WriteLine("Base.MyMethod()");

}

}

class Derived : Base(자식)

{

public new void MyMethod() <---------------------자식 클래스에서 new 한정자를 이용하여 재선언함으로써 기반(부모) 클래스 버전의 메소드를 숨김

{

Console.WriteLine("Derived.MyMethod()");

}

}

======================= 오버 라이딩봉인하기 ========================

[[[[[sealed 한정자를 이용하여 메소드를 선언하면 파생클래스에서는 해당 메소드를 오버라이딩할 수 없음]]]]]

class Base

{

public virtual void SealMe()

{

// …

}

}

class Derived : Base

{

public sealed void SealMe() //sealed <<-----------sealed로 메소드를 선언하면 이 클래스를 상속하는 클래스(부모[Base])에서는 SealMe() 메소드를 오버라이딩할 수 없음

{

// …

}

}

======================= 중첩 클래스 ========================

중첩 클래스는 클래스 안에 선언된 클래스를 말함

[[중첩 클래스 예제]]

class OuterClass

{

private int OuterMember;

class NestedClass

{

public void DoSomething()

{

OuterClass outer = new OuterClass();

outer.OuterMember = 10;

}

}

}

======================= 분할 클래스 ========================

분할 클래스(Partial Class)란, 여러 번에 나눠서 구현하는 클래스

[[예제]]

MyClass obj = new MyClass();

obj.Method1( );

obj.Method2( );

obj.Method3( );

obj.Method4( );

partial class MyClass

{

public void Method1( ) { }

public void Method2( ) { }

}

partial class MyClass

{

public void Method3( ) { }

public void Method4( ) { }

}

======================= 확장 메소드 ========================

기존 클래스의 기능을 확장하는 기법

기반 클래스를 물려받아 파생 클래스를 만든 뒤 여기에 필드나 메소드를 추가하는 상속과는 다름

확장 메소드를 이용하면 string 클래스에 문자열을 뒤집는 기능을 넣을 수도 있고, int 형식에 제곱 연산 기능을 넣을 수도 있음

[[기본형]]

namespace 네임스페이스이름

{ public static class 클래스이름

{

public static 반환형식 메소드이름( this 대상형식 식별자, 매개\_변수\_목록 )

{ ↑확장하고자 하는 클래스 또는 형식

//

}

}

}

[[예제]]

namespace MyExtension

{

public static class IntegerExtension

{

public static int Power(this int myInt, int exponent)

{

int result = myInt;

for (int i = 1; i < exponent; i++)

result = result \* myInt;

||

return result; ||

} ||

} ||

} ||

▼

using MyExtension; <------------------ 확장 메소드를 담는 클래스의

  네임스페이스를 사용합니다.

// …

int a = 2;

Console.WriteLine( a.Power( 3 ) ); <----- 마치 Power()가 원래부터 int형식의

Console.WriteLine( 10.Power(4) ); 메소드였던것 처럼 사용할수 있습니다.

======================= 구 조 체 =============================

구조체는 클래스하고는 사촌지간으로, 필드와 메소드를 가질 수 있는 등 상당 부분 비슷함

클래스 구조체

-----------|----------------------------------------------- | ----------------------------------------------------

키워드 | class | struct

-----------|----------------------------------------------- | ----------------------------------------------------

형식 | 참조형식 | 값 형식

-----------|----------------------------------------------- |----------------------------------------------------

복사 | 얕은 복사(Shallow Copy) | 깊은 복사(Deep Copy)

-----------|----------------------------------------------- | ----------------------------------------------------

인스턴스 | new연산자와 생성자 필요 | 선언만으로도 생성

생성 | |

-----------|----------------------------------------------- | ----------------------------------------------------

생성자 | 매개변수 없는 생성자 선언 가능 | 매개 변수 없는 생성자 선언 불가능

-----------|---------------------------------------------- | ----------------------------------------------------

상속 | 가능 | 모든 구조체는 System.Object 형식을

| | 상속하는 System.ValueType으로부터

| | 직접 상속받음.

