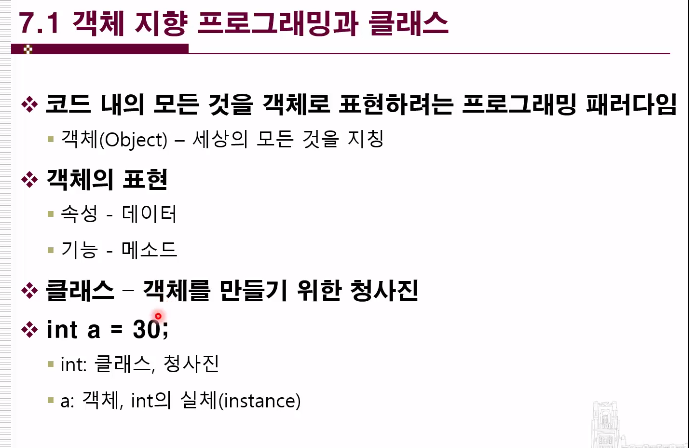
07.클래스

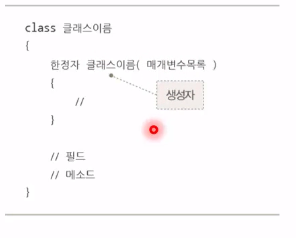


7.3 객체의 삶과 죽음을 관장하는 두가지 메소드

생성자와 종료자

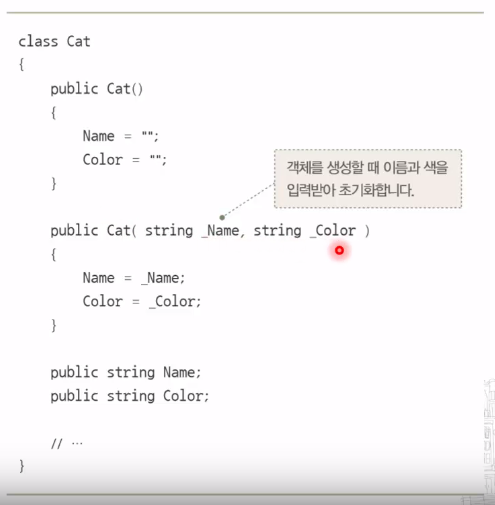
7.3.1생성자

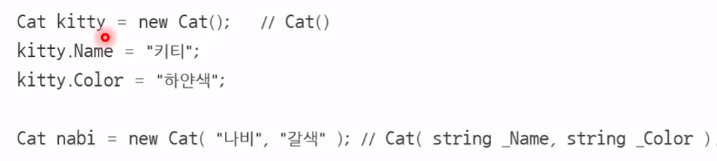
-클래스와 같은 이름이고 반환 형식은 없다.



컴파일러가 생성자를 만들어준다. 기본생성자

But 사용자가 직접 지정한 생성자





직접 생성자를 정하고 초기화 하는 경우.

7.3.2 종료자

-클래스 이름앞에 ~를 붙인 꼴.

#특징.

-매개변수도 없고 한정자도 사용하지 않음.

-오버로딩 불가능하고 직접 호출할 수 없음 -> CLR의 가비지컬렉터가 객체 소명 시점을 판단해서 종료자 호출

#종료자는 사용하지 말자

-CLR의 가비지 컬렉터의 동작 시점 예측 불가능

-명시적 종료자 구현은 성능 저하 초래 가능성 높음.

-CLR의 가비지 컬렉터는 객체 소멸 처리 **전문가**다.

namespace \_7\_3\_생성자와종료자

{

class Cat

{

public Cat()

{

Name = "";

Color = "";

}

public Cat(string \_Name, string \_Color)

{

Name = \_Name;

Color = \_Color;

}

~Cat() //종료자

{

Console.WriteLine($"{Name} : 잘가");

}

public string Name;

public string Color;

public void Meow()

{

Console.WriteLine($"{Name} : 야옹");

}

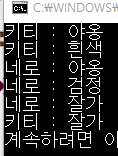
}

class MainApp

{

static void Main(string[] args)

