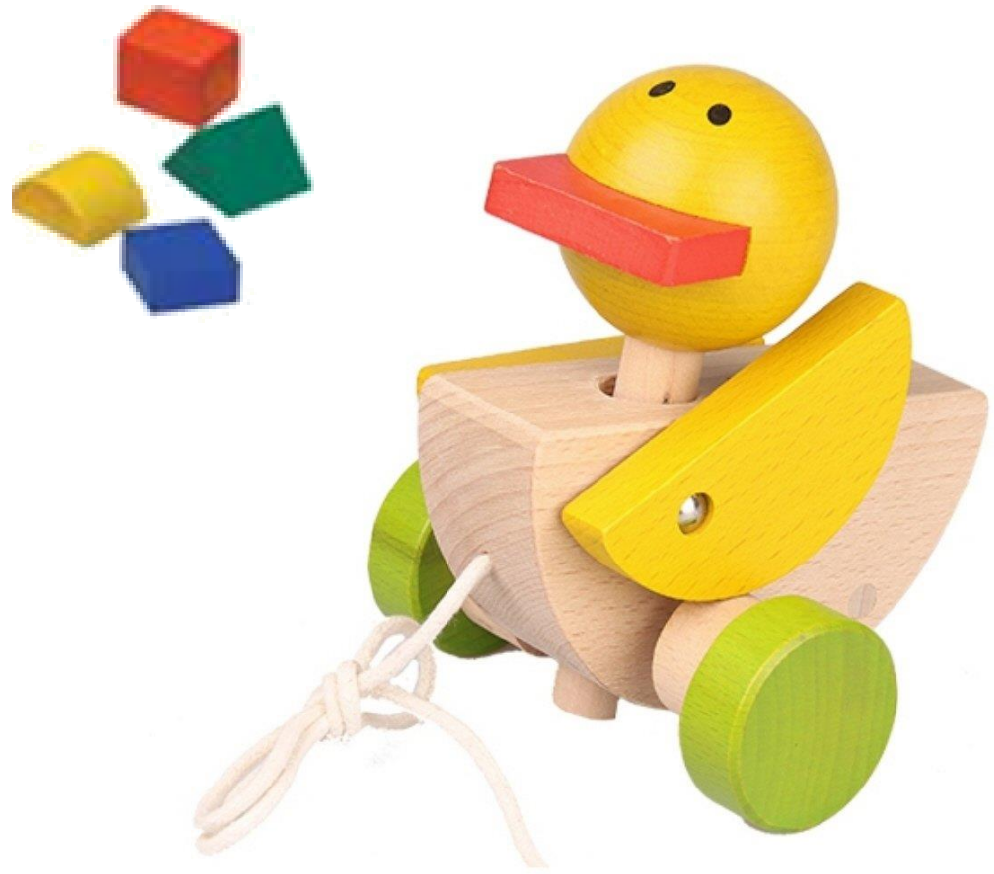


LESSON

# 프로그래밍 방식



- OOP(Object Oriented Programming)

먼저 부품 객체를 만들고 이것들을 하나씩 조립해서 완성된 프로그램을 만드는 기법.



## 장점

### 1. 코드의 재사용성 증가

어느 객체에서 이미 작성한 코드를 상속을 통해 재사용하기 쉽습니다.

### 2. 생산성 향상

클래스를 잘 만들어두면, 독립적인 객체를 사용할 수 있으며 개발의 생산성이 향상됩니다.

### 3. 유지보수의 편리성

수정, 추가 시 캡슐화가 되어있기 때문에 주변 코드에 영향을 크게 주지 않아 유지보수가 쉽습니다.

## 단점

### 1. 개발 속도가 느린 점

객체마다의 기능과 처리할 부분을 정확하게 이해하고 설계해야하기 때문에, 설계 단계부터 시간이 오래 걸립니다.

### 2. 실행 속도가 느린 점

객체 지향 언어로 프로그래밍 시 실행 속도가 느립니다.

### 3. 코딩의 난이도 상승

C++의 경우 다중 상속이 가능하여 굉장히 복잡해질 수 있어, 난이도가 상승합니다.

