3주 복습

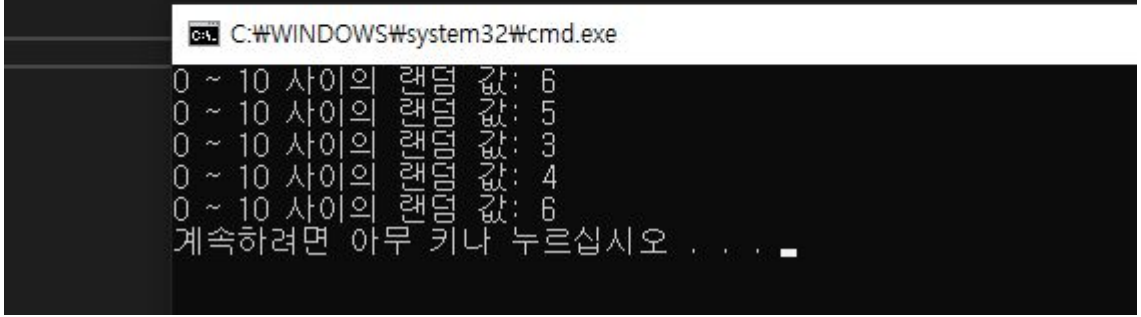
1. 통장 클래스를 만든다.
2. 해당 통장 클래스에는 잔고(int)를 가지고 있다.
3. 입금 / 출금 메소드가 있다.
4. 통장 클래스를 가지고 2개의 인스턴스를 만든다. (A통장 / B통장)
5. 수중에 100만원을 들고있는 것으로 시작한다.(int)
6. A통장에서 선택지를 띄운다 (1. 입금 / 2. 출금 /3. 이체)
7. 이체를 선택하고 금액을 입력하면 A통장의 잔액과 B통장의 잔액이 출력된다.
8. 입금을 선택하고 금액을 입력하면 수중의 돈이 줄고 A통장은 추가된다. A통장의 잔액이 출력된다.
9. 출금을 선택하면 금액 입력 -> 수중 돈 + , A 통장 -. -> A통장 잔액 출력.

1. <하나 은행/ 기업 은행/국민 은행> 클래스를 각각 제작한다.
2. 제작된 클래스들은 모두 통장 클래스를 상속 받는다.
3. while문으로 계속 진행된다. 특정 키를 입력하기 전까지.
4. 은행을 하나 선택한다.
5. 선택된 은행은 입금/출금/이체를 선택받을 수 있다.
6. 입금 , 출금 , 은 모두 같음.
7. 이체는 특정 은행을 선택해야 이체가 가능하다.

추가 내용

랜덤, enum, 프로퍼티

```
참조 0개
static void Main(string[] args)
{
    Random rand = new Random();

    for (int loop = 0; loop < 5; loop++)
    {
        Console.WriteLine("0 ~ 10 사이의 랜덤 값: " + rand.Next(0, 10));
    }
}
```



열거형(enum)

```
참조 5개
public enum thePlayerState
{
    idle,
    walk,
    run,
}
```

보통 상태를 정의할때 많이 사용한다.
서있기 , 걷기 , 뛰기 라던가
정보를 보내는 중, 받는 중, 등등으로 상태를 정의할때 많이 사용한다.

스위치문과 많이 사용한다.

```
참조 0개
static void Main(string[] args)
{
    thePlayerState theState = thePlayerState.idle;

    switch(theState)
    {
        case thePlayerState.idle:
            System.Console.WriteLine("idle");
            break;
        case thePlayerState.walk:
            System.Console.WriteLine("walk");
            break;
        case thePlayerState.run:
            System.Console.WriteLine("run");
            break;
    }
}
```

프로퍼티

```
참조 0개
public string Name { get; private set; }
```

private로 외부에서 데이터를
못가져가게 막고싶음.

하지만 데이터를 받는건 하고싶을때,

프로퍼티를 사용함.

```
class PlayerName
{
    private string userName;
    참조 0개
    public string UserName
    {
        get
        {
            return userName;
        }
        private set
        {
            userName = value;
        }
    }
}
```

get 을 사용해서 데이터를 가져올때 사용하고,
set 을 사용해서 데이터를 설정할때 사용한다.

