

# 파이썬 프로그래밍 강의 노트 #12

---

## 객체 지향 프로그래밍

# 프로그래밍 패러다임

---

- 프로그래밍 패러다임(Programming paradigm)
  - 프로그램을 작성하는 방향과 구조를 정하는 방법
- 파이썬
  - 절차적 프로그래밍(procedural programming)과 객체 지향 프로그래밍(object-oriented programming)을 지원함

# 절차적 프로그래밍

---

## □ 절차적 프로그래밍

- goto 문을 주로 사용하던 비구조적 프로그래밍 방법을 개선해서 구조화 프로그래밍을 지원
- 조건문, 반복문 등을 프로그래밍 언어에 포함시킴
- 함수 등을 이용해서 프로그램을 모듈화시킴

# 절차적 프로그래밍의 문제점

---

- 스마트폰으로 전화거는 과정을 절차적 프로그래밍 방식으로 작성
- 스마트폰을 표현하는데 필요한 정보
  - 휴대폰 소유자 이름(문자열)
  - 전화번호(문자열)
  - 배터리 충전 상태(%를 나타내는 정수형)
- 이런 정보들을 각각의 변수로 관리하는 것은 어려우므로 리스트에 저장하기로 함

# 절차적 프로그래밍의 문제점

## □ 스마트폰 두 개 생성

```
>>> phone1 = ["Ki1 Dong Hong", "010-1111-1111", 80]  
>>> phone2 = ["Dongsu Hong", "010-2222-2222", 60]
```

## □ 스마트폰에서 전화를 발신하는 call() 함수 구현

- 배터리 잔량이 10%이상이어야 전화 걸 수 있음
- 발신자의 휴대폰 번호에서 상대방 착신번호로 통화한다는 내용 출력
- 같은 이름의 휴대폰이지만 4G(LTE)와 5G용이 분리됨
  - 전화 앱의 인터페이스는 같지만, 내부적으로 다른 통신 칩을 사용함
  - hwCall()은 휴대폰에서 4G와 5G 하드웨어를 이용해서 통화하는 부분을 담당한다고 가정
  - 여기서는 추상화해서 문자열로 출력함

# 절차적 프로그래밍의 문제점

- ❑ call() 함수는 어떤 휴대폰을 사용할 지 인자로 전달 받아야 함

```
>>> def hwCall(phone, phoneNum):  
...     print("하드웨어를 제어해서 전화를 겁니다")  
  
>>> def call(phone, phoneNum):  
...     if phone[2] >= 10:  
...         print(f"{phone[1]}에서 {phoneNum}으로  
전화를 겁니다")  
...         hwCall(phone, phoneNum)  
...     else:  
...         print("배터리가 부족합니다")  
... 
```

# 절차적 프로그래밍의 문제점

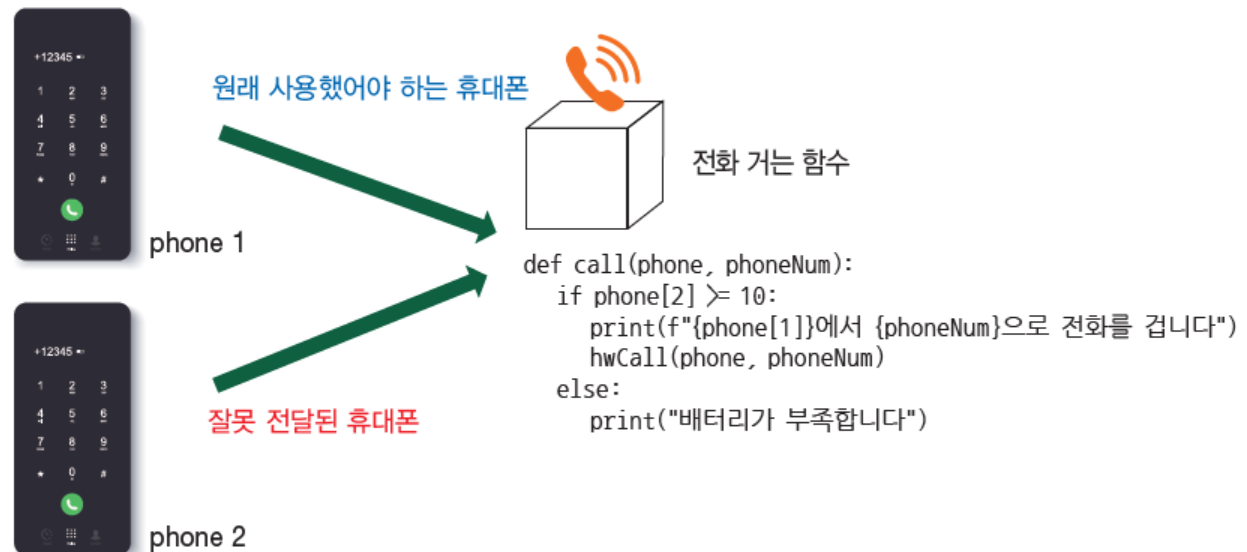
## ❑ call() 함수를 이용해서 전화 걸기

```
>>> call(phone1, "010-3333-3333")
```

010-1111-1111에서 010-3333-3333으로 전화를 겁니다  
하드웨어를 제어해서 전화를 겁니다

## ❑ 절차적 프로그래밍은 데이터와 동작(기능)이 분리됨

- call()함수에 phone이 잘못 전달되면 엉뚱한 스마트폰으로 전화를 걸게 됨



# 절차적 프로그래밍의 문제점

- 절차적 프로그래밍 방식은 재사용성도 떨어짐
  - 함수의 재사용을 위해 복사해서 붙인다고 가정할 때 그 함수가 의존하는 다른 함수들이나 자료들을 모두 찾아서 복사해서 붙여야 함
  - 예: call() 함수를 복사해서 붙임
    - 스마트폰 정보를 담고 있는 리스트 구조를 이해해야 함
    - hwCall() 함수도 복사해서 붙여야 함



# 객체 지향 프로그래밍

---

## □ 객체 지향 프로그래밍 방법

- 데이터와 코드를 객체로 함께 구성해서 한 개 자료형으로 취급
- 많은 객체 지향 언어들이 클래스(class)라는 이름으로 데이터와 코드를 묶을 수 있는 기능을 제공함