LESSON

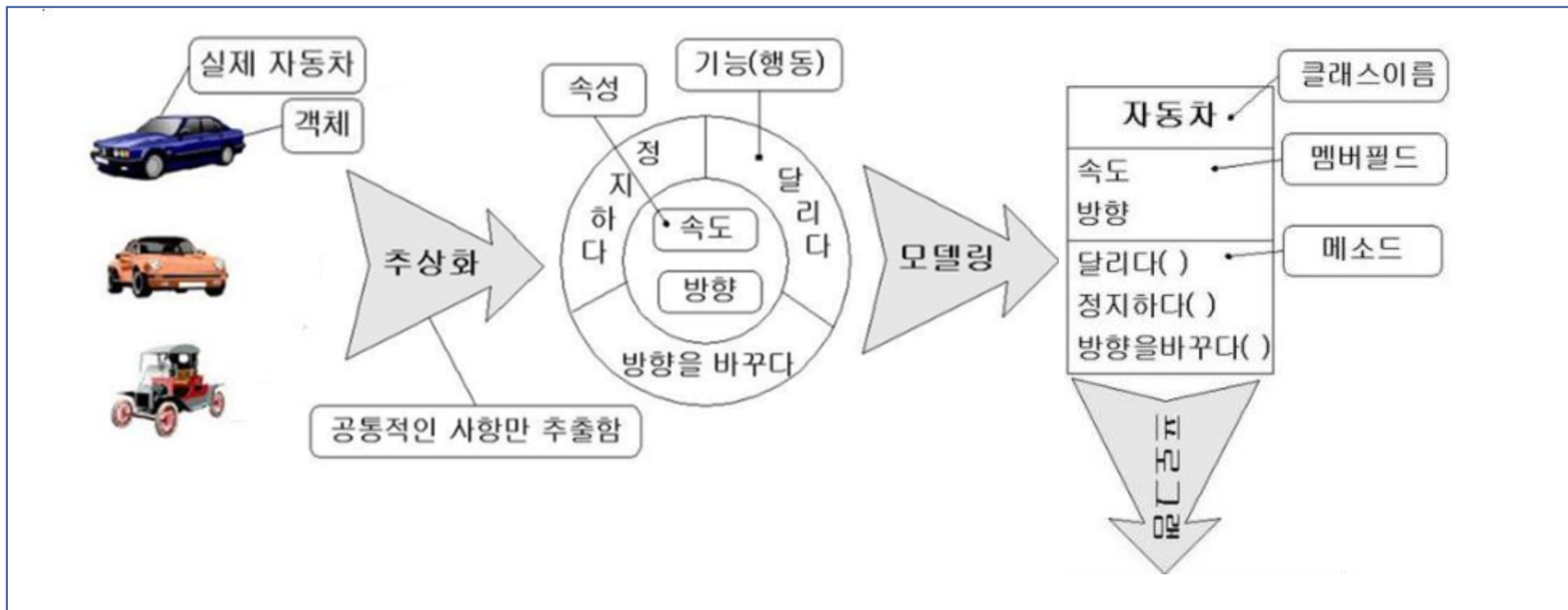
# 클래스(class)

## ● 객체(object)

실세계에 존재하는 또는 추상적인 사물이나 개념으로 소프트웨어 세계에 구현할 대상

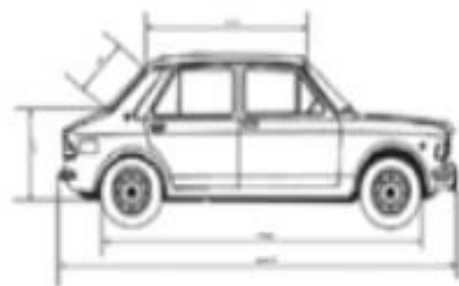
## ● 클래스(class)

추상화를 통해서 객체들의 공통된 사항을 모아 놓은 것



# 클래스(class)란?

- 객체를 정의하는 설계도.
- 클래스는 '현실 세계의 사물을 컴퓨터 안에서 구현하려고 고안된 개념'이다.



자동차를 클래스로 구현



```
# 자동차의 속성  
색상  
속도  
# 자동차의 기능  
속도 올리기()  
속도 내리기()
```

- 클래스 정의하기

```
class 클래스명:  
    필드 정의  
    메서드 정의
```

- ✓ 클래스명의 첫 문자는 반드시 대문자로 입력.
- ✓ 필드(field) : 클래스 내부에 정의된 변수.
- ✓ 메서드(method) : 클래스 내부에 정의된 함수.



- 클래스(설계도)에 의해 구현된 실체가 인스턴스이다.

Car 클래스(설계도)를 기반으로 실제 자동차를 제작하는 작업을 해야한다. 이렇게 생산되는 자동차가 인스턴스(instance)이며 객체(object)이다.

자동차 설계도(클래스)



여러 번  
찍어 내기



자동차(인스턴스)



자동차 설계도(클래스)

```
class 자동차 :  
    # 자동차의 속성  
    색상  
    속도  
    # 자동차의 기능  
    속도 올리기()  
    속도 내리기()
```



자동차(인스턴스)

```
자동차1=자동차()  
자동차2=자동차()  
자동차3=자동차()
```

- 인스턴스 생성

인스턴스명=클래스명()

```
class Car :  
    def __init__(self, color, speed):  
        self.color = color  
        self.speed = speed  
  
    def up_speed(self, value) :  
        self.speed += value  
  
    def down_speed(self, value) :
```

#메인 코드 부분

```
car1 = Car('red',30)  
car2 = Car('blue',60)
```

```
print("car1의 색상은 %s이며, 현재 속도는 %dkm입니다." % (car1.color, car1.speed))  
print("car2의 색상은 %s이며, 현재 속도는 %dkm입니다." % (car2.color, car2.speed))
```

# 생성자(constructor)

- 인스턴스를 생성할 때 자동으로 호출되는 메서드를 생성자라고 한다.
- 생성자는 `__init__()` 라는 이름을 가진다.

```
class 클래스명:  
    def __init__(self):  
        초기화할 코드
```



`__init__`는 initialize의 약자로 '초기화' 의미를 가진다. 언더바가 2개 붙은 것은 파이썬에서 예약해 놓은 것.

- ✓ 생성자(Constructor)로 인스턴스를 초기화할 수 있다.
- ✓ 생성자의 리턴 값은 존재하지 않는다.
- ✓ 생성자는 여러 개가 존재해서는 안된다.



클래스를 작성할 때 `__init__()` 라는 생성자를 만들지 않고도 인스턴스를 생성하고 필드의 사용이 가능하다. 이유는 파이썬 인터프리터에서 클래스 안에 생성자가 존재하지 않으면 자동으로 아무런 값을 설정하지 않은 기본생성자를 추가해 주기 때문이다.

