OOP With C#

객체 지향 프로그래밍

객체

어떤 대상이 가진 속성이나 기능에 의해 의미를 가지는 단위.
이는 ‘상태’ 와 ‘행위’ 로 이루어졌다고 볼 수 있다.

사람, 자동차, 의자, 건물 등 시각적인 단위로 객체를 정의 해 볼 수 있기도 하고,
탈것, 먹는것, 입는것 등 기능적인 단위로도 객체를 정의 할 수 있다.

실재하는 대상을 객체로 다룰 때, 그 관점에 따라 의미가 달라짐을 유의하자.

“사람” 이라는 대상을 예로 들면
- 학교에서는 사람을 학생, 교수, 직원 등의 의미로 분류할 수 있고
- 생물 그룹에서는 소, 말, 돼지, 사람 과 같은 동물의 하나로서 분류된다

객체 지향 프로그래밍?

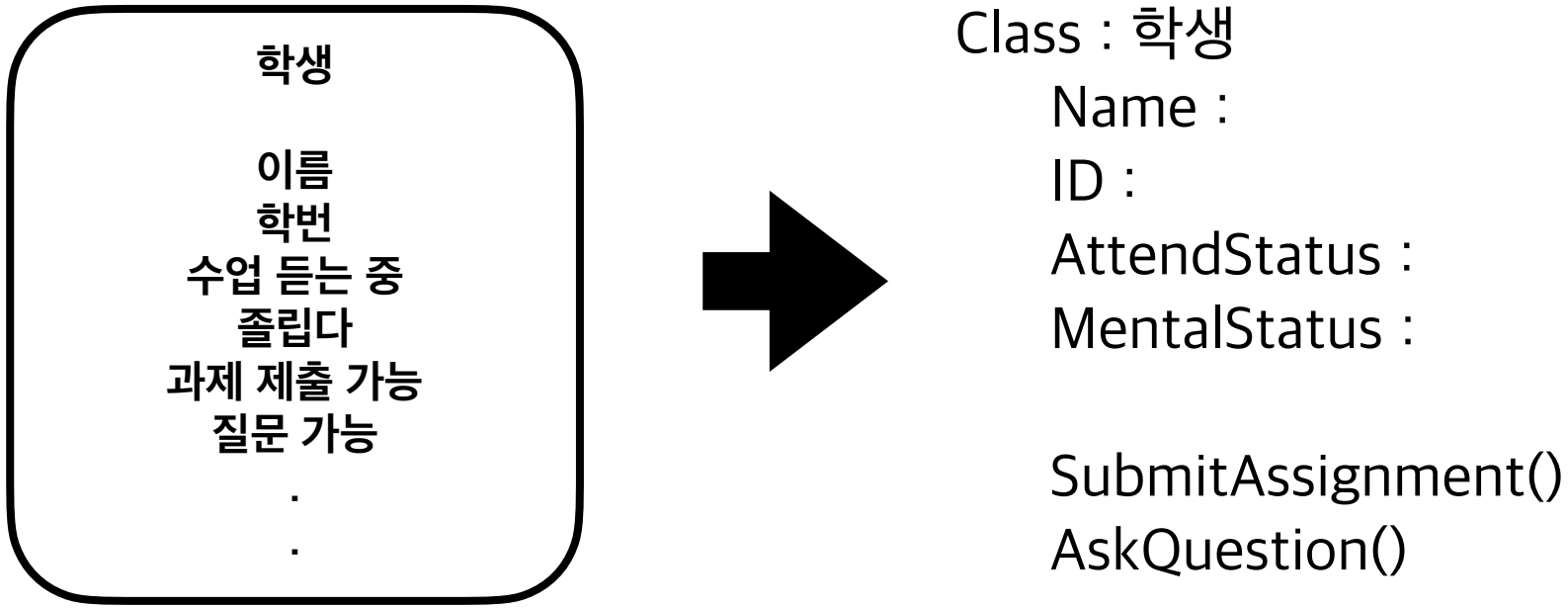
프로그램을 데이터와 처리 방법을 절차적 나열로서 표현하는 방법에서 벗어나, 객체를 정의하여 각각의 객체가 그에 적합한 데이터를 처리하고, 다른 객체와의 연관을 통해 프로그램의 결과를 도출하도록 설계하는 것.

왜?

