심화프로그래밍

8. 객체와 클래스

교수: 이은헌

강의 목표

- 객체 지향 프로그래밍(OOP)의 기본 개념을 이해한다.
- 클래스(Class)와 객체(Object)의 차이를 설명할 수 있다.
- 파이썬에서 클래스를 정의하고 객체를 생성할 수 있다.
- private 속성의 개념과 필요성을 이해한다.
- 상속(Inheritance)의 개념을 이해하고 구현할 수 있다.

목차

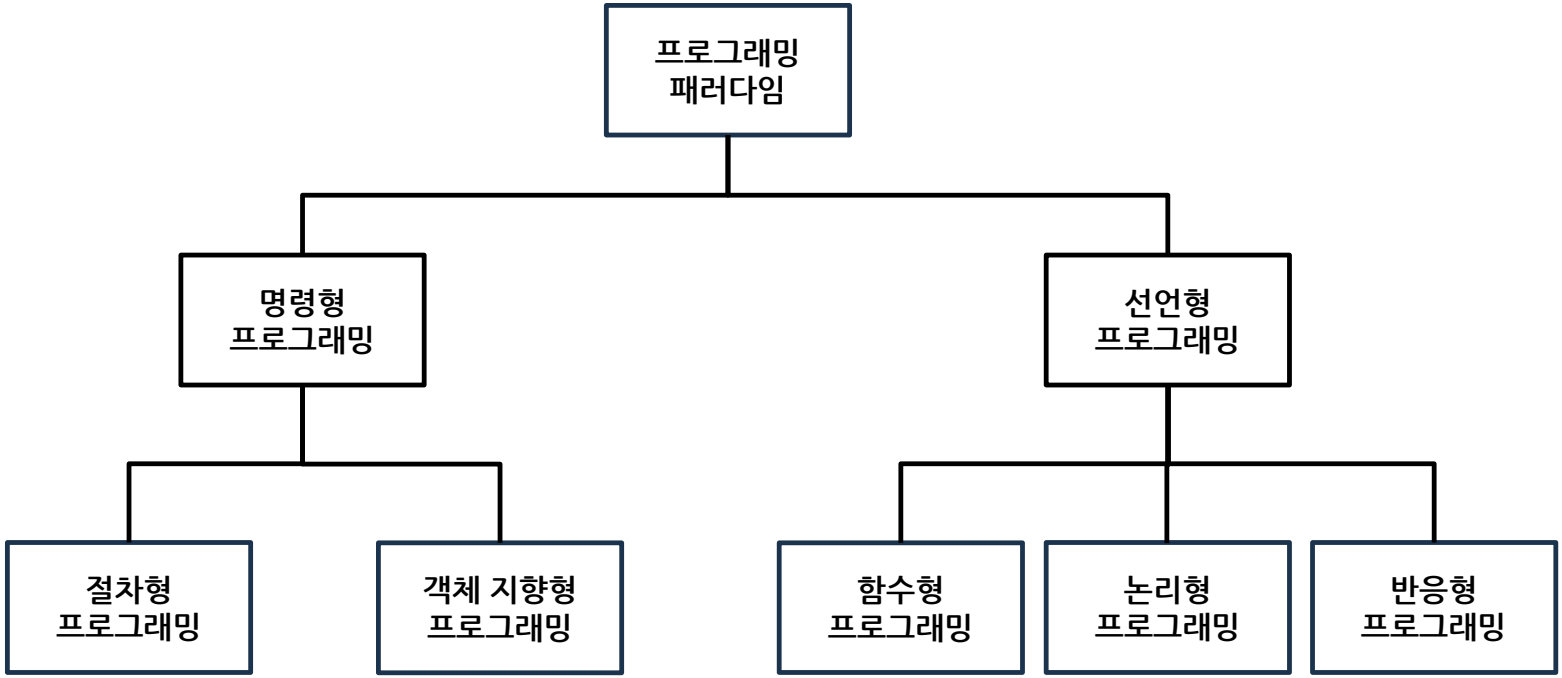
심화: 프로그래밍 패러다임

프로그래밍 패러다임

- 프로그래머가 프로그래밍을 할 때 사용하는 '사고의 방식', '스타일', 또는 '접근법'
  - **어떻게 코드를 구성하고, 어떻게 문제를 해결할 것인가**에 대한 전략 또는 철학

- 프로그래밍 패러다임은 크게 **어떻게**를 중시하는 **명령형**과, **무엇**을 중시하는 **선언형**으로 나뉜다.
  - 명령형 프로그래밍
    - 어떻게 문제를 해결할 것인가?
    - 컴퓨터가 수행해야 할 명령(절차)을 순서대로 기술
  - 선언형 프로그래밍
    - 이 프로그램한테 무엇을 원하는가?
    - 수행하는 방식은 컴퓨터(컴파일러)에 위임하고, 프로그램이 무엇을 하는지만 선언

프로그래밍 패러다임


프로그래밍
패러다임
명령형
프로그래밍
선언형
프로그래밍
절차형
프로그래밍
객체 지향형
프로그래밍
함수형
프로그래밍
논리형
프로그래밍
반응형
프로그래밍

명령형 프로그래밍

- 명령형 대표적인 두 가지 패러다임은 **절차형 프로그래밍**과 **객체 지향 프로그래밍**이다.

- **절차형 프로그래밍 (Procedural Programming)**
  - 프로그램의 순차적인(절차적인) 흐름을 중요시한다.
  - 데이터(변수)와 기능(함수)이 분리되어 관리된다.

- **객체 지향 프로그래밍 (Object-Oriented Programming, OOP)**
  - 프로그램을 '객체(Object)'들의 집합으로 본다.
  - 데이터(속성)와 기능(메서드)을 하나의 '객체'로 묶어서 관리한다.

절차형 프로그래밍

- 프로그램이 순차적인(절차적인) 흐름에 따라 실행되는 방식
  - **"무엇을" "어떤 순서로"** 실행할 것인지가 핵심
- 프로그램이 함수(Function)의 집합으로 구성된다.

- C언어가 대표적인 절차 지향 프로그래밍 언어

```c
// C 언어 예시 (절차 지향)
// 1. 데이터를 정의 (데이터)
int a = 10;
int b = 20;

// 2. 기능을 함수로 정의 (절차)
int add(int x, int y) {
    return x + y;
}

// 3. 순서에 따라 함수 호출 (흐름)
void main() {
    int result = add(a, b);
    printf("%d", result);
}
```

절차형 프로그래밍

- 절차형의 가장 큰 특징은 **데이터와 기능(함수)이 분리**되어 있다는 점이다.
  - 데이터(변수)는 특정 함수에 종속되지 않고, 여러 함수에서 접근하여 사용되는 경우가 많다.

- 프로그램의 규모가 커지고 복잡해 질수록 한계가 드러난다.
  - **낮은 유지보수성:** 데이터와 함수가 분리되어 있어, 데이터 구조가 변경되면 이 데이터를 사용하는 모든 함수를 찾아서 수정해야 한다.
  - **데이터 무결성 문제:** 여러 함수가 공용 데이터에 접근할 수 있어, 누가 데이터를 잘못 변경했는지 추적하기 어렵다.
  - **낮은 재사용성:** 특정 데이터 구조에 강하게 결합된 함수는 다른 곳에서 재사용하기 어렵다.