# 기본 생성자(매개변수로 self만 있는 생성자)

```
class Car :  
    # 기본 생성자  
    def __init__(self):  
        self.color = 'red'  
        self.speed = 0  
  
    def up_speed(self, value) :  
        self.speed += value  
  
    def down_speed(self, value) :  
        self.speed -= value
```

#메인 코드 부분

```
car1 = Car()  
car2 = Car()  
print("car1의 색상은 %s이며, 현재 속도는 %dkm입니다." % (car1.color, car1.speed))  
print("car2의 색상은 %s이며, 현재 속도는 %dkm입니다." % (car2.color, car2.speed))  
car1.color='white'  
car1.speed=100  
print("car1의 색상은 %s이며, 현재 속도는 %dkm입니다." % (car1.color, car1.speed))
```

car1의 색상은 red이며, 현재 속도는 0km입니다.  
car2의 색상은 red이며, 현재 속도는 0km입니다.

car1의 색상은 white이며, 현재 속도는 100km입니다.

# 매개변수가 있는 생성자

```
class Car :
    # 매개변수가 있는 생성자
    def __init__(self, color, speed):
        self.color = color
        self.speed = speed

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

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

#메인 코드 부분
myCar1 = Car('red',30)
myCar2 = Car('blue',60)

print("자동차1의 색상은 %s이며, 현재 속도는 %dkm입니다." % (myCar1.color, myCar1.speed))
print("자동차2의 색상은 %s이며, 현재 속도는 %dkm입니다." % (myCar2.color, myCar2.speed))
```

인스턴스 변수, 클래스 변수

- 인스턴스들이 생성될 때 각자의 값을 저장하기 위해 사용되는 변수.
- `__init__()` 메서드에서 생성된 변수.
- 모든 인스턴스 변수는 `self`를 앞에 붙인다.

## **self.변수**

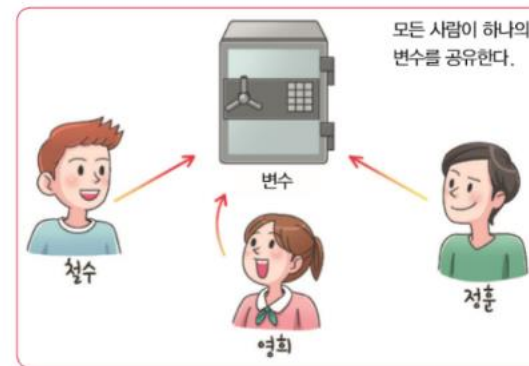
- 인스턴스 생성후 사용할 수 있는 변수.



```
class Car:
    car_count=0
    def __init__(self,name,color):
        self.name=name
        self.color=color
        Car.car_count+=1
        print(f'총 생산된 차량은 {Car.car_count}대')
```



- 모든 인스턴스가 공유하는 값을 저장하는 변수.
- 모든 인스턴스들이 메모리를 공유하게 된다.
- 클래스 변수는 인스턴스를 생성하지 않아도 사용이 가능하다.
- 클래스 변수에 접근하는 방법 :  
    클래스명.변수명 or 인스턴스명.변수명



클래스 변수는 클래스당 하나만 생성되어서 모든 객체가 공유합니다.



```
class Car:
    car_count=0
    def __init__(self,name,color):
        self.name=name
        self.color=color
        Car.car_count+=1
        print(f'총 생산된 차량은 {Car.car_count}대')
```

# 클래스 변수와 인스턴스 변수

# Car 인스턴스가 생성될 때마다 1씩 증가하는 클래스 변수 사용

```
class Car:
```

```
    car_count=0
```

```
    def __init__(self,name,color):
```

```
        self.name=name
```

```
        self.color=color
```

```
        Car.car_count+=1
```

```
        print(f'총 생산된 차량은 {Car.car_count}대')
```

클래스 변수는 앞에 클래스 이름을 붙여서 접근한다.

# 인스턴스 생성

```
car1=Car('Benz','red')
```

```
car2=Car('Audi','white')
```

# 인스턴스 생성없이 클래스 변수 사용

```
print(Car.car_count)
```

총 생산된 차량은 1대

총 생산된 차량은 2대

2

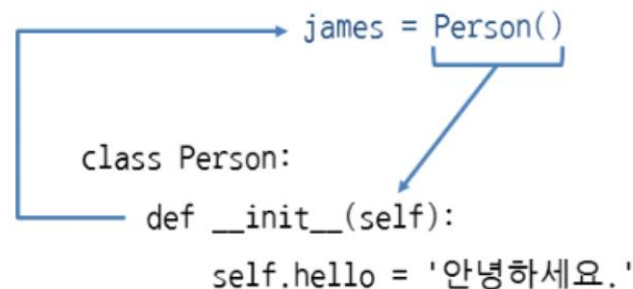
- 파이썬 메소드의 첫 매개변수 이름은 관례적으로 **self**를 사용한다. 다른 이름도 가능함.
- self는 인스턴스 자신.

```
class Person:
    def __init__(self):
        self.hello = '안녕하세요.'

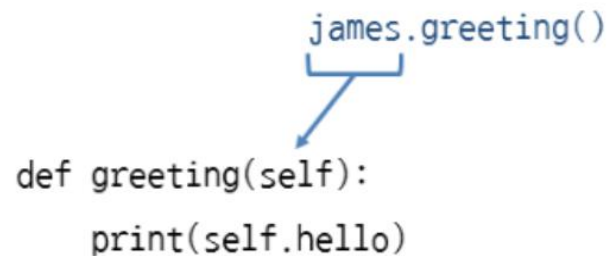
    def greeting(self):
        print(self.hello)

james = Person()
james.greeting()
```

- `__init__`의 매개변수 `self`에 들어가는 값은 `Person()`.



