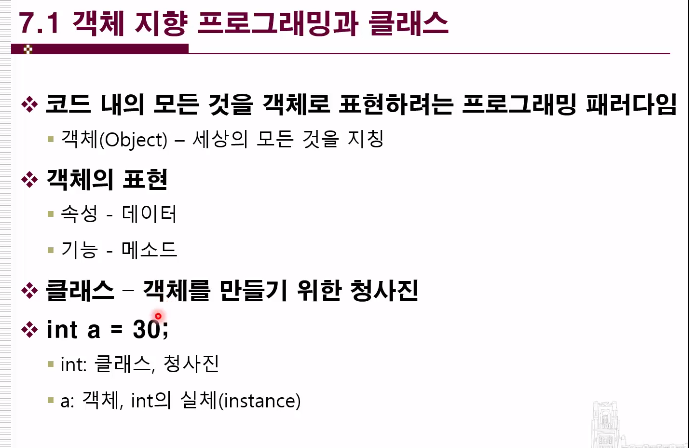
07.클래스

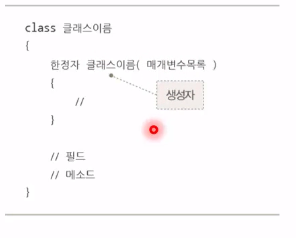


7.3 객체의 삶과 죽음을 관장하는 두가지 메소드

생성자와 종료자

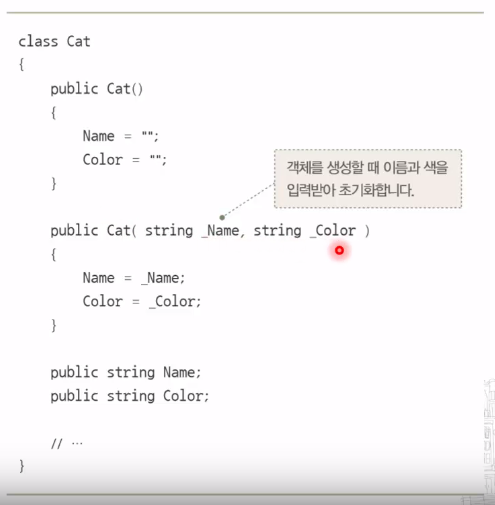
7.3.1생성자

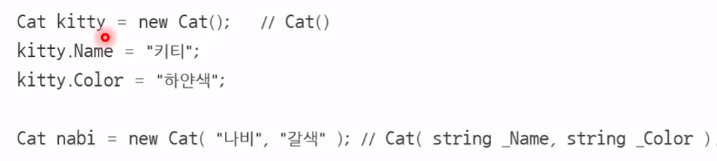
-클래스와 같은 이름이고 반환 형식은 없다.



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");

}

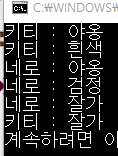
}

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}");

}

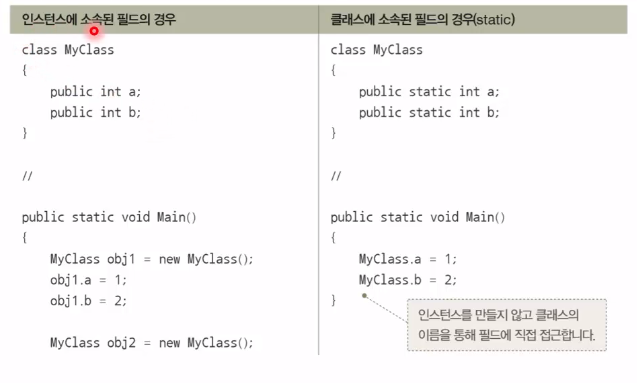
}

}

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();

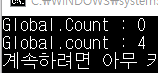
new ClassB();

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

}

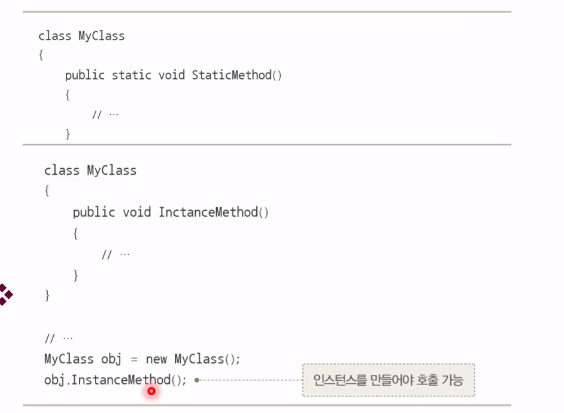
}

}



#정적메소드

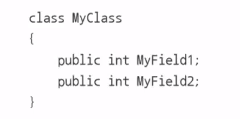
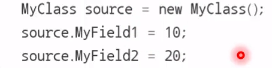


