

C#프로그래밍의 이해

6주차

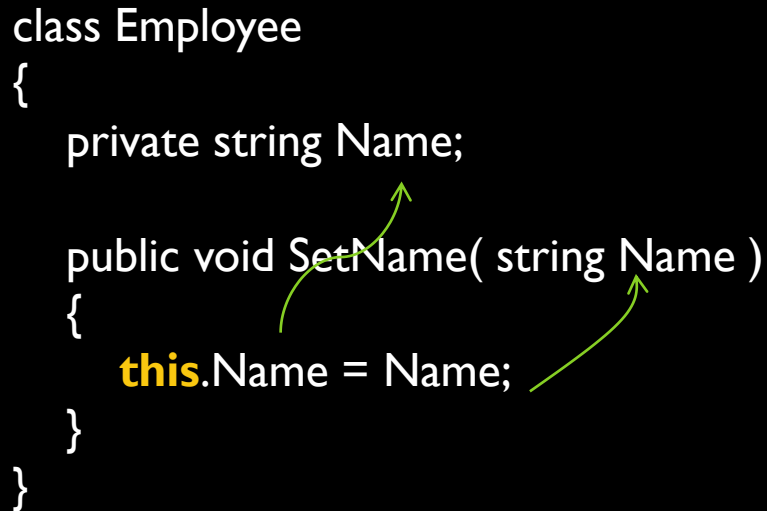
05. THIS 키워드

▶ this 키워드

- ▶ 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 Name;

    public void SetName( string Name )
    {
        this.Name = Name;
    }
}
```



06. 접근 한정자로 공개 수준 결정하기(1/2)

▶ 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

| 접근 한정자 | 설명 |
|---------------------------|--|
| public | 클래스의 내부/외부 모든 곳에서 접근할 수 있습니다. |
| protected | 클래스의 외부에서는 접근할 수 없지만, 파생 클래스에서는 접근이 가능합니다. |
| private | 클래스의 내부에서만 접근할 수 있습니다. 파생 클래스에서도 접근이 불가능합니다. |
| internal | 같은 어셈블리에 있는 코드에 대해서만 public 으로 접근할 수 있습니다. 다른 어셈블리에 있는 코드에서는 private 와 같은 수준의 접근성을 가집니다. |
| protected internal | 같은 어셈블리에 있는 코드에 대해서만 protected 로 접근할 수 있습니다. 다른 어셈블리에 있는 코드에서는 private 와 같은 수준의 접근성을 가집니다. |

06. 접근 한정자로 공개 수준 결정하기(2/2)

▶ 접근한정자 사용 예

```
class MyClass
{
    private int MyField_1;
    protected int MyField_2;

    int MyField_3;

    public int MyMethod_1 ( )
    {
        // ...
    }

    internal void MyMethod_1 ( )
    {
        // ...
    }
}
```

접근한정자를 명시하지 않으면
클래스의 필드는 private

07. 상속으로 코드 재활용하기 (1/3)

▶ 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

```
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.
}
```

07. 상속으로 코드 재활용하기 (2/3)

▶ 상속의 예

```
class Base
{
    public void BaseMethod()
    {
        Console.WriteLine( "BaseMethod" );
    }
}

class Derived : Base
{
}
```

Derived 클래스는 상속을 통해
Base.BaseMethod()를 얻음

▶ 파생클래스는 기반 클래스의 멤버에 새로운 멤버을 엮어 만드는 것임

파생 클래스

새로운 멤버

기반 클래스

07. 상속으로 코드 재활용하기 (3/3)

▶ base 키워드

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

▶ base 키워드 사용 예

```
class Base
{
    public void BaseMethod()
    { /* ... */ }
}

class Derived : Base
{
    public void DerivedMethod()
    {
        base.BaseMethod();
    }
}
```

base 키워드를 통해 기반 클래스에 접근할 수 있습니다.

08. 기반 클래스와 파생 클래스 사이의 형식 변환, 그리고 IS와 AS (1/4)

- ▶ 기반 클래스와 파생 클래스 사이에서는 족보를 오르내리는 형식 변환이 가능
 - ▶ 자식 클래스의 객체는 부모 클래스의 객체로 간주할 수 있음
 - ▶ 즉, 파생 클래스의 인스턴스는 기반 클래스의 인스턴스로서도 사용 가능

```
class Mammal
{
    public void Nurse() { ... }
}
```

```
class Dog : Mammal
{
    public void Bark() { ... }
}
```

```
class Cat : Mammal
{
    public void Meow() { ... }
}
```

```
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();
```


08. 기반 클래스와 파생 클래스 사이의 형식 변환, 그리고 IS와 AS (2/4)

- ▶ 기반클래스-파생클래스간 형변환을 어디에 사용할 수 있을까?