절차형 프로그래밍

```python
alice_account = {'owner': 'Alice', 'balance': 1000}
bob_account = {'owner': 'Bob', 'balance': 500}

def transfer_procedural(from_account, to_account, amount):
    # 'from_account'의 잔액을 확인하고, 'from_account'에서 출금
    if from_account['balance'] >= amount:
        from_account['balance'] -= amount
        to_account['balance'] += amount
        print(f"이체 성공!")
        print(f"결과: {from_account['owner']} 잔액 {from_account['balance']}, {to_account['owner']} 잔액 {to_account['balance']}")
    else:
        print(f"이체 실패: {from_account['owner']}의 잔액 부족")

print(f"초기 상태: {alice_account}, {bob_account}")

print("-"*40)
print("Alice가 Bob에게 300원 전송")
transfer_procedural(alice_account, bob_account, 300)

# 만약 프로그래머가 실수로 'from_account'와 'to_account'를 혼동한다면?
print("-"*40)
print("Alice가 Bob에게 500원 전송")
transfer_procedural(bob_account, alice_account, 500)
```

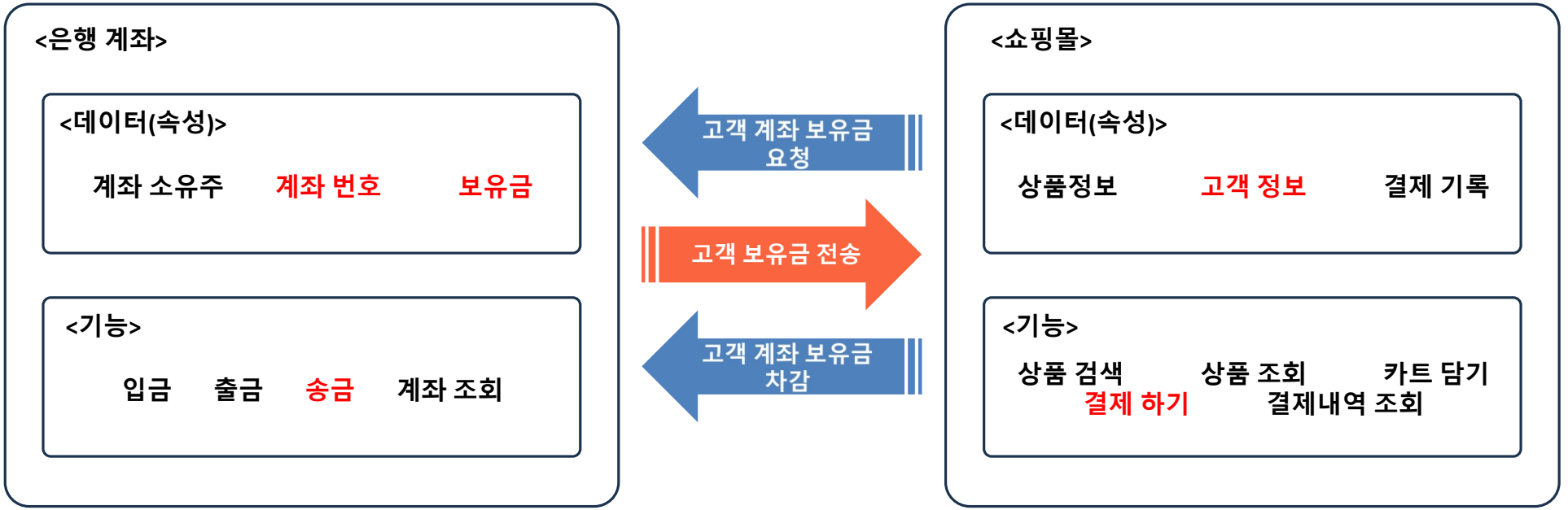
```
초기 상태: {'owner': 'Alice', 'balance': 1000}, {'owner': 'Bob', 'balance': 500}
----------------------------------------
Alice가 Bob에게 300원 전송
이체 성공!
결과: Alice 잔액 700, Bob 잔액 800
----------------------------------------
Alice가 Bob에게 500원 전송
이체 성공!
결과: Bob 잔액 300, Alice 잔액 1200
```

객체 지향 프로그래밍

- 프로그램을 **'객체(Object)'**들의 집합으로 본다.
- 객체들은 서로 메시지를 주고받으며 상호작용한다.

- OOP의 가장 큰 특징은 **데이터(속성)와 기능(메서드)**을 하나의 '객체'로 묶는다는 점이다.
  - **"누가" "무슨 일을"** 할 것인지가 핵심이다.

객체 지향 프로그래밍


<은행 계좌>
<데이터(속성)>
계좌 소유주
계좌 번호
보유금
<기능>
입금
출금
송금
계좌 조회
고객 계좌 보유금 요청
고객 보유금 전송
고객 계좌 보유금 차감
<쇼핑몰>
<데이터(속성)>
상품정보
고객 정보
결제 기록
<기능>
상품 검색
상품 조회
카트 담기
결제 하기
결제내역 조회

객체 지향 프로그래밍

- **높은 유지보수성**
  - 객체 내부의 데이터(속성)는 숨겨져 있고, 오직 허용된 메서드를 통해서만 접근 가능하다. (데이터 무결성 향상)
  - 수정이 필요할 때, 해당 객체 내부만 수정하면 되므로 파급 효과가 적다.
- **높은 재사용성**
  - 객체는 독립적인 단위(부품)이므로, 다른 프로그램에서 쉽게 가져다 쓸 수 있다.
  - **상속(Inheritance)**을 통해 기존 객체의 기능을 확장하여 **재사용**할 수 있다.
- **모듈화**
  - 프로그램을 여러 개의 독립적인 객체 단위로 나누어 설계할 수 있다.

파이썬과 패러다임

- 파이썬은 **다중 패러다임 (Multi-paradigm)** 언어이다.
  - 하나의 스타일만 강요하지 않는다.
  - 지금까지 파이썬을 이용해서 절차 지향 방식으로 코드를 작성했다.

- 하지만, 파이썬의 가장 핵심적인 철학은 **객체 지향 패러다임**이다.

파이썬과 패러다임

- 파이썬의 **모든 것은 객체(Object)**이다. (파이썬에서 제공하는 기본 변수들도 전부 객체임)

[1]:

```
int_var = 10
float_var = 3.14
str_var = "string"
bool_var = True

print(type(int_var))
print(type(float_var))
print(type(str_var))
print(type(bool_var))
```

```
<class 'int'>
<class 'float'>
<class 'str'>
<class 'bool'>
```

객체와 클래스

객체

- **객체(Object)란?**
  - 세상의 모든 것 (실존하는 것, 추상적인 것)을 표현하는 단위이다.
  - 예 (실존): 자동차, 사람, 고양이, 책상
  - 예 (추상): 주문, 계좌, 게임 캐릭터

- 객체는 두 가지 주요 요소로 구성된다.
  - **속성 (Attribute):** 객체의 상태나 데이터. **(명사적 특징)**
  - **메서드 (Method):** 객체가 수행할 수 있는 동작이나 기능. **(동사적 특징)**

객체

- 예시) 자동차 객체
  - **속성 (Attribute)**
    - 색상 (color)
    - 현재 속도 (speed)
    - 모델명 (model name) 등
  - **메서드 (Method)**
    - 전진 (move forward), 후진 (move backward)
    - 브레이크 (break)
    - 가속 (accelerate) 등

클래스

- **클래스(Class)란?**
  - 객체를 만들어내기 위한 **설계도 또는 틀(template)**이다.
  - 객체가 가져야 할 **속성(Attribute)**과 **메서드(Method)**를 정의한다.

- **객체 vs 클래스**
  - **클래스 (Class):** 객체의 설계도. 어떤 속성을 지녀야하는지, 어떤 작업을 수행할 수 있는지를 정의한다.
  - **객체 (Object):** 클래스를 바탕으로 실체화 된 구현물. 각 객체마다 다른 속성을 가질 수 있다.

클래스와 개체

- 클래스 = 붕어빵 틀 / 객체 = 붕어빵
  - 하나의 '붕어빵 틀'(클래스)로 여러 개의 '붕어빵'(객체)을 찍어낼 수 있다.
  - 모든 붕어빵은 같은 모양(속성과 메서드)을 가지지만, 속(팥, 슈크림 등)은 다를 수 있다.