즉, OriginValue를 가져올땐 originValue를 반환하고,
값을 수정할땐 originValue는 수정된 값 을 넣는다.

이때, set앞에 private가 붙어있어서 set은 외부에서 할 수없다.

```
참조 2개
class PlayerName
{
    private string userName;
    참조 2개
    public string UserName
    {
        get
        {
            return userName;
        }
        private set
        {
            userName = value;
        }
    }

    참조 0개
    public void SettingName()
    {
        userName = "Umin";  //Set
        System.Console.WriteLine(userName); //Get
    }
}
```

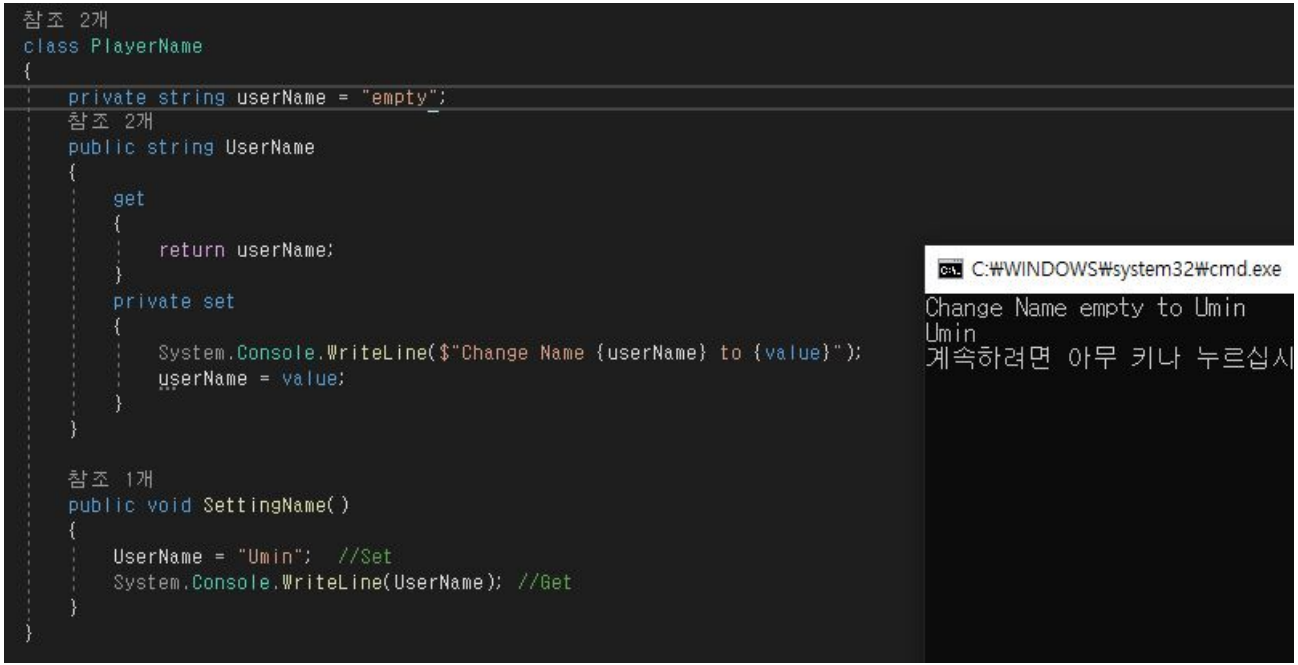
프로퍼티 예시

settingName() 에서
값을 설정하는 것은 set 이고,
값을 가져와 띄우는 내용은 get이다.

```
참조 0개
class OutsideClass
{
    참조 0개
    public void SettingName()
    {
        PlayerName pp = new PlayerName();
        pp.UserName = "Umin";  //Set
        System.Console.WriteLine(pp.UserName); //Get
    }
}
```

프로퍼티 예시

외부에서 쓰는건 set은 안되고
get은 가능하다



프로퍼티 응용예시

set에서 사용될때 수정될 값은 value라는 키워드에 담겨있다.
set 할때마다 값이 수정될때 마다 무언가를 할 수 있다.