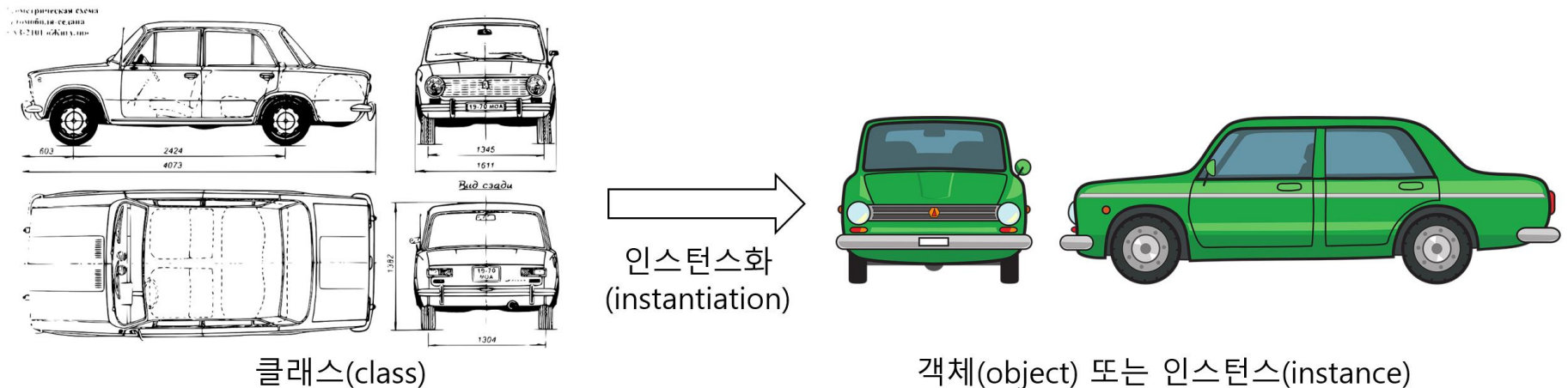
## □ 객체와 클래스

- 객체(object)
  - 프로그램이 실행될 때 클래스로부터 생성되는 실체
- 클래스는 객체의 속성(변수)과 속성들을 사용하고 처리하는 기능(함수 또는 메소드)를 포함하는 코드

# 객체와 클래스

## □ 자동차로 살펴보는 객체와 클래스

- 내가 A라는 차종의 자동차를 한 대 샀다면, 내가 소유한 차는 실제로 존재하는 객체
- 내 친구가 동일한 A 차종의 자동차를 소유하고 있다면, 그 친구는 다른 객체를 소유하고 있음
- 이때 A라는 차종의 클래스는 A 차종의 속성과 기능을 설명하고 구현하여 사용할 수 있도록 만들어진 설명서와 설계도



# 객체와 클래스

---

- 자동차는 다양한 속성을 가지고 있음
  - 색상
  - 엔진 크기
  - 소유주 정보
  - 차량 번호
  - 차대 번호(자동차마다 주어지는 고유 번호)
  - 자동차를 동작시키는데 필요한 부품들
- 자동차의 기능
  - 시동 걸기() 및 시동 끄기()
  - 앞으로 움직이기()
  - 가속하기()
  - 이러한 기능 등을 구현하는데 필요한 다른 함수들

# 객체와 클래스

---

## □ 자동차를 만드는 사람

- 내부 구조를 알고 있고, 어떻게 기능들이 구현되는지 알고 있어야 함

## □ 자동차를 사용하는 사람

- 자동차가 제공하는 사용법(인터페이스)만 알고 있으면 됨


## □ 클래스를 구현하는 사람은 클래스 내부 구조 및 구현 방법에 대해서 알고 있지만, 클래스를 사용하는 사람은 클래스에서 제공하는 인터페이스만 알고 있으면 됨

# 클래스 = 속성(변수)+기능(함수)+인터페이스

---

- 클래스는 객체를 생성하기 위한 설계도 또는 설명서
  - 속성을 나타내는 변수들과 이를 사용하는 함수들로 구성됨
  - 자료형이므로 직접 실행될 수 없음
  - 프로그래밍 일반적으로 인터페이스는 함수 형태로 제공됨
- 객체는 클래스로부터 생성되고, 인터페이스로 제공되는 함수들을 호출해서 사용함

# 스마트폰 클래스를 만든다면 포함될 내용

 SmartPhone
owner number battery appList
call(phoneNum) hwCall(phoneNum) getBatteryStatus()

이름

속성(변수)

기능(함수)

# 파이썬의 모든 자료형은 클래스

## ▣ 파이썬의 모든 자료형들은 클래스

- 정수, 실수, 문자열, 리스트, 튜플, 딕셔너리 등이 모두 클래스 자료형
- type() 명령을 이용하면 확인 가능

```
>>> type("문자열")
<class 'str'>
>>> type(3)
<class 'int'>
>>> type([]) # 빈 리스트의 자료형 확인
<class 'list'>
>>> type({}) # 빈 딕셔너리의 자료형 확인
<class 'dict'>
```

# 객체 지향 프로그래밍

---

## □ 객체 지향 프로그래밍의 특성

- 추상화
- 캡슐화
- 재사용성
- 상속
- 다형성



# 객체 지향 프로그래밍

## □ 추상화

- 실존하는 사물 또는 개념을 컴퓨터에서 처리할 수 있을 정도로 축약하고 핵심을 뽑아내는 것
- 휴대폰에 여러 기능이 있을 수 있지만, 주어진 문제를 해결하는데 전화를 거는 것만 필요하다면, 휴대폰의 나머지 기능은 제외하고 필요한 부분만으로 단순화시키는 것

## □ 캡슐화

- 필요한 정보와 인터페이스만 공개하고 나머지 자세한 구현 내용을 감추는 것 → 은닉성이라고도 부름
- 클래스의 인터페이스가 변경되지 않으면 사용자 코드를 변경할 필요가 없음
- TV는 거의 동일한 인터페이스(온/오프 스위치, 볼륨 조정, 채널 조정)로 CRT, PDP, LCD, LED 등으로 내부 구조가 계속 변경됨