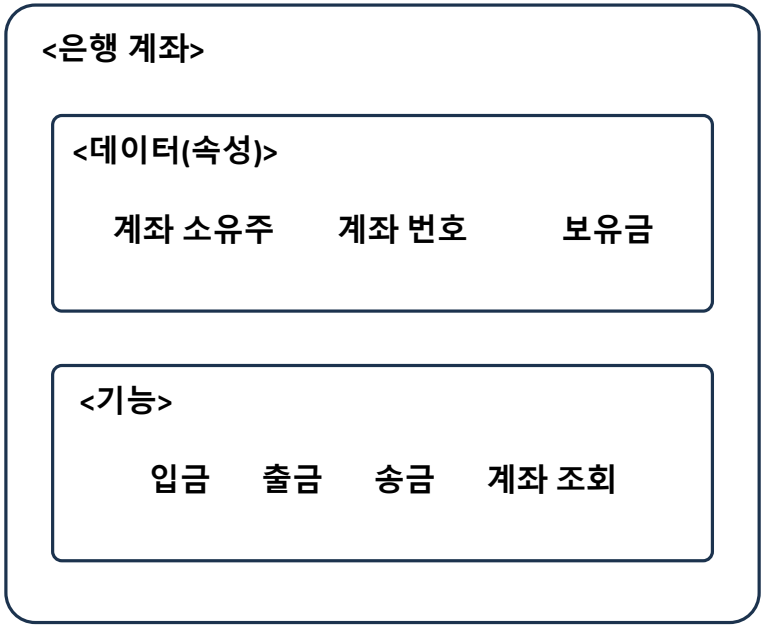
클래스 생성

- class 키워드를 사용하여 정의하며, 클래스 내부에는 속성과 메서드를 정의할 수 있다.
  - 속성에는 클래스에서 필요한 데이터(변수)를, 메서드는 클래스가 사용할 수 있는 기능을 정의

```
class 클래스명:
    # 클래스에서 사용할 속성(데이터)를 정의
    attribute1
    attribute2
    ...

    def 메서드1(매개 변수명):
        ...

    def 메서드1(매개 변수명):
        ...
```

클래스 생성 예시

- 은행 계좌(Account)라는 클래스를 생성한다고 가정하면,
- 이 클래스의 **속성**에는
  - **계좌 소유주**를 저장하는 문자열 데이터
  - **계좌 번호**를 저장하는 문자열 데이터
  - **보유금**을 저장하는 정수형 데이터가 있으면 된다.
- 이 클래스의 **기능(메서드)**에는
  - 돈을 계좌에 추가하는 **입금**
  - 계좌에서 돈을 꺼내는 **출금**
  - 다른 사람에게 돈을 보내는 **송금**
  - 계좌 정보를 조회하는 **계좌 조회**가 있으면 된다.



클래스 생성 예시

```python
class Account:
    owner = None
    number = None
    balance = 0

    def deposit(self, money):
        # 입금 기능 구현
        pass

    def withdraw(self, money):
        # 출금 기능 구현
        pass

    def transfer(self, account, money):
        # 송금 기능 구현
        pass

    def inquire(self):
        # 계좌 조회 기능 구현
        pass
```

<은행 계좌>

<데이터(속성)>

계좌 소유주 계좌 번호 보유금

<기능>

입금 출금 송금 계좌 조회

심화: None

- 파이썬에서 변수명만 정의를 하고 값은 나중에 채워 넣고 싶을 때 사용하는 키워드
- 해당 변수에 값이 있는지 없는지 알고 싶으면 None과 비교하면 된다.

[1]:
```
1 a
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[1], line 1
----> 1 a

NameError: name 'a' is not defined
```

[2]:
```
1 a = None
2 b = 10
3
4 print(a == None)
5 print(b == None)
```

```
True
False
```

심화: pass

- '아무것도 하지 말라'는 의미의 문법적 구문
- 문법적으로 코드가 필요하나, 실행할 동작이 없을 때 사용
  - 함수, 클래스, if문 블록 등의 빈 본문 채우기

[1]:
```
a = 10

if a == 10:

print(a)
```

```
  Cell In[1], line 5
    print(a)
    ^
IndentationError: expected an indented block after 'if' statement on line 3
```

[2]:
```
a = 10

if a == 10:
    pass

print(a)
```

```
10
```

객체 생성하기

- 클래스를 정의했다면(설계도), 이제 객체(실체)를 생성할 수 있다.
- 객체 생성 문법은 변수 생성이나 함수 호출과 유사하다.
- *객체명 = 클래스명()*

```
class Account: ●●●
```

[2]:
```
1 alice_account = Account()
2 bob_account = Account()
```

속성 접근

- 객체의 속성에 접근할 때는 점(.)을 사용한다.
- *객체.속성*
  - 이렇게 불러오고 변수처럼 사용하면 된다.

[2]:
```
alice_account = Account()
bob_account = Account()
```

[3]:
```
alice_account.owner = "Alice"
bob_account.owner = "Bob"

print(f"{alice_account.owner}'s balance = {alice_account.balance}")
print(f"{bob_account.owner}'s balance = {bob_account.balance}")
```

```
Alice's balance = 0
Bob's balance = 0
```

메서드 생성

- 클래스 내부에 함수(function)를 정의하면, 이 함수는 해당 클래스만 사용할 수 있는 기능인 **메서드**가 된다.
- **중요: 클래스 내부의 메서드는 반드시 첫 번째 매개변수로 self를 가져야 한다.**

```python
class Account:
    owner = None
    number = None
    balance = 0

    def deposit(self, money):
        self.balance += money
        print(f"계좌에 {money}원이 입금되었습니다.\n현재 계좌의 보유금은 총 {self.balance}원 입니다.")

    def withdraw(self, money):
        if self.balance >= money:
            self.balance -= money
            print(f"계좌에서 {money}원이 출금되었습니다.\n현재 계좌의 보유금은 총 {self.balance}원 입니다.")
        else:
            print(f"잔액이 부족하여 계좌에서 {money}원을 출금하는데 실패했습니다.")
```

self

- self는 객체 **자기 자신을 나타내는 매개변수**이다.
- 메서드를 호출하면, self에 대한 매개변수를 입력으로 명시하지 않더라도 파이썬은 객체 자신을 self라는 이름으로 자동으로 넘겨준다.
- 메서드 내에서 self를 사용하여 해당 객체의 속성에 접근할 수 있다.

```python
class Account:
    owner = None
    number = None
    balance = 0

    def deposit(self, money):
        self.balance += money
        print(f"계좌에 {money}원이 입금되었습니다.\n현재 계좌의 보유금은 총 {self.balance}원 입니다.")

    def withdraw(self, money):
        if self.balance >= money:
            self.balance -= money
            print(f"계좌에서 {money}원이 출금되었습니다.\n현재 계좌의 보유금은 총 {self.balance}원 입니다.")
        else:
            print(f"잔액이 부족하여 계좌에서 {money}원을 출금하는데 실패했습니다.")
```

메서드 호출

- *객체.메서드명(매개변수)* 형태로 호출한다.
  - self는 인수로 전달하지 않는다는 점에 주의

```
class Account: ...
```

[2]:
```
alice_account = Account()
alice_account.owner = "Alice"

alice_account.withdraw(1000)
alice_account.deposit(5000)
alice_account.withdraw(2000)
```

```
잔액이 부족하여 계좌에서 1000원을 출금하는데 실패했습니다.
계좌에 5000원이 입금되었습니다.
현재 계좌의 보유금은 총 5000원 입니다.
계좌에서 2000원이 출금되었습니다.
현재 계좌의 보유금은 총 3000원 입니다.
```

