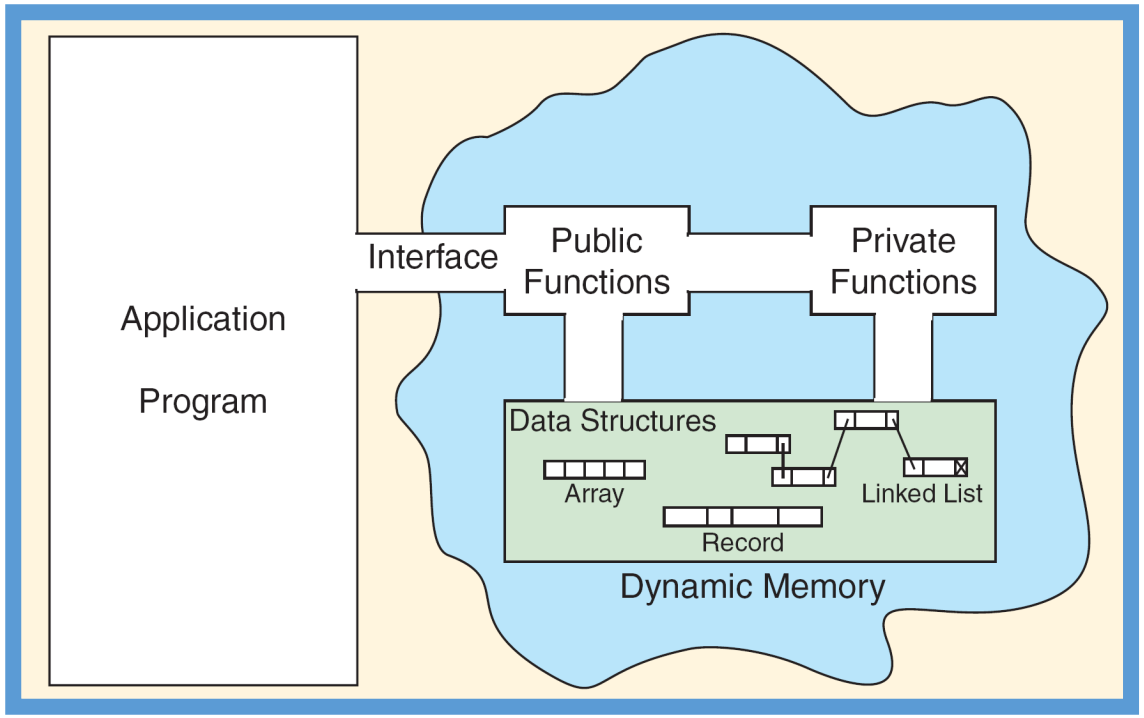
Object Oriented Programming examples 02

Contents

- 객체지향 설계에서 클래스 도출방법
- 클래스 다이어그램

ADT and class

- 객체지향언어는 ADT 개념을 class 명령어를 사용하여 구현
- Python에서 class는 데이터와 함수를 묶어서 선언할 수 있도록 만들어진 사용자 정의 자료형(User defined data type)이라 할 수 있음




```
#Python - Class

class  Car :
    color = ""
    speed = 0

    def  upSpeed(self, value) :
        self.speed += value

    def  downSpeed(self, value) :
        self.speed -= value

mycar1=Car() #객체 생성
```

객체지향 설계에서 클래스 도출 방법

- 시스템의 요구사항을 분석하고, 이를 효율적으로 구현할 수 있는 클래스로 변환하는 과정

1. 요구사항 분석

- 문제 이해하기: 시스템이 해결해야 할 문제를 이해
- 기능적 요구사항 정의: 시스템이 수행해야 할 기능을 명확히 도출
- 비기능적 요구사항 정의: 성능, 보안, 사용성 등의 비기능적 요소를 고려

2. 개념 모델링

- 도메인 개념 식별: 시스템과 관련된 주요 개념들을 식별(ex:'온라인 쇼핑 시스템'에서는 '상품', '고객', '주문’ 등)
- 속성 정의: 각 개념의 속성을 정의 (ex:'상품'에는 '가격', '재고', '설명’ 등)
- 관계 파악: 개념들 사이의 관계를 파악 (ex:'고객'은 여러 '주문'을 할 수 있음)