- `self`가 완성된 뒤 `james`에 할당되고 이후 메소드를 호출하면 현재 인스턴스(`james`)가 매개변수 `self`에 들어간다.



**캡슐화(encapsulation)**

# 캡슐화(encapsulation)의 필요성

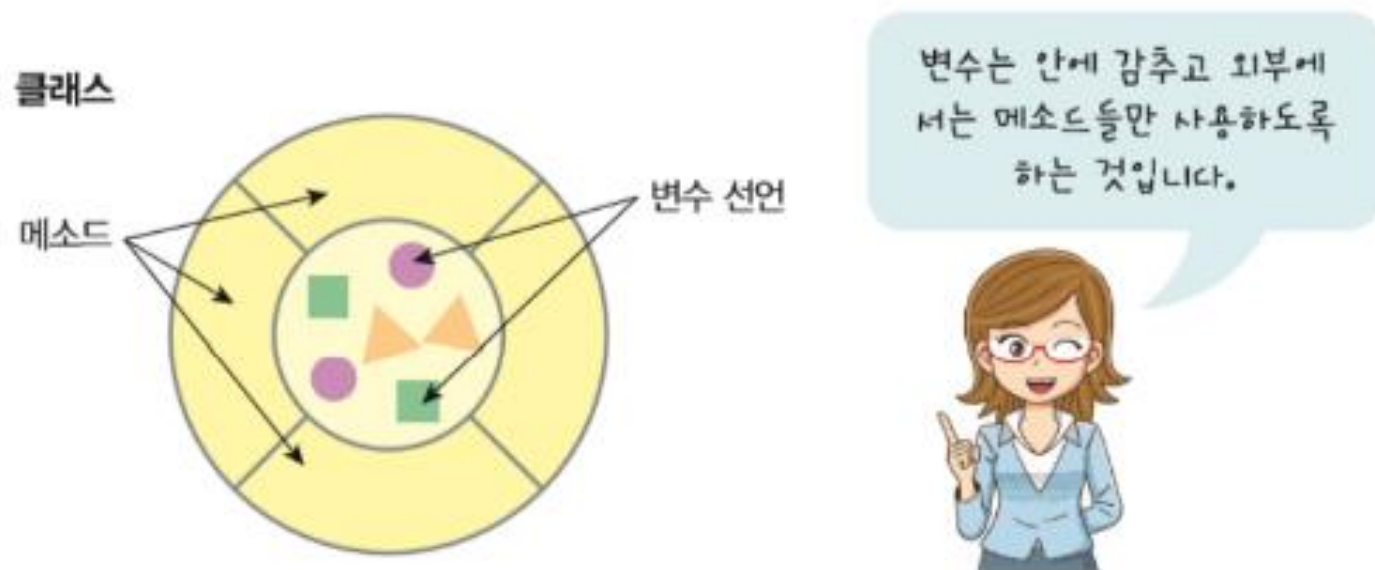
```
class Student:  
    def __init__(self, name, age=0):  
        self.name=name  
        self.age=age
```

```
st1=Student('hong', 20)  
st1.age=21  
print(st1.age)
```

```
st1.age=-10
```

나이를 음수로 변경하고 있다.

클래스 외부에서 마음대로 변경하지 못하게 구현의 세부 사항을 클래스 안에 감추는 것이 필요하다.



- 클래스 설계시 변수는 안에 감추고 외부에서는 메서드들만 사용하도록 설계하는 것이 좋다.
- 변수나 메서드를 `private`으로 정의하려면 **이름 앞에 언더바 2개(underscore)**를 붙이면 된다.
- 이런 변수나 메서드는 클래스 내부에서만 접근할 수 있다.

```
# step1. 공개 변수
class BankAccount:
    def __init__(self):
        self.balance=0

    def deposit(self,money):
        self.balance+=money

kim=BankAccount()
kim.deposit(100)
print(kim.balance)      # 결과: 100

# 클래스에 정의된 메서드를 거치지 않고 잔액 수정
kim.balance=5000
print(kim.balance)      # 결과: 5000
```

```
# step2. 캡슐화(비공개 변수)
class BankAccount:
    def __init__(self):
        self.__balance=0 ← 비공개로 선언

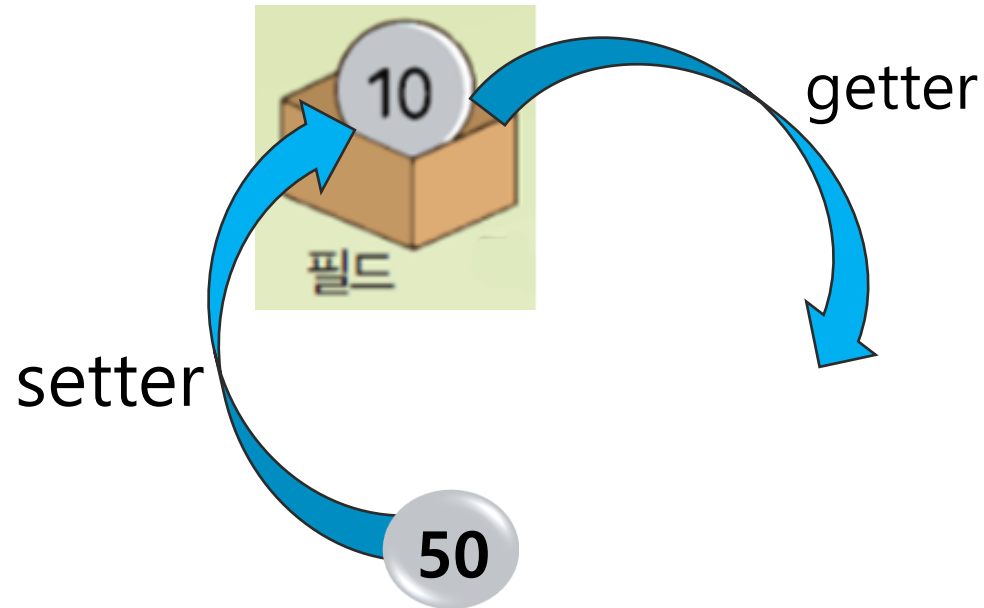
    def deposit(self,money):
        self.__balance+=money

    def get_balance(self):
        return self.__balance

kim=BankAccount()
kim.deposit(100)
print(kim.get_balance()) # 결과: 100

# 클래스에 정의된 메서드를 거치지 않고 잔액 수정
kim.__balance=5000
print(kim.get_balance()) # 결과: 100
```