생성자

- 아래의 Account 클래스는 속성의 값에 None을 넣어놓고 객체를 생성한 뒤에 속성을 새로 정의하고 있지만, 이런 식의 코드는 프로그래머의 실수를 유발할 수 있다.(실수로 클래스 속성 초기화 누락)
- 계좌를 생성할 때 계좌 번호와 계좌주에 대한 데이터가 정해져 있는 상태로 새로운 계좌를 생성하는 것처럼, 객체를 생성하는 순간 객체의 속성을 정의해야 할 때가 있다.
- 이 역할을 하는 특수한 메서드를 **생성자**라고 한다.

```
1 class Account:
2     owner = None
3     number = None
4     balance = 0
5 
6 alice_account = Account()
7 alice_account.owner = "Alice"
8 alice_account.number = "xxx-xx-xxxxxxx"
```

생성자

- 파이썬의 생성자는 클래스 내부에 정의된 __init__라는 특별한 메서드이다.
- __init__ 메서드는 객체가 생성될 때 동시에 호출된다.
  - __init__의 목적: 객체의 속성을 초기화하는 것.

```
class 클래스이름:
    def __init__(self, 매개변수1, ...):
        # 속성 초기화
        self.속성1 = 값1
        self.속성2 = 매개변수1
```

지정된 초기값으로 속성 초기화

- __init__ 메서드 내에서 self.속성 = 값 형태로 초기화할 수 있다.
- __init__ 메서드를 사용하지 않고 속성 = 값으로도 초기화할 수 있지만, __init__에서 초기화를 선언하는 것을 권장

```
1 class Account:
2     owner = None
3     number = None
4     balance = 0
```

```
1 class Account:
2     owner = None
3     number = None
4
5     def __init__(self):
6         self.balance = 0
```

매개변수로 속성 초기화

- 객체를 생성할 때 원하는 값을 전달하여 속성을 초기화 할 수 있다.
- __init__ 메서드를 생성할 때 self 이후 매개변수를 추가

```python
class Account:
    owner = None
    number = None

    def __init__(self):
        self.balance = 0

alice_account = Account()
alice_account.owner = "Alice"
alice_account.number = "xxx-xx-xxxxxxx"
```

```python
class Account:
    def __init__(self, owner, number, balance=0):
        self.owner = owner
        self.number = number
        self.balance = balance

alice_account = Account("Alice", "xxx-xx-xxxxxxx")
```

매개변수의 기본값 설정

- 생성자 매개변수에 기본값을 지정할 수 있다.
- 메서드의 매개변수에 =을 적고 기본값을 지정하면, 해당 값을 입력하지 않더라도 해당 값을 가지고 초기화를 할 수 있다. (기본값 설정은 생성자 뿐만 아니라 모든 메서드 및 함수에 전부 사용 가능하다)

```python
class Account:
    def __init__(self, owner, number, balance=0):
        self.owner = owner
        self.number = number
        self.balance = balance

alice_account = Account("Alice", "xxx-xx-xxxxxxx")
bob_account = Account("Bob", "yyy-yy-yyyyyyy", 10000)
print(alice_account.balance)
print(bob_account.balance)
```

```
0
10000
```

속성의 접근성

- 객체가 가진 속성은 **기본적으로 공개(Public)**이다.
  - 즉, 객체.속성을 통해 **외부에서 자유롭게 접근(읽기/쓰기)**이 가능하다.

```python
alice_account = Account("Alice", "xxx-xx-xxxxxxx", 10000000)

print(f"계좌주={alice_account.owner}, 계좌번호={alice_account.number}, 보유액={alice_account.balance}")
alice_account.owner = "Bob"
alice_account.balance = -1000000
print(f"계좌주={alice_account.owner}, 계좌번호={alice_account.number}, 보유액={alice_account.balance}")
```

```
계좌주=Alice, 계좌번호=xxx-xx-xxxxxxx, 보유액=10000000
계좌주=Bob, 계좌번호=xxx-xx-xxxxxxx, 보유액=-1000000
```

공개 속성의 문제점

- 객체 외부에서 속성을 마음대로 변경하면, 속성에 원하지 않는 변화가 생길 수 있는 문제를 안고 있다.
  - 예시
    - 계좌의 보유액을 외부에서 마음대로 수정하다가 보유액이 음수가 됨.
    - 계좌 번호를 외부에서 마음대로 수정하다가, 동일한 계좌번호를 가진 계좌가 2개가 생김 등

- **해결책: 정보 은닉/캡슐화 (Encapsulation)**
  - 마음대로 변해서는 안 되는 중요한 속성은 외부에서 직접 접근하지 못하도록 숨기고,
  - 숨겨진 속성은 검증된 메서드를 통해서만 접근하도록 허용한다.

속성 숨기기

- 속성 이름 앞에 밑 줄 2개(__)를 붙이면 해당 속성을 숨길 수 있다. 이러한 속성을 비공개(**private**)라고 한다.
  - 비공개 속성은 클래스 밖에서 직접 해당 속성에 접근하는 것이 불가능하다.

[1]:
```
class Account:
    def __init__(self, owner, number, balance=0):
        self.__owner = owner
        self.__number = number
        self.__balance = balance

alice_account = Account("Alice", "xxx-xx-xxxxxxx", 10000000)

print(f"계좌주={alice_account.__owner}, 계좌번호={alice_account.__number}, 보유액={alice_account.__balance}")
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Cell In[1], line 9
      5         self.__balance = balance
      7 alice_account = Account("Alice", "xxx-xx-xxxxxxx", 10000000)
----> 9 print(f"계좌주={alice_account.__owner}, 계좌번호={alice_account.__number}, 보유액={alice_account.__balance}")

AttributeError: 'Account' object has no attribute '__owner'
```

비공개 속성 접근

- 비공개(private) 속성을 외부에서 읽을 수 있게 하려면, 해당 값을 반환하는 공개(public) 메서드를 제공해야 한다.
- 이러한 메서드를 **접근자(Getter) 메서드**라고 부른다.

[1]:
```python
class Account:
    def __init__(self, owner, number, balance=0):
        self.__owner = owner
        self.__number = number
        self.__balance = balance

    def get_owner(self):
        return self.__owner

    def get_number(self):
        return self.__number

    def get_balance(self):
        return self.__balance

alice_account = Account("Alice", "xxx-xx-xxxxxxx", 10000000)

print(f"계좌주={alice_account.get_owner()}, 계좌번호={alice_account.get_number()}, 보유액={alice_account.get_balance()}")
```

```
계좌주=Alice, 계좌번호=xxx-xx-xxxxxxx, 보유액=10000000
```

비공개 속성 변경

- 비공개(private) 속성을 외부에서 변경할 수 있게 하려면, 해당 값을 변경하는 공개(public) 메서드를 제공해야 한다.
- 이러한 메서드를 **설정자(Setter) 메서드**라고 부른다.

```python
class Account:
    def __init__(self, owner, number, balance=0):
        self.__owner = owner
        self.__number = number
        self.__balance = balance

    def set_owner(self, owner):
        self.__owner = owner

    def set_balance(self, balance):
        self.__balance = balance

alice_account = Account("Alice", "xxx-xx-xxxxxxx", 10000000)
alice_account.set_owner("Bob")
alice_account.set_balance(0)

print(f"계좌주={alice_account.get_owner()}, 계좌번호={alice_account.get_number()}, 보유액={alice_account.get_balance()}")
```

```
계좌주=Bob, 계좌번호=xxx-xx-xxxxxxx, 보유액=0
```

비공개 속성 변경

- 설정자의 핵심: 값을 변경하기 전에 **유효성 검사**를 추가할 수 있다.

```python
class Account:
    def __init__(self, owner, number, balance=0):
        self.__owner = owner
        self.__number = number
        self.__balance = balance

    def set_owner(self, owner):
        if isinstance(owner, str):
            self.__owner = owner
        else:
            print("입력받은 변수가 문자열이 아닙니다.")

    def set_balance(self, balance):
        if isinstance(balance, int):
            if balance >= 0:
                self.__balance = balance
            else:
                print("입력받은 값이 양수가 아닙니다.")
        else:
            print("입력받은 값이 정수가 아닙니다.")
```

유효성 검사

유효성 검사

공개 메서드 / 비공개 메서드

- 속성과 마찬가지로 메서드도 외부에서 접근할 수 있는 공개 메서드와, 내부에서만 사용할 수 있는 비공개 메서드로 나눠진다.

