C# 2차 과제 2171027 안유경

1. 객체지향 프로그래밍이란?
• 절차지향 프로그래밍: 함수의 실행 순서가 매우 중요한 프로그래밍 방식
- 장점: 간단한 코드에서는 직관적
- 단점: 프로그램이 커질수록 유지보수가 힘들고 순서에 종속적
- ex: C언어
• 객체지향 프로그래밍: 프로그램을 '객체'라는 기본 단위로 나누어 이들의 상호작용을 서술하는 프로그래밍 방식
- 모든 엔티티들을 객체 단위로 생각 (캐릭터, 무기, 맵을 이루는 요소, 보이지 않는 규칙이나 개념등을 모두 포함

2. 객체지향의 4대 특징
• 상속
- 대상을 객체로 추상화 및 구현할 때, 기존에 구현한 클래스를 재활용하여 구현
- 기존에 구현되어 있던 클래스 = 상위 클래스, 부모 클래스
- 기존 클래스를 재활용하여 구현된 클래스 = 하위 클래스, 자식 클래스
• 다형성
- 어떤 객체의 속성이나 기능이 상황에 따라 여러 형태로 변할 수 있음
- 상속, 구현 상황에서의 메서드 오버라이딩/오버로딩
• 추상화
- 어떤 대상을 구현할 때, 그 대상의 본질적 특징을 정의하고, 이를 기반으로 대상을 객체로 구현하는 것
- abstract class와 interface로 구현됨
• 캡슐화
- 클래스 내의 연관된 속성이나 함수를 하나의 캡슐로 묶어 외부로부터 클래스로의 접근을 최소화하는 것
- public, static, private, protected와 같은 접근제한자를 통해 구현

3. 클래스와 객체, 인스턴스
• 클래스: 객체에 속성과 기능을 넣어줄 설계도
• 객체: 소프트웨어 세계에 구현할 대상이자 속성 및 기능을 가지는 프로그램 단위
• 인스턴스: 클래스에 따라 메모리상에 구현된 실체

4. 클래스 만들기
- 클래스를 만들 때는 클래스를 이루는 필드(=속성), 메소드(=기능)을 함께 구현해주어야 한다.

```
class Knight
{
    public int hp;
    public int attack;

    public void Move()
    {
        Console.WriteLine("Knight Move");
    }

    public void Attact()
    {
        Console.WriteLine("Knight Attact");
    }
}
```

```
class Program
{
    static void Main(string[] args)
    {
        // knight 객체 생성
        Knight knight = new Knight();

        // knight 객체의 필드에 접근
        knight.hp = 100;
        knight.attack = 10;

        // knight 객체의 메서드에 접근
        knight.Move();
        knight.Attack();
    }
}
```

- 클래스명 객체명 = new 클래스명(매개변수) 와 같은 문법으로 클래스 생성
- 객체명.필드명 = (값) 와 같은 문법으로 필드 생성
- 객체명.메소드(매개변수) 와 같은 문법으로 메소드 실행

• ref(복사값)과 copy(참조)

- 기본적으로 클래스는 값을 넘길 때 '참조값'을 넘기고, 구조체는 '복사값'으로 넘김
- 구조체 값을 참조값으로 쓰기 위해서는 ref나 out을 붙여서 넘겨줘야 함

• Deep copy

- Clone 함수는 새로운 Knight 객체인 knight2를 새로 선언하고, knight2의 각 요소에 자기 자신, 즉 knight의 값을 넣는 기능을 한다.

```
// 클래스는 ref, 즉 참조
class Knight
{
    public int hp;
    public int attack;

    public Knight Clone()
    {
        // 자기자신에게 접근할 때는 변수명만 적기
        // 자기자신에게 접근할 때 knight를 더이상 사용하지 않으므로
        // 아래 knight2는 모두 knight로 대체해서 사용가능함
        // 대체한 knight와 원래의 knight간에 연관은 없음!

        Knight knight2 = new Knight();
        knight2.hp = hp; // 본인의 hp값을 여기에 넣어주세요
        knight2.attack = attack; // 본인의 attack값을 여기에 넣어주세요
        return knight2;
    }
}
```

```
class Program
{
    static void KillKnight(Knight knight)
    {
        knight.hp = 0;
    }

    static void Main(string[] args)
    {
        Knight knight = new Knight();
        knight.hp = 100;
        knight.attack = 10;

        Knight knight2 = knight.Clone();
        KillKnight(knight2);
    }
}
```

=> 실행해보면 knight2는 KillKnight함수로 인해 hp값이 0이 되었지만, knight의 hp값은 그대로 100인 것을 확인할 수 있다. 별개의 knight와 knight2 객체가 생성된 것이다.