# 객체 지향 프로그래밍

---

## □ 재사용성

- 클래스는 절차적 프로그래밍의 함수에 비해 재사용성이 높음
- 클래스는 필요로 하는 모든 기능들을 포함시켜 독립적으로 만들 수 있음
- 클래스가 다른 클래스에 의존적이지 않으면, 재사용성이 높아짐

## □ 상속

- 객체 지향 프로그래밍의 상속과 다형성을 활용하면 기존 클래스 코드를 수정하지 않고 클래스의 기능을 확장하거나 다르게 만들 수 있음
- 자식 클래스가 부모 클래스로부터 상속을 받는다면, 부모의 모든 속성과 함수들을 포함하게 됨

# 휴대폰 클래스와 상속

## □ 휴대폰 발전 과정

- 3G는 피쳐폰과 스마트폰이 공존하던 시기
- 스마트폰은 응용 프로그램을 설치할 수 있는 휴대폰
- 그 뒤에 4G(LTE) 휴대폰들이 보급되었고, 현재는 5G 네트워크를 사용하는 휴대폰이 많이 사용되고 있음

## □ 4G와 5G 휴대폰 클래스 만들기

- 앞에서 보인 SmartPhone 클래스(3G용)를 이용해서 4G와 5G 휴대폰 클래스를 만들어보기
  - 3G, 4G, 5G 휴대폰은 모두 같은 종류이고 같은 기능과 앱을 가지고 있지만, 사용하는 통신망만 다름
  - 각 세대별 휴대폰들은 자신의 통신망만 사용한다고 가정

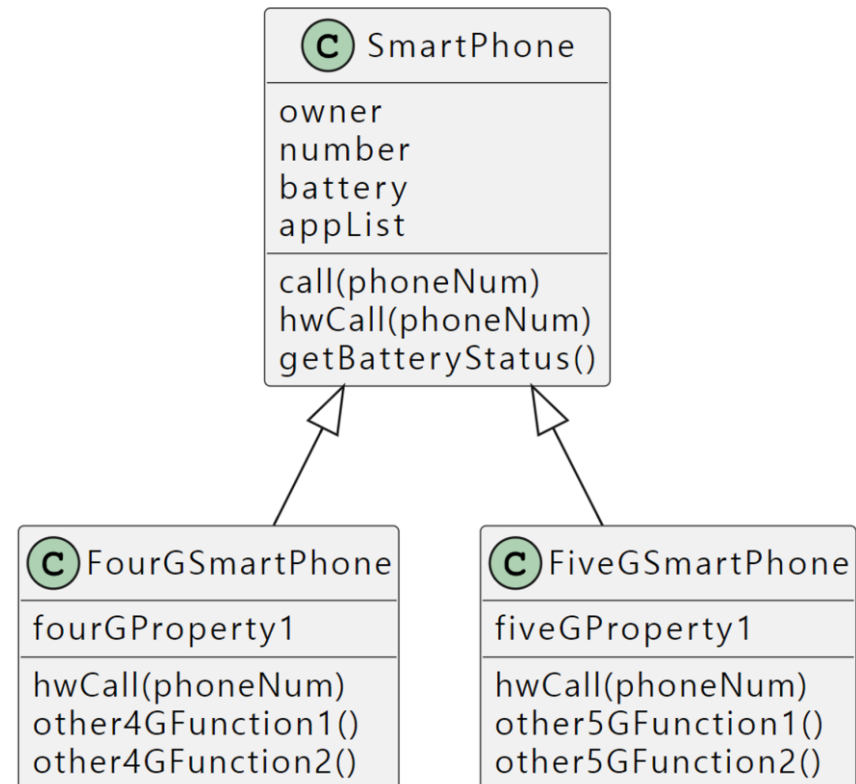
# 휴대폰 클래스와 상속

- 복사해서 붙이기로 4G와 5G 휴대폰 클래스 만들기
  - SmartPhone 클래스를 복사해서 4G용(FourGPhone)과 5G용(FiveGPhone)을 만든다면?
    - SmartPhone 클래스 코드를 복사해서 붙인 후에, 4G 또는 5G 휴대폰에 필요한 새로운 속성들과 함수들을 추가
    - 같은 속성들은 그대로 사용하고, 주소록이나 앱 등을 관리하는 기능들은 기존 코드를 사용할 수 있음
    - 새로운 통신망을 사용해야 하는 함수들만 수정
- 문제점
  - 중복 코드가 많아져서 유지 보수가 어려움(3G, 4G, 5G 클래스에 동일한 코드가 많음)
  - 같은 코드가 반복되면서 메모리를 더 많이 차지하면서 효율성이 떨어짐

# 휴대폰 클래스와 상속

## ■ 상속으로 4G와 5G 휴대폰 클래스 만들기

- 상속을 사용하면 자식 클래스는 부모 클래스의 모든 속성과 함수를 물려받음
- 새로 변경되는 함수들만 다시 구현하면 됨
- 상속 클래스 다이어그램
- hwCall() 함수는 하드웨어를 이용해서 실제 전화를 거는 함수
- 4G와 5G 휴대폰은 다른 통신망을 사용하므로, 부모 클래스에 있던 hwCall() 함수는 각 클래스에서 다시 구현되어야 함 → 오버라이딩