- 비공개 메서드를 왜 사용하는가?
  - **객체 외부에서 알 필요 없는 내부 로직이나 보조 기능을 숨기기 위해**

공개 메서드 / 비공개 메서드

```python
class Account:
    def __init__(self, owner, number, balance=0):
        self.__owner = owner
        self.__number = number
        self.__balance = balance
        self.__log = []

    def __log_transaction(self, kind, amount):
        log = f"[{kind}] {amount}원 (잔액: {self.__balance}원)"
        self.__log.append(log)

    def deposit(self, money):
        self.__balance += money
        self.__log_transaction("입금", money)
        print(f"계좌에 {money}원이 입금되었습니다.\n현재 계좌의 보유금은 총 {self.__balance}원 입니다.")

    def withdraw(self, money):
        if self.__balance >= money:
            self.__balance -= money
            self.__log_transaction("출금", money)
            print(f"계좌에서 {money}원이 출금되었습니다.\n현재 계좌의 보유금은 총 {self.__balance}원 입니다.")
        else:
            self.__log_transaction("출금 실패", money)
            print(f"잔액이 부족하여 계좌에서 {money}원을 출금하는데 실패했습니다.")
```

입출금 내역 로그를 기록하는 메서드는, 입금과
출금시에만 작동하면 되고 외부에서 불러올 이유가 없다.
그렇기 때문에 비공개 메서드로 정의를 한다.

심화: __dict__

- 모든 객체에는 __dict__라는 내장 딕셔너리가 존재한다. 이 딕셔너리는 객체에 저장되어 있는 속성을 딕셔너리 형태로 보여준다.

```
class Account: ...
```

[2]:
```
alice_account = Account("Alice", "xxx-xx-xxxxxxx", 10000000)
```

[3]:
```
pprint(alice_account.__dict__)
```

```
{'_Account__balance': 10000000,
 '_Account__log': [],
 '_Account__number': 'xxx-xx-xxxxxxx',
 '_Account__owner': 'Alice'}
```

심화: PEP8과 클래스

- 예전에도 한 번 다룬 적이 있는 PEP8 규칙은 클래스에 대해서도 몇 가지 가이드라인을 제공하고 있다.

1. 클래스 이름 규칙
   - 클래스 이름은 대문자로 시작하고, 여러 영단어가 이어진 클래스 이름이라면 각 단어의 첫 글자를 대문자로 한다.
   - 예: MyClass, DatabaseConnection, UserProfile 등
2. 공백 규칙
   - 클래스 정의 위아래에는 두 줄의 빈 줄을 둡니다.
   - 클래스 내부의 메서드 사이에는 한 줄의 빈 줄을 둡니다.

실습 1

- 원을 정의하는 클래스인 Circle을 만들고자 한다. Circle 클래스는 다음의 속성과 기능을 가진다고 한다.
  - 객체가 생성될 때 반지름(radius)를 초기 속성값으로 정의한다.
  - 외부에서 반지름을 확인할 수 있는 접근자 메서드 get_radius가 있다.
  - 외부에서 반지름을 변경할 수 있는 설정자 메서드 set_radius가 있다. 반지름은 음수가 될 수 없어서 음수값이 들어오면 메시지를 출력한다.
  - 원의 넓이를 계산해서 반환해주는 get_area 메서드가 있다.
  - 원의 둘레를 계산해서 반환해주는 get_circumference 메서드가 있다.
- 이 정의를 만족하는 Circle 클래스를 완성하시오. (속성은 비공개로 생성)

```
PI = 3.14 ●●●
```

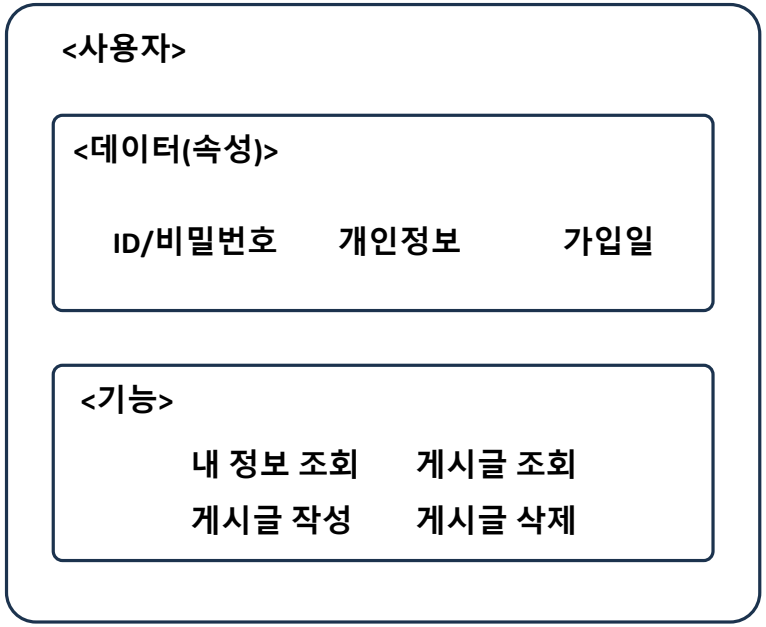
[2]:

```
1 circle = Circle(10)
2
3 print(circle.get_radius())
4 print(circle.get_area())
5 print(circle.get_circumference())
6 circle.set_radius(-10)
7 circle.set_radius(20)
8 print(circle.get_radius())
```

```
10
314.0
62.800000000000004
반지름은 음수가 될 수 없습니다.
20
```

클래스 상속

상속

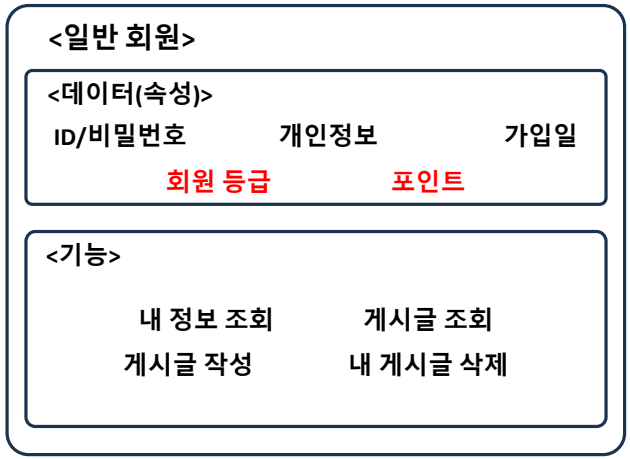


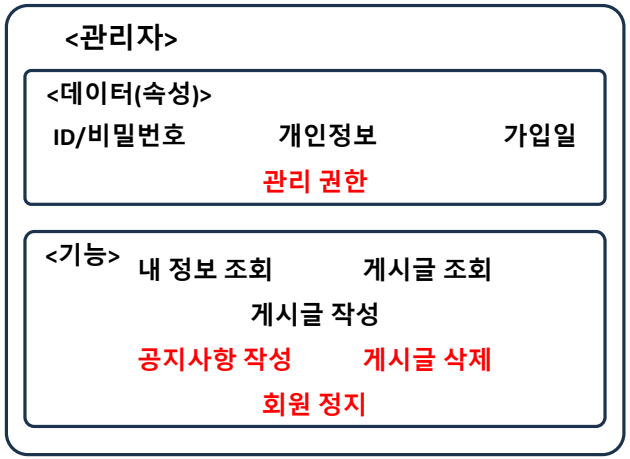
- 어떤 웹사이트 게시판의 사용자들을 User라는 클래스를 이용해서 정의한다고 가정해보자.