객체지향 설계에서 클래스 도출 방법

3. 클래스 도출

- 클래스 후보 선정: 개념 모델링에서 식별된 개념들을 바탕으로 클래스 후보를 선정
- 클래스 정의: 각 클래스의 속성과 메소드를 정의 (속성은 개념의 속성에서, 메소드는 시스템의 기능적 요구사항에서 도출)
- 상속 및 연관 관계 설정: 클래스 간의 상속, 연관, 집합, 구성 등의 관계를 정의

4. 반복 및 정제

- 반복적 검토 및 수정: 설계는 반복적인 과정이며, 새로운 요구사항이 추가되거나 변경될 때마다 클래스를 재검토하고 필요에 따라 수정
- 정제: 중복을 제거하고, 불필요한 클래스를 줄이며, 클래스의 책임을 명확하게 작성

객체지향 설계에서 클래스 도출 방법

5. 문서화 및 검증

- UML 다이어그램 작성: 클래스 다이어그램을 통해 클래스와 그 관계를 시각적으로 표현
- 검증: 설계가 모든 요구사항을 충족하는지 검증

- 팁 및 주의사항

-실세계 모델링: 객체지향 설계는 실세계 문제를 모델링하는 것이므로, 실세계의 개념과 프로세스를 잘 이해하는 것이 중요

- 유연성 유지: 요구사항 변경에 유연하게 대응할 수 있는 설계를 고려

- 과도한 세분화 방지: 너무 많은 클래스로 세분화하는 것은 복잡성을 증가시킬 수 있으므로 주의

- 객체지향 설계에서 클래스 도출은 시스템의 효율적인 구현과 유지보수에 중요한 역할이며, 이 과정은 체계적이고 반복적인 접근을 필요

과일가게 프로그램 요구사항 분석서

1. 개요

이 프로그램은 과일가게의 과일 재고 관리 및 판매를 돕는 시스템

과일의 종류, 가격, 재고 수량을 관리하고, 고객에게 판매하는 기능을 포함

2. 기능적 요구사항

1) 과일 정보 관리

- 과일의 이름과 가격 정보를 관리
- 과일은 `Fruit` 클래스를 통해 표현되며, 각 과일은 이름과 가격 속성

2) 특정 과일 클래스

- `Apple`, `Banana` 등 특정 과일은 `Fruit` 클래스로부터 상속받아 구현
- 이들 클래스는 과일의 특정 종류를 나타냄

과일가게 프로그램 요구사항 분석서

3.할인된 과일 처리

- 할인된 과일은 `DiscountedFruit` 클래스를 통해 처리
- 이 클래스는 할인율을 적용하여 최종 가격을 계산

4. 과일 재고 관리

- `FruitShop` 클래스는 과일의 재고를 관리
- 과일의 추가, 재고 확인, 판매 등의 기능을 포함

5. 과일 판매 기능

- 고객이 과일을 구매할 경우, 재고에서 해당 수량만큼 감소
- 재고가 부족할 경우, 판매가 불가능함을 알림

6. 할인 이벤트 적용

- 특정 과일에 대한 할인 이벤트를 적용할 수 있어야 함
- 할인율을 적용하여 과일의 가격을 조정

과일가게 프로그램 요구사항 분석서

3. 비기능적 요구사항

1) 확장성

- 새로운 과일 종류나 기능을 쉽게 추가할 수 있어야 함

2) 사용자 친화적 인터페이스

- 재고 관리 및 판매 과정이 명확하고 이해하기 쉬워야 함

3) 성능 요구사항

- 프로그램은 과일의 추가, 조회, 판매 등의 기능을 신속하게 처리할 수 있어야 함

4. 구현 기술

- 프로그래밍 언어: Python
- 주요 클래스: `Fruit`, `Apple`, `Banana`, `DiscountedFruit`, `FruitShop`

과일가게 프로그램 알고리즘

1. 초기 설정

- 과일가게(`FruitShop`) 인스턴스 생성
- 기본 과일(`Apple`, `Banana`) 인스턴스 생성

2. 과일 재고 추가

- `FruitShop`의 `add_fruit` 메소드를 사용하여 과일과 해당 수량을 재고에 추가

3. 할인 이벤트 적용

- 특정 과일에 대한 할인율 설정
- `FruitShop`의 `apply_discount` 메소드를 사용하여 할인율 적용

과일가게 프로그램 알고리즘

1. Fruit 클래스:
   - 이 클래스는 모든 과일의 기본 클래스
   - 속성: 과일의 이름(`_name`)과 가격(`_price`)
   - private 속성으로, 클래스 외부에서 직접 접근할 수 없음
   - 메소드: `get_price()`는 과일의 가격을 반환
   - `__str__()`는 과일의 이름과 가격을 문자열로 반환