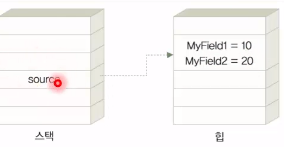


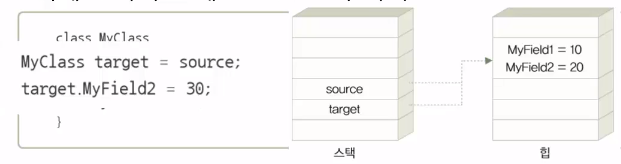
7.5 객체 복사하기:얕은 복사와 깊은 복사

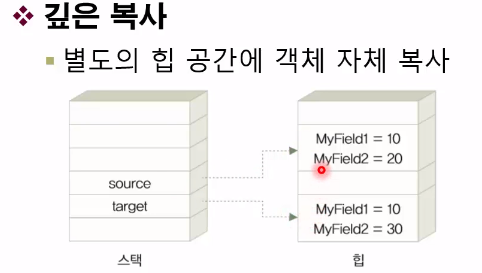
#얕은 복사

-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}");

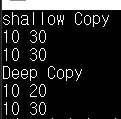
Console.WriteLine($"{target.MyField1} {target.MyField2}");

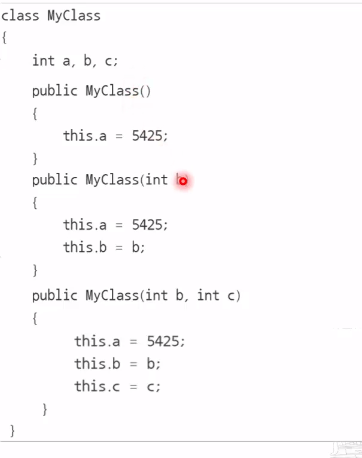
}

}

}

}

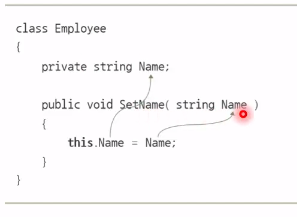




7.6 this

$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");

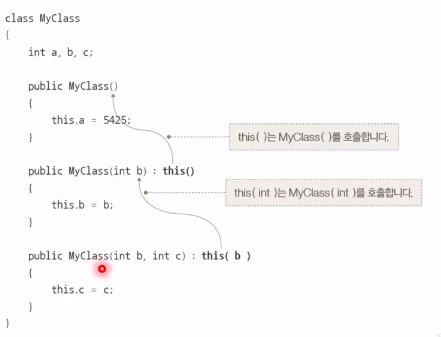
Console.WriteLine($"{tigger.GetName()} {tigger.GetPosition()}");

}

}

}

# this() 생성자

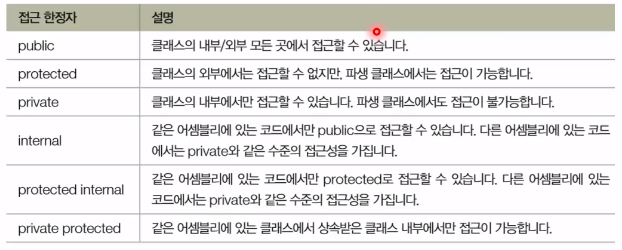


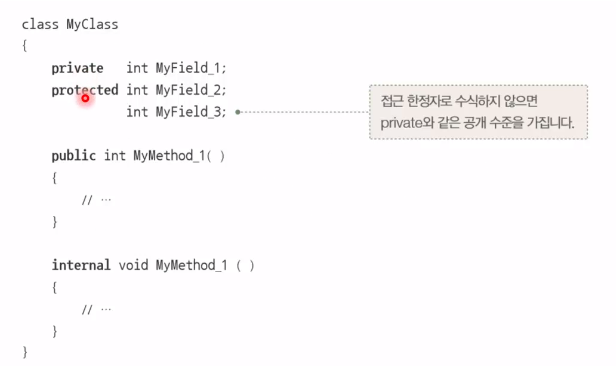
7.7 접근한정자로 공개수준 결정하기

#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);