- 각 객체의 개별적인 수정이 가능하므로 보수와 변경이 용이
- 인간이 인지하기 쉬운 단위로 객체를 설계하기 때문에, 직관적인 코드 작성 가능
- 단위별로 작성된 객체들의 코드를 재사용하기 용이
- 형식과 형식에 대한 동작(함수)를 직관적으로 연결해둘 수 있다

객체화

- 다루고자 하는 대상의 행위, 상태, 정보를 정의한다

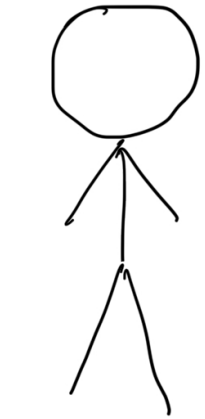


- 이 때, 해당 객체에 속한 함수를 Method , 변수를 Member 라 한다.

* Method 와 Member function은 모두 Class 정의에 포함된 함수를 지칭하는 말입니다

<게임 프로그램에서의 객체>

1. 데이터와 상태의 관점



player

Monster

Item

Class Player :

Health
Atk
Speed
Exp
.

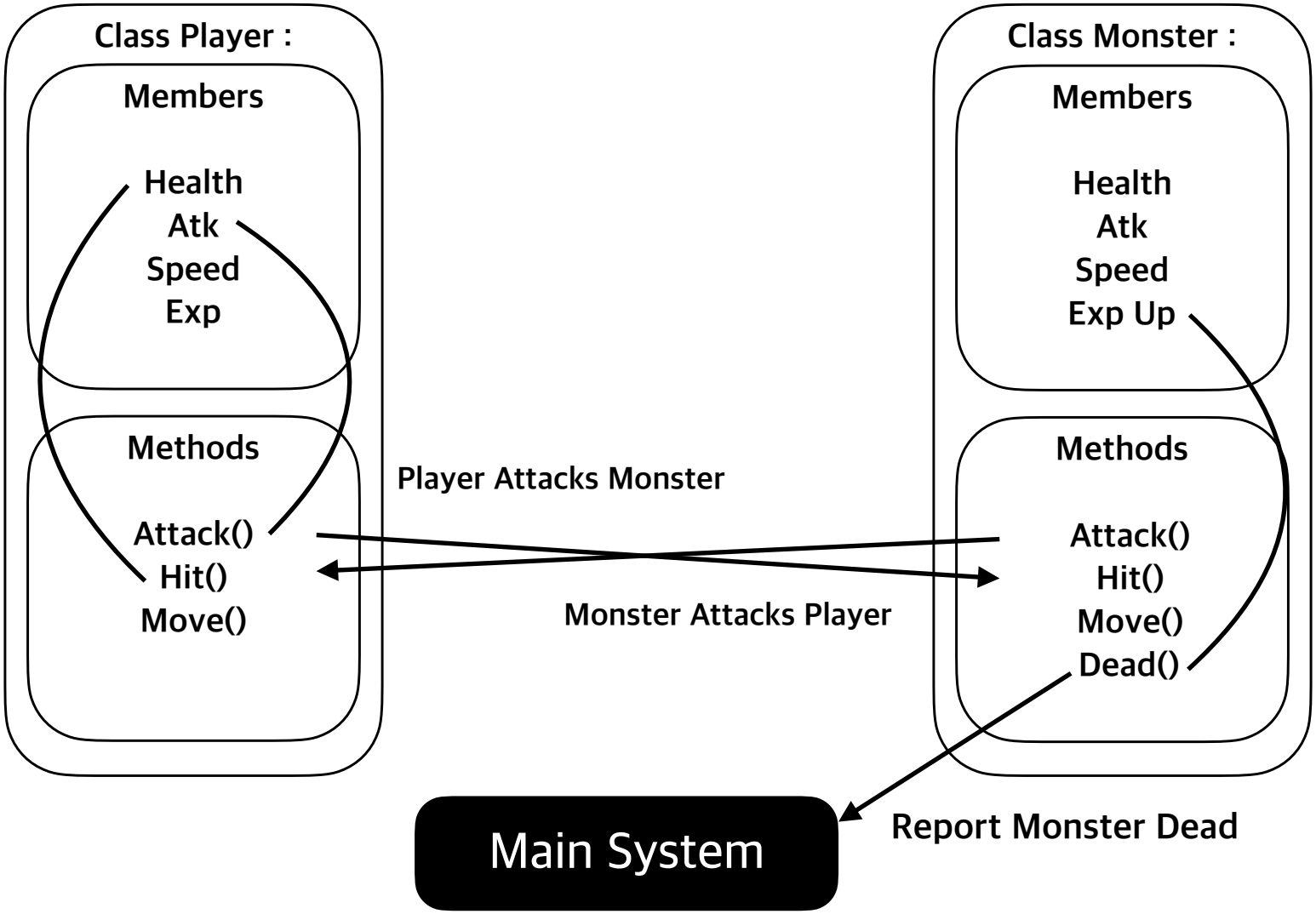
Class Monster :