8장

인터페이스, virtual, abstract

인터페이스?

예를 들어

내가 어둠 속성으로 공격한다고 가정하자.

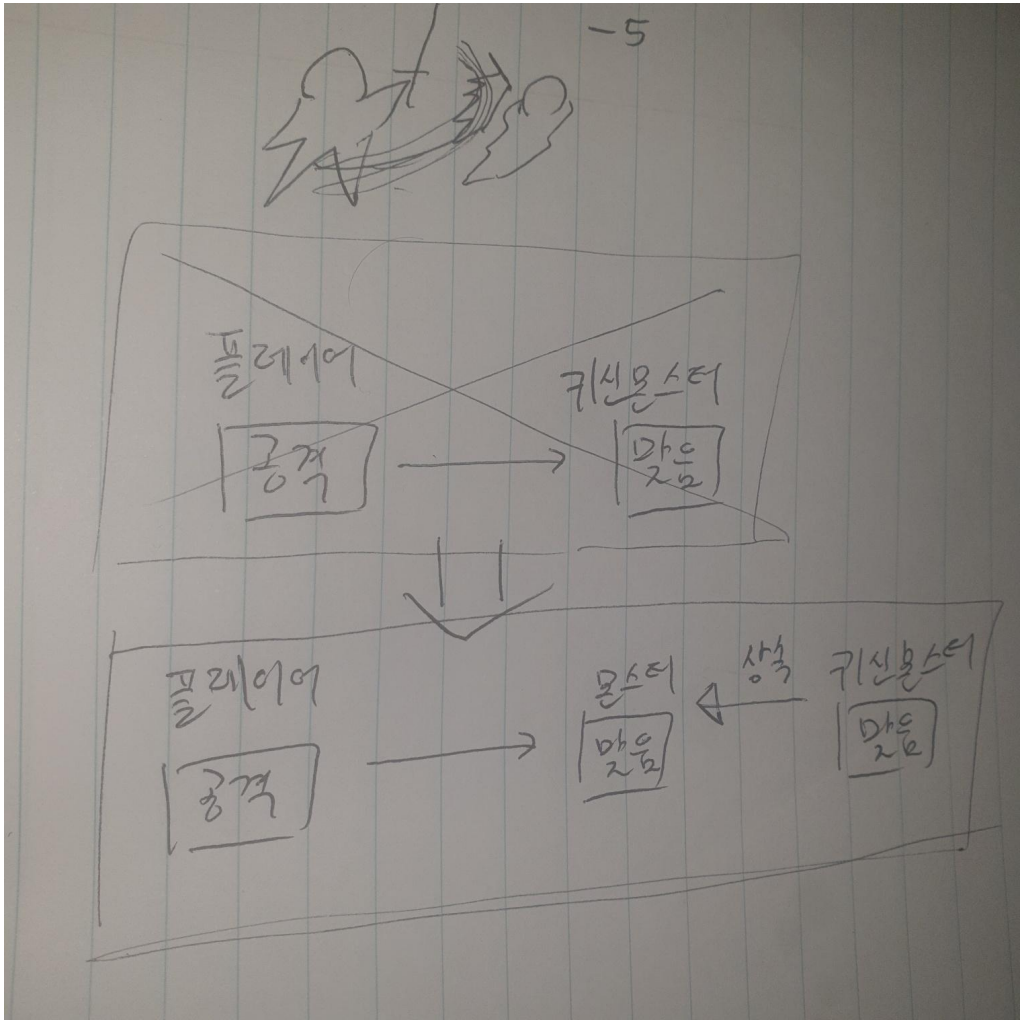
어둠속성의 공격이 어둠 몬스터에겐 1.5배 대미지이고, 빛 몬스터에겐 0.5배 데미지라고 하자.

그럼 내가 공격을 했고,
몬스터가 맞았을때
어떤 몬스터인지 내가 알아야할까?

그렇지않다.

나는 공격을 할 뿐

맞은 대상이 어둠이면 1.5배 맞는거고,
빛이면 0.5배 맞는 것 뿐.