{



Cat kitty = new Cat("키티", "흰색");

kitty.Meow();

Console.WriteLine($"{kitty.Name} : {kitty.Color}");

Cat nero = new Cat("네로", "검정");

nero.Meow();

Console.WriteLine($"{nero.Name} : {nero.Color}");

}

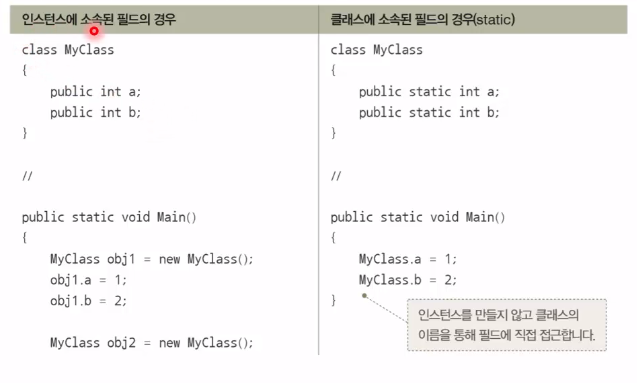
}

}

7.4 정적 필드와 메소드

#static은 메소드나 필드가 클래스 자체에 소속되도록 지정하는 한정자

#인스턴스 소속 필드vs 클래스 소속 필드



#프로그램 전체에 공유하는 변수에 사용

namespace \_7\_4\_정적필드와메소드

{

class Global

{

public static int Count = 0;

}

class ClassA

{

public ClassA()

{

Global.Count++;

}

}

class ClassB

{

public ClassB()

{

Global.Count++;

}

}

class Program

{

static void Main(string[] args)

{

Console.WriteLine($"Global.Count : {Global.Count}");

new ClassA();

new ClassA();

new ClassB();

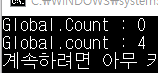
new ClassB();

Console.WriteLine($"Global.count : {Global.Count}");

}

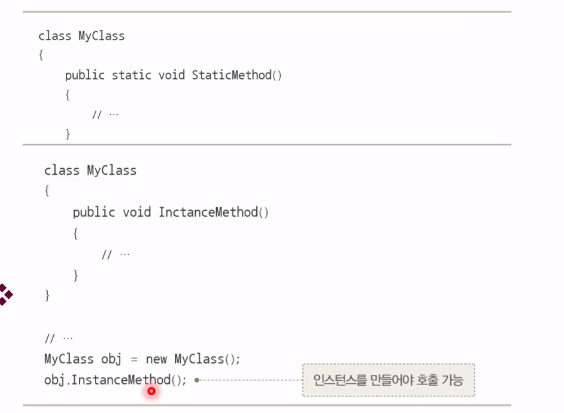
}

}



#정적메소드

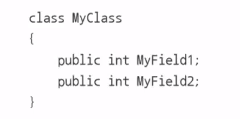
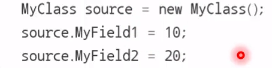


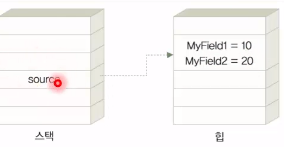


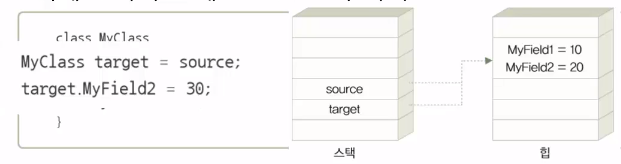
7.5 객체 복사하기:얕은 복사와 깊은 복사

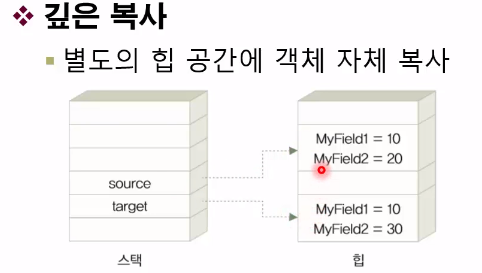
#얕은 복사

-객체를 복사할 때 참조만 살짝 복사.