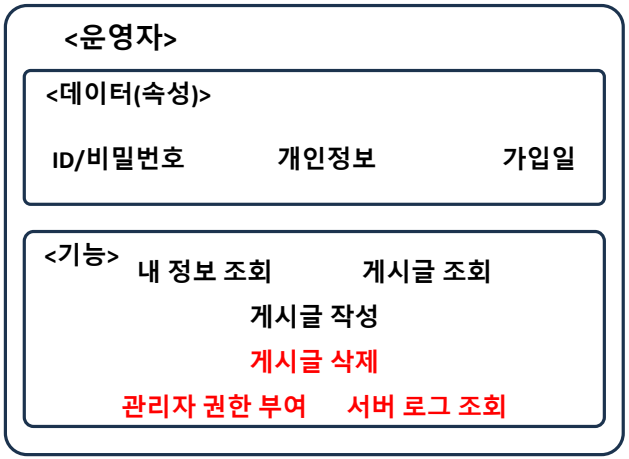
- User는 다음과 같은 속성 및 기능을 가질 수 있다.
  - 속성
    - 사용자 ID/비밀번호
    - 개인 정보 (이메일,전화번호 등)
    - 가입일 등
  - 기능
    - 내 정보 조회
    - 게시글 조회
    - 게시글 작성
    - 내 게시글 삭제 등

상속

- 웹사이트 게시판 사용자를 User로 정의했는데, 게시판 사용자에는 운영자, 관리자, 일반 회원 등의 다양한 사용자가 존재한다.
- 이 때, 각각의 사용자 역할군에 따라 할 수 있는 기능이 따로 있다.

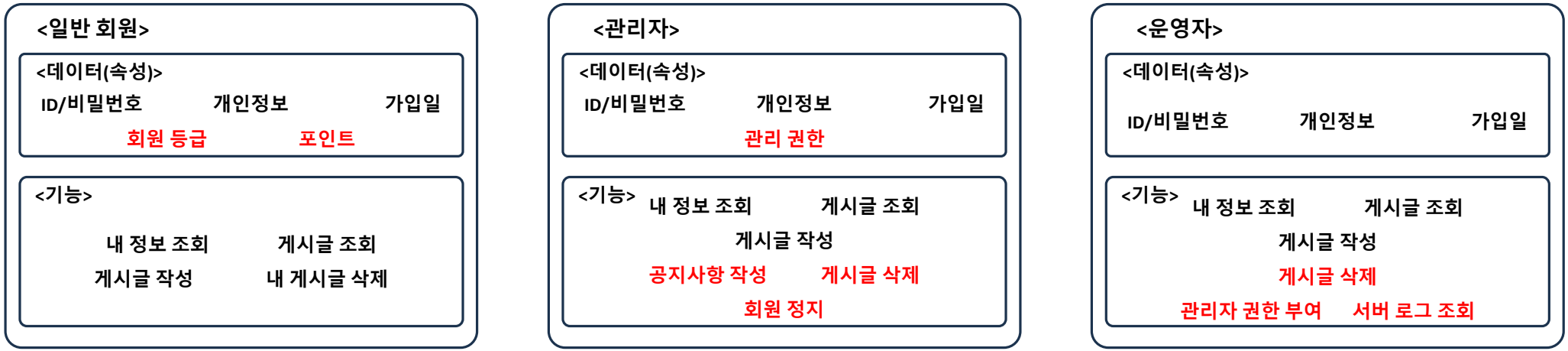






상속

- 이 때, 각 사용자 역할군에 공통으로 존재하는 속성과 기능이 존재하는 것을 확인할 수 있다.
  - 일반 회원, 관리자, 운영자에 대한 클래스를 별도로 생성하면 이러한 중복되는 속성과 기능에 대한 코드를 여러 번 작성해야 할 것이다.
- 이렇게 중복되는 속성과 기능을 여러 번 작성하는 것을 방지하기 위한 기능이 **상속**이다.

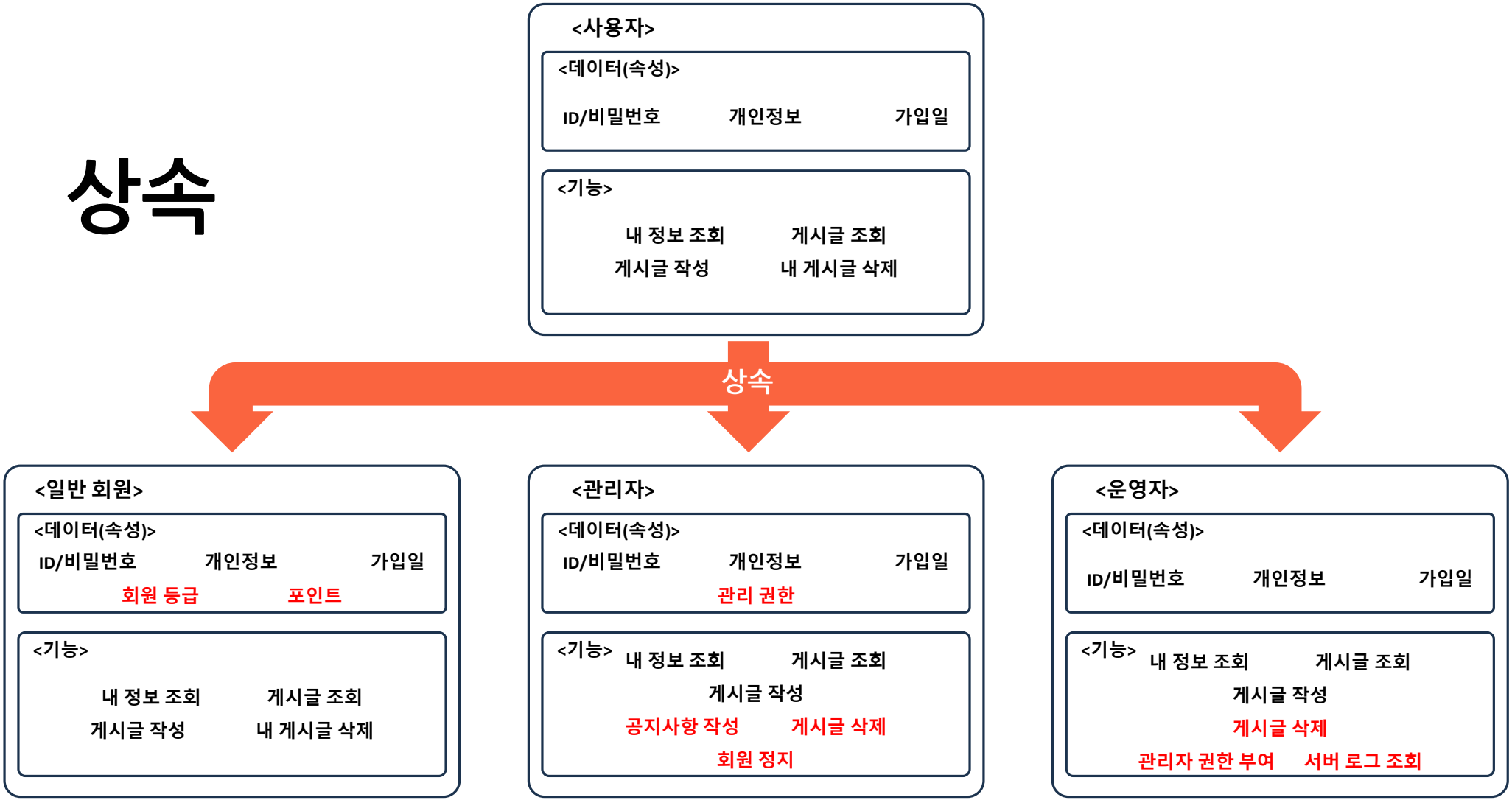


상속

- 상속이란 기존 클래스(부모)의 모든 속성과 메서드를 새로운 클래스(자식)가 물려받아 사용할 수 있게 하는 기능이다.
- 코드의 재사용성(reusability)을 극대화한다.