2. Apple, Banana 클래스:
   - 이들은 `Fruit` 클래스의 하위 클래스
   - `Apple` 클래스는 이름을 "사과"로, `Banana` 클래스는 이름을 "바나나"로 자동 설정
   - 각 클래스는 고유의 가격을 받아 부모 클래스의 생성자에 전달

과일가게 프로그램 알고리즘

3. DiscountedFruit 클래스:

   - `Fruit`의 하위 클래스
   - 할인된 가격을 계산하는 기능을 추가
   - 속성: 할인율(`_discount`)
   - 메소드: `get_discounted_price()`는 할인된 가격을 계산
   - `__str__()`는 할인된 가격과 원래 가격을 문자열로 반환

과일가게 프로그램 알고리즘

4. FruitShop 클래스:
   - 과일 가게를 나타내며, 다양한 과일과 그 수량을 관리
   - 속성: 과일 목록(`_fruits`)을 저장하는 딕셔너리. 이는 private 속성
   - 메소드:
     - `add_fruit()`: 새로운 과일을 가게에 추가하거나, 이미 있는 과일의 수량을 업데이트
     - `sell_fruit()`: 특정 과일을 판매하고, 수량을 감소
     - `show_inventory()`: 현재 가게에 있는 모든 과일과 그 수량을 표시
     - `apply_discount()`: 특정 과일에 대해 할인을 적용

과일가게 프로그램 알고리즘

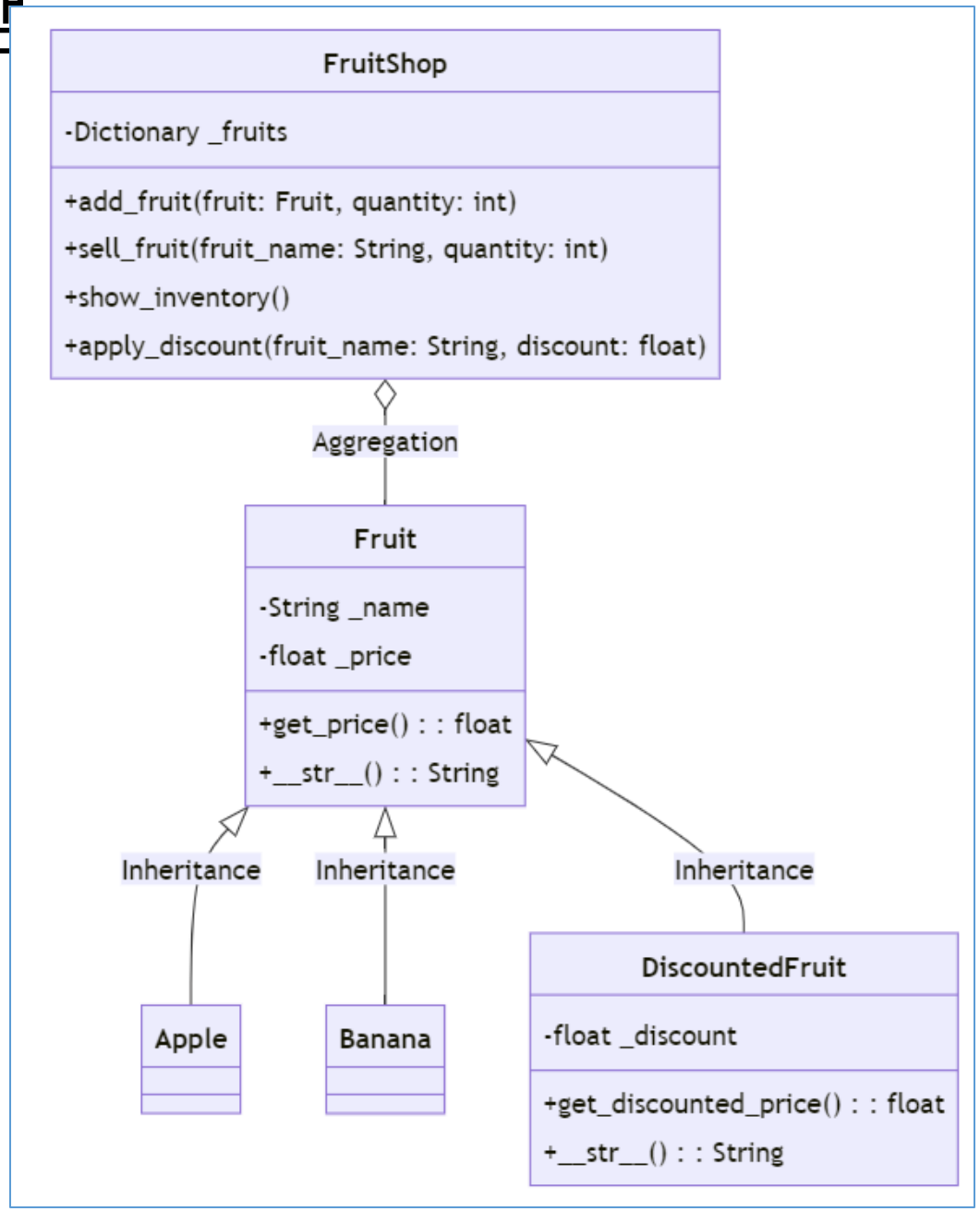
5. 실행 코드:

   - `FruitShop` 객체를 생성하고, `Apple`과 `Banana` 객체를 만들어 가게에 추가

   - `Apple`에 대해 할인을 적용하고, 재고를 확인한 후, 일부 과일을 판매하고 다시 재고를 확인

- 객체지향 프로그래밍의 기본 원칙을 따르며, 클래스 상속, 데이터 은닉, 메소드 오버라이딩 등의 개념을 사용

클래스 다이어그램


FruitShop
-Dictionary _fruits
+add_fruit(fruit: Fruit, quantity: int)
+sell_fruit(fruit_name: String, quantity: int)
+show_inventory()
+apply_discount(fruit_name: String, discount: float)
Aggregation
Fruit
-String _name
-float _price
+get_price() : : float
+__str__() : : String
Inheritance
Inheritance
Inheritance
Apple
Banana
DiscountedFruit
-float _discount
+get_discounted_price() : : float
+__str__() : : String

클래스 다이어그램

1. 상속 관계(Inheritance):

   - `Fruit` 클래스는 `Apple`, `Banana`, `DiscountedFruit` 클래스의 부모 클래스

   - `Apple`, `Banana`, `DiscountedFruit` 클래스는 `Fruit` 클래스로부터 상속을 받음

   - `Fruit` 클래스의 속성과 메소드를 `Apple`, `Banana`, `DiscountedFruit` 클래스가 사용할 수 있음을 의미

   - 다이어그램에서 이 관계는 화살표(`--▷`)로 표시되며, 화살표는 부모 클래스(`Fruit`)에서 자식 클래스(`Apple`, `Banana`, `DiscountedFruit`)의 관계에서 표시

2. 연관 관계(Association):

   - `Apple`, `Banana`, `DiscountedFruit` 클래스는 `Fruit` 클래스의 인스턴스를 참조할 수 있음을 의미

   - 다이어그램에서 이 관계는 점선(`..>`)으로 표시

클래스 다이어그램

3. 집합 관계(Aggregation):

   - `FruitShop` 클래스는 `Fruit` 클래스와 집합 관계를 가짐

   - `FruitShop` 클래스가 `Fruit` 인스턴스들을 관리하지만, `Fruit` 인스턴스들이 `FruitShop`가 없어도 독립적으로 존재할 수 있음

   - 다이어그램에서 이 관계는 빈 다이아몬드와 선(`--◇`)으로 표시되며, 다이아몬드는 `FruitShop` 클래스 쪽에 위치

이러한 관계들은 객체지향 프로그래밍에서 클래스 간의 상호작용과 의존성을 나타내며, 코드의 재사용성과 유지보수성을 높이는 데 중요한 역할

과일가게 예제

```
class Fruit:
    def __init__(self, name, price):
        self._name = name  # 데이터 은닉을 위해 속성을 private으로 설정
        self._price = price

    def get_price(self):
        return self._price

    def __str__(self):
        return f"{self._name} (가격: {self._price}원)"

class Apple(Fruit):
    def __init__(self, price):
        super().__init__("사과", price)

class Banana(Fruit):
    def __init__(self, price):
        super().__init__("바나나", price)

class DiscountedFruit(Fruit):
    def __init__(self, name, price, discount):
        super().__init__(name, price)
        self._discount = discount

    def get_discounted_price(self):
        return self._price * (1 - self._discount)

    def __str__(self):
        # 메소드 오버라이딩을 사용하여 __str__ 메소드를 재정의
        discounted_price = self.get_discounted_price()
        return f"{self._name} (할인가: {discounted_price}원, 원래 가격: {self._price}원)"
```