namespace \_7\_5\_얕은복사와깊은복사

{

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;

}

}

class Program

{

static void Main(string[] args)

{

Console.WriteLine("shallow Copy");

{

MyClass source = new MyClass();

source.MyField1 = 10;

source.MyField2 = 20;

MyClass target = source;

target.MyField2 = 30;

Console.WriteLine($"{source.MyField1} {source.MyField2}");

Console.WriteLine($"{target.MyField1} {target.MyField2}");

}

Console.WriteLine("Deep Copy");

{

MyClass source = new MyClass();

source.MyField1 = 10;

source.MyField2 = 20;

MyClass target = source.DeepCopy();

target.MyField2 = 30;

Console.WriteLine($"{source.MyField1} {source.MyField2}");

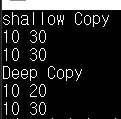
Console.WriteLine($"{target.MyField1} {target.MyField2}");

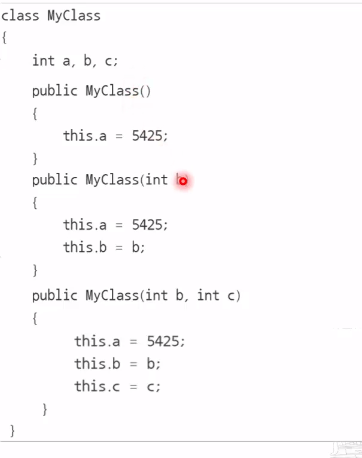
}

}

}

}

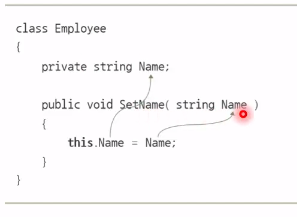




7.6 this

$객체가 자신을 지칭할 때 사용하는 키워드 this

-객체 내부에서 자신의 필드나 메소드에 접근할 때 사용



namespace \_7\_6\_this

{

class Employee

{

private string Name;

private string Position;

public void SetName(string Name)

{

this.Name = Name;

//필드네임 = 네임매개변수

}

public string GetName()

{

return Name;

}

public void SetPosition(string Position)

{

this.Position = Position;

}

public string GetPosition()

{

return this.Position;

}

}

class Program

{

static void Main(string[] args)

{

Employee pooh = new Employee();

pooh.SetName("Pooh");

pooh.SetPosition("Waiter");

Console.WriteLine($"{pooh.GetName()} {pooh.GetPosition()}");

Employee tigger = new Employee();

tigger.SetName("tigger");

tigger.SetPosition("Cleaner");

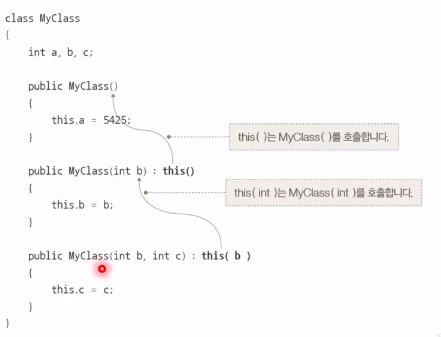
Console.WriteLine($"{tigger.GetName()} {tigger.GetPosition()}");

}

}

}

# this() 생성자

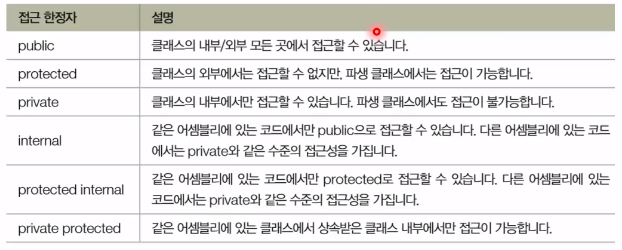


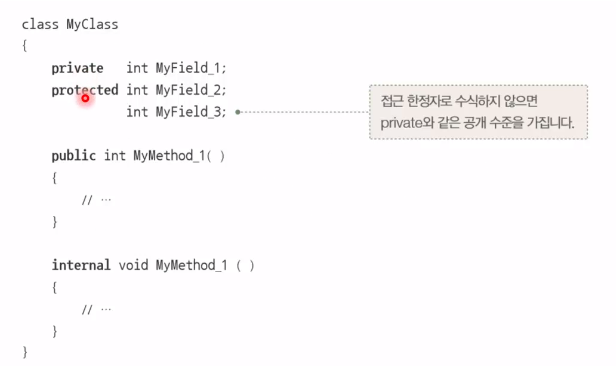
7.7 접근한정자로 공개수준 결정하기

#은닉성(캡슐화)의 구현

-감추고 싶은 것은 감추고, 보여주고 싶은 것만 보여준다.

