실습내용 정리 / 복습자료

```
public class Lecture{
    7 references
    private int _firstMember;
    1 reference
    private bool _firstMemberIsModified = false;
    0 references
    public int FirstMember{
        get => _firstMember;
        set {
            _firstMember = value;
            _firstMemberIsModified = true;
        }
    }

    0 references
    public Lecture()
    {
        _firstMember = 0;
        Console.WriteLine("Initializer With no args");
    }

    0 references
    public Lecture(int initialFirstMember)
    {
        _firstMember = initialFirstMember;
        Console.WriteLine($"Initializer With args {initialFirstMember}");
    }

    0 references
    public int FirstMember2 => _firstMember;

    0 references
    public int GetFirstMember() // getter
    {
        return _firstMember;
    }

    0 references
    public void SetFirstMember(int aValue) // setter
    {
        _firstMember = aValue;
    }
}
```

클래스 선언 - 네이밍, 상속, 유형

멤버 변수 선언

프로퍼티를 통한 멤버 변수 접근

기본 생성자 정의

매개변수 입력 생성자 정의

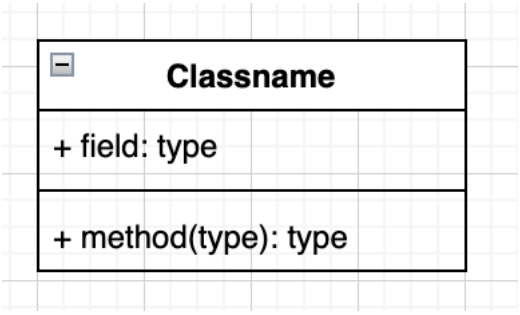
암시적 Get 프로퍼티 선언

Getter 동작의 예

Setter 동작의 예

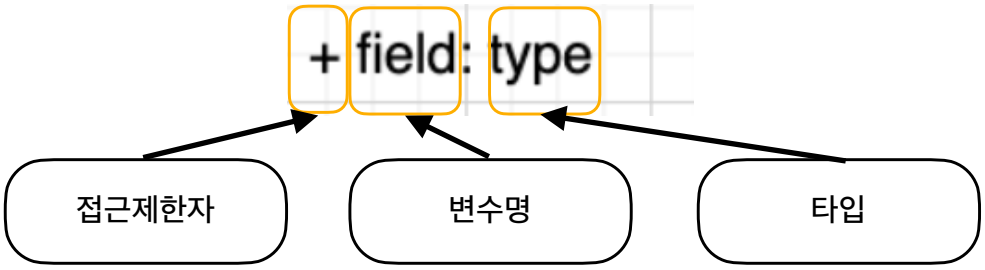
접근제한자를 통해 멤버 변수를 클래스 외부로부터 보호 할 수 있다고 했습니다.
예제에서는 보호된 멤버 변수의 값을 클래스 외부에서 안전하게 얻어오고,
값 수정의 경로를 제한함으로써 해당 클래스가 그러한 상황을 통제할 수 있음을 보여줍니다.

* 프로그램 설계를 관리하고 공유하기 위해서 UML 이라는 특별한 표준을 사용합니다. 설계 구조내에서 클래스의 관계나 클래스의 구현 상황을 표현하며 , 다루는 언어나 툴에 따라 조금씩 다른 키워드를 사용하기도 하지만, 기본적인 틀은 같으니 어떻게 작성하고 읽는지 정도를 알아두시면 좋습니다.



위는 클래스의 기본적인 UML 다이어그램 입니다.
클래스 이름과, 멤버 변수, 메소드를 각각의 칸을 지정하여 나열합니다.

멤버 변수를 설명하자면



접근제한자 형식은 아래와 같습니다.

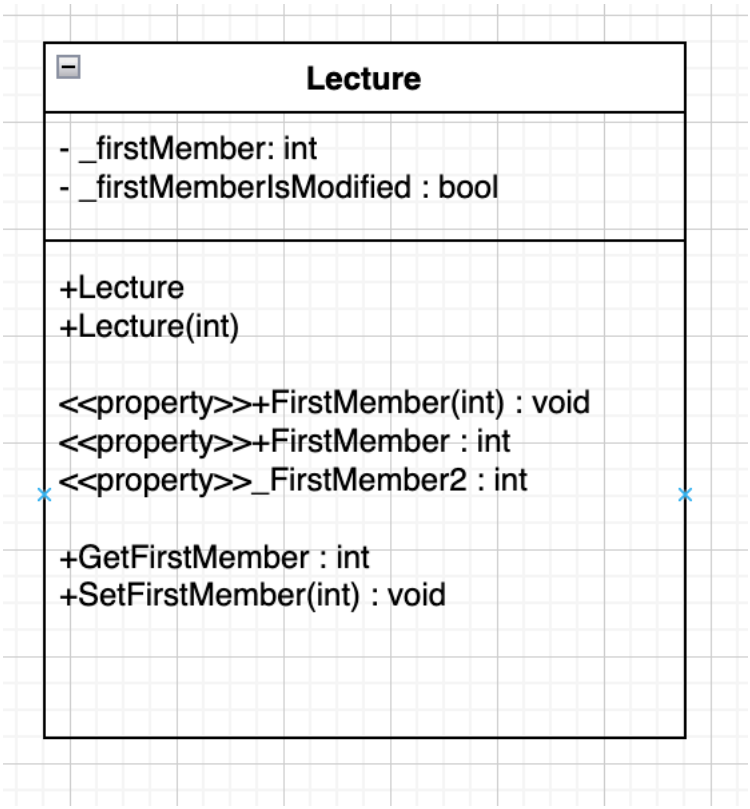
private	: -
public	: +
protected	: #
static	: 밑줄