과일가게 예제

```
class FruitShop:
    def __init__(self):
        self._fruits = {}  # 데이터 은닉

    def add_fruit(self, fruit, quantity):
        if fruit._name in self._fruits:
            self._fruits[fruit._name]['quantity'] += quantity
        else:
            self._fruits[fruit._name] = {'fruit': fruit, 'quantity': quantity}

    def sell_fruit(self, fruit_name, quantity):
        if fruit_name in self._fruits and self._fruits[fruit_name]['quantity'] >= quantity:
            self._fruits[fruit_name]['quantity'] -= quantity
            print(f"{fruit_name} {quantity}개 판매 완료!")
        else:
            print("재고가 부족합니다.")

    def show_inventory(self):
        for fruit in self._fruits.values():
            print(f"{fruit['fruit']} - 재고: {fruit['quantity']}개")

    def apply_discount(self, fruit_name, discount):
        if fruit_name in self._fruits:
            original_fruit = self._fruits[fruit_name]['fruit']
            discounted_fruit = DiscountedFruit(original_fruit._name, original_fruit.get_price(), discount)
            self._fruits[fruit_name]['fruit'] = discounted_fruit
            print(f"{fruit_name}에 {discount*100}% 할인이 적용되었습니다.")
```

과일가게 예제

```
# 과일가게 객체 생성
shop = FruitShop()

# 과일 객체 생성 및 재고 추가
apple = Apple(1000)
banana = Banana(500)
shop.add_fruit(apple, 30)
shop.add_fruit(banana, 20)

# 특별 이벤트 - 사과 할인 적용
shop.apply_discount("사과", 0.2)

# 재고 확인
shop.show_inventory()

# 과일 판매
shop.sell_fruit("사과", 5)
shop.sell_fruit("바나나", 2)

# 재고 확인
shop.show_inventory()
```

```
사과에 20.0% 할인이 적용되었습니다.
사과 (할인가: 800.0원, 원래 가격: 1000원) - 재고: 30개
바나나 (가격: 500원) - 재고: 20개
사과 5개 판매 완료!
바나나 2개 판매 완료!
사과 (할인가: 800.0원, 원래 가격: 1000원) - 재고: 25개
바나나 (가격: 500원) - 재고: 18개
```

클래스 다이어그램 (집합 관계)

- 클래스 다이어그램에서 집합관계는 객체 간의 특별한 형태의 연관 관계를 나타내며, 집합관계는 크게 집약(aggregation)과 구성(composition)으로 표현

1. 집약 (Aggregation):
   - 집약은 전체와 부분의 관계를 나타내지만, 부분이 전체와 생명주기를 공유하지 않는 경우를 의미
   - 예를 들어, '자동차' 클래스와 '바퀴' 클래스가 있을 때, 바퀴는 자동차의 일부분이지만, 자동차가 없어져도 바퀴는 독립적으로 존재할 수 있음
   - 클래스 다이어그램에서는 보통 빈 다이아몬드(`◇`)로 표시되며, 전체 클래스 쪽에 다이아몬드가 위치

2. 구성 (Composition):
   - 구성은 전체와 부분의 관계를 나타내지만, 부분이 전체의 생명주기에 의존하는 경우를 의미
   - 예를 들어, '집' 클래스와 '방' 클래스가 있을 때, 방은 집이 없어지면 존재할 수 없으며, 집이 소멸되면 방도 함께 소멸
   - 클래스 다이어그램에서는 채워진 다이아몬드(`◆`)로 표시되며, 마찬가지로 전체 클래스 쪽에 다이아몬드가 위치

- 집약과 구성 모두 연관 관계의 한 형태이지만, 객체 간의 의존성과 생명주기에 따라 두 관계는 명확하게 구분
- 이러한 관계를 통해 객체 지향 프로그래밍에서 클래스들 간의 복잡한 상호작용과 의존성을 명확하게 표현