- **부모 클래스 (Parent Class)**
  - 기능을 물려주는 클래스
- **자식 클래스 (Child Class)**
  - 기능을 물려받는 클래스

상속


<사용자>
<데이터(속성)>
ID/비밀번호 개인정보 가입일
<기능>
내 정보 조회 게시글 조회
게시글 작성 내 게시글 삭제
상속
<일반 회원>
<데이터(속성)>
ID/비밀번호 개인정보 가입일
회원 등급 포인트
<기능>
내 정보 조회 게시글 조회
게시글 작성 내 게시글 삭제
<관리자>
<데이터(속성)>
ID/비밀번호 개인정보 가입일
관리 권한
<기능>
내 정보 조회 게시글 조회
게시글 작성
공지사항 작성 게시글 삭제
회원 정지
<운영자>
<데이터(속성)>
ID/비밀번호 개인정보 가입일
<기능>
내 정보 조회 게시글 조회
게시글 작성
게시글 삭제
관리자 권한 부여 서버 로그 조회

상속

- 상속을 받기 위해서는 클래스를 생성할 때, 상속받을 클래스를 괄호 안에 적어주면 된다.

```
# 부모 클래스
class 부모클래스:
    # ... 속성 및 메서드 ...


# 자식 클래스
# 괄호 안에 부모 클래스 이름을 적는다.
class 자식클래스(부모클래스):
    # ... 추가적인 속성 및 메서드 ...

```

상속

- 아래의 코드는 사람(Person)이라는 부모 클래스를 상속받는 학생(Student)와 교사(Teacher) 클래스의 예시이다

[1]:
```python
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age
        print(f"Person '{self.name}' 생성됨")

    def walk(self):
        print(f"{self.name}이(가) 걷는다.")

    def introduce(self):
        print(f"나는 {self.name}, {self.age}세입니다.")
```

[2]:
```python
class Student(Person):
    pass

class Teacher(Person):
    pass
```

상속

- Student와 Teacher를 생성할 때 walk와 introduce 메서드를 정의하지 않았지만, 부모인 Person으로부터 상속받았기 때문에 walk와 introduce를 사용할 수 있다.

```
class Person: ...
```

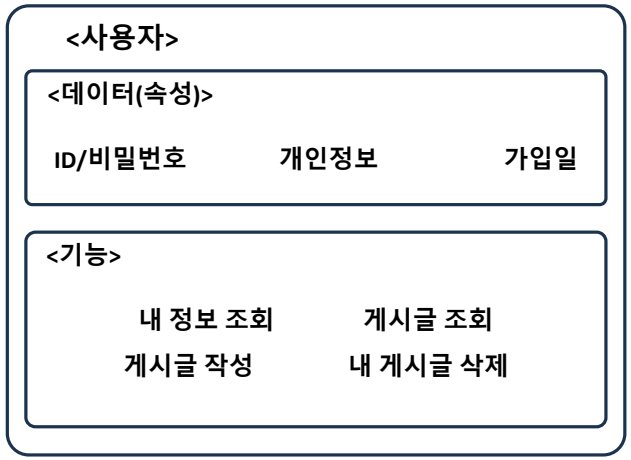
```
class Student(Person): ...
```

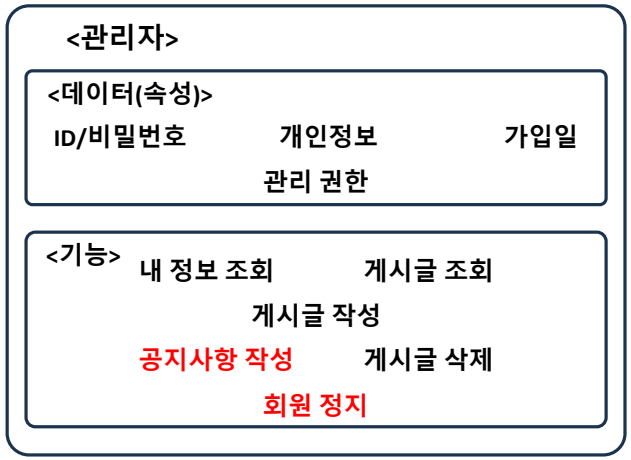
[3]:
```
student = Student("Tom", 10)
teacher = Teacher("Jamie", 40)

student.walk()
student.introduce()
teacher.walk()
teacher.introduce()
```

```
Person 'Tom' 생성됨
Person 'Jamie' 생성됨
Tom이(가) 걷는다.
나는 Tom, 10세입니다.
Jamie이(가) 걷는다.
나는 Jamie, 40세입니다.
```

상속과 기능

- 자식 클래스가 부모 클래스로부터 상속을 받으면, 부모 클래스가 가지고 있는 모든 속성과 기능(메서드)를 전부 물려받는다.
- 만약 자식 클래스가 부모 클래스한테 없는 새로운 기능이 필요하면 어떻게 하면 될까?
  - 새롭게 기능을 추가해주면 된다!





상속과 기능

```
class Person: ...
```

[2]:

```
class Student(Person):
    # Student만의 고유한 기능 추가
    def study(self):
        print(f"{self.name}이(가) 공부한다.")

class Teacher(Person):
    # Teacher만의 고유한 기능 추가
    def teach(self):
        print(f"{self.name}이(가) 가르친다.")
```

Student에 study라는 새로운 메서드 추가

Teacher에 teach라는 새로운 메서드 추가

[3]:

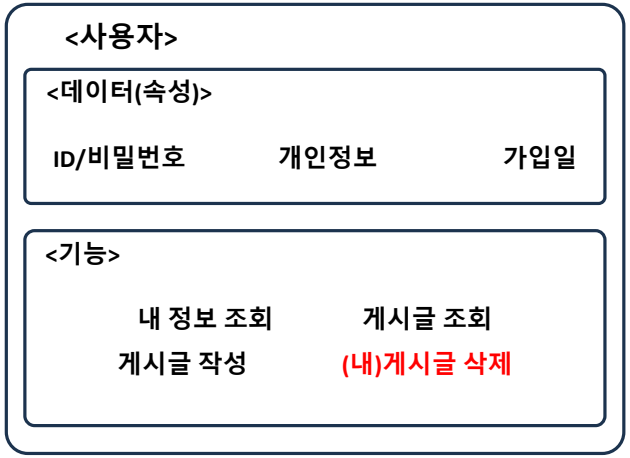
```
student = Student("Tom", 10)
teacher = Teacher("Jamie", 40)

student.walk()
student.introduce()
student.study()
teacher.walk()
teacher.introduce()
teacher.teach()
```