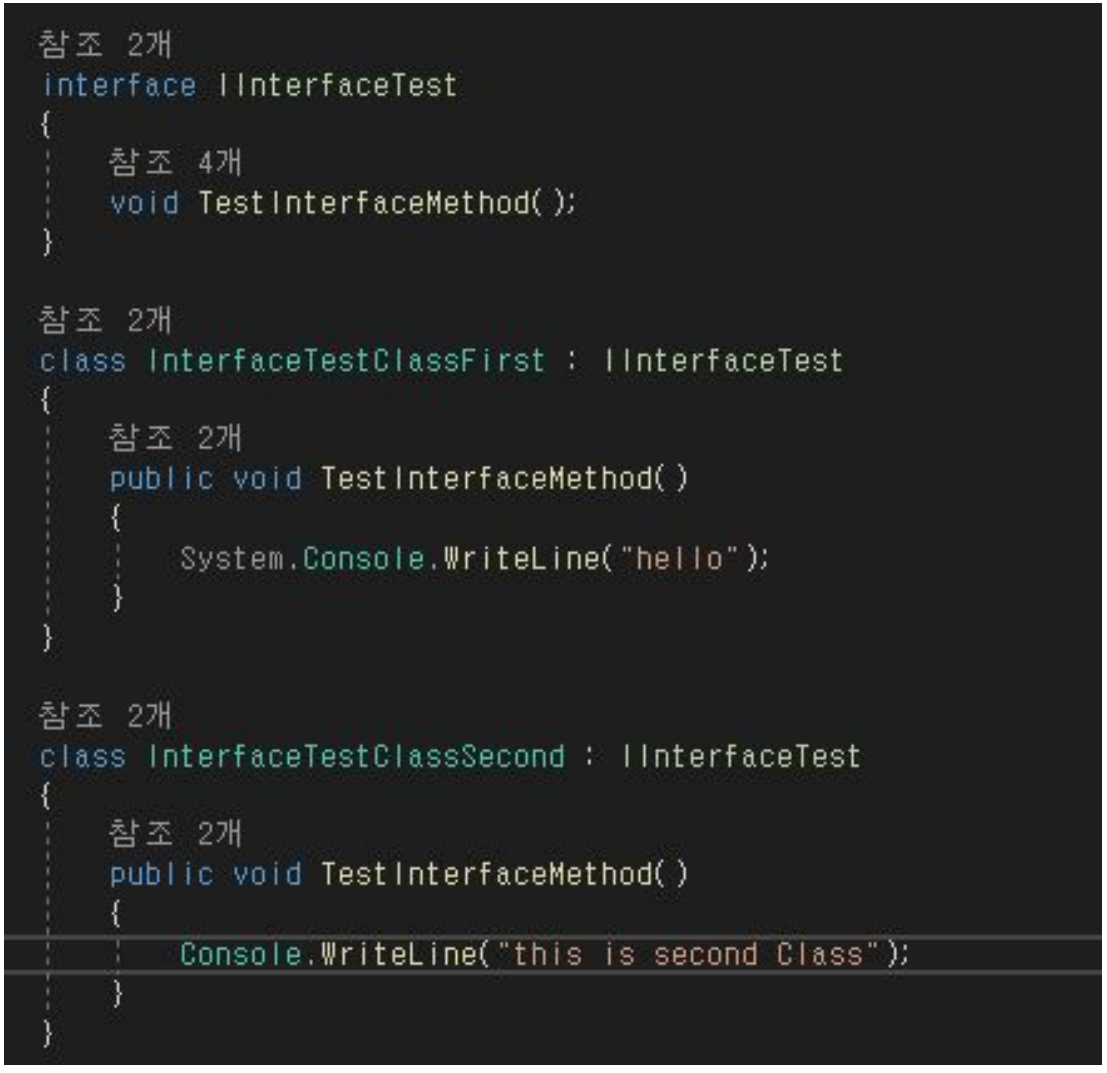
인터페이스?

플레이어 입장에선 공격 할 뿐.
맞는 대상이 귀신인지, 버섯인지 알필요 없다.

맞은 대상이 몬스터인지만 확인하고
공격하면 된다.