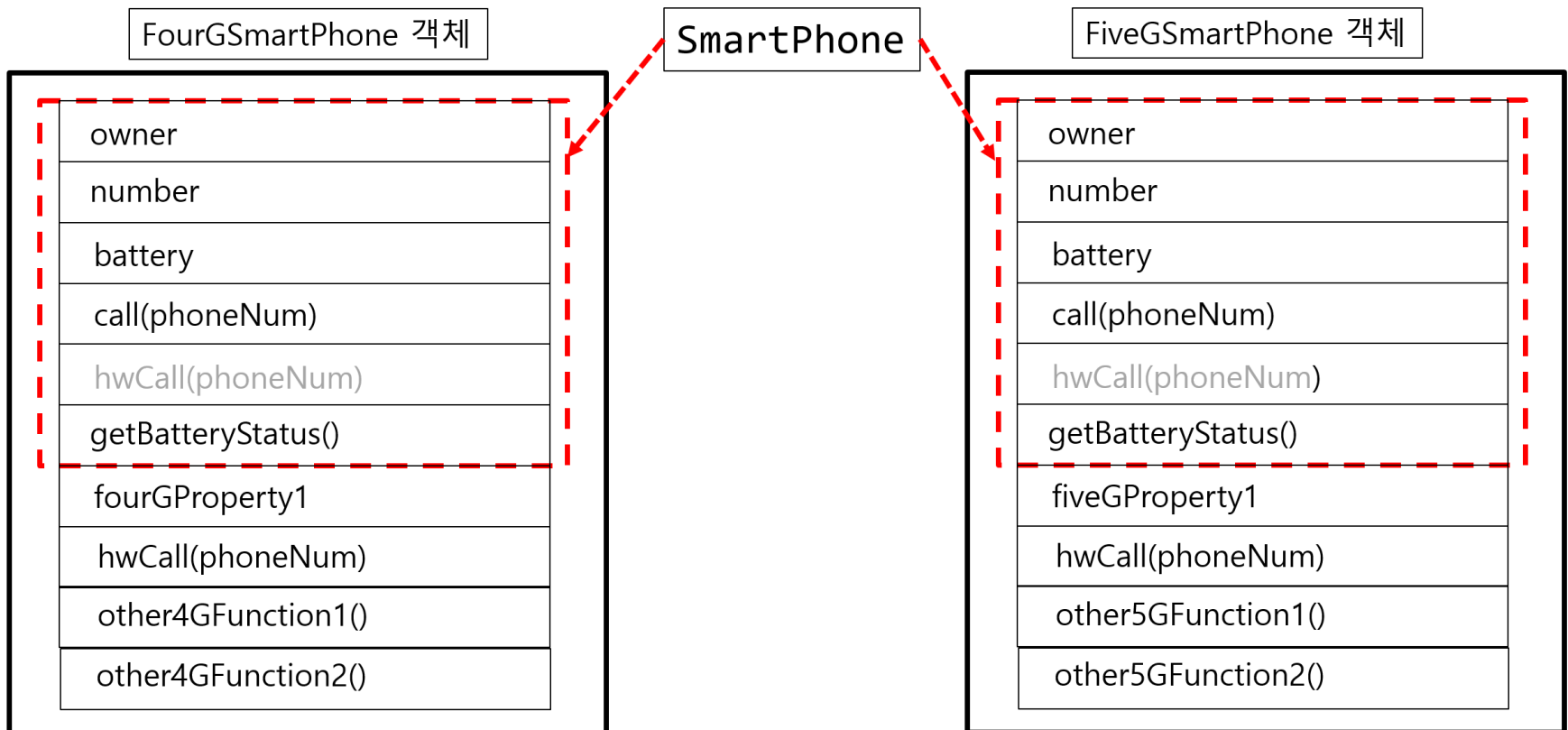
# 휴대폰 클래스와 상속

- call() 함수는 휴대폰의 사용자 인터페이스를 담당하며, hwCall() 함수를 내부에서 호출함
- 3G~5G 휴대폰의 전화앱 사용자 인터페이스는 동일하다고 가정했기 때문에 FourGSmartPhone과 FiveGSmartPhone은 SmartPhone에서 구현한 call() 함수를 사용할 수 있음
- call 함수를 의사코드(pseudo-code)로 표현

```
def call():  
    상대방 전화번호를 입력받고 문자열 phoneNum에 저장  
    number변수에서 phoneNum으로 전화 건다는 문구 출력  
    hwCall(phoneNum)    # 하드웨어를 이용해서 전화 걸기
```

# 휴대폰 클래스와 상속

- 다음 그림은 FourGSmartPhone과 FiveGSmartPhone 클래스에서 생성된 객체를 보임
  - hwCall() 함수는 각 클래스에서 따로 구현하므로 SmartPhone의 hwCall() 함수는 회색으로 흐릿하게 표현



# 다형성

## □ 다형성

- 동일한 함수가 다르게 작동한다는 의미
- 상속 관계의 여러 클래스들에서 동일한 명칭으로 구현된 함수들이 각각 다르게 동작하는 것
- 부모 클래스에 있는 함수가 자식 클래스들에서 오버라이딩 되면, 실행 시점에 자신의 객체에 연결되어 오버라이딩된 함수들이 호출됨
- 앞에서 만들어진 SmartPhone의 call() 함수에서 사용하는 hwCall() 함수가 SmartPhone이 아니라 FourGSmartPhone과 FiveGSmartPhone에서 만들어진 hwCall() 함수를 호출할 수 있게 해줌



# 다형성

- SmartPhone 객체를 인자로 전달받고 전화를 걸어주는 makeAPhoneCall() 함수 구현

```
def makeAPhoneCall(smartPhone):  
    smartPhone.call(phoneNum)
```