- 비공개(private)인 필드와 메서드는 클래스 내부에서만 접근될 수 있다. 클래스 외부에서는 접근이 불가능하다. 하지만 **외부에서 이들 필드가 필요한 경우에는 어떻게 해야 할까? 내부의 값을 외부로 전달해 주는 메서드를 사용하면 된다.**



- 인스턴스 변수값을 반환하는 getter : 메서드 이름이 get으로 시작
- 인스턴스 변수값을 변경하는 setter : 메서드 이름이 set으로 시작



# get 접근자(getter)와 set 설정자(setter)



접근자와 설정자 메소드  
만을 통하여 필드에  
접근하여야 합니다.



# getter와 setter의 생성과 사용

```
# getter와 setter로 안전하게 hp 접근 가능하게 설계
class Car:
    def __init__(self, name, hp):
        self.name = name
        self.__hp = hp          # 생성자에서 self.set_hp(hp) 메서드 호출로도 가능

    def set_hp(self, hp):
        if hp < 0:
            raise ValueError('마력은 양수로 입력하세요!!!')
        self.__hp = hp

    def get_hp(self):
        return self.__hp

car1 = Car('benz', 200)
print(car1.get_hp())  # 200
car1.set_hp(220)
print(car1.get_hp())  # 220
car1.__hp = -5000
print(car1.get_hp())  # 220    -5000을 입력해도 car1 인스턴스의 hp는 변경x
```

# 은행 계좌를 클래스로 모델링 해보자.

- 계좌 번호(acc\_no)와 잔액(balance)을 입력받아 인스턴스를 생성하고 출금 메서드 withdraw(), 입금 메서드 deposit(), 조회 메서드 inquiry() 를 작성한다.  
단, 계좌 번호와 잔액은 비공개 필드로 설정한다.
- 출금(withdraw)시 잔액보다 출금액이 많으면 "잔액 부족"을 출력한다.

```
a = BankAccount(1234, 1000)
a.deposit(100)
a.withdraw(500)
a.withdraw(3000)
a.inquiry()
```

1234 번 계좌가 생성되었습니다.  
통장에 100 원이 입금되었습니다.  
통장에서 500 원이 출금되었습니다.  
잔액이 부족합니다.  
현재 잔액: 600원

```
class BankAccount:
    def __init__(self, acc_no, balance):
        self.__acc_no=acc_no
        self.__balance=balance
        print(self.__acc_no, '번 계좌가 생성되었습니다.')
    # 출금
    def withdraw(self, money):
        if self.__balance < money:
            print("잔액이 부족합니다.")
        else:
            self.__balance -= money
            print("통장에서", money, "원이 출금되었습니다.")
    # 입금
    def deposit(self, money):
        self.__balance += money
        print("통장에", money, "원이 입금되었습니다.")
    # 잔액 조회
    def inquiry(self):
        print("현재 잔액:", self.__balance, "원")
```

```
# 인스턴스 생성
a=BankAccount(1234, 1000)
a.deposit(100)
a.withdraw(1000)
a.withdraw(5000)

a.inquiry()
```

1234 번 계좌가 생성되었습니다.  
통장에 100 원이 입금되었습니다.  
통장에서 500 원이 출금되었습니다.  
잔액이 부족합니다.  
현재 잔액: 600원

- 계좌 번호(acc\_no)와 잔액(balance)을 입력받아 인스턴스를 생성하고 출금 메서드 withdraw(), 입금 메서드 deposit(), 조회 메서드 inquiry() 를 작성한다.

단, 계좌 번호와 잔액은 비공개 필드로 설정한다.

- 출금(withdraw)시 잔액보다 출금액이 많으면 "잔액 부족"을 출력한다.

- 이체 메서드 transfer()를 추가한다.**

```
me=BankAccount(1234,1000)
you=BankAccount(5555,50000)
me.transfer(you,100)
me.inquiry()
you.inquiry()
```

```
1234 번 계좌가 생성되었습니다.
5555 번 계좌가 생성되었습니다.
5555 계좌로 100 원이 송금되었습니다.
1234 의 잔액은 900 원입니다.
5555 의 잔액은 50100 원입니다.
```

```
def transfer(self,your_acc,money):
    if money>self.__balance:
        print('송금할 잔액이 부족합니다.')
    else:
        your_acc.deposit(money)
        self.__balance-=money
```

# 메서드 유형

- 인스턴스 메서드

```
def 함수명(self, ...):  
    수행할 코드
```

- 클래스 메서드

**클래스 변수를 사용하는 메서드이다.**

인스턴스를 생성하지 않아도 호출할 수 있다.  
"@classmethod" 데코레이터를 반드시  
표시해야 한다.

```
@classmethod  
def 함수명(cls, ...):  
    수행할 코드
```



**cls** 는 클래스라는 뜻

- 정적 메서드(static method)

모든 인스턴스들이 공유한다.