예로 들었던 어둠인 경우 1.5배,
빛인 경우 0.5배는 몬스터가 스스로
계산하면된다.

이처럼 불필요한 정보 공유를 막고자
중간다리를 하는 것을 인터페이스라고 한다.



Interface

클래스 상속과는 다르다.

인터페이스에서 써놓은 메소드는 반드시 비어있어야 한다.

상속받은 클래스는 해당 메소드를 반드시 정의해야 한다.

Interface

정의한 메소드 임으로 사용할 수 있다.

```
참조 2개
interface Monster
{
    참조 2개
    void HitFromPlayer(int damage);
}

참조 2개
class DarkMonster : Monster
{
    int hp = 100;

    참조 2개
    public void HitFromPlayer(int damage)
    {
        hp = hp - damage;

        System.Console.WriteLine($"Monster : Im Hit! my Hp is {hp}");
    }
}
```

Interface 예시

Monster 라는 Interface 제작.

DarkMonster는 Monster라는 Interface를 상속받는 Class

DarkMonster는 체력이 100이다.

인터페이스에서 HitFromPlayer라는 메소드를 정의했기 때문에 DarkMonster는 이 메소드를 꼭 정의해야 한다.

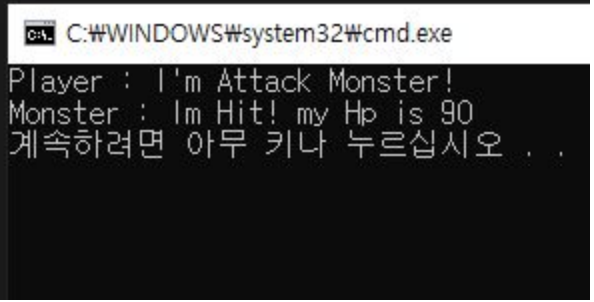
```
참조 2개
class Player
{
    int AttackDamage = 10;

    참조 1개
    public void AttackMonster(Monster monster)
    {
        System.Console.WriteLine("Player : I'm Attack Monster!");
        monster.HitFromPlayer(AttackDamage);
    }
}
```

Interface 예시

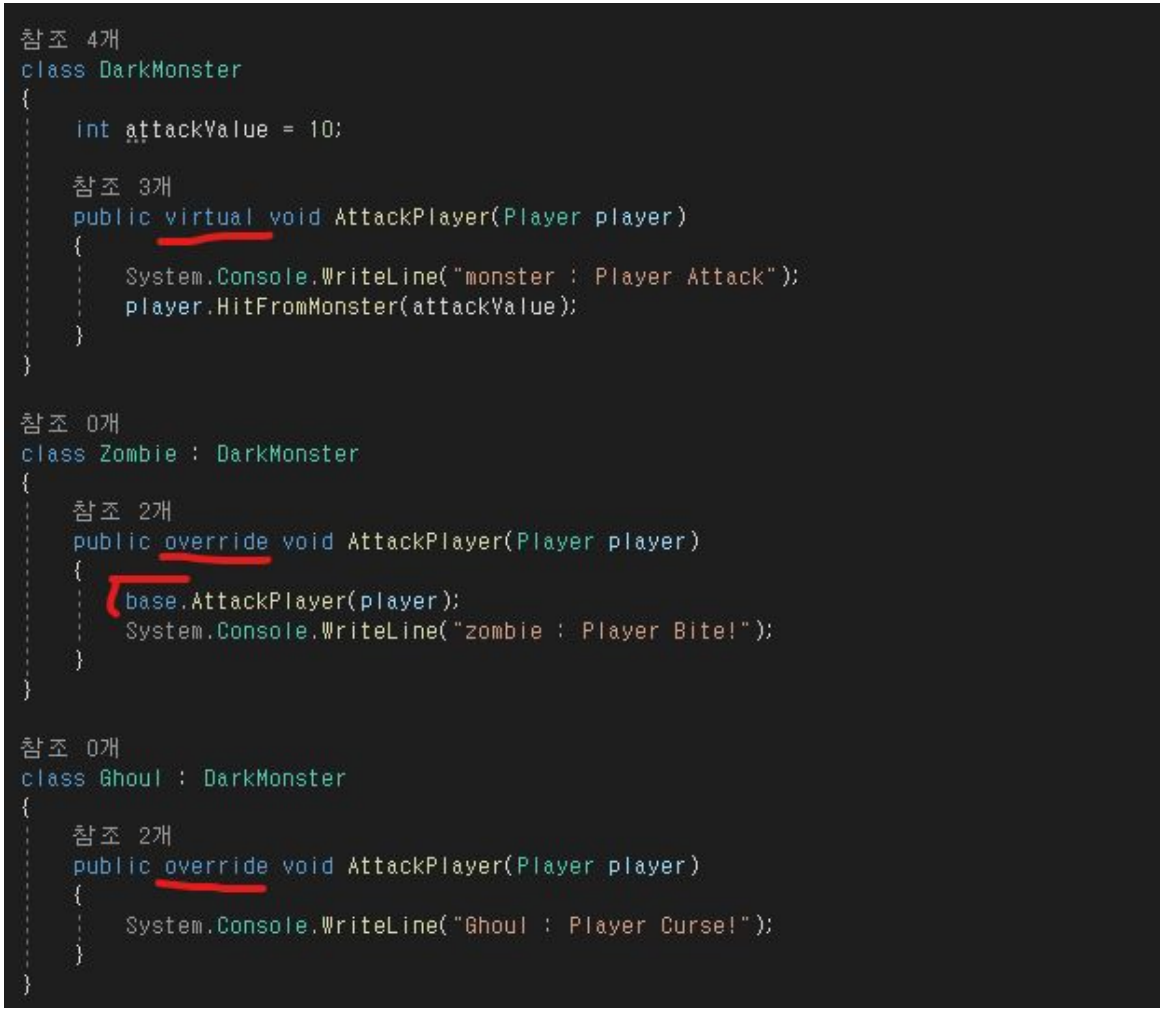
Player 클래스 제작.
해당 클래스는 AttackMonster라는 메소드를 가지고 있는데,