```
class Zookeeper
{
    public void Wash( Dog dog ) { /* ... */ }
    public void Wash( Cat cat ) { /* ... */ }
    public void Wash( Elephant elephant ) { /* ... */ }
    public void Wash( Lion lion ) { /* ... */ }

    // ... 300개 버전의 Wash() 메소드 선언
}
```

VS

```
class Zookeeper
{
    public void Wash( Mammal mammal ) { /* ... */ }
}
```

- ▶ 상속 관계를 이용한 형변환을 이용하여 Zookeeper를 구현하면 오버로딩 없이 한가지 버전의 메소드만으로 다양한 클래스 형식의 매개변수를 처리할 수 있음

08. 기반 클래스와 파생 클래스 사이의 형식 변환, 그리고 IS와 AS (3/4)

▶ is 연산자와 as 연산자

| 연산자 | 설명 |
|-----|---|
| is | 객체가 해당 형식에 해당하는지를 검사하여 그 결과를 boolean 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. |

08. 기반 클래스와 파생 클래스 사이의 형식 변환, 그리고 IS와 AS (4/4)

▶ is 연산자 사용 예

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

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

mammal 객체가 Dog 형식임을 확인했으므로 안전하게 형식 변환이 이루어집니다.

▶ as 연산자의 사용 예

```
Mammal mammal2 = new Cat();  
  
Cat cat = mammal2 as Cat;  
if (cat != null)  
{  
    cat.Meow();  
}
```

mammal2가 Cat 형식 변환에 실패했다면 cat은 null이 됩니다. 하지만 이 코드에서는 mammal2는 Cat 형식에 해당하므로 안전하게 형식 변환이 이루어집니다.

09. 오버라이딩과 다형성 (1/2)

▶ 다형성(Polymorphism)

- ▶ OOP에서 다형성은 객체가 여러 형태를 가질 수 있음을 의미
- ▶ 하위 형식 다형성 (Subtype Polymorphism) 의 준말
- ▶ 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)

09. 오버라이딩과 다형성 (2/2)

- ▶ ArmorSuite 클래스는 IronMan, WarMachine 등 다양한 모습으로 파생

```
class ArmorSuite
{
    public virtual void Initialize()
    {
        Console.WriteLine("Armored");
    }
}
```

부모 클래스에서 미리 virtual
로 메소드를 선언

자식 클래스에서는 override 로
메소드 재선언(재정의)

```
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");
    }
}
```

10. 메소드 숨기기 (1/2)

- ▶ 오버라이딩을 하기 위해서는 기반 클래스가 단단하게 설계되고 구현되어야 함.
 - ▶ 파생 클래스에서 `override` 로 재정의 하려면 기반 클래스의 메소드가 `virtual`로 선언되어 있어야 함.
- ▶ 메소드 숨기기는 기반 클래스의 메소드가 `virtual`로 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)

10. 메소드 숨기기 (2/2)

▶ 메소드 숨기기 예

```
class Base
{
    public void MyMethod()
    {
        Console.WriteLine("Base.MyMethod()");
    }
}

class Derived : Base
{
    public new void MyMethod()
    {
        Console.WriteLine("Derived.MyMethod()");
    }
}
```

부모 클래스에서 메소드를
virtual로 선언하지 않음

자식 클래스에서 **new** 한정자를 이용
하여 재선언함으로써 기반 클래스
버전의 메소드를 숨김

II. 오버라이딩 봉인하기

- ▶ `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`로 메소드를 선언하면 이 클래스를 상속하는 클래스에서는 `SealMe()` 메소드를 오버라이딩할 수 없음

12. 중첩 클래스

- ▶ 중첩 클래스는 클래스 안에 선언된 클래스를 말함
- ▶ 아래는 중첩 클래스의 예

```
class OuterClass
{
    private int OuterMember;

    class NestedClass
    {
        public void DoSomething()
        {
            OuterClass outer = new OuterClass();
            outer.OuterMember = 10;
        }
    }
}
```

OuterClass의 private 멤버에 접근하여 값을 할당하거나 읽을 수 있습니다.

13. 분할 클래스

- ▶ 분할 클래스(Partial Class)란, 여러 번에 나눠서 구현하는 클래스
 - ▶ 그 자체로 특별한 기능이 있는 것은 아님

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

```
partial class MyClass
{
    public void Method3 ( ) { }
    public void Method4 ( ) { }
}
// ...
```

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

```
obj.Method1 ( );
```

```
obj.Method2 ( );
```

```
obj.Method3 ( );
```

```
obj.Method4 ( );
```

14. 구조체 (1/2)

- ▶ 구조체는 클래스하고는 사촌지간으로, 필드와 메소드를 가질 수 있는 등 상당 부분 비슷함
 - ▶ 하지만 차이점도 상당수 있음. 다음표를 참조

| 특징 | 클래스 | 구조체 |
|---------|---------------------|--|
| 키워드 | class | struct |
| 형식 | 참조 형식 | 값 형식 |
| 복사 | 얕은 복사(Shallow Copy) | 깊은 복사(Deep Copy) |
| 인스턴스 생성 | new 연산자와 생성자 필요 | 선언만으로도 생성 |
| 생성자 | 매개 변수 없는 생성자 선언 가능 | 매개 변수 없는 생성자 선언 불가능 |
| 상속 | 가능 | 모든 구조체는 System.Object 형식을 상속하는 System.ValueType으로부터 직접 상속받음. |

14. 구조체 (2/2)

▶ 구조체 선언 및 사용 예

```
struct MyStruct
{
    public int MyField1
    public int MyFiled2

    public void MyMethod()
    {
        // ...
    }
}
```

```
MyStruct s;
s.MyField1 = 1;
s.MyField2 = 2;
```

```
MyStruct t;
t = s;
s.MyField1 = 3;
```

s의 MyField1은 3, MyField2는 2이지만,
t의 MyField1은 1, MyField2는 2입니다.