메소드에 대해서는

변수와 구조가 같으나, 메소드의 인수 부분을 괄호로 지정하고 내부에 인수의 변수 타입을 작성해줍니다.

다이어그램 작성 사이트 draw.io 에서 하단 UML 항목의 내용으로 연습할 수 있습니다.

예제로 제시한 Lecture 클래스의 UML 다이어그램은 다음과 같습니다



클래스 외부의 한 지점에서 예제 클래스 Lecture 의 참조 변수와 인스턴스를 생성하고 접근 가능한 메소드를 호출해봅니다.

```
Lecture aLecture = new Lecture();

aLecture.SetFirstMember(10);
int valueOfFirstMemeber = aLecture.GetFirstMember();
```

게임의 한 장면을 예제 삼아 설계해봅니다.

각각의 체력과 공격력이 존재하는 3가지 타입의 객체가 있다고 가정합니다.

이 객체는 각각 Player, Monster, Boss 입니다.

동작을 정의해봅니다.

Player 와 Monster 는 서로 공격이 가능하며 공격에 의해 데미지를 입기도 합니다.

Boss 는 몬스터와 동일한 속성이지만 다른 특수한 패턴이 있을 수 있으므로 미리 확장해둡니다.

이 세 객체가 공유하는 공통적인 속성을 추출해보면

체력, 공격력 이라는 고유 데이터와, '공격' , '피해입음' 이라는 동작으로 정리가 가능합니다.

이 공통된 동작을 먼저 정의하고 나머지 객체들이 상속해가는 방식으로 설계하면 효율적이므로

이를 Actor 라는 이름으로 작성합니다.

```
public class Actor {
    1 reference
    private int _health;
    2 references
    protected int _atk;

    6 references
    public virtual void Attack(Actor target){
        target.Hit(_atk);
    }

    2 references
    public virtual void Hit(int damage)
    {
        _health -= damage;
    }
}
```

이 때, 상속받을 클래스들의 공격과 피해받음 동작시 고유의 역할이 예상되므로
(게임에서 플레이어가 맞을때와 몬스터가 맞을때는 소리부터 조금 다르죠?)

이를 가상 메소드로 선언하여 오버라이드 가능성을 만들어둡니다.

Actor 를 부모클래스 삼아 나머지 객체를 정의합니다.

```
public class Player : Actor {
    3 references
    public override void Attack(Actor target)
    {
        target.Hit(_atk);
        Console.WriteLine("Player overrided Attack");
    }
}
```

```
public class Monster : Actor {
    3 references
    public override void Attack(Actor target)
    {
        base.Attack(target);
        Console.WriteLine("Monster overrided Attack");
    }
}

2 references
public class Boss : Actor {

}
```

가상함수이기 때문에 오버라이드 하지 않아도 부모 클래스에서 정의된 동작을 하겠죠.

보스몬스터는 아직 기획이 없기 때문에 놔둡니다.

플레이어 클래스는 유저가 컨트롤한 입력을 처리해야하는 특수한 Actor 입니다.

그러나 이 동작은 Actor 에 넣기에는 말그대로 특수하고, Actor의 단일 책임 원칙을 벗어납니다. (원칙을 벗어난다 라고 말하기보다는 클래스가 쓸데없이 복잡해진다. 라고 이해하셔도 됩니다)

이럴 때 , interface 를 사용하여 필요한 기능을 다중상속 합니다.

