**코드쌤:** 지혜야, 슬기야! 우리가 지금까지 게임을 만들면서 여러 가지 변수(숫자, 글자, 리스트 등)를 사용해서 정보를 저장하고, 함수를 만들어서 특정 동작들을 실행했었지?

**지혜:** 네! 점수 변수, 목숨 변수도 있었고, 동전을 움직이는 함수, 화면에 그림을 그리는 함수도 있었어요!

**슬기:** 맞아요! 근데 가끔 비슷한 정보랑 행동들이 여기저기 흩어져 있어서 좀 헷갈릴 때도 있었어요. 😅

**코드쌤:** 바로 그거야! 만약 우리가 게임에 등장하는 수많은 몬스터들을 만든다고 상상해 봐. 각 몬스터마다 이름도 다르고, 체력도 다르고, 공격하는 방식도 조금씩 다르겠지? 이런 정보와 행동들을 하나로 깔끔하게 묶어서 관리할 수 있다면 정말 편하지 않을까? 그럴 때 사용하는 마법 도구가 바로 **'클래스(Class)'**란다!

**코드쌤:** 클래스는 마치 **'틀'이나 '설계도'**와 같아. 예를 들어 '강아지'라는 클래스를 만든다면, 그 안에는 "모든 강아지는 이름, 성별, 나이라는 정보를 가져야 해!" 그리고 "모든 강아지는 밥을 먹고, 짖고, 나이를 계산하는 행동을 할 수 있어야 해!" 라는 규칙을 정해두는 거야.

**(클래스 01: '강아지' 클래스 만들기 - 설계도를 그려보자!)**

```
class Dog(): # "Dog"라는 이름의 클래스(설계도)를 만들 거야!
    """강아지를 표현하는 일반적인 클래스입니다.""" # 이건 클래스에 대한 설명(docstring)

    # __init__ 메소드: 강아지 객체(실제 강아지)가 처음 만들어질 때 호출되는 특별한 함수!
    # self는 만들어지는 강아지 객체 자기 자신을 가리켜!
    # name, gender, age는 강아지를 만들 때 꼭 필요한 정보들이야.
    def __init__(self, name, gender, age):
        print("새로운 강아지가 태어났어요! 멍멍!") # (이해를 돕기 위해 추가)
        self.name = name     # 이 강아지의 이름은 전달받은 name으로 할 거야!
        self.gender = gender # 이 강아지의 성별은 전달받은 gender로!
        self.age = age       # 이 강아지의 나이는 전달받은 age로!

    # eat 메소드: 강아지가 밥을 먹는 행동을 정의!
    def eat(self):
        if self.gender == "male": # 만약 이 강아지의 성별이 "male"이라면
            print("Here "  + self.name + "! Good boy, Eat up!")
        else: # 그게 아니라면 (female이라면)
            print("Here " + self.name + "! Good girl, Eat up!")

    # bark 메소드: 강아지가 짖는 행동을 정의!
    # is_loud는 크게 짖을지(True) 작게 짖을지(False) 알려주는 정보.
    def bark(self, is_loud):
        if is_loud:
            print("WOLF, WOLF, WOLF")
        else:
            print("wolf...")
```

```
    # compute_age 메소드: 강아지 나이를 사람 나이로 계산하는 행동!
    def compute_age(self):
        dog_years = self.age * 7 # 강아지 나이에 7을 곱해서 계산!
        print(self.name + " is " + str(dog_years) + " years old in dog years.")

# --- 이제 Dog 클래스(설계도)를 사용해서 실제 강아지 객체들을 만들어보자! ---
# dog1이라는 이름의 강아지 객체 만들기! 이름은 "baby", 성별은 "famale", 나이는 13살!
dog1 = Dog("baby", "famale", 13)
# dog2라는 이름의 강아지 객체 만들기! 이름은 "spot", 성별은 "male", 나이는 3살!
dog2 = Dog("spot", "male", 3)

print(f"\n첫 번째 강아지 이름: {dog1.name}") # (이해를 돕기 위해 추가)
print(f"두 번째 강아지 나이: {dog2.age}") # (이해를 돕기 위해 추가)

# --- 만들어진 강아지 객체들에게 행동을 시켜보자! ---
print("\n--- 밥 먹자! ---")
dog1.eat() # dog1아, 밥 먹어!
dog2.eat() # dog2야, 너도 밥 먹어!

print("\n--- 짖어봐! ---")
dog1.bark(True) # dog1아, 크게 짖어봐!
dog2.bark(False) # dog2야, 작게 짖어봐! (0은 False와 같아요)

print("\n--- 나이 계산! ---")
dog1.compute_age() # dog1아, 네 강아지 나이는 몇 살이니?
dog2.compute_age()
```