- 이 함수가 제대로 동작하려면 smartPhone에 전달되는 객체에 call() 함수가 있어야 함

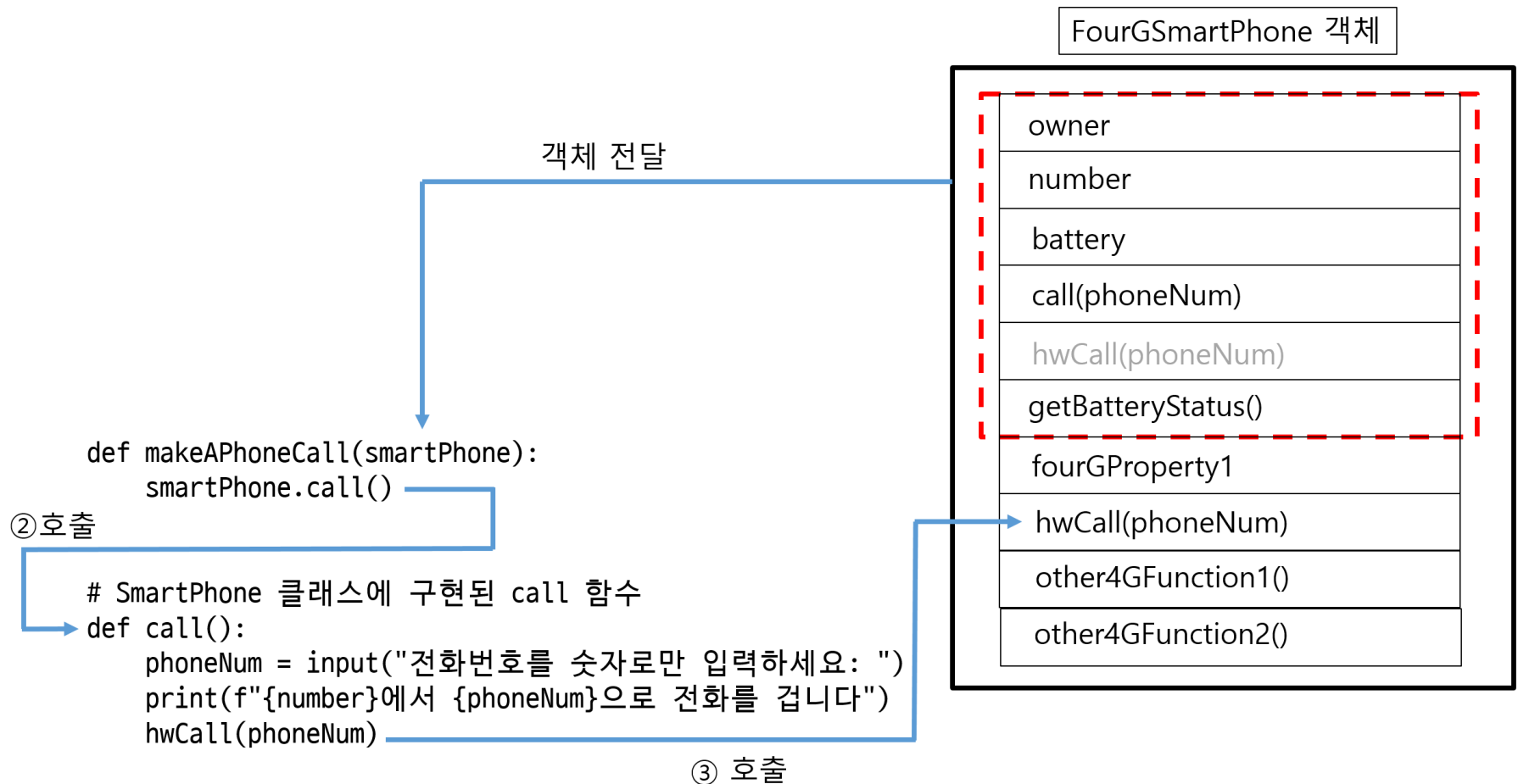
- SmartPhone에서 사용할 call() 함수 구현

- 스마트폰처럼 그래픽 인터페이스를 통해 전화번호를 받는 것은 어려우므로, 키보드로 입력 받는 것으로 구현

```
def call():  
    phoneNum = input("전화번호를 숫자로만 입력하세요: ")  
    print(f"{number}에서 {phoneNum}으로 전화를 겁니다")  
    hwCall(phoneNum)
```

# 다형성

- makeAPhoneCall() 함수에 FourGSmartPhone 객체가 전달되었을 때 실행되는 과정을 보임



# 다형성

---

- 상속과 오버라이딩을 통해서 SmartPhone 클래스의 call() 함수에서 사용하는 hwCall() 함수가 상속받은 FourGSmartPhone과 FiveGSmartPhone 클래스에 포함된 것을 호출할 수 있도록 해주는 것이 다형성
- 6G 통신망이 만들어지고, 6G용 휴대폰을 추상화한 SixGSmartPhone 클래스에 hwCall() 함수가 오버라이딩 된다면 makeAPhoneCall() 함수나 SmartPhone 클래스의 코드를 수정하지 않고도 SixGSmartPhone 객체를 전달해서 전화를 걸 수 있음

# 클래스 선언 및 사용

## □ 클래스 선언

```
class 클래스_이름:
    # 생성자
    def __init__(self, 매개_변수_리스트):
        멤버_변수_생성

    # 메소드
    def 이름(self, 매개_변수_리스트):
        코드_블록
```

## □ 객체 또는 인스턴스(instance) 생성

```
객체_변수_이름 = 클래스_이름(인자)
```

- 객체 생성할 때 전달되는 인자는 생성자에 전달됨

# 클래스 선언 및 사용

- 클래스의 인스턴스 메소드(method) 또는 멤버 함수
  - 클래스 내부에 구현된 함수
    - 첫 번째 매개 변수로 self를 지정해야 함
  - 클래스 내부에서 메소드 호출
    - self.함수\_이름으로 호출
  - 클래스 외부에서 메소드 호출
    - 객체\_이름.함수\_이름으로 호출
  - 멤버 함수의 첫 번째 매개변수: self 변수
    - 생성자나 멤버 함수의 첫 번째 인자는 객체 자신을 가리키는 변수
    - 관례적으로 "self"라는 이름을 사용
  - 멤버 함수의 두 번째 변수부터는 일반 함수처럼 사용
  - 클래스의 멤버 함수를 호출할 때 첫 번째 인자는 생략

# SmartPhone 클래스

## □ 첫 번째 버전(속성 없음)

```
class SmartPhone:
    def call(self, phoneNum):
        print(f"call(): {phoneNum}")

    def hwCall(self, phoneNum):
        print(f"hwCall(): {phoneNum}")

    def getBatteryStatus(self):
        print("getBatteryStatus")
```

## □ 객체 생성

```
phone1 = SmartPhone()
```

## □ 객체의 메소드 호출(self에는 인자 전달 안함)

```
phone1.call("010-1111-1111")
```

## □ 객체 생성 및 사용 코드

```
>>> class SmartPhone:
...     def call(self, phoneNum):
...         print(f"call(): {phoneNum}")
...
...     def hwCall(self, phoneNum):
...         print(f"hwCall(): {phoneNum}")
...
...     def getBatteryStatus(self):
...         print("getBatteryStatus")
...
>>> phone1 = SmartPhone()
>>> phone2 = SmartPhone()
>>> phone1.call("010-1111-1111")
call(): 010-1111-1111
>>> phone1.getBatteryStatus()
getBatteryStatus
>>> phone2.call("010-2222-2222")
call(): 010-2222-2222
>>> phone2.getBatteryStatus()
getBatteryStatus
```

