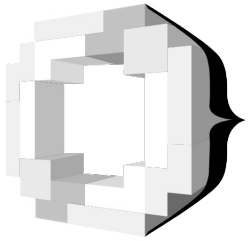
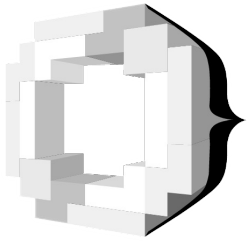
2023 유니티 중급반 5주차

오파츠 12기 여정인, 유태환



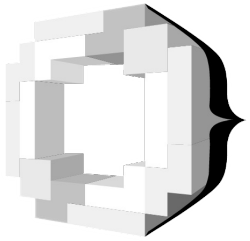
목차

- OOP (객체 지향 프로그래밍) 다시 훑어보기
- 팩토리 패턴 알아보기



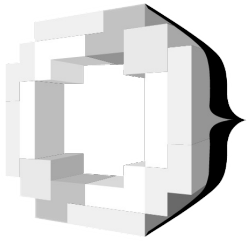
시작하기 전에

- 오늘은 저번주에 잠깐 알아본 몬스터 생성을 객체 지향 개념으로 알아보아요



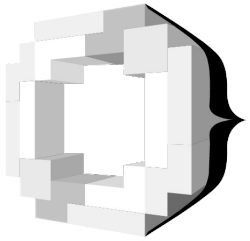
객체 지향 프로그래밍

- 객체 지향 프로그래밍 (Object Oriented Programming)
  - 데이터와 동작을 캡슐화하는 객체를 만들자!



객체 지향 프로그래밍

- 동물농장을 프로그램으로 만든다고 생각해봅시다



객체 지향 프로그래밍

- 동물농장을 프로그램으로 만든다고 생각해봅시다

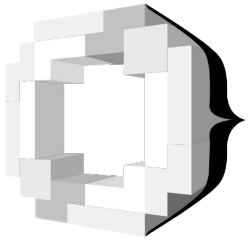
```
string name = 'none'; int id = 0
```

```
string name = 'body'; int id = 1
```

```
string name = 'human'; int id = 2
```

```
string name = 'dog'; int id = 3
```

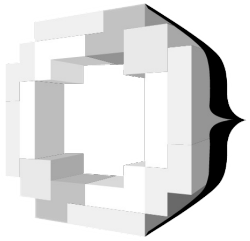
```
string name = 'cat'; int id = 4
```



객체 지향 프로그래밍

- 사람은 두발로 걷고 개는 네발로 걷는 코드를 짜주세요!

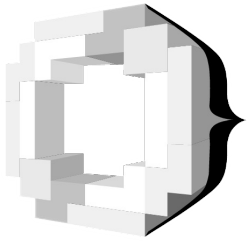
?



객체 지향 프로그래밍

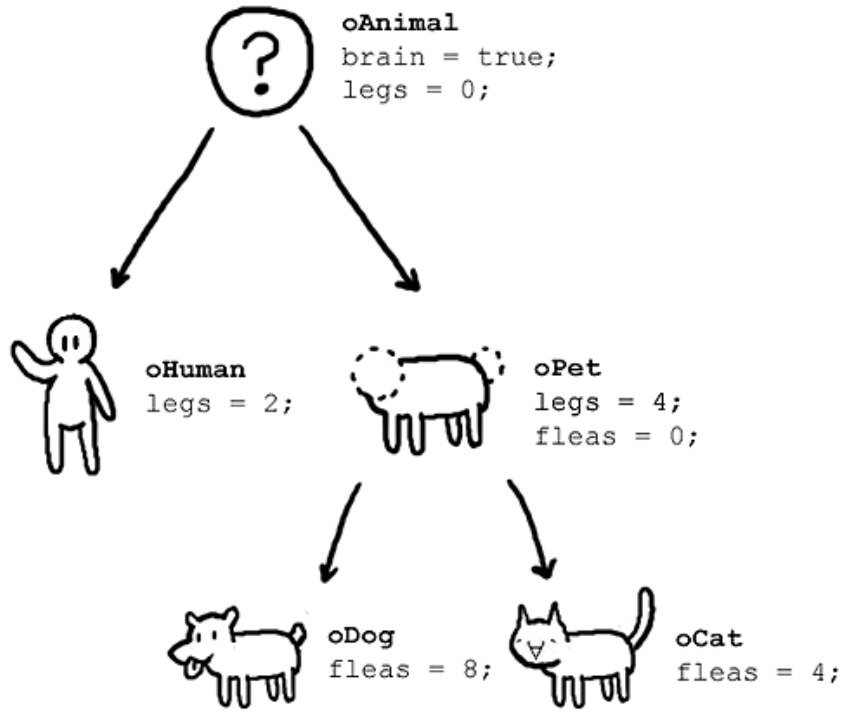
- 고양이 200마리 만들어주세요!