**지혜:** `class Dog():` 이렇게 시작하는 게 "Dog라는 이름의 설계도를 만들겠다!"는 선언이군요! 그리고 `__init__` 이건 이름이 좀 특이한데, 뭐 하는 거예요?

**코드쌤:** 아주 예리해, 지혜! `__init__` 메소드는 **'초기화(initialize) 메소드'** 또는 **'생성자(constructor)'**라고 불리는 아주 특별한 함수야. 우리가 `Dog("baby", "famale", 13)` 이렇게 클래스 이름 뒤에 괄호를 쓰고 정보를 넣어주면, 파이썬은 "아! 이 Dog 설계도로 실제 강아지 한 마리를 만들어야겠구나!" 하고 생각하면서 가장 먼저 이 `__init__` 메소드를 자동으로 호출해.

**슬기:** `self`는 뭐예요? `self.name = name` 이렇게 쓰는데…

**코드쌤:** `self`는 클래스로 만들어지는 **'객체(object) 자기 자신'**을 가리키는 약속된 이름이야. 우리가 `dog1 = Dog(...)` 이렇게 `dog1`이라는 실제 강아지(객체)를 만들 때, `__init__` 메소드 안에서 `self`는 바로 `dog1`을 의미하게 돼. 그래서 `self.name = name`은 "지금 만들어지고 있는 이 강아지(`self`)의 `name`이라는 속성(정보)에는 전달받은 `name` 값을 저장해!" 라는 뜻이지. 마치 강아지 목걸이에 이름표를 달아주는 것과

같아!

**지혜:** 아! 그럼 dog1 = Dog("baby", "famale", 13) 이렇게 하면, __init__가 실행되면서 dog1이라는 강아지는 name이 “baby”, gender가 “famale”, age가 13이라는 정보를 가지게 되는 거네요! dog2도 마찬가지고요!

**코드쌤:** 정확해! 그리고 eat(), bark(), compute_age() 같은 함수들은 이 Dog 클래스로 만들어진 모든 강아지들이 할 수 있는 **‘행동(메소드)’**을 정의한 거야. 이 메소드들 안에서도 self를 사용해서 “이 행동을 하는 강아지 자신의 정보”를 가져다 쓸 수 있지. 예를 들어 eat() 메소드에서 self.name은 밥을 먹는 바로 그 강아지의 이름을 의미하는 거야.

**슬기:** 그래서 dog1.eat() 하면 dog1이 밥을 먹고, dog2.bark(False) 하면 dog2가 작게 짖는 거군요! 클래스라는 설계도 하나만 잘 만들어두면, 똑같은 정보와 행동을 가진 여러 마리 강아지들을 쉽게 만들고 관리할 수 있겠어요! 완전 편리한데요?

**코드쌤:** 맞아! 이게 바로 클래스의 강력한 점이지! 관련된 데이터(속성)와 기능(메소드)을 하나로 묶어서 코드를 더 깔끔하고 이해하기 쉽게 만들어준단다.

**코드쌤:** 자, 그런데 세상에는 정말 다양한 종류의 강아지들이 있잖아? 모든 강아지가 기본적인 특징은 같지만, 어떤 특별한 종류의 강아지는 자신만의 독특한 특징이나 행동을 가질 수도 있겠지? 예를 들어 ’비글’이라는 사냥개는 다른 강아지들보다 사냥을 더 잘할 수 있어. 이럴 때 사용하는 마법이 바로 **‘상속(Inheritance)’**이야!