\_\_\_\_\_\_\_\_\_\_|\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_|\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

[[예제]]

struct MyStruct

{

public int MyField1

public int MyFiled2

public void MyMethod()

{

// …

}

}

-------

MyStruct s;

s.MyField1 = 1;

s.MyField2 = 2;

  <=====s의 MyField1은 3, MyField2는 2이지만,

MyStruct t; t의 MyField1은 1, MyField2는 2입니다.

t = s;

s.MyField1 = 3;

**PPT.8**

====================== 인터페이스 ===========================

----------------[[ 기 본 형 ]]---------------------------

interface 인터페이스이름

{

반환형식 메소드이름1( 매개변수 목록 );

반환형식 메소드이름2( 매개변수 목록 );

반환형식 메소드이름3( 매개변수 목록 );

// …

}

-----------[[ 예 제 : 인터페이스 응용 ]]---------------

[[[1]]]

class ClimateMonitor

{

\*인터페이스\*--> private ILogger logger;

public ClimateMonitor(ILogger logger) <----- [[[3]]]- ClimateMonitor monitor

ClimateMonitor(new FileLogger("MyLog.txt"));

{

this.logger = logger;

}

public void start()

{

while ( true )

{

Console.Write( "온도를 입력해주세요.: " );

string temperature = Console.ReadLine();

if (temperature == "")

break;

\*인터페이스\*--> logger.WriteLog( "현재 온도 : " + temperature );

}

}

}

[[[2]]]

\*인터페이스\*--> class FileLogger : ILogger

{

private StreamWriter writer;

public FileLogger(string path)

{ ↑\_[[[3]]]]\_\_ ClimateMonitor monitor = new ClimateMonitor(new FileLogger("MyLog.txt"));

writer = File.CreateText(path);

writer.AutoFlush = true;

}

public void WriteLog(string message)

{

writer.WriteLine("{0} {1}", DateTime.Now.ToShortTimeString(), message);

}

}

[[[3]]]

ClimateMonitor monitor = new ClimateMonitor(new FileLogger("MyLog.txt"));

↑ [[[[1]]]] ↑[[[2]]]]

monitor.start();

-----------------------------------------------------------------------------------------------------------

인터페이스에 선언되어 있는 메소드를 구현하기만 한다면 해당 인터페이스를

지원하는 코드에는 그 인터페이스의 모든 파생 클래스를 사용할 수 있음

C#은 인터페이스의 다중상속은 지원한다.

인터페이스는 내용이 아닌 외형을 물려주기 때문에 죽음의 다이아 몬드 문제가 발생하지 않음

앞 페이지에서 선언한 ILogger 인터페이스를 상속받는 클래스는 WriteLog() 메소드를 구현할 때

로그를 파일에 기록하도록 할 수도 있고 네트워크 너머로 보내도록 할 수도 있음.

\* 인터페이스를 상속할 수 있는 것은 클래스 뿐만이 아님.

\* 구조체도 인터페이스 상속 가능하며

\* 인터페이스도 인터페이스를 상속할 수 있음.

\* 대체로 다음과 같은 경우에 인터페이스의 인터페이스 상속 이용

상속하려는 인터페이스가 소스 코드가 아닌 어셈블리로만 제공되는 경우

상속하려는 인터페이스의 소스 코드를 갖고 있어도 이미 인터페이스를 상속하는 클래스들이 존재하는 경우

[[인터페이스가 인터페이스를 상속]]

interface ILogger

{

void WriteLog(string message);

}

===============================추상 클래스=============================

추상클래스는 인터페이스와는 달리 "구현"을 가질 수 있음

그러나 클래스와는 다르게 인스턴스(메모리 할당)를 가질 수 없음---> 메모리 할당을 받을 수 없다.