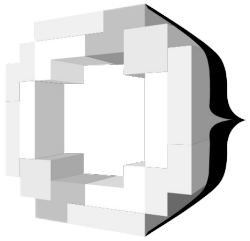
???



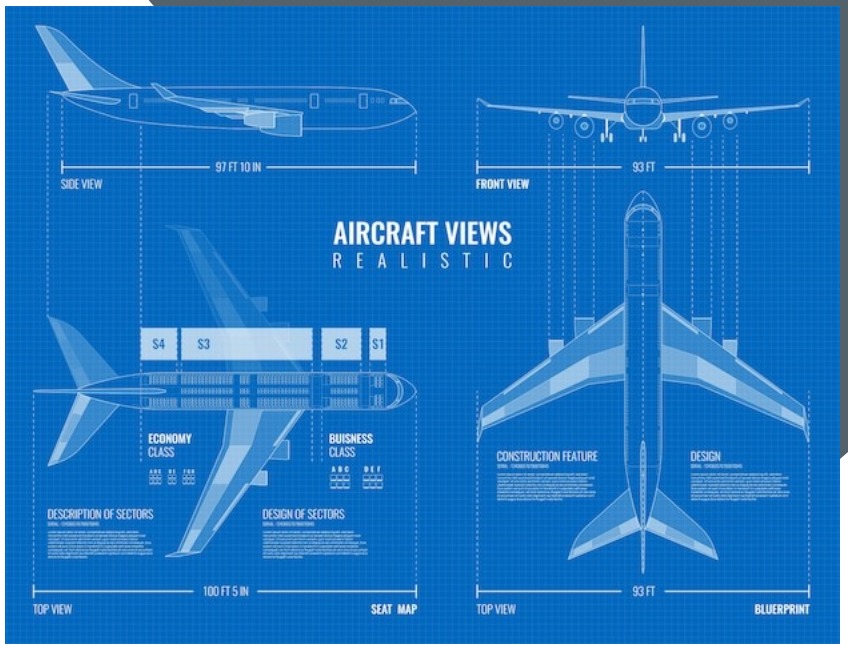
객체 지향 프로그래밍

- 객체 지향 프로그래밍 (Object Oriented Programming)



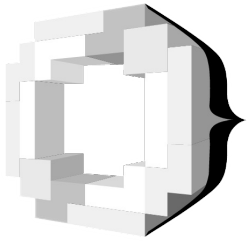


객체 지향 프로그래밍



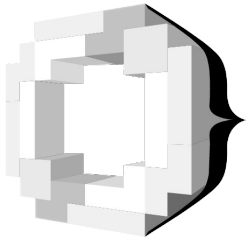
- 클래스와 객체
  - 비행기 설계도는 클래스이고
  - 비행기는 객체에요
  - 설계도 한번만 잘 만들면

    비행기는 수백대도 찍어낼 수 있어요



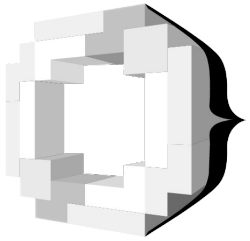
객체 지향 프로그래밍

- 상속
  - 부모의 속성과 동작을 상속하는 자식 클래스
  - 다른 클래스의 특성을 유지하면서도 새 속성이나 동작을 추가하고 싶을 때
  - 코드 재사용 + 계층 구조



