



# C#


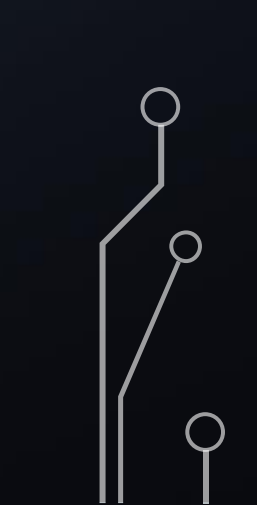
## -CHAPTER3-

SOUL SEEK



# 목차

---

1. 클래스, 인터페이스, 추상 클래스
  2. 프로퍼티
- 
- 

# 클래스와 프로퍼티

# 1. 클래스, 인터페이스, 추상 클래스

## 클래스

- 구조체는 **Value Type**, 클래스는 **Reference Type**이다.
- **C#**에서는 특정 값 타입을 반환하는 클래스는 사용 불가능하다.
- 클래스를 할당하면 인스턴스(**Instance**)라고 하며, 그렇지 않은 것은 클래스라고 한다.
- 클래스는 프로그램에서 하나만 존재하며, 인스턴스는 여러 개 존재할 수 있다.

### Class Cat

```
{  
    public string Name;  
    public string Color;  
  
    public void Meow()  
    {  
        Console.WriteLine("{0} : 야옹", Name);  
    }  
  
    public Cat();  
    public ~Cat();  
}
```

# 1. 클래스, 인터페이스, 추상 클래스

## New

- **New** 연산자와 생성자는 모든 데이터 형식에 사용이 가능하다.
- **GC**(가비지컬렉터)가 직접 수거하기 때문에 **delete**가 따로 필요하지 않지만 **GC**의 관리 이슈가 있다.

```
int a = new int();
```

```
A = 3;
```

```
string b = new string(new char []{'한', '글'});
```

하지만 이런 귀찮은 행동은 할 필요가 없다. 보여지는 예는 **new**를 사용할 때 조건을 간단하게 보여준 것에 불과하다. 클래스에 사용한다면,

```
class A = new class();
```

```
class A = new class(b, c);
```

이와 같이 클래스를 할당하면서 자연스럽게 생성자로 할당하는 모습이 연출된다. 기본값을 설정할 필요성이 있을 때 생성자에 다양한 오버로딩 함수로 활용하면 클래스의 데이터 관리가 편리하다.

단, 클래스가 처럼 **new**로 할당될 때의 설정이므로 이미 할당 받은 적이 있는 클래스는 따로 셋팅 메소드를 준비해서 변경해주는 구현이 필요하다.

# 1. 클래스, 인터페이스, 추상 클래스

## Static Filed(정적필드)와 Method

### static

**Method**나 **Filed**가 클래스의 인스턴스가 아닌 클래스 자체에 소속되도록 지정.  
지정한 필드는 인스턴스에 속하고 지정하지 않은 필드는 클래스에 속한다.

#### Class MyClass

```
{  
    public int a;  
    public int b;  
}
```

#### Class staticClass

```
{  
    public static int a;  
    public static int b;  
}
```

#### Public static void Main()

```
{  
    MyClass obj1 = new MyClass();  
    obj1.a = 1;  
    obj2.b = 2;  
  
    staticClass.a = 1;  
    staticClass.b = 2;  
}
```

# 1. 클래스, 인터페이스, 추상 클래스

## 객체의 복사 : 얇은 복사와 깊은 복사

- 클래스는 태생적으로 참조 형식이기 때문에 인스턴스로 사용하면 일반 대입형식으로 값을 복사 할 수 없다.
- 일반 대입은 스택에 있는 값들을 복사하는 것인데 참조형식으로 선언되는 인스턴스는 참조 값만 복사하기 때문에 **C#**에서는 포인터가 존재하지 않기 때문에 모든 구성이 동일한 깊은 복사를 위해서는 따로 구현이 필요하다.

