Python 객체지향 프로그래밍(2)

상속(Inheritance) · 오버라이딩(Overriding)

GOAL

- **상속**의 개념과 사용 이유 이해
- **메서드 오버라이딩**의 동작 원리 익히기
- **전직(직업 상속 구조)**를 통해 실제 예제 구성
- **부모/자식 클래스 관계**에서 `super()`의 역할 익히기
- **전투 시스템(공격 → 경험치 → 레벨업)** 구현

상속(Inheritance)이란?

상속(Inheritance)이란?

- 기존 클래스를 **물려받아 재사용**할 수 있는 기능
- 부모 클래스(Parent / Superclass)의 속성과 메서드를 **자식 클래스(Child / Subclass)**가 그대로 사용 가능
- 중복 코드를 줄이고 유지보수를 쉽게 만듦

```
class Character:  # 부모 클래스
    def move(self):
        print("캐릭터가 이동 중입니다.")

class Archer(Character):  # 자식 클래스
    pass

legolas = Archer()
legolas.move()  # 부모의 move() 사용 가능
```

왜 상속을 사용할까?

왜 상속을 사용할까?

상황	상속 전	상속 후
공통 코드 중복	전사, 궁수, 마법사 각각 move(), attack() 따로 정의	Character에 정의 후 상속받아 사용
유지보수	직업 추가 시 모든 클래스 수정	Character만 수정하면 전체 반영

기본 상속 구조 예제

```
class Character:
    def __init__(self, name):
        self.name = name
        self.level = 1

    def attack(self):
        print(f"{self.name} 기본 공격!")

class Warrior(Character):
    def __init__(self, name):
        super().__init__(name)  # 부모 초기화
        self.weapon = "검"

hero = Warrior("유저1")
hero.attack()  # 부모의 attack() 사용 가능
```

`super()`의 역할

- 부모 클래스의 기능을 **그대로 가져오거나 + 확장할 때** 사용
- **중복을 줄이고 가독성을 높임**

```
class Warrior(Character):
    def __init__(self, name):
        super().__init__(name)
        self.weapon = "검"
        print(f"{self.name}이(가) 무기 {self.weapon}을 장착했습니다!")
```

메서드 오버라이딩 (Overriding)

- 부모의 메서드를 **자식이 재정의**해서 자신의 방식으로 바꾸는 것
- 이름은 같지만 **내용을 다르게 구현**
- **같은 메서드 이름이지만 직업마다 다른 동작** = 다형성(Polymorphism)

메서드 오버라이딩 (Overriding)

```
class Archer(Character):
    def attack(self):
        print(f"{self.name} 활 공격!")

class Mage(Character):
    def attack(self):
        print(f"{self.name} 마법 공격!")

characters = [Archer("궁수유저"), Mage("법사유저"), Warrior("전사유저")]
for c in characters:
    c.attack()
```

상속 + 오버라이딩을 사용하는 이유

- “기본 구조는 같지만 행동은 다를 때”
- 전투 게임, 동물 행동, 결제 시스템 등에서 매우 자주 사용됨

공통	직업별 차이
이름, 레벨, HP	공격 방식, 스킬, 무기

➡️ 부모에 공통 정의, 자식에 개별 구현

전직 구조 설계 예시

```
class Character:
    def __init__(self, name):
        self.name = name
        self.hp = 100
        self.exp = 0
    def attack(self):
        print(f"{self.name}의 기본 공격!")

class Warrior(Character):
    def attack(self):
        print(f"{self.name}의 검 공격! 데미지 10")

class Archer(Character):
    def attack(self):
        print(f"{self.name}의 활 공격! 데미지 8")

class Mage(Character):
    def attack(self):
        print(f"{self.name}의 파이어볼! 데미지 12")
```

상속 트리 다이어그램

```
Character
├── Warrior
├── Archer
└── Mage
```

모든 직업은 Character의 **공통 속성(hp, exp)** 을 가짐.

각 클래스는 자신만의 attack()을 가짐.

`super()`로 부모 로직 재사용

```
class Warrior(Character):
    def attack(self):
        super().attack()
        print(f"{self.name}의 검 공격! 데미지 10")
```

출력:

```
전사유저의 기본 공격!
전사유저의 검 공격! 데미지 10!
```

전투 시스템 기본 설계

```
class Monster:
    def __init__(self, name, exp, hp=30):
        self.name = name
        self.hp = hp
        self.exp = exp

    def take_damage(self, dmg):
        self.hp = max(0, self.hp - dmg)
        print(f"{self.name}이(가) {dmg} 피해를 입음 (남은 HP: {self.hp})")
```

🔟 캐릭터와 몬스터 전투 예시

```
class Warrior(Character):
    def attack(self, monster):
        dmg = 10
        monster.take_damage(dmg)
        print(f"{self.name}의 검 공격! {monster.name}에게 {dmg} 피해!")

class Archer(Character):
    def attack(self, monster):
        dmg = 8
        monster.take_damage(dmg)
        print(f"{self.name}의 활 공격! {monster.name}에게 {dmg} 피해!")

m = Monster("주황버섯")
hero = Archer("메르세데스")
hero.attack(m)
```

경험치 시스템 추가

```
class Character:
    def __init__(self, name):
        self.name = name
        self.level = 1
        self.hp = 100
        self.exp = 0

    def gain_exp(self, amount):
        self.exp += amount
        print(f"EXP +{amount} (현재 {self.exp}/100)")
        if self.exp >= 100:
            self.level += 1
            self.exp = 0
            self.hp += 10
            print(f"🎉 레벨업! Lv.{self.level} HP +10")
```

전투 후 보상

```
def grant_reward(character, monster):
    print(f"{monster.name} 처치! EXP {monster.exp}, Gold +10")
    character.gain_exp(monster.exp)
```

다형성 (Polymorphism)

- 같은 이름의 메서드를 호출해도 **객체 타입에 따라 동작이 달라짐**

```
party = [Warrior("전사유저"), Archer("궁수유저"), Mage("법사유저")]
for c in party:
    c.attack()
```

결과:

```
전사유저의 검 공격! 데미지 10
궁수유저의 활 공격! 데미지 8
법사유저의 파이어볼! 데미지 12
```

다형성의 장점

- 직업이 많아져도 동일한 방식으로 코드 실행 가능
- 조건문 없이 객체의 메서드만 호출하면 됨

```
for c in party:
    c.attack(monster)
```

➡️ 전사든 궁수든 마법사든 알아서 자신의 공격 방식으로 실행!

상속 구조로 확장하기 쉬운 게임 설계

- 새로운 직업을 추가해도 부모 클래스의 구조를 그대로 따름

```
class Thief(Character):
    def attack(self, monster):
        print(f"{self.name}의 표창 공격! 12 피해!")
        monster.take_damage(dmg)
```

✅ 기존 코드를 수정하지 않고도 확장 가능!

핵심 정리

개념	설명
상속	부모의 속성과 기능을 자식이 물려받음
super()	부모의 메서드를 명시적으로 호출할 때 사용
오버라이딩	부모의 메서드를 자식이 재정의
다형성	같은 이름의 메서드가 다른 방식으로 동작

요약

- 상속은 공통 코드를 재사용하는 강력한 수단.
- 오버라이딩으로 자식이 자신만의 행동을 정의 가능.
- `super()` 로 부모 기능을 활용하면서 확장 가능.
- 다형성을 이용하면 직업이 늘어나도 코드가 간결.