인스턴스를 생성할 필요가 없는 메서드이다.

"@staticmethod" 데코레이터를 사용한다.

```
@staticmethod  
def 함수명():  
    수행할 코드
```

# 클래스 메서드와 인스턴스 메서드

# 클래스 메서드와 인스턴스 메서드

```
class Car:
```

```
    __car_count=0
```

```
    def __init__(self,name,color):
```

```
        self.__name=name
```

```
        self.__color=color
```

```
        Car.__car_count+=1
```



**self**가 있는 메서드는 인스턴스 메서드이다. 인스턴스를 생성한 후에 사용 가능해진다.

```
    def show_info(self):
```

```
        print(f'차량이름:{self.__name}, 색상:{self.__color}')
```

인스턴스 메서드.

```
    @classmethod
```

```
    def show_count(cls):
```

```
        print(f'생산된 차량 수:{cls.__car_count}')
```

클래스 메서드로 선언.

# 인스턴스 생성

```
car1=Car('Benz','red')
```

```
car1.show_info()
```

```
car1.show_count()
```

# 클래스 메서드는 인스턴스, 클래스 모두 사용.

```
# Car.show_info()
```

# show\_info()는 인스턴스 메소드이므로 반드시 인스턴스 생성후 사용 가능함.

차량이름: Benz, 색상: red  
생산된 차량 수: 1



# 정적 메서드: 인스턴스를 생성하지 않고 사용할 메소드

# 인스턴스를 생성하지 않고 사용할 메소드를 만들때 사용

```
class Calc:
    @staticmethod
    def add(first, second):
        return first+second

    @staticmethod
    def subtract(first,second):
        return first-second
```

# 인스턴스를 생성하지 않고 메서드 사용

```
print(Calc.add(3,5))          # 8
print(Calc.subtract(3,5))     # -2
```

# 인스턴스를 생성하고 메서드 사용

```
c1=Calc()
print(c1.add(5,6))           # 11
```



인스턴스를 생성하지 않고 사용할  
메소드를 만드는 경우 @staticmethod 가  
생략돼도 된다.

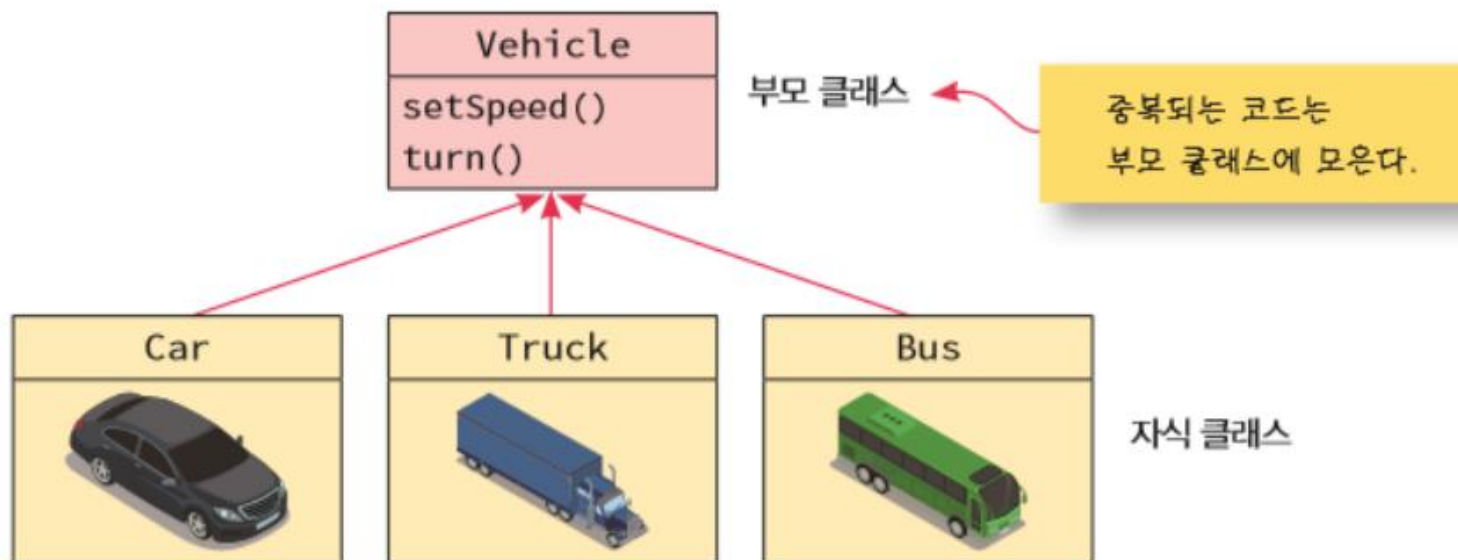
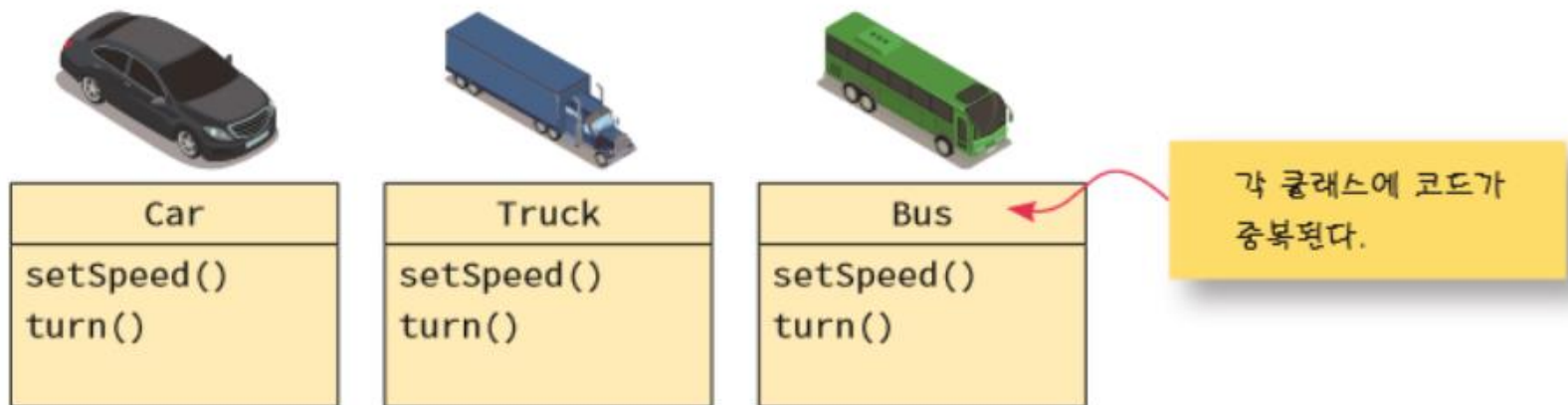
**상속(inheritance)**