# 멤버 변수 생성 및 사용

## □ 클래스의 멤버 변수(속성)

- 초기자(생성자)나 멤버 함수에서 "self.변수\_이름"에 처음으로 값을 저장할 때 생성됨
- 멤버 변수는 객체마다 다른 메모리 공간에 존재
- 클래스 외부에서는 객체\_이름.변수\_이름 형태로 접근
- 클래스의 멤버 함수에서는 해당 클래스의 인스턴스 변수에 자유롭게 접근 가능
  - self.변수\_이름 형태로 사용
- 변수 이름 앞에 밑줄 문자('\_')를 두 개 붙이면 클래스 내부에서만 사용할 수 있는 멤버 변수(private 변수) 생성
- 변수 이름 앞에 밑줄 문자('\_')를 한 개 붙이면 클래스 내부와 상속된 클래스에서만 사용할 수 있는 멤버 변수(protected 변수) 생성



# 생성자(constructor) 또는 초기자(initializer)

## □ 생성자

- `__init__(self, 매개_변수_리스트)`로 정해짐
- 클래스 이름으로 객체를 생성할 때 호출되는 함수
- 없을 수도 있음
- 주로 멤버 변수를 생성하고 초기화시키는 목적으로 사용

## □ 멤버 변수를 포함하는 SmartPhone 클래스 선언

```
class SmartPhone:  
    def _init_(self, name, phoneNum, battery):  
        self.owner = name  
        self.number = phoneNum  
        self.battery = battery # (1)
```

# SmartPhone 클래스

- 멤버 변수를 화면에 출력하는 `printMemberVariables()` 함수를 SmartPhone 클래스에 추가

```
class SmartPhone:
    def __init__(self, name, phoneNum, battery):
        self.owner = name
        self.number = phoneNum
        self.battery = battery # (1)

    def printMemberVariables(self):
        print(f"owner = {self.owner}, phone number = {self.number}, battery = {self.battery}, privateInt = {self._privateInt}")
```

# SmartPhone 클래스

## ▣ 객체 생성 및 멤버 함수 호출

```
>>> sp1 = SmartPhone("Cho", "010-1111-1111", 80) #  
함수 호출하듯이 인자 전달  
>>> sp2 = SmartPhone("Lim", "010-2222-2222", 70)  
>>> sp1.printMemberVariables()  
owner = Cho, phone number = 010-1111-1111, battery  
= 80, privateInt = 30  
>>> sp2.printMemberVariables()  
owner = Lim, phone number = 010-2222-2222, battery  
= 70, privateInt = 30
```

# SmartPhone 클래스 – 두 번째 버전

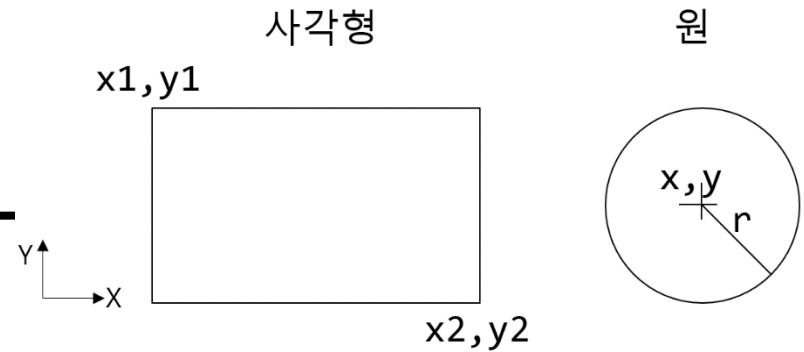
```
class SmartPhone:
    def __init__(self, name, phoneNum, battery):
        self.owner = name
        self.number = phoneNum
        self.battery = battery
    def call(self, phoneNum):
        if self.battery >= 10:
            print(f"{self.number}에서 {phoneNum}으로
전화를 겁니다")
            self._hwCall(phoneNum)
        else:
            print("배터리가 부족합니다")
    def getBatteryStatus(self):
        return self.battery
    def _hwCall(self, phoneNum): # protected 함수
        print(f"하드웨어를 제어해서 {phoneNum}으로 전화를
겁니다")
```

# SmartPhone 클래스 – 두 번째 버전

## ▣ 객체 생성 및 사용 코드

```
sp1 = SmartPhone("Cho", "010-1111-1111", 80)
sp2 = SmartPhone("Lim", "010-2222-2222", 70)
sp1.call("010-3333-3333")
sp2.call("010-4444-4444")
print(f"sp1의 배터리 충전 상태: {sp1.getBatteryStatus()}")
print(f"sp2의 배터리 충전 상태: {sp2.getBatteryStatus()}")
```

# 실습문제 1



## □ 문제

- 사각형 또는 원의 면적 계산에 필요한 데이터를 3회 입력 받고, 면적을 계산
- 입력 데이터는 리스트에 저장한 후, 순차적으로 면적을 계산해서 출력

## □ 요구사항

- 사각형과 원을 클래스로 구현
- 사각형은 왼쪽 상단  $(x1, y1)$ 과 오른쪽 하단  $(x2, y2)$ 의 좌표를 입력 받고, 원은 중심 좌표  $(x, y)$ 와 반지름을 입력 받기
- 좌표값은 정수
- 사용자로부터 입력받을 때, 모양을 문자열로 입력받고, 좌표값이나 반지름 등을 입력 받기
- 사용자가 어떤 순서로 모양을 입력할 지 모름(오류 없음)

# 실습문제 1

## ▣ 최종 코드

```
import math

class Rectangle:
    def __init__(self, x1, y1, x2, y2):
        self.x1 = x1
        self.y1 = y1
        self.x2 = x2
        self.y2 = y2
    def calcArea(self):
        return (self.x2 - self.x1) * (self.y1 - self.y2)
```

# 실습문제 1

---

```
class Circle:
    def __init__(self, x, y, r):
        self.x = x
        self.y = y
        self.r = r

    def calcArea(self):
        return math.pi * self.r * self.r
```



# 실습문제 1

```
shapeList = []
for i in range(3):
    s = input("도형 모양을 입력하세요: ")
    if s == "사각형":
        x1 = int(input("왼쪽 상단의 x좌표를 입력: "))
        y1 = int(input("왼쪽 상단의 y좌표를 입력: "))
        x2 = int(input("오른쪽 하단의 x좌표를 입력: "))
        y2 = int(input("오른쪽 하단의 y좌표를 입력: "))
        shapeList.append(Rectangle(x1, y1, x2, y2))
    elif s == "원":
        x = int(input("원의 중심 x 좌표를 입력: "))
        y = int(input("원의 중심 y 좌표를 입력: "))
        r = int(input("원의 반지름을 입력: "))
        shapeList.append(Circle(x, y, r))

for s in shapeList:
    print(f"면적: {s.calcArea()}")
```