이 메소드는 공격 받는 대상이 DarkMonster이든 아니든 상관없이 Monster이면 공격한다.
(Monster 클래스는 HitFromPlayer를 가지고 있기 때문에 사용 할 수 있다)

```
참조 0개
internal class Program
{
    참조 0개
    static void Main(string[] args)
    {
        Player player = new Player();
        DarkMonster zombie = new DarkMonster();

        player.AttackMonster(zombie);
    }
}
```



결과



```
참조 4개
class DarkMonster
{
    int attackValue = 10;

    참조 3개
    public virtual void AttackPlayer(Player player)
    {
        System.Console.WriteLine("monster : Player Attack");
        player.HitFromMonster(attackValue);
    }
}

참조 0개
class Zombie : DarkMonster
{
    참조 2개
    public override void AttackPlayer(Player player)
    {
        base.AttackPlayer(player);
        System.Console.WriteLine("zombie : Player Bite!");
    }
}

참조 0개
class Ghoul : DarkMonster
{
    참조 2개
    public override void AttackPlayer(Player player)
    {
        System.Console.WriteLine("Ghoul : Player Curse!");
    }
}
```

Virtual

virtual은 부모가 정의한 메소드를 자식이 다시정의할 수 있게 열어두는 키워드 이다.

부모의 메소드에 virtual이 있으면 자식은 그 메소드를 다시 정의할 수있다.

다시 정의할 때는 override를 사용하고, 내용에 base. 를 이용해서 기존 메소드를 사용한다는 내용을 정의할 수 있다.

```
참조 5개
class Player
{
    int HP = 100;

    참조 1개
    public void HitFromMonster(int number)
    {
        System.Console.WriteLine($"player : Hit From Monster {HP - number}");
    }
}
```

Virtual

플레이어 클래스 내용

```
참조 0개
static void Main(string[] args)
{
    Player player = new Player();

    DarkMonster monster = new DarkMonster();
    Zombie zombie = new Zombie();
    Ghoul ghoul = new Ghoul();

    monster.AttackPlayer(player);
    System.Console.WriteLine("");

    zombie.AttackPlayer(player);
    System.Console.WriteLine("");

    ghoul.AttackPlayer(player);
}
```