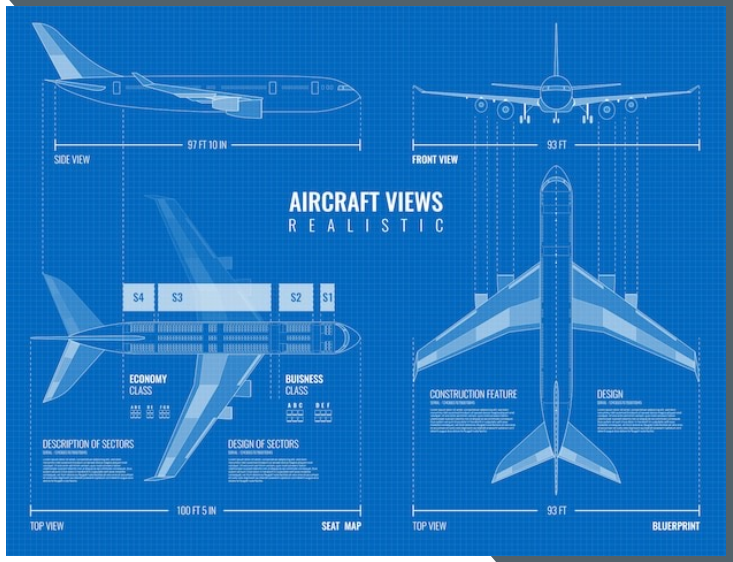
C#의 OOP

- Interface
  - 인터페이스는 이런 기능을 구현해두자! 하고 정한 일종의 계약이에요.
  - 이 계약을 체결(상속)한 클래스는 반드시 그 기능들을 구현해야 해요.
  - 기능을 어떻게 **구현**하느냐는 상관 없어요.
  - 유연한 기능 구현과 유지보수에 유리해요

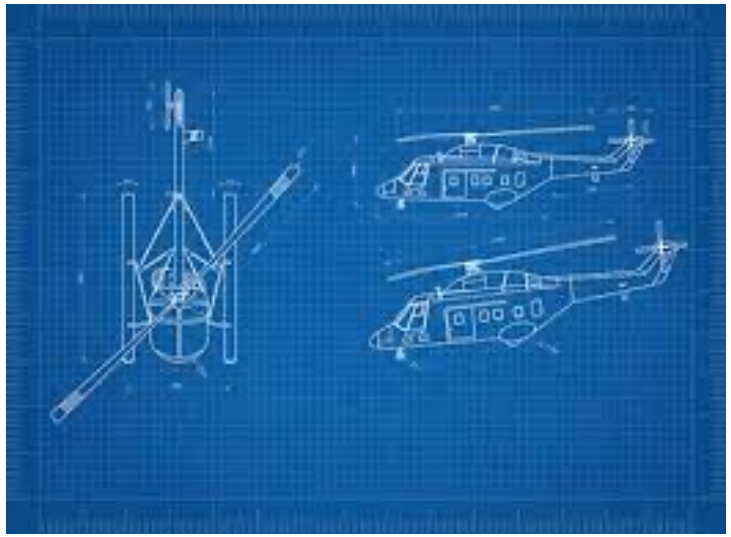


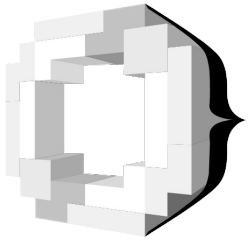
상속

- 비행기로 다시 생각해볼까요
  - 비행기의 기능을 조금만 개선시키면
    헬리콥터도 만들어 볼 수 있을 것 같아요
    (둘 다 날아다니잖아요!)
  - 비행기를 헬리콥터의 부모라고 생각하고
    날아다니는 기능을 물려달라고 하면 편하게
    헬리콥터를 만들 수 있어요



날아다니는 기능 상속!



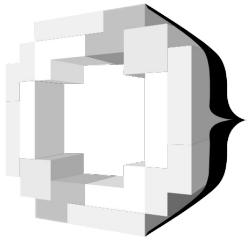


인터페이스

- 자동차와 헬리콥터가 모두 주차 가능한 주차장이 있다고 해요
  - 자동차와 헬리콥터 모두 주차 기능을 구현하겠다는 계약을 체결해요
  - 주차장에서는 자동차와 헬리콥터에게 주차 명령을 내려요

  (헬리콥터와 자동차가 어떤 방식으로 주차하는지는 몰라도 돼요! 그건 **구현**의 영역이에요)

주차 기능을 구현
하겠다고 계약!