# 상속과 다형성 구현

- 상속 관계를 표현하는 것은 클래스를 선언하면서 부모 클래스의 이름을 명시

```
class 자식_클래스_이름(부모_클래스_이름):  
    <코드 블록>
```

- FourGSmartPhone 클래스 구현(첫 번째 버전)

```
class FourGSmartPhone(SmartPhone):  
    def __init__(self):  
        self.fourGProperty1 = "4G LTE Network"  
    def other4GFunction1(self):  
        print(f"{self.fourGProperty1}")  
    def other4GFunction2(self):  
        print("4G 기능 2를 처리합니다")  
    def _hwCall(self, phoneNum):  
        print(f"4G 통신망을 사용해서 {phoneNum}으로  
전화를 겁니다")
```

# 상속과 다형성 구현

## ■ FourGSmartPhone 사용 코드

```
>>> fsp1 = FourGSmartPhone()  
>>> fsp1.other4GFunction1()  
4G LTE Network  
>>> fsp1.other4GFunction2()  
4G 기능 2를 처리합니다
```

## ■ SmartPhone 클래스의 멤버 함수를 호출

```
>>> fsp1.call("010-3333-3333")  
Traceback (most recent call last):  
File "<stdin>", line 1, in <module>  
File "<stdin>", line 7, in call  
AttributeError: 'FourGSmartPhone' object has  
no attribute 'battery'
```

# 상속과 다형성 구현

- 오류 발생 원인은 SmartPhone 클래스의 생성자가 호출되지 않았기 때문
  - 자식 클래스의 생성자에서 부모 클래스의 생성자를 호출해야 함
  - 부모 클래스의 객체를 가리키는 값은 `super()` 함수를 사용
  - 부모 클래스의 함수를 호출: `super().함수_이름()`

```
super().__init__() # 부모 클래스의 생성자 호출  
super().func()     # 부모 클래스의 func() 호출
```

# 상속과 다형성 구현

## ▣ FourGSmartPhone 클래스 구현(두 번째 버전)

```
class FourGSmartPhone(SmartPhone):  
    def __init__(self, name, phoneNum, battery):  
        super().__init__(name, phoneNum, battery)  
        self.fourGProperty1 = "4G LTE Network"  
    def other4GFunction1(self):  
        print(f"{self.fourGProperty1}")  
    def other4GFunction2(self):  
        print("4G 기능 2를 처리합니다")  
    def _hwCall(self, phoneNum):  
        print(f"4G 통신망을 사용해서 {phoneNum}으로  
전화를 겁니다")
```

# 상속과 다형성 구현

## ▣ 사용 코드

```
>>> fsp1 = FourGSmartPhone("cho", "010-1111-1111", 80)
>>> fsp1.call("010-3333-3333")
```

010-1111-1111에서 010-3333-3333으로 전화를 겁니다  
4G 통신망을 사용해서 010-3333-3333으로 전화를 겁니다

# 상속과 다형성 구현

## ▣ FiveGSmartPhone 클래스 구현

```
class FiveGSmartPhone(SmartPhone):  
    def __init__(self, name, phoneNum, battery):  
        super().__init__(name, phoneNum, battery)  
        self.fiveGProperty1 = "5G Network"  
    def other5GFunction1(self):  
        print(f"{self.fiveGProperty1}")  
    def other5GFunction2(self):  
        print("5G 기능 2를 처리합니다")  
    def _hwCall(self, phoneNum):  
        print(f"5G 통신망을 사용해서 {phoneNum}으로 전화를  
        겁니다")
```

# 상속과 다형성 구현

## ▣ 사용 코드

```
>>> fsp2 = FiveGSmartPhone("lim", "010-2222-2222", 70)
>>> fsp2.other5GFunction1()
5G Network
>>> fsp2.other5GFunction2()
5G 기능 2를 처리합니다
>>> fsp2.call("010-4444-4444")
010-2222-2222에서 010-4444-4444으로 전화를 겁니다
5G 통신망을 사용해서 010-4444-4444으로 전화를 겁니다
```



## 실습문제 2

---

### □ 문제

- 실습문제 1에서 만들었던 Rectangle과 Circle을 클래스에서 상속받도록 다시 구현
- Shape 클래스는 도형의 모양을 나타내는 문자열을 포함하고, 이 문자열을 반환하는 멤버 함수 구현

### □ 요구사항

- 실습문제 1에서 했던 것처럼 3개의 도형 정보를 입력받아 면적 출력
- 도형 정보를 "도형 모양: 사각형" 또는 "도형모양: 원" 형태로 출력하고 다음 줄에 면적을 출력

## 실습문제 2

### ▣ 최종 코드

```
import math

class Shape:
    def __init__(self, shapeStr):
        self.shapeStr = shapeStr
    def getShapeStr(self):
        return self.shapeStr

class Circle(Shape):
    def __init__(self, shapeStr, x, y, r):
        super().__init__(shapeStr)
        self.x = x
        self.y = y
        self.r = r
    def calcArea(self):
        return math.pi * self.r * self.r
```

## 실습문제 2

```
class Rectangle(Shape):  
    def __init__(self, shapeStr, x1, y1, x2, y2):  
        super().__init__(shapeStr)  
        self.x1 = x1  
        self.y1 = y1  
        self.x2 = x2  
        self.y2 = y2  
    def calcArea(self):  
        return (self.x2 - self.x1) * (self.y1 - self.y2)
```

## 실습문제 2

```
shapeList = []
for i in range(3):
    s = input("도형 모양을 입력하세요: ")
    if s == "사각형":
        x1 = int(input("왼쪽 상단의 x좌표 입력: "))
        y1 = int(input("왼쪽 상단의 y좌표 입력: "))
        x2 = int(input("오른쪽 하단의 x좌표 입력: "))
        y2 = int(input("오른쪽 하단의 y좌표 입력: "))
        shapeList.append(Rectangle("사각형",x1,y1,x2, y2))
    elif s == "원":
        x = int(input("원의 중심 x 좌표 입력: "))
        y = int(input("원의 중심 y 좌표 입력: "))
        r = int(input("원의 반지름 입력: "))
        shapeList.append(Circle("원", x, y, r))
for s in shapeList:
    print(f"도형 모양: {s.getShapeStr()}")
    print(f"면적: {s.calcArea()}")
```

## 실습문제 3

---

### □ 문제 - 기본 클래스 정의와 객체 생성

- 간단한 Car 클래스를 정의하고, 이 클래스의 객체를 생성한 후 속성을 출력하는 프로그램을 작성하세요.