#접근 한정자의 종류





namespace AccessModifier

{

class WaterHeater

{

protected int temperature;

public void SetTemperature(int temperature)

{

if (temperature < -5 || temperature > 42)

{

throw new Exception("Out of temperature range");

}

this.temperature = temperature;

}

internal void TurnOnWater( )

{

Console.WriteLine($"Turn on water : {temperature}");

}

}

class MainApp

{

static void Main(string[] args)

{

try

{

WaterHeater heater = new WaterHeater();

heater.SetTemperature(20);

heater.TurnOnWater();

heater.SetTemperature(-2);

heater.TurnOnWater();

heater.SetTemperature(50);

heater.TurnOnWater();

}

catch (Exception e)

{

Console.WriteLine(e.Message);

}

}

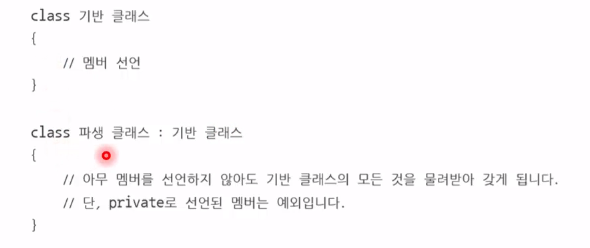
}

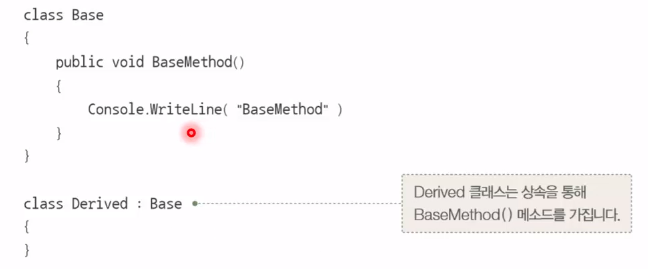
}

7.8 상속으로 코드 재활용하기

#물려받는 클래스가 물려줄 클래스 지정

#상속의 형식







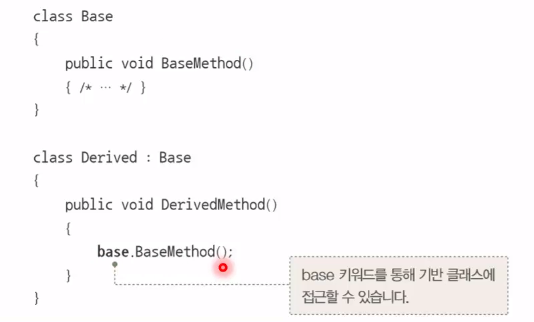
#파생클래스 = 자신만의 고유 멤버 + 기반 클래스 멤버

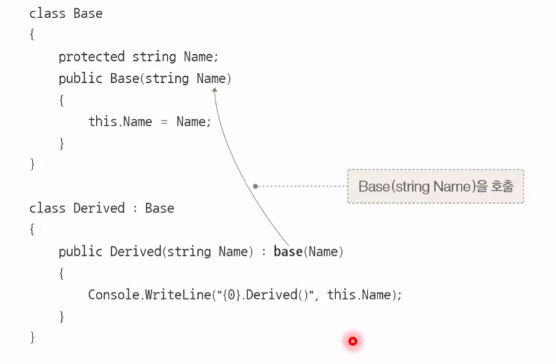
#파생 클래스의 수명 주기

기반생성자 -> 파생생성자 -> 파생종료자 -> 기반종료자

#기반 클래스의 멤버 호출 -> base

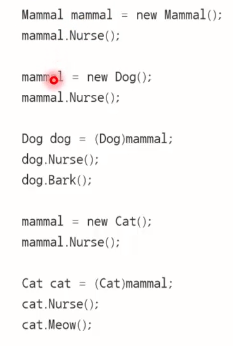
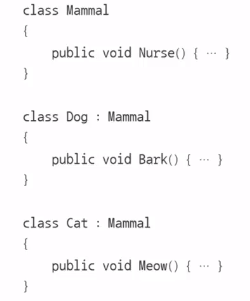
파생클래스의 생성자에서 기반 클래스 생성자에 매개변수 전달





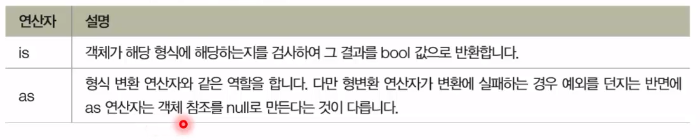
7.9 기반 클래스와 파생 클래스 사이의 형식 변환