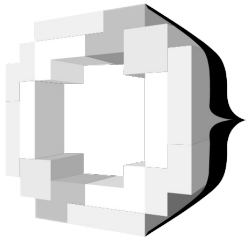
C#의 OOP

- Class

```
class ClassName
{
    // 각종 변수와 함수들
}
```

```
access-modifier를 변수, 함수 앞에 붙일 수 있어요

- public, protected, default, private
```

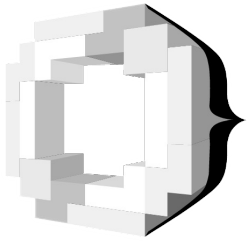
- 나만 가지고 싶으면 private
- 자식한테만 물려주고 싶으면 protected
- 모두에게 뿌리고 싶으면 public
- 외부의 접근으로부터 데이터와 기능을 보호해요
- 필요한 부분만을 보여줄 수 있어요



C#의 OOP

- Abstract class
  - 추상 클래스는 다른 클래스의 기본클래스(부모)로만 사용이 가능합니다
  - 따라서 인스턴스화가 불가능합니다
  - abstract가 붙은 메소드는 반드시 자식클래스에 재정의 해야합니다.

```
access-modifier abstract class ClassName
{
    // 각종 변수와 함수들
    public abstract void AbstractMethod();
}
```



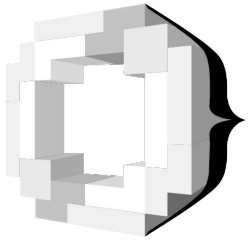
C#의 OOP

- 상속할 땐

```
access-modifier class ClassName : ParentClass,
Interface
{
    // 각종 변수와 함수선언 (구현 없음)
}
```

  - 클래스는 단 하나의 부모 클래스만을 가집니다.
  - 인터페이스는 여러 개 상속이 가능합니다.

    (계약은 여러 개 체결해도 돼요!)

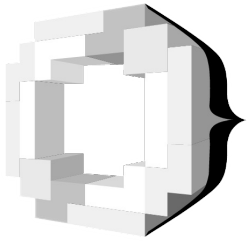


C#의 OOP

- 다형성 (런타임 다형성, 서브타입 다형성)

```
class Parent { void foo() { }; }
class Child1 : Parent { void foo() { print(“A”) } }
class Child2 : Parent { void foo() { print(“B”) } }

Parent pa = new Child1();
Parent pb = new Child2();
pa.foo() // A
pb.foo() // B
```

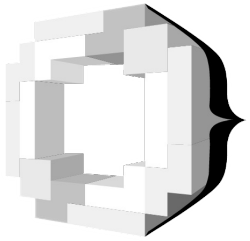
  - 같은 타입이라도 서로 다른 자식 객체가 들어있다면 다른 동작을 한다.



예제 살펴보기

- 이 게임에는 플레이어와 적이 있어요

플레이어와 적 모두 비슷하게 생겼어요. 왠지 같은 부모를 가지고 있을 것 같아요



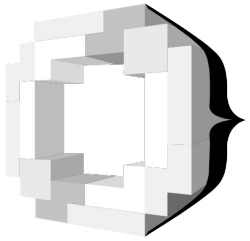
예제 살펴보기

- 플레이어나 적이나 모두 중력의 영향을 받고 땅과 충돌해요

?

사실 같은 부모를 가지고 있어서 그렇다고 생각해볼까요?
그럼 이 부모는 어떤 기능을 가지고 있을까요?

- 땅과의 충돌 감지를 해요
- 밖으로 떨어지면 죽어요
- 왼쪽으로 움직이면 왼쪽으로 방향을 틀고, 오른쪽으로 움직이면 오른쪽으로 방향을 틀어요



예제 살펴보기

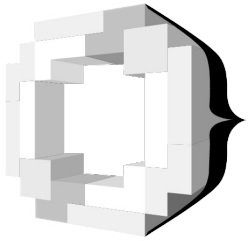
- 플레이어나 적이나 모두 HP를 가져요

?