유저의 입력을 처리할 interface 를 정의 합니다.
인터페이스는 구현이 없으므로 이를 상속받으면 내용을 구현해야 합니다.

```
1 reference
public interface Controllable{
    1 reference
    public void OnUserInput(float x,float y);
}
```

```
2 references
public class Player : Actor , Controllable {
    2 references
    public override void Attack(Actor target)
    {
        target.Hit(_atk);
        Console.WriteLine("Player overrided Attack");
    }

    1 reference
    public void OnUserInput(float x,float y)
    {
        Console.WriteLine($"Player Move Toward Vector ({x},{y})");
    }
}
```

위 내용까지 구현했다면, 클래스 참조 구문에서 아래와 같이 타입캐스팅을 통한 접근과 처리가 가능합니다.
플레이어와 몬스터와 보스를 모두 Actor 타입으로서 관리 할 수 있게 됩니다.

```
Player player = new Player();
Monster monster0 = new Monster();
Monster monster1 = new Monster();
Monster monster2 = new Monster();
Monster monster3 = new Monster();
Boss boss = new Boss();

List<Actor> allActors = new List<Actor>();
allActors.Add(player);
allActors.Add(monster0);
allActors.Add(monster1);
allActors.Add(monster2);
allActors.Add(monster3);
allActors.Add(boss);
```

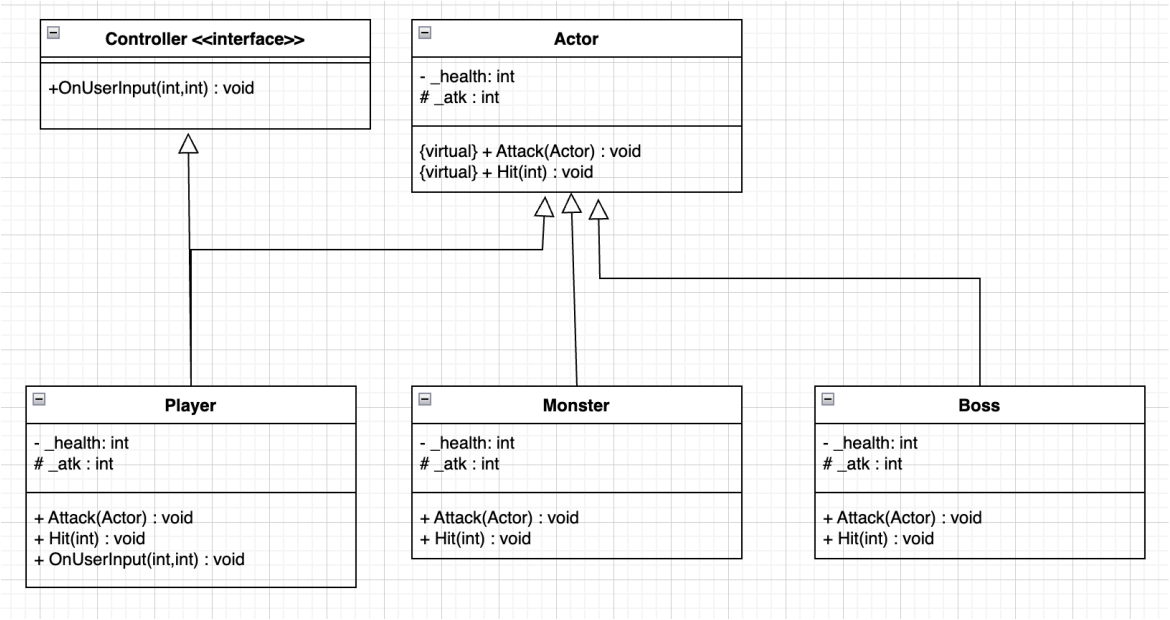
```
List<Actor> allActors = new List<Actor>();
allActors.Add(player);
allActors.Add(monster0);
allActors.Add(monster1);
allActors.Add(monster2);
allActors.Add(monster3);
allActors.Add(boss);

Controllable controllerFound = null;
Player playerFound;

for(int i = 0; i < allActors.Count ; i++)
{
    if(allActors[i] is Controllable)
    {
        controllerFound = allActors[i] as Controllable;

        break;
    }
}

controllerFound.OnUserInput(1,2);
playerFound = controllerFound as Player;
playerFound.OnUserInput(5,6);
```

클래스가 상속한 타입에 따라 자유롭게 캐스팅하거나 타입 비교를 할 수 있습니다.
(물론 실제로 구현 하지 않았는데 캐스팅 하면 오류가 발생 하겠죠)

여기서 is 키워드는 인스턴스가 비교하려는 타입이라면 bool 타입을 반환하고
As 키워드는 동작이 가능하다면 (이 경우 명시적으로 상속이 되었다면)
해당 타입으로 인스턴스를 타입 변환 해주고, 불가능하다면 nullable 값을 반환합니다.

값 타입 수업시간에 char <-> int 변환을 예로 들며 사용했던 괄호 캐스팅은
캐스팅이 불가능할 경우에 Exception 즉, 에러를 반환하지만 as 키워드는 nullable 을 반환한다는 차이가 있습니다.

위에서 구현한 Actor, Player, Monster, Boss, Controller 를 UML 에 표현하면
아래와 같은 모습이 됩니다.



각 클래스를 연결하는 표현을 '관계' 라고 합니다.
이 관계에는 여러가지 표현이 있으나, 여기서는 점선과 실선으로 표현된 상속과 구현 (추상 클래스의) 만 소개합니다.