}

}

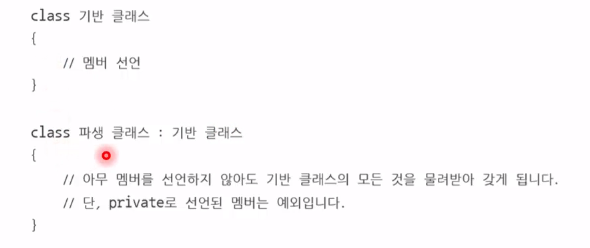
}

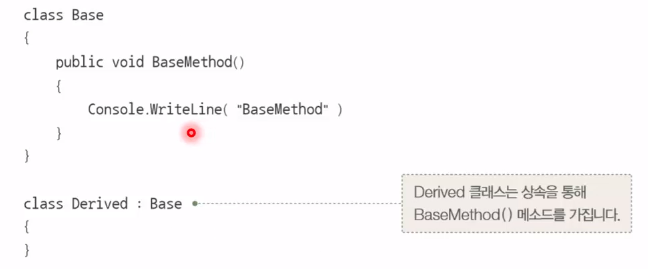
}

7.8 상속으로 코드 재활용하기

#물려받는 클래스가 물려줄 클래스 지정

#상속의 형식







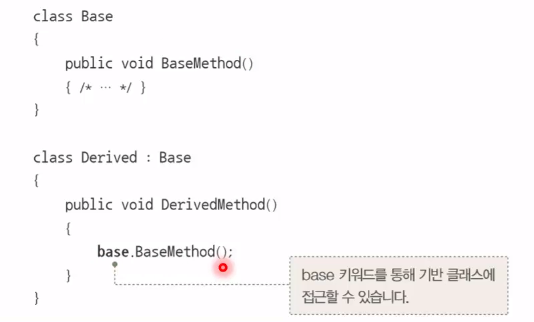
#파생클래스 = 자신만의 고유 멤버 + 기반 클래스 멤버

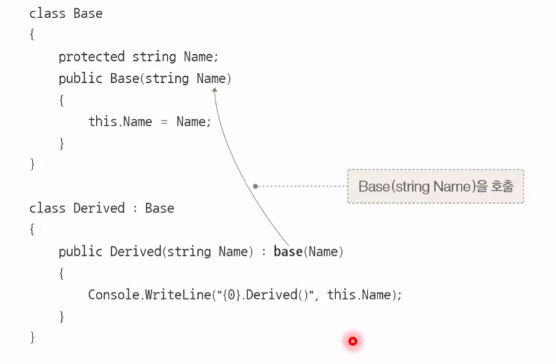
#파생 클래스의 수명 주기

기반생성자 -> 파생생성자 -> 파생종료자 -> 기반종료자

#기반 클래스의 멤버 호출 -> base

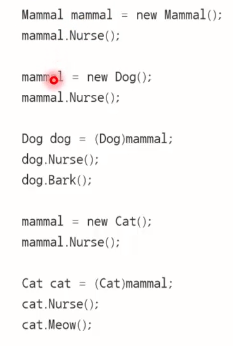
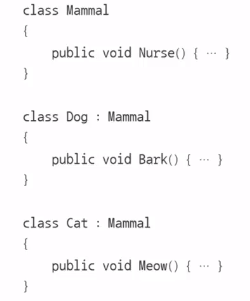
파생클래스의 생성자에서 기반 클래스 생성자에 매개변수 전달





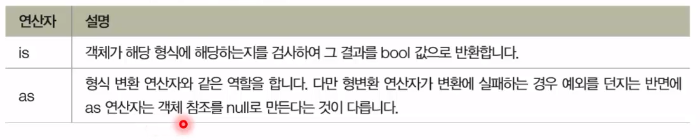
7.9 기반 클래스와 파생 클래스 사이의 형식 변환

#기반 클래스와 파생클래스 사이에 족보를 오르내리는 현식변환이 가능

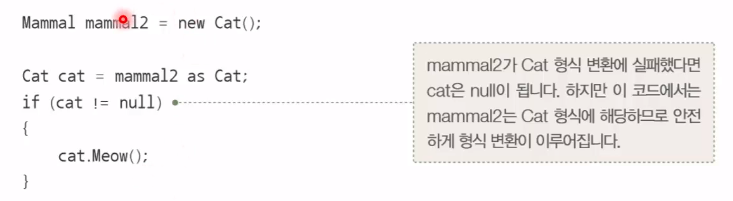
개와 고양이 -> 포유류

7.9 is와 as

@ C#의 형 변환 연산자







as연산자를 사용하면 jump되지 않고 null로 만듬