Health
Atk
Exp Up
.
.

Class Item :

Atk Up
Health Up
.
.

2. 행위와 관계 관점


Class Player :
Members
Health
Atk
Speed
Exp
Methods
Attack()
Hit()
Move()
Player Attacks Monster
Monster Attacks Player
Class Monster :
Members
Health
Atk
Speed
Exp Up
Methods
Attack()
Hit()
Move()
Dead()
Main System
Report Monster Dead

객체지향 프로그래밍의 특징

1. 캡슐화 (Encapsulation)

- 객체 고유의 독립적인 동작 설계를 위해 해당 객체 외부에서 접근이 가능한 부분과 그렇지 않은 부분을 정의하는 것

- 필드를 클래스 외부로부터 숨길지, 그렇지 않을지는 언어에서 지원하는 접근제한자를 통해 이루어진다.

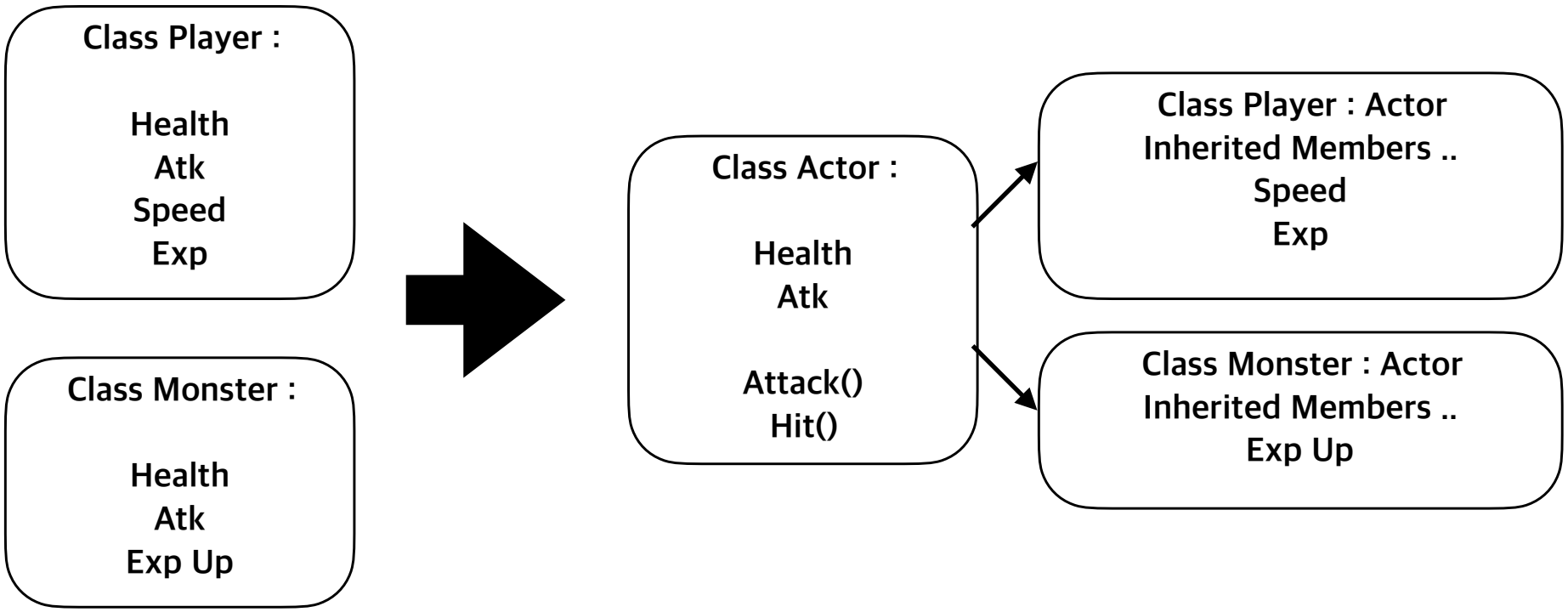
접근제한자	클래스 내부	상속된 클래스 내부	클래스 외부
private	O	X	X
protected	O	O	X
public	O	O	O