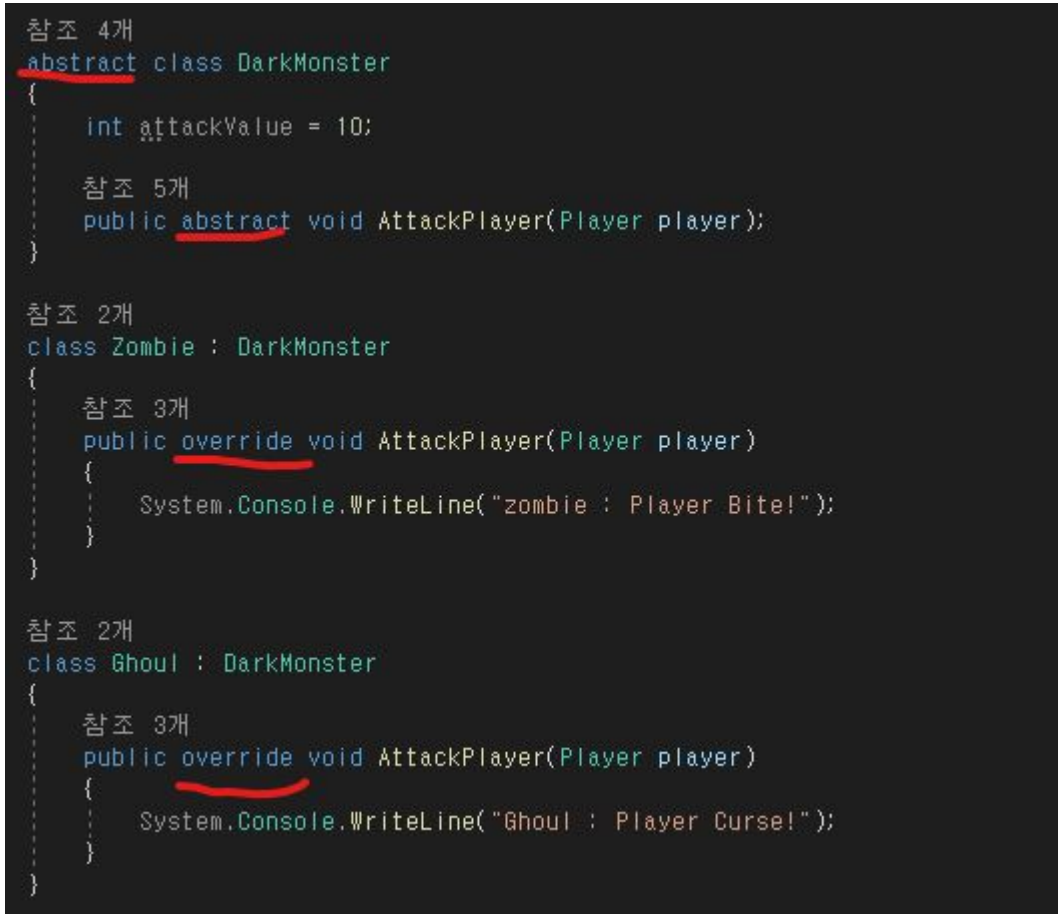
결과

```
C:₩WINDOWS₩system32₩cmd.exe
monster : Player Attack
player : Hit From Monster 90

monster : Player Attack
player : Hit From Monster 90
zombie : Player Bite!

Ghoul : Player Curse!
계속하려면 아무 키나 누르십시오
```

몬스터는 기존 내용대로 공격함.

zombie는 원래 monster 사용하고, 다음 추가 정의했음.

ghoul는 새로운 내용을 정의했음.



Abstract

abstract는 virtual과 굉장히 비슷하다.

단, abstract는 이런 형태의 클래스를 만들어야 해 라는 느낌이 강하다.

abstract은 클래스 앞에도 붙여야하고, 메소드도 그 내용을 정의할 수 없다.