# 실습문제 3

## □ 최종 코드

```
class Car:
    def __init__(self, make, model, year):
        self.make = make
        self.model = model
        self.year = year

def create_and_print_car():
    my_car = Car("Toyota", "Corolla", 2021)
    print(f"차량 정보: {my_car.year} {my_car.make} {my_car.model}")

# 함수 호출
create_and_print_car()
```

## 실습문제 4

---

### □ 문제 - 메소드 추가

- Dog 클래스를 정의하고, bark 메소드를 추가하세요. 이 메소드는 호출 시 "Woof!"를 출력합니다. 객체를 생성하고 메소드를 호출하세요.

# 실습문제 4

## □ 최종 코드

```
class Dog:
    def bark(self):
        print("Woof!")

def make_dog_bark():
    my_dog = Dog()
    my_dog.bark()

# 함수 호출
make_dog_bark()
```



## 실습문제 5

---

### □ 문제 - 상속

- Vehicle 클래스를 만들고, 이를 상속받는 Truck 클래스를 정의하세요. Truck 클래스는 추가적으로 load\_capacity 속성을 갖습니다. 객체를 생성하고 정보를 출력하세요.

# 실습문제 5

## □ 최종 코드

```
class Vehicle:
    def __init__(self, make, model):
        self.make = make
        self.model = model

class Truck(Vehicle):
    def __init__(self, make, model, load_capacity):
        super().__init__(make, model)
        self.load_capacity = load_capacity

def create_and_print_truck():
    my_truck = Truck("Ford", "F-150", 1000)
    print(f"트럭 정보: {my_truck.make} {my_truck.model}, 적재 용량: {my_truck.load_capacity}kg")

# 함수 호출
create_and_print_truck()
```

## 실습문제 6

---

- 문제 - 인스턴스 메소드, 클래스 메소드, 스태틱 메소드
  - MathOperations 클래스를 정의하고, 인스턴스 메소드로 add, 클래스 메소드로 multiply, 스태틱 메소드로 subtract를 구현하세요. 각 메소드의 기능은 이름과 동일하게 연산을 수행합니다.

# 실습문제 6

## □ 최종 코드

```
class MathOperations:
    def add(self, a, b):
        return a + b

    @classmethod
    def multiply(cls, a, b):
        return a * b

    @staticmethod
    def subtract(a, b):
        return a - b

def perform_operations():
    math_op = MathOperations()
    print("Add:", math_op.add(2, 3))
    print("Multiply:", MathOperations.multiply(4, 5))
    print("Subtract:", MathOperations.subtract(10, 3))

# 함수 호출
perform_operations()
```

## 실습문제 7

---

### □ 문제 - 프라이빗 속성과 메소드 활용

- Account 클래스를 정의하고, balance (잔액)를 프라이빗 속성으로 갖습니다. 잔액을 조작하는 프라이빗 메소드와, 이를 호출하는 공개 메소드를 작성하세요.

# 실습문제 7

## □ 최종 코드

```
class Account:
    def __init__(self, balance):
        self.__balance = balance

    def __update_balance(self, amount):
        self.__balance += amount

    def deposit(self, amount):
        if amount > 0:
            self.__update_balance(amount)
            print(f"Deposited: ${amount}")
            print(f"New Balance: ${self.__balance}")
        else:
            print("Invalid amount")

# 함수 호출
my_account = Account(100)
my_account.deposit(50)
```

## 실습문제 8

---

### □ 문제 – 클래스 정의

- BankAccount 클래스를 정의하세요. 이 클래스는 초기 잔액(balance)을 매개변수로 받으며, 입금(deposit) 메소드와 출금(withdraw) 메소드를 포함해야 합니다. 출금 시 잔액보다 많은 금액을 출금하려고 하면, "Insufficient funds"를 출력해야 합니다.

# 실습문제 8

## □ 최종 코드

```
class BankAccount:
    def __init__(self, balance=0):
        self.balance = balance

    def deposit(self, amount):
        self.balance += amount
        print(f"Deposited: ${amount}. New balance:
${self.balance}")

    def withdraw(self, amount):
        if amount > self.balance:
            print("Insufficient funds")
        else:
            self.balance -= amount
            print(f"Withdrew: ${amount}. New balance:
${self.balance}")

# 함수 호출
account = BankAccount(100)
account.deposit(50)
account.withdraw(200)
```



## 실습문제 9

---

### □ 문제

- 상속 및 메소드 오버라이딩
- Animal 클래스를 정의하고, speak 메소드를 포함하세요. Dog와 Cat 클래스가 Animal 클래스를 상속받도록 하고, 각 클래스에서 speak 메소드를 오버라이드하여 "Woof"와 "Meow"를 각각 출력하세요.

# 실습문제 9

## □ 최종 코드

```
class Animal:
    def speak(self):
        print("Some generic sound")

class Dog(Animal):
    def speak(self):
        print("Woof")

class Cat(Animal):
    def speak(self):
        print("Meow")

# 함수 호출
my_dog = Dog()
my_cat = Cat()
my_dog.speak()
my_cat.speak()
```

# 실습문제 10

---

## □ 문제

- Setter,getter
- Product 클래스를 정의하고, 이름(name), 가격(price) 속성을 포함하세요.

# 실습문제 10

□ 초

```
class Product:
    def __init__(self, name, price):
        self.name = name
        self._price = price

    @property
    def price(self):
        return self._price

    @price.setter
    def price(self, value):
        if value < 0:
            print("Price cannot be negative.")
        else:
            self._price = value

# 함수 호출
product = Product("Coffee", 5)
print(product.price) # Output: 5
product.price = -10 # Should raise an error message
product.price = 15
print(product.price) # Output: 15
```