#기반 클래스와 파생클래스 사이에 족보를 오르내리는 현식변환이 가능

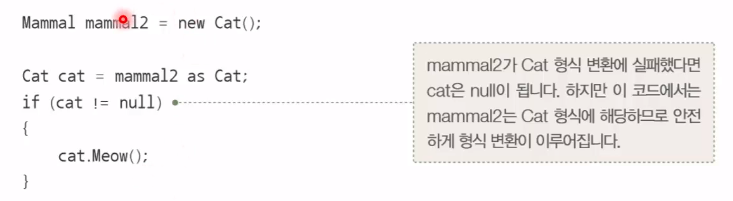
개와 고양이 -> 포유류

7.9 is와 as

@ C#의 형 변환 연산자







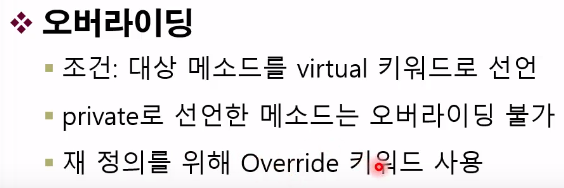
as연산자를 사용하면 jump되지 않고 null로 만듬

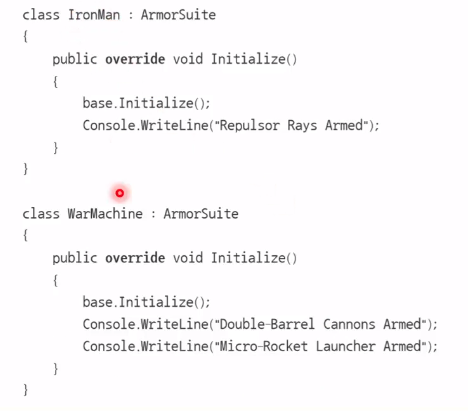
클래스2

7.10 오버라이딩과 다형성

@객체가 여러 형대를 가질 수 있음을 의미

원래 있는 메소드를 상속하여 더욱 업그레이드.





namespace \_7\_10\_오버라이딩

{

class ArmorSuite //부모클래스

{

public virtual void Initialize() //버츄얼로 선언

{

Console.WriteLine("Armored");

}

}

class IronMan : ArmorSuite

{

public override void Initialize() //오버라이드 라는 한정자 사용

{

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");

}

}

class MainApp

{

static void Main(string[] args)

{

Console.WriteLine("Creating ArmorSuite...");

ArmorSuite armorsuite = new ArmorSuite();

armorsuite.Initialize();

Console.WriteLine("\nCreating IronMan...");

ArmorSuite ironman = new IronMan();

ironman.Initialize();

Console.WriteLine("\nCreating WarMaching...");

ArmorSuite warmachine = new WarMachine();

warmachine.Initialize();

}

}

}

7.11 메소드 숨기기

@기반 클래스의 메소드를 감추고 파생 클래스 구현만 표시

@파생 클래스 버전의 메소드를 new한정자로 수식

@오버라이드와 다름 -> 완전한 다형성 표현의 한계

namespace \_7\_11\_메소드숨기기

{

class Base //부모클래스

{

public void MyMethod()

