2026.01.22.목

생성일	@2026년 1월 22일 오후 1:32
태그	

목차

▼ 1) 객체지향 (OOP)

```
from datetime import datetime

class Car:
    """자동차를 표현하는 클래스"""

    # 클래스 변수
    wheels = 4
    total_cars = 0  # 생성된 자동차 수를 추적

    def __init__(self, make, model, year, color):
        self.make = make
        self.model = model
        self.year = year
        self.color = color
        self.speed = 0
        self.odometer = 0

        # 새 자동차가 생성될 때마다 클래스 변수 카운트 증가
        Car.total_cars += 1

    @classmethod
    def count_cars(cls):
        """생성된 총 자동차 수를 반환합니다."""
        return f"지금까지 {cls.total_cars}대의 자동차가 생성되었습니다."

    @classmethod
    def create_sedan(cls, make, year, color):
        """세단 타입의 자동차를 생성하는 팩토리 메서드"""
        # 모델명에 자동으로 " 세단"을 붙여서 새로운 인스턴스를 생성합니다.
        return cls(make, make + " 세단", year, color)

    @staticmethod
    def is_valid_year(year):
        """연식이 유효한지 확인합니다."""
        # 최초의 가솔린 자동차 발명 연도(1886년)부터 현재 연도 사이인지 검사합니다.
        return 1886 <= year <= datetime.now().year

    @staticmethod
    def get_vehicle_type(model):
        """모델명을 바탕으로 차량 유형을 추측합니다."""
        # 대소문자 구분 없이 검색하기 위해 소문자로 변환합니다.
        model = model.lower()

        if "suv" in model or "지프" in model:
```

```
            return "SUV"
        elif "트럭" in model or "픽업" in model:
            return "트럭"
        elif "세단" in model:
            return "세단"
        else:
            return "알 수 없음"

# 1. 클래스 메서드 사용
# 인스턴스를 생성할 때마다 클래스 변수인 total_cars가 자동으로 증가합니다.
car1 = Car("현대", "쏘나타", 2020, "검정")
car2 = Car("기아", "K5", 2022, "흰")

# 클래스 이름을 통해 전체 생성 대수를 확인합니다.
print(Car.count_cars())  # 출력: 지금까지 2대의 자동차가 생성되었습니다.

# 2. 팩토리 메서드 사용
# 복잡한 생성 로직을 캡슐화한 메서드를 통해 특정 타입의 객체를 쉽게 만듭니다.
sedan = Car.create_sedan("현대", 2023, "파랑")

# 3. 정적 메서드 사용
# 객체 생성 없이 클래스 이름을 통해 유틸리티 기능을 바로 사용합니다.
print(Car.is_valid_year(2025))          # 출력: False (미래 연식)
print(Car.is_valid_year(1950))          # 출력: True
print(Car.get_vehicle_type("싼타페 SUV")) # 출력: SUV
print(Car.get_vehicle_type("쏘나타 세단")) # 출력: 세단

# ================================================================

class Resource:
    """리소스 할당 및 해제를 관리하는 클래스"""

    def __init__(self, name):
        # 객체가 생성될 때 호출되는 생성자입니다.
        self.name = name
        print(f"{self.name} 리소스가 초기화되었습니다.")

    def __del__(self):
        # 객체가 메모리에서 제거될 때 호출되는 소멸자입니다.
        print(f"{self.name} 리소스가 해제되었습니다.")

    def use(self):
        # 리소스 사용을 시뮬레이션하는 인스턴스 메서드입니다.
        print(f"{self.name} 리소스를 사용 중입니다.")

# 1. 객체 생성
# Resource 인스턴스를 만들면 __init__이 자동으로 실행됩니다.
res = Resource("데이터베이스")

# 2. 메서드 사용
res.use()

# 3. 객체 제거
# 변수에 None을 할당하여 참조를 끊으면, 가비지 컬렉터에 의해 __del__이 호출됩니다.
res = None

# ================================================================
# 상속

class Vehicle: # 부모 클래스
```

```
    def __init__(self, name):
        self.name = name

    def move(self):
        print("이동합니다.")

class Car(Vehicle): # Vehicle을 상속받은 자식 클래스
    super().__init__()

    def honk(self):
        print("빵빵!")

my_car = Car()
my_car.move()  # 부모의 기능을 사용 [출력: 이동합니다.]
my_car.honk()  # 자신의 기능을 사용 [출력: 빵빵!]

# ================================================================

class Animal:
    def __init__(self, name):
        self.name = name

    def make_sound(self):
        """동물이 소리를 냅니다."""
        pass  # 구현하지 않음 (추상 메서드와 유사)

class Dog(Animal):
    def make_sound(self):  # 오버라이딩
        return f"{self.name}(이)가 멍멍 짖습니다."

class Cat(Animal):
    def make_sound(self):
        return f"{self.name}(이)가 야옹 웁니다."

class Duck(Animal):
    def make_sound(self):
        return f"{self.name}(이)가 꽥꽥 소리를 냅니다."

# ================================================================

from abc import ABC, abstractmethod

class Shape(ABC):
    @abstractmethod
    def area(self):
        """도형의 넓이를 계산합니다."""
        pass

    @abstractmethod
    def perimeter(self):
        """도형의 둘레를 계산합니다."""
        pass

    def describe(self):
        """도형을 설명합니다. (선택적 구현)"""
        return f"이것은 도형입니다."

class Rectangle(Shape):
    def __init__(self, width, height):
        # 가로(width)와 세로(height) 길이를 초기화합니다.
```

```
        self.width = width
        self.height = height

    def area(self):
        # 추상 메서드 area를 구현하여 직사각형의 넓이를 계산합니다.
        return self.width * self.height

    def perimeter(self):
        # 추상 메서드 perimeter를 구현하여 직사각형의 둘레를 계산합니다.
        return 2 * (self.width + self.height)

    # describe는 선택적으로 오버라이드 가능
    def describe(self):
        # 부모 클래스의 메서드를 재정의하여 직사각형의 크기 정보를 반환합니다.
        return f"이것은 {self.width}x{self.height} 크기의 직사각형입니다."

# 추상 클래스는 인스턴스화할 수 없음
try:
    # Shape는 추상 클래스이므로 직접 객체를 생성할 수 없습니다.
    shape = Shape()  # TypeError 발생
except TypeError as e:
    # 발생한 오류 메시지를 출력합니다.
    print(f"오류: {e}")

# 구현 클래스 사용
# Shape를 상속받아 모든 추상 메서드를 구현한 Rectangle 클래스를 사용합니다.
rect = Rectangle(5, 3)

# 구현된 메서드들을 호출하여 결과를 출력합니다.
print(f"직사각형 넓이: {rect.area()}")
print(f"직사각형 둘레: {rect.perimeter()}")
print(rect.describe())
```