얇은 복사

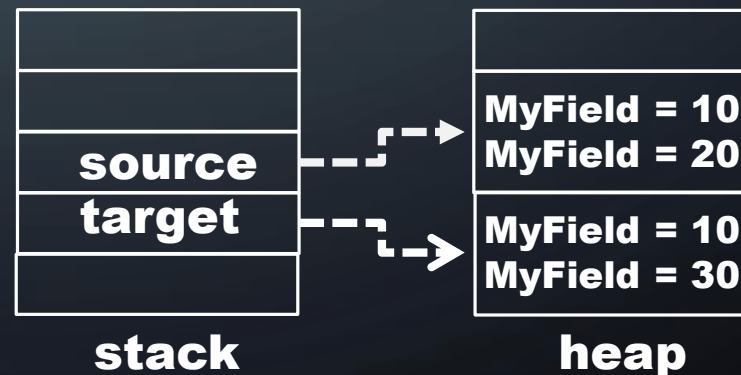
```
class MyClass
{
    public int MyField1;
    public int MyField2;
}
```



```
MyClass source = new MyClass();
source.MyField1 = 10;
source.MyField2 = 20;
```

```
MyClass target = source;
Target.MyField2 = 30;
```

깊은 복사



# 1. 클래스, 인터페이스, 추상 클래스

## **ICloneable.Clone() Method**

- **.Net Framework**의 **System** 네임스페이스에서 지원하는 인터페이스.
- 깊은 복사 기능을 가질 클래스가 다른 프로그래머가 작성한 코드와 호환되도록 하고 싶다면 사용하면 된다.

```
class MyClass : ICloneable
{
    public int MyField1;
    public int MyField2;

    public Object Clone()
    {
        MyClass newCopy = new MyClass();
        newCopy.MyField1 = this.MyField1;
        newCopy.MyField2 = this.MyField2;

        return newCopy;
    }
}
```



# 1. 클래스, 인터페이스, 추상 클래스

## This 키워드

객체 내부에서 자기 자신을 지칭해서 자기 자신의 모든 멤버를 사용할 수 있게 하는 것.

### Class Employee

```
{  
    private string Name;  
  
    public void SetName(string Name)  
    {  
        //Name = Name; // 컴파일러가 대략 @,.@ 하고 있다.  
        this.Name = Name;  
    }  
}
```

## is 와 as

연산자	설명
<b>is</b>	객체가 해당 형식에 해당하는지를 검사하여 그 결과를 <b>bool</b> 값으로 반환한다.
<b>as</b>	형식 변환 연산자와 같은 역할을 한다. 다만 형변환 연산자가 변환에 실패하는 경우 예외를 던지는 반면에 <b>as</b> 연산자는 객체 참조를 <b>null</b> 로 만든다는 것이 다르다.

# 1. 클래스, 인터페이스, 추상 클래스

**is** 연산자 사용.

```
Mammal mammal = new Dog();  
Dog dog;
```

```
if(mammal is Dog)  
{  
    dog = (Dog)mammal;  
    dog.Bark();  
}
```

**Mammal** 객체가 **Dog** 형식임을 확인했으므로  
안전하게 형식 변환이 이루어진다.

**as** 연산자 사용.

```
Mammal mammal = new Cat();  
Cat cat = mammal as Cat;
```

```
if(cat != null)  
{  
    cat.Meow();  
}
```

**Mammal**이 **Cat** 형식 변환에 실패했다면  
**cat**은 **null**이 된다.  
하지만 이 코드에서는 **mammal**은 **Cat** 형식에  
해당하므로 안전하게 형식 변환이 이루어진다.