{

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

}

}

class Derived : Base //재정의한 클래스

{

public new void MyMethod()

{

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

}

}

class MainApp

{

static void Main(string[] args)

{

Base baseObj = new Base(); // 메소드숨기기 하더라도 원본메소드 선언가능

baseObj.MyMethod();

Derived derivedObj = new Derived();

derivedObj.MyMethod();

Base baseOrDerived = new Derived(); // 원본 메소드가 호출됨

baseOrDerived.MyMethod();

}

}

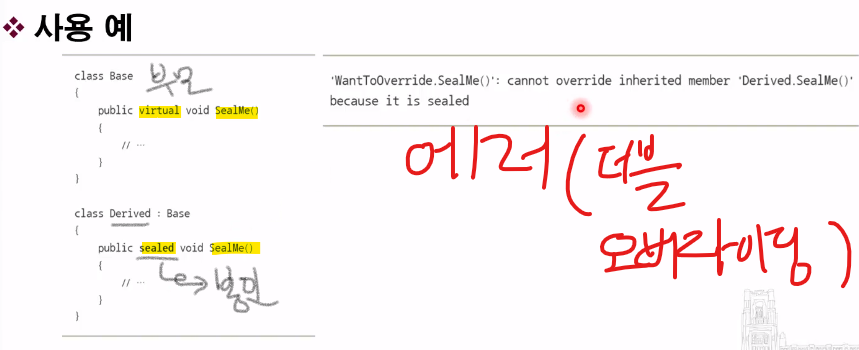
}

7.12 오버라이딩 봉인하기

@메소드의 오버라이딩 봉인

-대상- virtual 가상 메소드를 오버라이딩한 메소드

-오작동 위험이 있거나 잘못 오버라이딩 함으로써 문제가 예상되는 경우



namespace \_7\_12\_오버라이딩봉인

{

class Base

{

public virtual void SealMe()

{

}

}

class Derived : Base

{

public sealed override void SealMe()

{

}

}

class WantToOverride : Derived

{

public override void SealMe() //컴파일 하지 않아도 오류

{

}

}

class MainApp

{

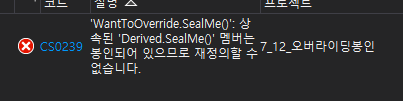
static void Main(string[] args)

{

}

}

}

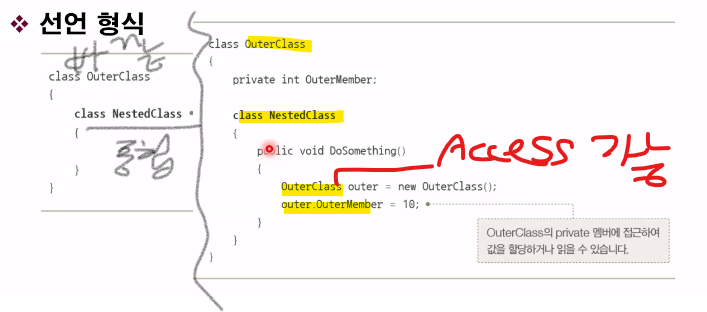


7.13 중첩 클래스

@클래스 안에 선언되어 있는 클래스

-소속되어 있는 클래스의 멤버에 자유롭게 접근

( 심지어 private멤버 포함)



@사용하는 이유

-클래스 외부에 공개하고 싶지 않은 형식을 만들고자 할 때

- 현재 클래스의 일부처럼 표현 가능한 클래스를 만들고자 할때

namespace \_7\_13\_중첩클래스

{

class Configuration

{

List<ItemValue> listConfig = new List<ItemValue>();

public void SetConfig(string item, string value)

{

ItemValue iv = new ItemValue();

iv.SetValue(this, item, value);

}

public string GetConfig(string item)

{

foreach (ItemValue iv in listConfig)

{

if (iv.GetItem() == item)

return iv.GetValue();

}

return "";