그건 사실 부모가 HP 기능을 구현하겠다고 계약을 했기 때문이 아닐까요?

```
public interface IHasHP
{
    void LoseHP(int damage);
    void RecoverHP(int cure);
}
```

이 계약을 꼭 캐릭터가 아니더라도 아이템이 체결할 수도 있잖아요?



예제 살펴보기

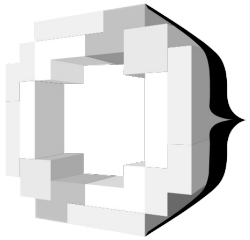
- 플레이어나 적이나 모두 HP를 가져요

?

그건 사실 부모가 HP 기능을 구현하겠다고 계약을 했기 때문이 아닐까요?

```
public interface IHasHP
{
    void LoseHP(int damage);
    void RecoverHP(int cure);
}
```

이 계약을 꼭 캐릭터가 아니더라도 아이템이 체결할 수도 있잖아요?



예제 살펴보기

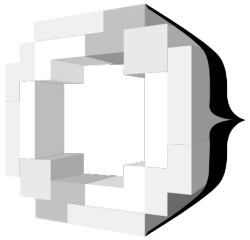
- 부모 클래스는 protected로 자식에게 물려줄 변수와 함수를 지정해요
  (public도 당연히 자식이 볼 수 있어요!)

?

```
protected Rigidbody2D rgbody;
protected SpriteRenderer spriteRenderer;

protected const int MAX_HP = 20;
protected int hp;

protected bool isOnGround = false;
```

```
protected virtual void Awake() {
    rgbody = GetComponent<Rigidbody2D>();
    spriteRenderer = GetComponent<SpriteRenderer>();

    hp = MAX_HP;
}
```

유니티 생명주기 함수들도 자식들에게 보여 주려면 protected virtual로 선언해야해요



예제 살펴보기

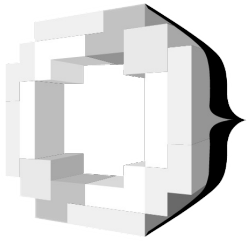
- 부모 클래스는 protected로 자식에게 물려줄 변수와 함수를 지정해요
  (public도 당연히 자식이 볼 수 있어요!)

?

```
protected Rigidbody2D rgbody;
protected SpriteRenderer spriteRenderer;

protected const int MAX_HP = 20;
protected int hp;

protected bool isOnGround = false;
```

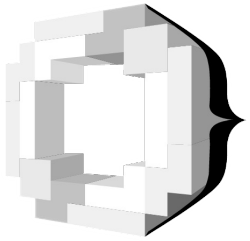
```
protected virtual void Awake() {
    rgbody = GetComponent<Rigidbody2D>();
    spriteRenderer = GetComponent<SpriteRenderer>();

    hp = MAX_HP;
}
```

유니티 생명주기 함수들도 자식들에게 보여 주려면 protected virtual로 선언해야해요



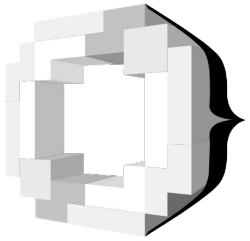
예제 살펴보기

- 플레이어는 키보드 입력으로 이동할 수 있어요
- 스페이스를 누르면 총알도 나가는 기능을 가져요
- 하지만 부모의 모든 기능들도 가지고 있어요!



예제 살펴보기

- 몬스터는 일정 시간마다 플레이어를 향해 이동해요
- 하지만 부모의 모든 기능들도 가지고 있어요!

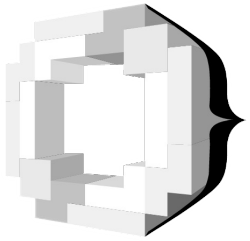


예제 살펴보기

- 음… 다르게 움직이는 몬스터도 만들고 싶어요.
- 아까 만들었던 몬스터를 부모로 가지는 다른 캐릭터를 만들어봐요

```
protected override void Move()
{
    Vector2 deltaPos = player.transform.position - transform.position;

    rgbody.velocity = new Vector2(deltaPos.normalized.x * MOVE_MAGNITUDE, rgbody.velocity.y);
}
```

override 키워드를 붙이면 부모가 가지는 함수를 다시 정의할 수 있어요



예제 살펴보기

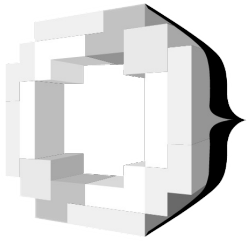
- 몬스터를 생성하는 EnemyFactory에서는

```
enemy.GetComponent<EnemyBase>().InitEnemy(player.gameObject, new Vector2(0, 0));
```

EnemyBase라는 이름으로 두 몬스터 모두 관리할 수 있어요!
자식은 부모의 타입으로 불릴 수 있거든요.