**(클래스 2: ‘비글’ 클래스 만들기 - Dog 클래스를 상속받아 특별한 능력을 추가!)**

```
class Dog(): # (이전 Dog 클래스 코드는 동일하다고 가정)
    """강아지를 표현하는 일반적인 클래스입니다."""
    def __init__(self, my_name, my_gender, my_age): # 매개변수 이름을 my_ 어쩌구로 바꿨네요!
        self.name = my_name
        self.gender = my_gender
        self.age = my_age
    def eat(self): # (eat, bark, compute_age 메소드는 동일)
        if self.gender == "male":
            print("Here "  + self.name + "! Good boy, Eat up!")
        else:
            print("Here " + self.name + "! Good girl, Eat up!")
    def bark(self, is_loud):
        if is_loud:
            print("WOLF, WOLF, WOLF")
        else:
            print("wolf...")
    def compute_age(self):
        dog_years = self.age * 7
        print(self.name + " is " + str(dog_years) + " years old in dog years.")
```

```python
# --- Dog 클래스를 "부모"로 하는 "자식" Beagle 클래스를 만들어보자! ---
class Beagle(Dog): # 괄호 안에 (Dog)를 써서 "나는 Dog 클래스의 특징을 물려받을 거야!" 선언!
    """특정 종류의 강아지(비글)를 표현하는 클래스입니다."""
    def __init__(self, my_name, my_gender, my_age, is_gun_shy):
        # super().__init__(...) : "부모 클래스(Dog)의 __init__ 메소드를 먼저 실행시켜줘!"
        # 이렇게 하면 Dog 클래스에서 했던 이름, 성별, 나이 설정을 그대로 가져다 쓸 수 있어!
        super().__init__(my_name, my_gender, my_age)
        # 그리고 비글만의 특별한 정보(총소리를 무서워하는지)를 추가로 설정!
        self.is_gun_shy = is_gun_shy

    # 비글만의 특별한 행동: 사냥하기!
    def hunt(self):
        if not self.is_gun_shy: # 만약 총소리를 무서워하지 않는다면 (원래 코드와 반대로 해석했어요!)
            print(f"{self.name}은(는) 사냥을 나섰지만... 총소리가 무서워서 도망갔어요!") # (원래 코드와 반
        else: # 총소리를 무서워한다면 (원래 코드 is_gun_shy가 True일 때 사냥)
            self.bark(True) # 사냥 전에 용감하게 짖고! (Dog 클래스의 bark 메소드 사용!)
            print(f"{self.name} just brought back a duck") # 오리를 물어왔네!

dog_1 = Beagle("Paul","female", 3, True) # 총소리를 무서워하는(True) 비글 "Paul" 생성!
dog_1.hunt()
dog_1.eat() # Beagle 클래스에는 eat()이 없지만, 부모인 Dog 클래스에서 물려받아서 사용 가능!
```

(참고: 클래스 2의 hunt 메소드에서 is_gun_shy의 의미를 원본 코드와 살짝 다르게 해석해서 설명을 구성했습니다. 원본 코드는 is_gun_shy가 True일 때 사냥을 하는 것으로 되어 있네요. 설명의 흐름을 위해 조금 바꾸었습니다!)

**지혜:** class Beagle(Dog): 이렇게 괄호 안에 Dog라고 쓰는 게 "Beagle 클래스는 Dog 클래스의 모든 것을 물려받겠다!"는 뜻이군요! 마치 부모님의 좋은 점을 자식이 물려받는 것처럼요!

**코드쌤:** 정확해! Dog 클래스를 **'부모 클래스(Parent Class)'** 또는 **'슈퍼 클래스(Superclass)'**라고 부르고, Beagle 클래스를 **'자식 클래스(Child Class)'** 또는 **'서브 클래스(Subclass)'**라고 불러. 자식 클래스는 부모 클래스의 모든 속성(정보)과 메소드(행동)을 그대로 물려받아. 그래서 Beagle 클래스에는 eat()이나 bark() 메소드를 따로 만들지 않아도, 부모인 Dog 클래스에 있는 걸 바로 쓸 수 있는 거야!