}

private class ItemValue //중첩클래스

{

private string item;

private string value;

public void SetValue(Configuration config, string item, string value)

{

this.item = item;

this.value = value;

bool found = false;

for (int i = 0; i < config.listConfig.Count; i++)

{

if (config.listConfig[i].item == item)

{

config.listConfig[i] = this;

found = true;

break;

}

}

if (found == false)

config.listConfig.Add(this);

}

public string GetItem()

{ return item; }

public string GetValue()

{ return value; }

}

}

class MainApp

{

static void Main(string[] args)

{

Configuration config = new Configuration();

config.SetConfig("Version", "V 5.0");

config.SetConfig("Size", "655,324 KB");

Console.WriteLine(config.GetConfig("Version"));

Console.WriteLine(config.GetConfig("Size"));

config.SetConfig("Version", "V 5.0.1");

Console.WriteLine(config.GetConfig("Version"));

}

}

}

(사실 코드이해 잘 안감)

7.14 분할 클래스

@여러 번에 나눠서 구현하는 클래스

-클래스의 구현이 길어질 경우 여러 파일에 나눠서 구현 -> 소스코드 관리의 편의를 제공

partial키워드 사용



namespace \_7\_14분할클래스

{

partial class MyClass //원본클래스

{

public void Method1()

{

Console.WriteLine("Method1");

}

public void Method2()

{

Console.WriteLine("Method2");

}

}

partial class MyClass //클래스는 1개인데 2개로 분리가능

{

public void Method3()

{

Console.WriteLine("Method3");

}

public void Method4()

{

Console.WriteLine("Method4");

}

}

class MainApp

{

static void Main(string[] args)

{

MyClass obj = new MyClass();

obj.Method1();

obj.Method2();

obj.Method3();

obj.Method4();

//하나의 클래스로 인식함

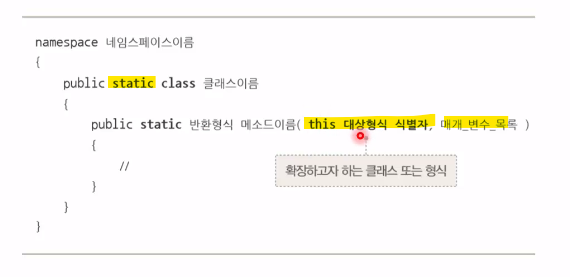
}

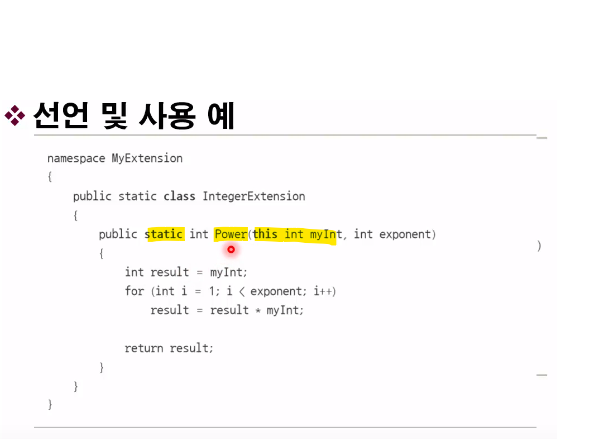
}

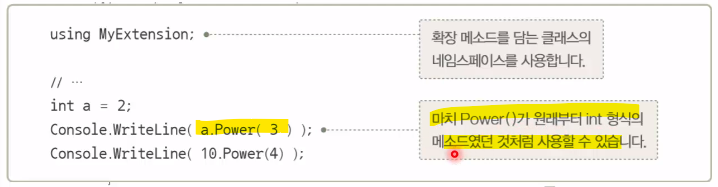
}

7.15 확장 메소드

@기존 클래스의 기능을 확장하는 기법







using \_7\_15\_확장메소드;

namespace \_7\_15\_확장메소드

{

public static class IntegerExtension

{

public static int Square(this int myInt)

{

return myInt \* myInt;

}

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

{

int result = myInt;

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

result = result \* myInt;

return result;