- 다른 클래스의 기능을 물려받아 쓸 수 있게 하는 것.
- 다중 상속도 가능하다.
- 기존 클래스를 변경하지 않고 **새로운 기능을 추가하거나 기존 기능을 변경해서** 사용할 수 있다.


```
class 클래스명(슈퍼 클래스명):  
    코드
```


- 관련 용어들  
부모 클래스=슈퍼(super) 클래스=기반(base) 클래스  
자식 클래스=서브(sub) 클래스=파생(derived) 클래스

# 상속이 필요한 이유



내가 만든 클래스가 상속될 가능성이 있다면 나중을 위해 완벽한 설계 대신 공통 사항만 서술한다.

 동물 클래스 : 네 발로 달린다, 고기를 먹는다, 8시간씩 잔다...

 동물 클래스 : 달린다, 먹는다, 잔다...

```
class FourCalc:
    def __init__(self, first, second):
        self.first = first
        self.second = second
    def add(self):
        return self.first + self.second
    def subtract(self):
        return self.first - self.second
    def multiply(self):
        return self.first * self.second
    def divide(self):
        return self.first / self.second
```

```
class MoreCalc(FourCalc):
    def pow(self):
        return self.first ** self.second
```

```
a=MoreCalc(5,4)
print(a.pow())      # 결과 : 625
print(a.add())      # 결과 : 9
```

기존 FourCalc클래스를 상속  
받아서 새로운 기능을 추가한  
MoreCalc클래스를 정의하고  
있다.

# 클래스 상속-메서드 오버라이딩(overriding)

```
class FourCalc:
    def __init__(self, number1, number2):
        self.number1 = number1
        self.number2 = number2

    def add(self):
        return self.number1 + self.number2

    def subtract(self):
        return self.number1 - self.number2

    def multiply(self):
        return self.number1 * self.number2

    def divide(self):
        return self.number1 / self.number2
```

```
class MoreCalc(FourCalc):
    # 나누기는 0으로 나누지 못하게 변경
    def divide(self):
        if self.number2==0:
            return 0
        else:
            return self.number1 / self.number2
```

```
a=MoreCalc(5,0)
print(a.add())
print(a.divide())
```



## 메소드 오버라이딩(overriding)

부모 클래스에 있는 메소드를 동일한 이름으로 재정의하는 것

# 클래스 상속-슈퍼 클래스의 생성자 호출

- 상속시 주의사항

- 서브 클래스의 생성자(`__init__`) 구현시 반드시 슈퍼 클래스의 생성자를 먼저 호출해야 슈퍼 클래스의 변수를 사용할 수 있다.
- 서브 클래스의 생성자는 부모 클래스의 인스턴스 변수를 먼저 나열하고 서브 클래스의 인스턴스 변수들을 나열한다.

`super().__init__()`

부모=> `__init__(self, x, y):`  
자식=> `__init__(self, x, y, w, h):`

```
class Shape:
    def __init__(self, x, y):
        self.x = x
        self.y = y
    # 면적
    def area(self):
        pass
    # 둘레
    def perimeter(self):
        pass
```

```
class Rectangle(Shape):
    def __init__(self, x, y, w, h):
        super().__init__(x, y)
        self.w = w
        self.h = h
    def area(self):
        return self.w*self.h
    def perimeter(self):
        return 2*(self.w+self.h)

r = Rectangle(0, 0, 100, 200)
print("사각형의 면적", r.area())
print("사각형의 둘레", r.perimeter())
```



# 클래스 상속-메서드 오버라이딩(overriding)

```
class Car:
    def __init__(self, name, color, speed):
        self.name = name
        self.color = color
        self.speed = speed

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

    def down_speed(self, value):
        self.speed -= value
```

```
class Bus(Car):
    def __init__(self, name, color, speed, seat):
        super().__init__(name, color, speed)
        self.seat = seat

    def show_numSeats(self):
        return self.seat

class Truck(Car):
    def __init__(self, name, color, speed, payload):
        super().__init__(name, color, speed)
        self.payload = payload

    def show_payload(self):
        return self.payload

bus1 = Bus('붕붕', '노랑', 60, 10)
trk1 = Truck('붕붕', '노랑', 60, 340)
print(bus1.show_numSeats())          # 10
print(trk1.show_payload())           # 340
trk1.up_speed(30)
print(trk1.speed)                    # 90
```

# 클래스 상속과 다형성(polymorphism)

- 동일한 메서드가 다르게 동작하도록 하는 것으로 객체지향 프로그래밍의 특징 중 하나이다.