그 안에 들어있는 게 초록색 몬스터인지 분홍색 몬스터인지 중요하지 않아요.

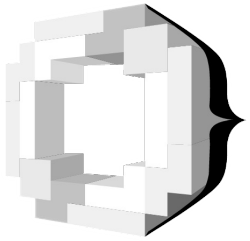
이게 바로 다형성이에요.



예제 살펴보기

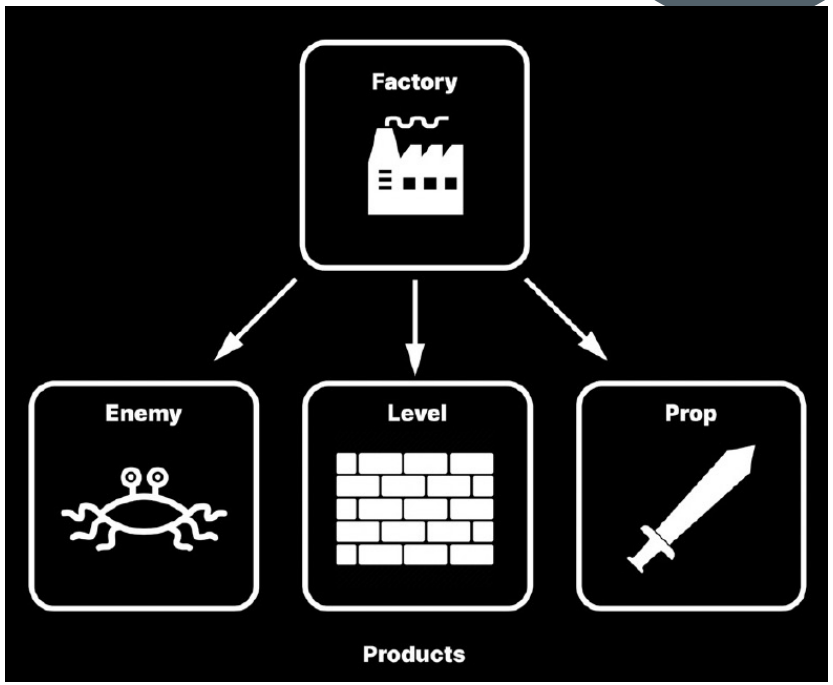
- 총알에서도 충돌판정을 할 때 부모 타입으로 확인할 수 있어요.
- Player, EnemyBase, EnemyOther 모두 대상이 될 수 있어요
- 모두 IHasHP 계약을 체결한 CharacterBase의 자식이에요.

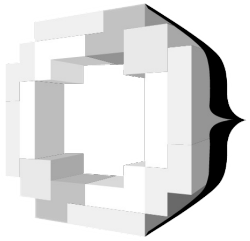
```
CharacterBase target = collision.gameObject.GetComponent<CharacterBase>();

if (target != null && !collision.tag.Equals("Player"))
{
    target.LoseHP(DAMAGE);
    Destroy(gameObject);
}

if (collision.tag == "Wall")
{
    Destroy(gameObject);
}
```



팩토리 패턴?

- EnemyFactory와 같이 다른 캐릭터, 맵, 아이템 등을 생성해서 게임 내에 뿌려주는 역할을 하는 일종의 공장을 하나 만들어 줄 수 있어요.
- 상속과 인터페이스를 잘 활용하면 구현하기 쉬워요





해보면 좋아요

- 직접 새로운 몬스터를 만들어봐요

- 유니티 기능들은 자세히 안 다뤘어요
  - 유니티 공식 매뉴얼을 살펴봐요
  - https://docs.unity3d.com/kr/2021.3/Manual/UnityManual.html