추상 클래스는 클래스와는 달리 추상 메소드도 가질 수 있다

[[[추상 클래스 선언 형식]]]

abstract class 클래스이름

{

// 클래스와 동일하게 구현

}

interface IFormattableLogger : ILogger

{

void WriteLog(string format, params Object[] args);

}

추상 메소드는 구현을 갖지 않으며, 파생 클래스에서만 해당 메소드를 구현할 수 있음

[[추상 클래스 선언 예]]

abstract class AbstractBase

{

public abstract void SomeMethod();

}

class Derived : AbstractBase

{

public override void SomeMethod()

{

// Something

}

}

**PPT.9**

============ 프로퍼티 =========

[[[프로퍼티 선언 형식]]]

class 클래스이름’

{

데이터형식 필드이름;

접근한정자 데이터형식 프로퍼티이름

{

get

{

return 필드이름;

}

set

{

필드이름 = value;

}

}

}

[[[프로퍼티의 선언과 사용 예]]

class MyClass

{

private int myField;

public int MyField

{

get [데이터 받기, 초기화 필수]

{

return myField;

}

set [데이터 입력, 데이터 입력]

{

myField = value;

}

}

}

|||

||| 필드에 접근하듯 데이터에 접근

|||

▼

MyClass obj = new MyClass();

obj.MyField = 3 ;

Console.WriteLine( obj.MyField );

-\*-\*-\*-\*-\*-\*-\*-\*-\*-\*-\*-\*-\*- 자동 구현 프로퍼티 -\*-\*-\*-\*-\*-\*-\*-\*-\*-\*-\*-\*-\*-\*-\*-\*-\*-\*-\*-\*-\*-\*

C#은 자동 구현 프로퍼티를 제공함으로써 프로그래머가 의미없는 코드에

들이는 수고를 제거하도록 도움.

[[[프로퍼티의 자동 구현 예]]

ublic class NameCard

{

private string name;

private string phoneNumber;

public string Name

{

get { return name; }

set { name = value; }

}

public string PhoneNumber

{

get { return phoneNumber; }

set { phoneNumber = value; }

}

}

|||

프로퍼티 자동 구현

|||

▼

public class NameCard

{

public string Name

{

get; set;

}

public string PhoneNumber

{

get; set;

}

}

=======프로퍼티와 생성자======================

객체를 생성할 때 프로퍼티를 이용한 초기화 가능

[[[프로퍼티를 이용한 초기화]]]

클래스이름 인스턴스 = new 클래스이름()

{

프로퍼티1 = 값,

프로퍼티2 = 값, <===값 뒤에는 " , " <<\*\*콤마이다.\*\*

프로퍼티3 = 값

};

[[예시]]

BirthdayInfo birth = new BirthdayInfo()

{

Name = "서현",

Birthday = new DateTime(1991, 6, 28)

};

======= 무명 형식 =========

선언과 동시에 인스턴스를 할당하는 형식

“1회용” 데이터 형식(Data Type)으로 사용하며,

2번 이상 반복 사용할 데이터 형식이 필요한 경우에는 class나 struct를

선언하는 것이 나음

var 키워드와 프로퍼티를 이용하여 선언함

var myInstance = new { Name=”박상현”, Age=”17” };

Console.WriteLine( myInstance.Name, myInstace.Age );

========= 인터페이스의 프로퍼티 ==========

프로퍼디나 인덱서를 가진 인터페이스를 상속하는 클래스는

"반드시" 해당 프로퍼티와 인덱서를 구현해야함

[[1]]

interface IProduct

{

string ProductName

{

get;

set;

}

}

---> 자동 구현 프로퍼티처럼 보이지만 인터페이스 안에 선언된 프로퍼티는

“구현이 없는 상태”

[[2]]

 class Product : IProduct class

{

private string productName;

public string ProductName

{

get{ return productName; }

set{ productName = value; }

}

}

---> 인터페이스를 상속하는 클래스는 인터페이스에

선언되어 있는 프로퍼티는 반드시 구현해야 함