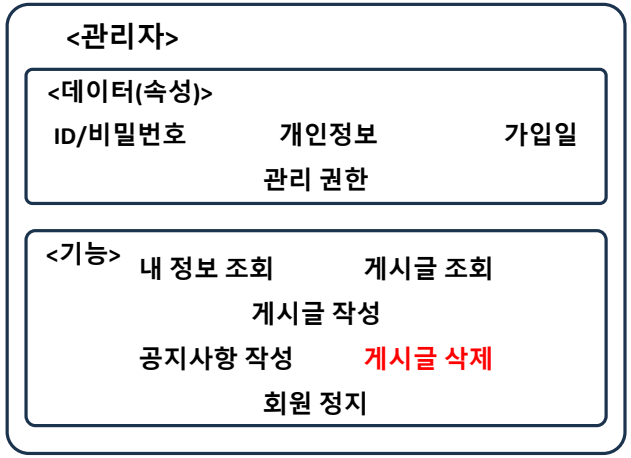
Student 객체는 상속받은 walk와 introduce 뿐만 아니라,
새롭게 정의한 study 메서드도 사용할 수 있게 된다.

```
Person 'Tom' 생성됨
Person 'Jamie' 생성됨
Tom이(가) 걷는다.
나는 Tom, 10세입니다.
Tom이(가) 공부한다.
Jamie이(가) 걷는다.
나는 Jamie, 40세입니다.
Jamie이(가) 가르친다.
```

상속과 기능 재정의

- 자식 클래스가 부모 클래스로부터 상속을 받으면, 부모 클래스가 가지고 있는 모든 속성과 기능(메서드)를 전부 물려받는데, 만약 기존 부모 클래스가 가지고 있는 기능을 새로운 기능으로 바꾸고 싶은 경우에는 어떻게 해야할까?





상속과 기능 재정의

- 자식 클래스가 부모 클래스에 존재하고 있는 메서드과 **동일한 이름을 가진 메서드를 새롭게 정의**하는 경우, 부모 클래스에 있는 메서드는 자식 클래스에서 **새롭게 정의한 메서드로 대체**된다.
- 이를 **메서드 재정의(Override)**라고 한다.

[1]:
```python
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age
        print(f"Person '{self.name}' 생성됨")

    def walk(self):
        print(f"{self.name}이(가) 걷는다.")

    def introduce(self):
        print(f"나는 {self.name}, {self.age}세입니다.")
```

[2]:
```python
class Student(Person):
    # Student만의 고유한 기능 추가
    def study(self):
        print(f"{self.name}이(가) 공부한다.")

    # 부모 클래스(Person)에서 물려받은 introduce 메서드를 재정의
    def introduce(self):
        print(f"나는 학생인 {self.name}, {self.age}세입니다.")
```

[3]:
```python
student = Student("Tom", 10)

student.walk()
student.introduce()
student.study()
```

```
Person 'Tom' 생성됨
Tom이(가) 걷는다.
나는 학생인 Tom, 10세입니다.
Tom이(가) 공부한다.
```

상속과 생성자

- 학생(Student)이라는 클래스는 Person에서 상속받은 name, age 외에 학번(student_id)과 학점(grade) 속성이 필요하다고 가정해보자.

- 어떻게 하면 Student 클래스에 새로운 속성을 정의할 수 있을까?

상속과 생성자

- 앞에서 배운 메서드 재정의(override)를 생성자인 __init__ 메서드에도 적용할 수 있다.

- 그런데 부모에서 name과 age 속성을 이미 정의했는데, 재정의를 할 때 이 부분을 다시 언급할 필요가 있을까?

[1]:
```python
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age
        print(f"Person '{self.name}' 생성됨")

    def walk(self):
        print(f"{self.name}이(가) 걷는다.")

    def introduce(self):
        print(f"나는 {self.name}, {self.age}세입니다.")
```

[2]:
```python
class Student(Person):
    # __init__ 메서드를 재정의
    def __init__(self, name, age, student_id, grade):
        self.name = name
        self.age = age
        self.student_id = student_id
        self.grade = grade
        print(f"Person '{self.name}' 생성됨")

    # Student만의 고유한 기능 추가
    def study(self):
        print(f"{self.name}이(가) 공부한다.")

    # 부모 클래스(Person)에서 물려받은 introduce 메서드를 재정의
    def introduce(self):
        print(f"나는 학생인 {self.name}, {self.age}세이고, 학번은 {self.student_id}, 학점은 {self.grade}입니다.")
```

super() 키워드

- 부모 클래스에 이미 작성되어 있는 코드를 완전히 새롭게 정의하는 것이 아니라, 기존에 작성된 코드에 새로운 내용을 추가할 수 있다. 이 때, 파이썬에서는 super()라는 키워드를 사용한다.

- super()는 부모 클래스 자체를 의미하여, 부모 클래스에서 정의된 속성이나 메서드를 그대로 가져와서 사용할 수 있다.

super() 키워드

[1]:
```python
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age
        print(f"Person '{self.name}' 생성됨")

    def walk(self):
        print(f"{self.name}이(가) 걷는다.")

    def introduce(self):
        print(f"나는 {self.name}, {self.age}세입니다.")
```

부모 클래스 Person에서 name과 age 속성을 지정하는 코드를 그대로 가져와서 사용한다

[2]:
```python
class Student(Person):
    # __init__ 메서드를 재정의
    def __init__(self, name, age, student_id, grade):
        super().__init__(name, age) # 부모 클래스 Person의 __init__을 가져와서 그대로 사용
        self.student_id = student_id
        self.grade = grade
        print(f"Person '{self.name}' 생성됨")

    # Student만의 고유한 기능 추가
    def study(self):
        print(f"{self.name}이(가) 공부한다.")

    # 부모 클래스(Person)에서 물려받은 introduce 메서드를 재정의
    def introduce(self):
        print(f"나는 학생인 {self.name}, {self.age}세이고, 학번은 {self.student_id}, 학점은 {self.grade}입니다.")
```

실습 2

- 수업 내용에 상속에 대한 예시로 게시판 이용자(User)를 상속받는 일반 회원(Member), 관리자(Admin), 운영자(Operator)를 이야기했다. 게시판 board 딕셔너리와 User 클래스가 주어질 때, 이 User 클래스를 상속받는 Member를 생성해보자.
  - Member는 회원 등급(level)과 회원 포인트(point)를 새로운 속성으로 갖는다. 이 때, 별도로 회원 등급과 포인트를 입력받지 않는다면, 등급은 1, 포인트는 0으로 초기화되어야 한다.
  - my_info 메서드를 수정해서, 기존에 출력되던 내용에 회원 등급과 회원 포인트까지 출력해줘야한다.

```
from datetime import datetime ●●●
```

```
class User: ●●●
```

```
class Member(User): ●●●
```

[4]:
```
member = Member("User", "Password", "user@mail.com", "010-0000-0000")

member.my_info()
```

```
--- User님의 정보 ---
이메일: user@mail.com
전화번호: 010-0000-0000
가입일: 2025-10-27
회원등급: 1
포인트: 0
```

실습 3

- 이번에는 User를 상속받는 관리자(Admin) 클래스를 생성해보자. 이 클래스는 User와 비교해서 다음의 변경사항이 있다.
  - 관리자의 관리 권한을 저장하는 속성 admin_level이 추가된다. 이 admin_level은 별도로 정의하지 않으면 1로 초기화된다.
  - my_info 메서드를 수정해서, 기존에 출력되던 내용에 관리자 권한 등급까지 출력해줘야 한다.
  - del_post 메서드를 수정한다. User는 자기가 작성한 게시글만을 삭제할 수 있었지만, Admin은 자기가 작성하지 않은 모든 게시글을 전부 삭제할 수 있다.

```
from datetime import datetime ●●●
```

```
class User: ●●●
```

```
class Admin(User): ●●●
```

[4]:
```
# admin = Admin("admin", "adpassword", "admin@mail.com", "010-0000-0000")

admin.my_info()
admin.del_post(board)
```

```
--- admin님의 정보 ---
이메일: admin@mail.com
전화번호: 010-0000-0000
가입일: 2025-10-27
관리레벨: 1
게시판 번호:  1
삭제할 게시글 번호:  2
성공(관리자 admin): user_B의 2번 게시글이 삭제되었습니다.
```