2. 추상화 (abstraction)

- 다루려 하는 대상이 가진 본질 및 공통적인 특성, 기능을 추출하여 정의하는 것

- 추상 클래스는 설계의 기초만을 제공하며 인스턴스화 되지 않는다
때문에 해당 클래스를 상속받아서만 구현 (implementation) 이 가능하다.

- C# 에서는 abstract 클래스와 interface 로 작성된다



3. **상속 (Inheritance)**
   - 먼저 작성된 어떤 클래스의 내용을 그대로 이어받으면서 , 추가로 필요한 동작이나 속성을 정의하는 것

   - 중복된 코드를 줄일 수 있고, 타입캐스팅을 통해 다양한 클래스에 대해 간편한 참조를 만들 수 있다.

   - C# 은 기본적으로 단일 상속이지만, 인터페이스 등의 기능을 통해 다중 상속을 구현 할 수 있다.

4. **다형성 (Polymorphism)**
   - 객체의 속성이나 기능이 사용된 맥락에 따라 각기 다른 기능을 수행하는것

   - 상속, 메소드 오버라이딩, 상위 클래스를 통한 하위클래스에 대한 참조 (타입 캐스팅) 으로 달성 가능하다

   - 상호 클래스의 직접 의존성 (결합도) 를 낮추는 방안으로 고려된다

* **Overriding**
  - 부모 클래스의 메소드를 재정의 하는것

  - 오버라이드 가능성 :
    1) 추상 클래스인 부모 클래스에서 abstract 선언
    3) 부모 클래스에서 가상 함수 (virtual)로 선언
    2) 부모 클래스에서도 오버라이드 된 함수

* **Overloading**
  - 인수의 타입이나 갯수에 관계없이 같은 이름의 함수를 정의하고,
  컴파일러가 호출 시점에 해당 함수에 부여된 인수에 따라 실제 호출될 함수를 선택

  - 반환 타입과 이름이 같아야 함

객체지향 설계 원칙 (SOLID)

- 객체 지향 설계 5대 원칙

SRP , **O**CP , **L**SP , **I**SP , **D**IP

SRP - Single Responsibility Principle (단일 책임 원칙)

- 하나의 클래스는 하나의 책임(기능) 만 가져야 한다
- 기능이 많으면 결합도가 높아지며 이는 수정시 파급력이 커지므로 유지보수에 불리해진다

OCP - Open-Closed Principle (개방-폐쇄 원칙)

- 프로그램의 요소 전체는 확장에 대해 열려있어야 하나, 변경에는 닫혀 있어야 한다.
- 기존의 코드를 변경하지 않으면서도 추가하거나 수정 가능해야 한다
- 이를 달성하기 위한 좋은 기능의 예로 인터페이스가 있다.

LSP - Liskov Substitution Principle (리스코프 치환 원칙)

- 하위 타입은 상위 타입에서 가능한 행위를 수행할 수 있어야 한다
- 즉 상속받은 하위 클래스에서 상위 클래스의 동작을 누락해선 안된다
  -> 상위 클래스의 사용처에 하위 클래스를 사용하더라도
  동작이 가능해야 한다.

ISP - Interface Segregation Principle (인터페이스 분리 원칙)

- 인스턴스 사용처에서는 사용하는 메소드에만 의존해야 한다
- 인터페이스는 인터페이스 사용처에 따라 분리되어야 한다
- 하나의 인터페이스가 수정되더라도 다른 인터페이스에 영향을 주면 안된다

DIP - Dependency Inversion Principle (의존 역전 원칙)

- 프로그램을 작성할 때 구현(implementation) 에 의존하면 안되며
추상화(abstraction)에 의존해야 한다는 원칙
- 인터페이스를 정의하고 사용하면 그 구현부가 변경되더라도 유연한
구현이 가능하지만, 구현된 코드에 의존하면 구현부가 변경될 때
사용부에서도 변경이 필요한 경우가 생긴다