}

}

}

namespace \_7\_15\_확장메소드

{

class MainApp

{

static void Main(string[] args)

{

Console.WriteLine($"{3}^2 : {3.Square()}");

Console.WriteLine($"{3}^{4} : {3.Power(4)}");

Console.WriteLine($"{2}^{10} : {2.Power(10)}");

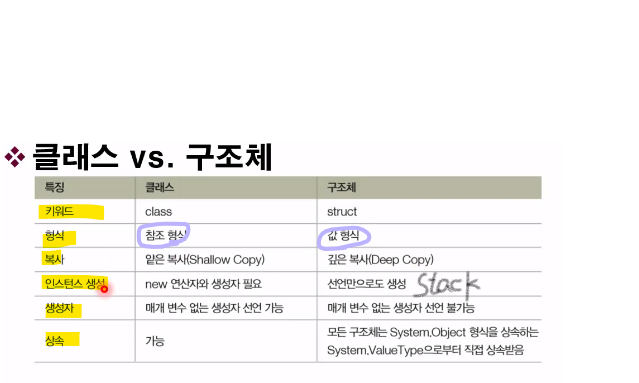
}

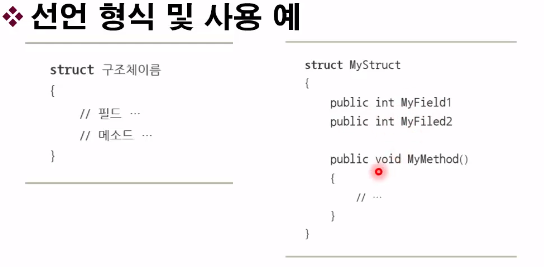
}

}

7.16 구조체

@클래스 vs 구조체





namespace \_7\_16\_구조체

{

struct Point3D //구조체

{

public int X;

public int Y;

public int Z;

public Point3D(int X, int Y, int Z) // 반드시 매개변수 있어야함 생성자

{

this.X = X;

this.Y = Y;

this.Z = Z;

}

public override string ToString()

{

return string.Format($"{X}, {Y}, {Z}");

}

}

class MainApp

{

static void Main(string[] args)

{

Point3D p3d1;

p3d1.X = 10;

p3d1.Y = 20;

p3d1.Z = 40;

// 초기화 선언 가능

Console.WriteLine(p3d1.ToString());

Point3D p3d2 = new Point3D(100, 200, 300);

Point3D p3d3 = p3d2;

p3d3.Z = 400;

Console.WriteLine(p3d2.ToString());

Console.WriteLine(p3d3.ToString());

}

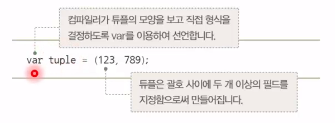
}

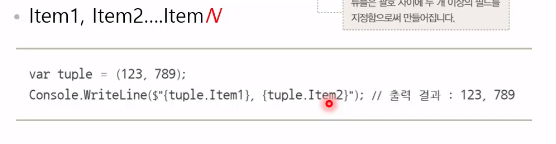
}

7.17 튜플

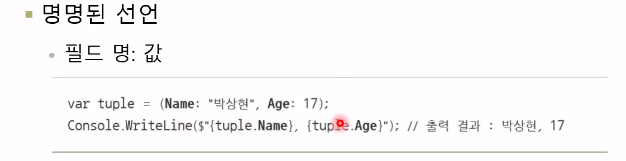
@여러 필드를 담을 수 있는 구조체

* 형식의 이름을 갖지 않는다
* 임시적으로 사용할 복합 데이터 형식 선언에 적합하다.

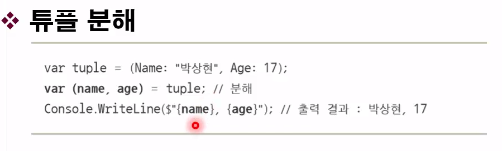
@튜플 선언

* 명명되지 않은 선언
* 
* 이름을 붙일 수 있음

(필드명 : 값)



@튜플 분해도 가능하다



특정필드는 무시하고 싶을때( \_ 밑줄사용)