하지만 상속받으면 override를 쓰는것은 똑같다.

```
참조 5개
class Player
{
    int HP = 100;

    참조 1개
    public void HitFromMonster(int number)
    {
        System.Console.WriteLine($"player : Hit From Monster {HP - number}");
    }
}
```

Abstract

플레이어 클래스 내용

```
참조 0개
static void Main(string[] args)
{
    Player player = new Player();

    DarkMonster monster = new DarkMonster();
    Zombie zombie = new Zombie();
    Ghoul ghoul = new Ghoul();

    monster.AttackPlayer(player);
    System.Console.WriteLine("");

    zombie.AttackPlayer(player);
    System.Console.WriteLine("");

    ghoul.AttackPlayer(player);
}
```

결과

Abstract으로 정의된 클래스는
생성할 수 없기 때문에
에러를 띄운다.

1. 생명체 인터페이스 생성.
2. 생명체에는 공격하기, 공격 받기, 회피하기, 피해받기 가 있음.
3. 생명체는 플레이어와 몬스터 2개가 상속받는다.
4. 플레이어와 몬스터가 둘중 하나가 죽을때 까지 계속 공격하고 공격 받고를 주고받는다.
5. 공격하면 공격한다고 출력하고
6. 공격 받는 대상은 확률(랜덤)로 회피 또는 피해받기를 실행한다.
7. 플레이어와 몬스터 둘다 이렇게 진행된다.

위와 같은 내용을 인터페이스, abstract, virtual 총 3개를 각각 제작해본다.

추가** 몬스터의 종류를 늘려본다.

8장

콜랙션
List, Dictionary

```
List<int> intList = new List<int>() { 1, 2, 3, 4 };

intList.Add(5);
intList.Add(6);

for (int index = 0; index < intList.Count; index++)
{
    Console.WriteLine(intList[index]);
}
```

```
C:₩WINDOW
1
2
3
4
5
6
계속하려면 0
```

List

배열과 같은 방식으로 사용됨.
단, 최대 크기를 정하지
않아도 되고, 계속 추가할 수
있다.

대신, 배열보단 무겁다.

```
intList.Remove(2);

for (int index = 0; index < intList.Count; index++)
{
    Console.WriteLine(intList[index]);
}
```

```
1
3
4
5
6
```

Add로 추가

Remove로 번째 녀석 삭제

List

```
intList.Insert(3, 9);

for (int index = 0; index < intList.Count; index++)
{
    Console.WriteLine(intList[index]);
}
```

```
1
3
4
9
5
6
```

중간에 추가하기

```
intList.Clear();
for (int index = 0; index < intList.Count; index++)
{
    Console.WriteLine(intList[index]);
}
```

Clear로 모두 제거

1. int 리스트 5개 만들기
2. string 리스트 5개 만들기
3. float 리스트 5개 만들기

4. 몬스터 클래스 제작.
5. 몬스터 클래스를 활용한 몬스터 5종류 제작(new 로 5개 만들기)
6. 몬스터 클래스엔 IntroduceMonster() 메소드를 가지고 몬스터 소개를 출력함.
7. List<Monster>로 모두 담고 소개하기. (for문)

8. City 인터페이스 제작.
9. 해당 인터페이스를 상속받는 5개의 도시 클래스 제작.
10. 인터페이스는 자기 소개를 하는 메소드를 가지고 있음.
11. List<City> 로 모두 담고 소개하기.

Dictionary

key,value 를 한 쌍으로 가지고 있는 값들.

Add로 추가해도 되고,
그냥 바로 Key에 값 넣어도 된다.

1. int , string 을 쌍으로 하는 딕셔너리 5개 제작
2. string ,int 를 쌍으로 하는 딕셔너리 5개 제작
3. int , int 를 쌍으로 하는 딕셔너리 제작.

4. monster 클래스를 제작.
5. 몬스터 클래스는 공격력을 가지고 있음. (int)
6. string , monster 딕셔너리를 제작한다.
7. string (key)에는 각 몬스터의 이름을 제작해 준다.
8. monster를 5개 제작하고(instance) 각각 다른 공격력을 부여해준다.
9. 각각 딕셔너리에 넣어준다.
10. monster 이름에 해당하는 몬스터의 공격력을 출력해 본다.

1. 기획한 내용 발표 및 만들기 시작