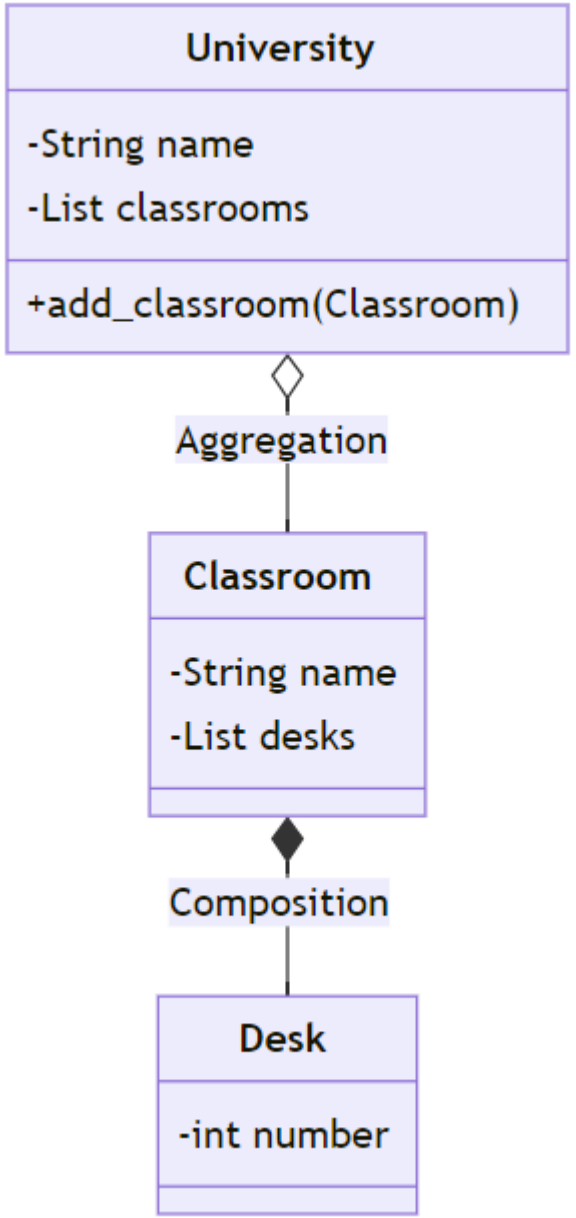
대학과 강의실 예제

- '대학(University)', '강의실(Classroom)', 책상(Desk) 클래스를 사용

1. 집약 (Aggregation) 관계:
   - '대학' 클래스는 여러 '강의실'을 가질 수 있음
   - 강의실은 대학에 속하지만, 대학이 없어져도 강의실은 독립적으로 존재할 수 있음
   - 이는 대학과 강의실이 집약 관계에 있다는 것을 의미

2. 구성 (Composition) 관계:
   - '강의실' 클래스 내에는 '책상(Desk)' 클래스가 있을 수 있음
   - 책상은 강의실에 속하며, 강의실이 없어지면 책상도 존재할 수 없음
   - 이는 강의실과 책상이 구성 관계에 있다는 것을 의미

클래스 다이어그램


University
-String name
-List classrooms
+add_classroom(Classroom)
Aggregation
Classroom
-String name
-List desks
Composition
Desk
-int number

예제 코드

```
class Desk:
    def __init__(self, number):
        self.number = number

class Classroom:
    def __init__(self, name):
        self.name = name
        self.desks = [Desk(i) for i in range(1, 101)]  # 강의실은 100개의 책상을 포함

class University:
    def __init__(self, name):
        self.name = name
        self.classrooms = []

    def add_classroom(self, classroom):
        self.classrooms.append(classroom)

# 객체 생성
university = University("POSTECH")
classroom1 = Classroom("506")
classroom2 = Classroom("502")

# 대학에 강의실 추가
university.add_classroom(classroom1)
university.add_classroom(classroom2)

# 대학에 추가된 강의실 목록 출력
print(f"{university.name} has the following classrooms:")
for classroom in university.classrooms:
    print(classroom.name)

# 각 강의실의 책상 수 출력
for classroom in university.classrooms:
    print(f"Classroom {classroom.name} has {len(classroom.desks)} desks")
```

```
POSTECH has the following classrooms:
506
502
Classroom 506 has 100 desks
Classroom 502 has 100 desks
```