# 1. 클래스, 인터페이스, 추상 클래스

## Partial Class(분할 클래스)

- 하나의 클래스를 여러 번에 나눠서 구현하는 클래스
- 특별한 기능이 있는 것이 아니라 구현이 길어질 경우 여러 파일로 나눠서 구현 할 수 있게 함으로써 소스코드 관리의 편의를 제공한다.
- 클래스 멤버 변수가 확정적으로 확립되었을 때, **Method**를 따로 분할 할 때 주로 사용한다.
  - ➔ 테이블 데이터들을 로드 할 때, 데이터 타입을 나열해서 클래스를 작성하기 때문에 거의 고정이고 기능적인 역할을 하는 **Method**를 구현하기 위한 **Class**를 분할하여 사용 할 때 주로 사용.

```
partial class MyClass
{
    public int a;
    public string b;
}
```

```
partial class MyClass
{
    public void Method1(){}
    public void Method2(){}
}
```

```
MyClass obj = new MyClass();
```

```
int c = Obj.a;
string d = Obj.b;
obj.Method1();
obj.Method2();
```

# 1. 클래스, 인터페이스, 추상 클래스

## 접근제어

- **C#**에서는 **private**와 **public**을 **C++**처럼 선언하지 않고 각 변수마다 **public int a**라는 형식으로 직접적으로 선언해줘야 한다.
- 구조체의 경우 **struct**가 **public** 이라고 암묵적으로 구성변수도 **public** 으로 되는 것이 아니다.
- 이루고 있는 변수들도 같이 **public** 으로 다 선언해줘야 한다.

## 인터페이스(Interface)

**Interface** 인터페이스 이름 → 클래스와의 혼용을 방지하기 위해 이름 앞에 **I**를 붙여 사용한다.

```
{  
    메소드 1;  
    메소드 2;  
    메소드 3;  
}
```

- **C#**의 클래스는 한 클래스가 여러 부모를 상속하지 못하기 때문에 **Interface**를 많이 활용한다.
- 클래스와 비슷하지만 메소드, 이벤트, 프로퍼티만 제한 적으로 가질 수 있다.
- 인스턴스를 가질 수 없기 때문에 구현부가 따로 없다. → 도구만 제공한다.
- 메소드, 이벤트, 프로퍼티를 제공하고 직접 상속받아 사용 하는 객체가 정의해야 한다.  
→ 기능은 만들어 사용하더라.

# 1. 클래스, 인터페이스, 추상 클래스

## 추상 클래스

- 추상 클래스는 인터페이스와 달리 **public**과 **private**를 지정해 줄 수 있다.
- 인터페이스와 달리 구현부를 가질 수 있다.
- 클래스와 달리 인스턴스를 가질 수 없지만 추상 메소드를 가질 수 있다.
  - ➔ 추상 클래스를 상속하는 클래스에 구현을 강제하기 때문에 상속하는 클래스는 반드시 해당 메소드를 가지고 있을 거라는 약속인 것이다.

```
abstract class 클래스 이름
{
    // 클래스와 동일하게 구현
}
```

```
abstract class AbstractBase
{
    public abstract void SomeMethod();
}
```

```
class Derived : AbstractBase
{
    public override void SomeMethod()
    {
        // Something
    }
}
```

프로퍼티

## 2. 프로퍼티

- 접근제어에 의해 차단되어 있는 객체로의 접근방법을 제시한다.
- **C/C++**에서는 반환형 **return** 과 **inline** 형식을 이용해서 제공했다.
  - ➔ 접근방법으로 제공한 것이 아닌 기능을 응용한 것이므로 제공하는 기능이 있다면 그것을 사용하자.
- **Method**를 만드는 것보다 **get, set**을 이용한 프로퍼티를 사용하는 것이 좋다.

### get, set

- 접근자라고 하며 접근제어로 감추어진 값을 대입하거나 얻어오게 하는 역할을 한다.
- 직접적인 참조나 대입이 아닌 간접적인 허락에 의한 행동.
- **Method**를 사용하는 것보다 유리
- **set**사용하면 **private**로 선언한 의미가 너무 퇴색된다 다른 곳에 선언해서 함부로 사용되는 데이터를 보호하기 위해 사용되는 것인데 **set**을 통해 저장하게 되면 자기가 원하는 순간에 대입해서 사용하려 하는 의미가 줄어들게 되는 것이다. 그래서 **set**이 존재하지만 따로 대입 함수를 만들어서 사용하는 걸 추천한다.

**private** 데이터 형식 필드이름;

**pubic** 데이터 형식 프로퍼티

{

**get** { **return** 필드이름;}

**set** { 필드이름 = **value**;}

}