# 내장함수와 다형성의 예

```
print(len([1,2,3]))  
print(len('Python'))  
print(len({'name':'kim','age':20}))
```

# + 연산자의 다형성 예

```
print(1+2+3)  
print('a'+'c'+'e')
```

# 클래스 상속과 다형성(polymorphism)

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

    def speak(self):
        pass

class Cat(Animal):
    def speak(self):
        return '야옹~'

class Dog(Animal):
    def speak(self):
        return '멍~멍~'

animals=[Dog('dog1'), Dog('dog2'), Cat('cat1')]

for animal in animals:
    print(animal.name+':'+animal.speak())
```

```
dog1:멍~멍~
dog2:멍~멍~
cat1:야옹~
```

- 상속을 받은 서브 클래스에서 반드시 오버라이딩 해야 하는 메소드가 있을 때 추상 메소드로 선언함.
- 사용법:

```
raise NotImplementedError()
```

```
class Shape:
    def draw(self):
        raise NotImplementedError()

class Circle(Shape):
    def draw(self):
        print('원을 그립니다.')

class Rectangle(Shape): # draw()메소드를 오버라이딩하지 않음.
    pass

c=Circle()
c.draw()
r=Rectangle()
r.draw()           # 오버라이딩하지 않았으므로 오류가 발생됨.
```

특수 메서드

# 특수 메서드(special method or magic method)

- 메서드 앞뒤에 언더바(\_) 2개가 붙은 메서드.
- 이 메소드들은 특별한 기능을 갖고 있다. 대표적으로 `__init__()`도 매직 메소드이다.
- `__del__()` 메소드 : 소멸자(destructor)라고 부른다. 인스턴스가 삭제될 때 자동으로 호출된다.  
인스턴스를 삭제하는 `del(인스턴스명)` 명령 사용시 자동으로 호출된다.  
따라서 인스턴스가 삭제될 때 특별히 작업할 명령이 있다면 이 안에 작성하면 됨.
- `__str__()` 메소드 : 인스턴스를 `print()`문으로 출력할 때 실행되는 메소드이다.  
`print(인스턴스명)` 시에 자동 호출된다.
- `__add__()` 메소드 : 인스턴스 사이에 덧셈 작업이 일어날 때 자동 호출되어 실행된다.
- `__lt__()`, `__le__()`, `__gt__()`, `__ge__()`, `__eq__()`, `__ne__()` : 인스턴스 사이에 비교 연산자 사용시 자동호출된다.

# 특수 메서드(special method)

```
class Car:
    car_cnt=0
    def __init__(self,color,speed):
        self.color=color
        self.speed=speed
        Car.car_cnt+=1

    def __str__(self):
        return(f"total:{Car.car_cnt}, color:{self.color}, speed:{self.speed}")

    def __del__(self):
        Car.car_cnt-=1

benz=Car("red",5)
bmw=Car("blue",4)
morning=Car("gray",3)
print(benz)
del benz
print(bmw)
```

total:3, color:red, speed:5  
total:2, color:blue, speed:4



# 특수 메서드와 관련된 연산자들

연산자	메소드	설명
<code>x + y</code>	<code>__add__(self, y)</code>	덧셈
<code>x - y</code>	<code>__sub__(self, y)</code>	뺄셈
<code>x * y</code>	<code>__mul__(self, y)</code>	곱셈
<code>x / y</code>	<code>__truediv__(self, y)</code>	실수나눗셈
<code>x // y</code>	<code>__floordiv__(self, y)</code>	정수나눗셈
<code>x % y</code>	<code>__mod__(self, y)</code>	나머지
<code>divmod(x, y)</code>	<code>__divmod__(self, y)</code>	실수나눗셈과 나머지
<code>x ** y</code>	<code>__pow__(self, y)</code>	지수
<code>x &lt;&lt; y</code>	<code>__lshift__(self, y)</code>	왼쪽 비트 이동
<code>x &gt;&gt; y</code>	<code>__rshift__(self, y)</code>	오른쪽 비트 이동
<code>x &lt;= y</code>	<code>__le__(self, y)</code>	less than or equal(작거나 같다)
<code>x &lt; y</code>	<code>__lt__(self, y)</code>	less than(작다)
<code>x &gt;= y</code>	<code>__ge__(self, y)</code>	greater than or equal(크거나 같다)
<code>x &gt; y</code>	<code>__gt__(self, y)</code>	greater than(크다)
<code>x == y</code>	<code>__eq__(self, y)</code>	같다
<code>x != y</code>	<code>__neq__(self, y)</code>	같지않다

- 파이썬에서는 클래스 작성시, 부모 클래스를 명시적으로 지정하지 않으면 object 클래스의 자식 클래스로 암묵적으로 간주된다. 따라서 모든 클래스의 가장 위에는 object 클래스가 있다고 생각하면 된다.
- object 클래스는 모든 클래스에 공통적인 메서드를 구현한다.