▼ 2) 객체지향-은행 계좌 실습

```
class BankAccount:
    interest_rate = 0.01

    def __init__(self, owner, balance=0):
        self.owner = owner
        self.balance  = balance
        self.transaction_history = []

        self._log_transaction("계좌 개설", balance)

    def deposit(self, amount):
        if amount <= 0:
            print ("입금액은 0보다 커야합니다.")
            return False

        self.balance += amount
        self._log_transaction("입금", amount)
        print(f"{amount:,}원이 입금되었습니다. 현재 잔액: {self.balance:,}원")
        return True

    def withdraw(self, amount):
        if amount <= 0:
            print("출금액은 0보다 커야합니다.")
            return False
```

```
        if amount > self.balance:
            print(f"잔액 부족. 현재 잔액: {self.balance:,}원")
            return False

        self.balance -= amount
        self._log_transaction("출금", amount)
        print(f"{amount:,}원이 출금되었습니다. 현재 잔액: {self.balance:,}원")

    def get_balance(self):
        print(f"{self.owner}님의 계좌 잔액: {self.balance:,}원")
        return self.balance

    def apply_interest(self):
        interest = self.balance * BankAccount.interest_rate
        self.balance += interest

        self._log_transaction("이자", interest)
        print(f"이자 {interest:,.2f}원이 추가되었습니다. 현재 잔액: {self.balance:,}원")

    def _log_transaction(self, transaction_type, amount):
        import datetime

        timestamp = datetime.datetime.now().strftime("%Y-%m-%d %H:%M:%S")

        self.transaction_history.append({
            "type": transaction_type,
            "amount": amount,
            "timestamp": timestamp,
            "balance": self.balance
        })

    def print_transaction_history(self):
        print(f"\n{self.owner}님의 거래 내역:")
        print("-" * 60)
        print(f"{'일시':<20} {'종류':<10} {'금액':<15} {'잔액':<15}")
        print("-" * 60)

        for transaction in self.transaction_history:
            print(f"{transaction['timestamp']:<20} "
              f"{transaction['type']:<10} "
              f"{transaction['amount']:,}원".ljust(15) +
              f"{transaction['balance']:,}원".ljust(15))
        print("-" * 60)

# 1. 계좌 생성
# 초기 잔액을 지정하거나 기본값(0)을 사용하여 인스턴스를 생성합니다.
my_account = BankAccount("홍길동", 1000000)
your_account = BankAccount("김철수")

# 2. 계좌 조작
my_account.get_balance()     # 결과: 홍길동님의 계좌 잔액: 1,000,000원
my_account.deposit(500000)   # 결과: 500,000원이 입금되었습니다. 현재 잔액: 1,500,000원
my_account.withdraw(200000)  # 결과: 200,000원이 출금되었습니다. 현재 잔액: 1,300,000원
my_account.withdraw(2000000) # 결과: 잔액 부족. 현재 잔액: 1,300,000원

# 3. 이자 적용
# 클래스 변수 interest_rate(0.01)를 기준으로 이자를 계산하여 합산합니다.
my_account.apply_interest()  # 결과: 이자 13,000.00원이 추가되었습니다. 현재 잔액: 1,313,000원
```

```
# 4. 거래 내역 출력
# 지금까지의 모든 활동 로그를 표 형식으로 출력합니다.
my_account.print_transaction_history()
```

▼ 3) SOLID

1. Solid Responsiblity Principle: 단일 책임 원칙
   - 한 클래스는 하나의 책임만 가져야 함
   - 클래스를 변경해야 하는 이유는 오직 하나뿐이어야 함
2. Opne Close Principle: 개방 폐쇄 원칙
   - 확장에는 열려 있으나 수정에는 닫혀 있어야 함
   - 기존 코드를 변경하지 않고도 기능을 확장할 수 있어야 함
3. Liskov Substitution Principle: 리스코프 치환 법칙
   - 자식 클래스는 부모 클래스의 역할을 대체할 수 있어야 함
   - 부모 클래스의 인스턴스 대신 자식 클래스의 인스턴스를 사용해도 프로그램이 올바르게 동작해야 함
4. Interface Segregation Principle: 인터페이스 분리 법칙
   - 클라이언트가 자신이 사용하지 않는 메서드에 의존하지 않아야 함
   - 하나의 일반적인 인터페이스보다 여러 개의 구체적인 인터페이스가 나음
5. Dependeny Inversion Principle: 의존성 역전 법칙
   - 추상화에 의존해야 하며, 구체화에 의존하면 안 됨
   - 고수준 모듈은 저수준 모듈의 구현에 의존해서는 안 됨