5. 생성자와 this

- 클래스의 새로운 객체를 선언하는 동시에 객체의 요소들에 정해둔 값을 부여해주는 기능

• 생성자는 무조건 클래스 이름과 같게 설정, 생성자 이름 앞에는 어떤 타입도 입력하면 안된다.

```
class Knight
{
    public int hp;
    public int attack;

    // 생성자 버전1
    public Knight()
    {
        hp = 100;
        attack = 10;
    }
```

```
    // 생성자 버전2
    public Knight(int hp)
    {
        this.hp = hp;
        attack = 10;
    }
```

- this.hp는 매개변수로 받은 hp가 아니라 생성자 자신의 hp라는 것을 명시적으로 나타내고 있음

• sealed(): 상속 불가능한 클래스를 지정해주는 한정자

• base() : 기반클래스/ 부모클래스의 생성자

- public Derived(string Name) : base(Name)

- 기반 클래스의 생성자를 먼저 실행하고, 파생 클래스의 생성자로 덮어씌우는 과정

6. 오버라이딩
• 오버라이딩: 상위 클래스가 가지고 있는 메서드를 하위 클래스가 재정의해서 사용
- virtual: 부모 클래스의 메소드에 virtual을 붙여준다.
- override: 자식 클래스의 메소드에 override를 붙여준다

7. 접근한정자

접근 제한자	설명
public	모든 외부(파생클래스 포함)에서 이 타입(Type: 클래스, 구조체, 인터페이스, 델리게이트 등)을 엑세스할 수 있다. (개별 타입 멤버의 엑세스 권한은 해당 멤버의 접근 제한자에 따라 별도로 제한될 수 있다)
internal	동일한 Assembly 내에 있는 다른 타입들이 엑세스 할 수 있다. 하지만, 다른 어셈블리에서는 접근이 불가하다.
protected	파생클래스에서 이 클래스 멤버를 엑세스할 수 있다.
private	동일 클래스/구조체 내의 멤버만 접근 가능하다.

💡 접근한정자를 따로 지정하지 않으면 private가 default로 지정된다.

---

is / as 키워드:
• 'is': 주어진 객체가 특정 클래스/인터페이스의 인스턴스인지를 확인
- "if (obj is String)"은 obj가 String 인스턴스인지 확인
• 'as': 객체를 특정 타입으로 안전하게 변환
- 변환할 수 없는 경우 null을 반환
- "String s = obj as String"은 obj를 String으로 변환

abstract 클래스:
- abstract 클래스는 인스턴스화할 수 없는 클래스
- 즉, 객체를 직접 생성할 수 없으며, 상속을 통해 자식 클래스에서 구현하도록 하는 용도로 사용

인터페이스:
- 인터페이스는 메서드, 프로퍼티, 이벤트 등의 '시그니처'만을 정의하고, 실제 구현은 인터페이스를 구현하는 클래스나 구조체에서 이루어짐

델리게이트, 액션/이벤트:
- 델리게이트는 메서드에 대한 참조를 저장하는 타입
- 이를 통해 메서드를 매개변수로 전달하거나, 이벤트와 콜백을 구현할 수 있음
- 액션은 반환 값이 없는 메서드를 참조하는 델리게이트.
- 이벤트는 특정 조건이나 상황이 발생했을 때 알림을 전달하는 메커니즘으로 델리게이트를 기반

제네릭:
- 제네릭은 타입을 매개변수로 가지는 클래스나 메서드를 정의할 수 있는 기능으로, 이를 통해 타입 안전성을 유지하면서 코드 재사용성을 높일 수 있음