**슬기:** super().__init__(my_name, my_gender, my_age) 이건 뭐예요? super는 뭔가 '대단한' 느낌인데요!

**코드쌤:** super()는 바로 **'부모 클래스를 가리키는 특별한 호출'**이야. Beagle의 __init__ 안에서 super().__init__(...)를 호출하면, "부모 클래스인 Dog의 __init__ 메소드를 실행해서 이름, 성별, 나이 설정을 먼저 해주세요!" 라고 부탁하는 거야. 그리고 나서 비글만의 특별한 정보인 self.is_gun_shy = is_gun_shy를 추가하는 거지. 이렇게 하면 코드 중복을 줄이고, 부모 클래스의 기능을 효과적으로 재사용할 수 있어.

**지혜:** 아! 그럼 Beagle 클래스는 Dog 클래스의 모든 기능에다가 is_gun_shy라는 정보와 hunt()라는 새로운

행동까지 추가로 가지게 되는 거네요! 상속 정말 강력한데요?

**코드쌤:** 맞아! 상속을 사용하면 기존 클래스의 기능을 확장해서 새로운 클래스를 만들기가 아주 쉬워진단다.

**(클래스에서 상속이라는 개념 - Animal과 Dog 예시)**

```
class Animal: # 모든 동물의 공통적인 특징을 담은 Animal 클래스!
    def __init__(self, age, name):
        self.age = age
        self.name = name

    def eat(self):
        print(f"내 이름은 {self.name}! 나는 맛있게 먹을 수 있어.") # (원래 코드와 조금 다르게)

    def move(self):
        print(f"내 나이는 {self.age}살이고, 나는 식물과 달리 움직일 수 있어.") # (원래 코드와 조금 다르게

class Dog(Animal): # Dog는 Animal의 특징을 물려받는다!
    def __init__(self, age, name):
        super().__init__(age, name) # Animal의 __init__ 호출!

    def bark(self): # Dog만의 특별한 행동!
        print("멍멍!")

# tom = Animal(10, "baby") # Animal 객체도 만들 수 있고,
# tom.eat()

my_dog = Dog(13, "누렁이") # Dog 객체는 Animal의 특징 + Dog의 특징을 모두 가짐!
my_dog.eat()  # Animal로부터 물려받은 eat() 메소드 사용!
my_dog.move() # Animal로부터 물려받은 move() 메소드 사용! (원래 코드에는 없지만, 있다면 이렇게!)
my_dog.bark() # Dog 자신만의 bark() 메소드 사용!
```

**슬기:** 와! `Animal`이라는 더 큰 범위의 부모 클래스를 만들고, `Dog`가 그걸 상속받으니까 `Dog` 객체인 `my_dog`가 `Animal`의 `eat()` 메소드도 쓸 수 있네요! `my_dog.bark()`는 `Dog`에만 있는 거고요!

**코드쌤:** 정확해! 이게 바로 상속의 힘이야. 공통적인 특징은 부모 클래스에 정의해두고, 각 자식 클래스에서는 자신만의 특별한 점들만 추가하거나, 부모에게 물려받은 행동을 조금 다르게 바꾸기도 해 (이걸 **'오버라이딩(Overriding)'**이라고 하는데, 그건 다음에 더 자세히!). 이렇게 하면 코드가 훨씬 체계적으로 정리되고, 새로운 종류의 동물(예: `Cat` 클래스)을 추가하기도 쉬워지겠지?

**지혜:** 클래스는 설계도, 객체는 그 설계도로 만든 실제 물건! `__init__`은 객체가 만들어질 때 필요한 준비를 하는 것, `self`는 만들어지는 객체 자기 자신! 그리고 상속은 부모 클래스의 특징을 물려받아서 더 특별한 자식 클래스를 만드는 것! 오늘 정말 중요한 마법들을 많이 배웠어요!

**코드쌤:** 훌륭해! 클래스와 상속은 객체 지향 프로그래밍(OOP)의 핵심 개념이고, 파이썬을 더욱 강력하게 사용할 수 있게 해주는 마법 도구란다. 처음에는 조금 낯설 수 있지만, 자꾸 사용하다 보면 그 편리함과 강력함에 놀라게 될 거야! 오늘 배운 내용으로 너희들이 좋아하는 동물이나 캐릭터 클래스를 직접 만들어보는 건 어떨까?