OBJECT-ORIENTED PROGRAMMING

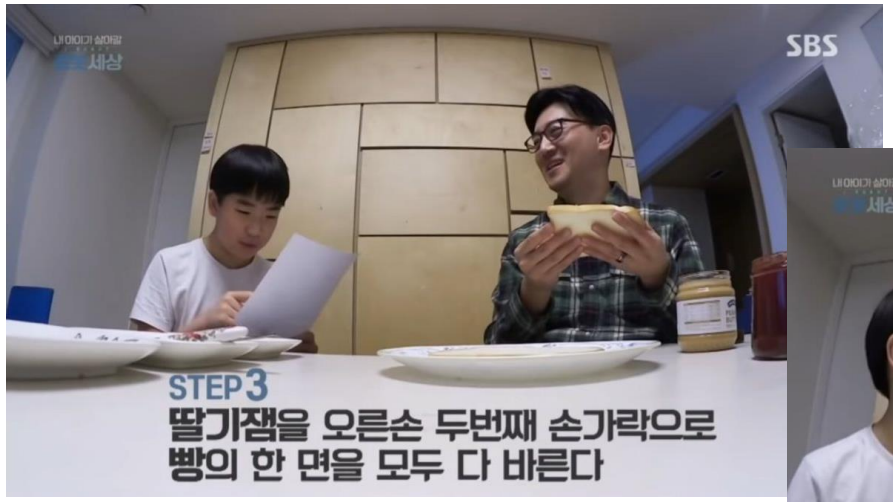
Procedure Programming

택배 전달 프로그램을 만들어 보자!

1. 택배를 포장한다.
2. 택배 송장을 작성한다.
3. 택배를 택배 회사에 전달한다.
4. 택배를 허브 터미널로 전달한다.
5. 택배를 지방 영업소로 전달한다.
6. 택배 기사에게 택배를 배정한다.
7. 택배 기사가 도착지에 전송한다.

택배 보내는 절차를 차례대로 작성
→ 절차 지향 프로그래밍

Procedure Programming



아버지와 함께하는

절차 지향 프로그래밍

https://www.youtube.com/watch?t=199&v=ZTxs0II20Iw&feature=youtu.be

Problem of Procedure Programming

중간의 코드를 수정한다면?

1. 택배를 포장한다.
2. 택배 송장을 작성한다.
3. 택배를 택배 회사에 전달한다.
4. **택배를 허브 터미널로 전달한다.**
5. 택배를 지방 영업소로 전달한다.
6. 택배 기사에게 택배를 배정한다.
7. 택배 기사가 도착지에 전송한다.

허브터미널에서 택배 송장 양식 변경

→송장 작성부터 도착지 전송까지 모든 코드 수정이 필요할 수도 있음

협업을 한다면?

1. 택배를 포장한다.
2. 택배 송장을 작성한다.
3. 택배를 택배 회사에 전달한다.
4. 택배를 허브 터미널로 전달한다.
5. 택배를 지방 영업소로 전달한다.
6. 택배 기사에게 택배를 배정한다.
7. 택배 기사가 도착지에 전송한다.

주황색: 프로그래머 1 **연두색: 프로그래머 2**

큰 코드를 여러 명이서 함께 작성

→한 프로그래머의 수정 사항이 다른 프로그래머에게 큰 영향

Idea of Object-Oriented Programming

코드를 객체 단위로 나눌 필요가 있다!

1. 택배를 포장한다.
2. 택배 송장을 작성한다.
3. 택배를 택배 회사에 전달한다.
4. 택배를 허브 터미널로 전달한다.
5. 택배를 지방 영업소로 전달한다.
6. 택배 기사에게 택배를 배정한다.
7. 택배 기사가 도착지에 전송한다.

고객

택배 회사

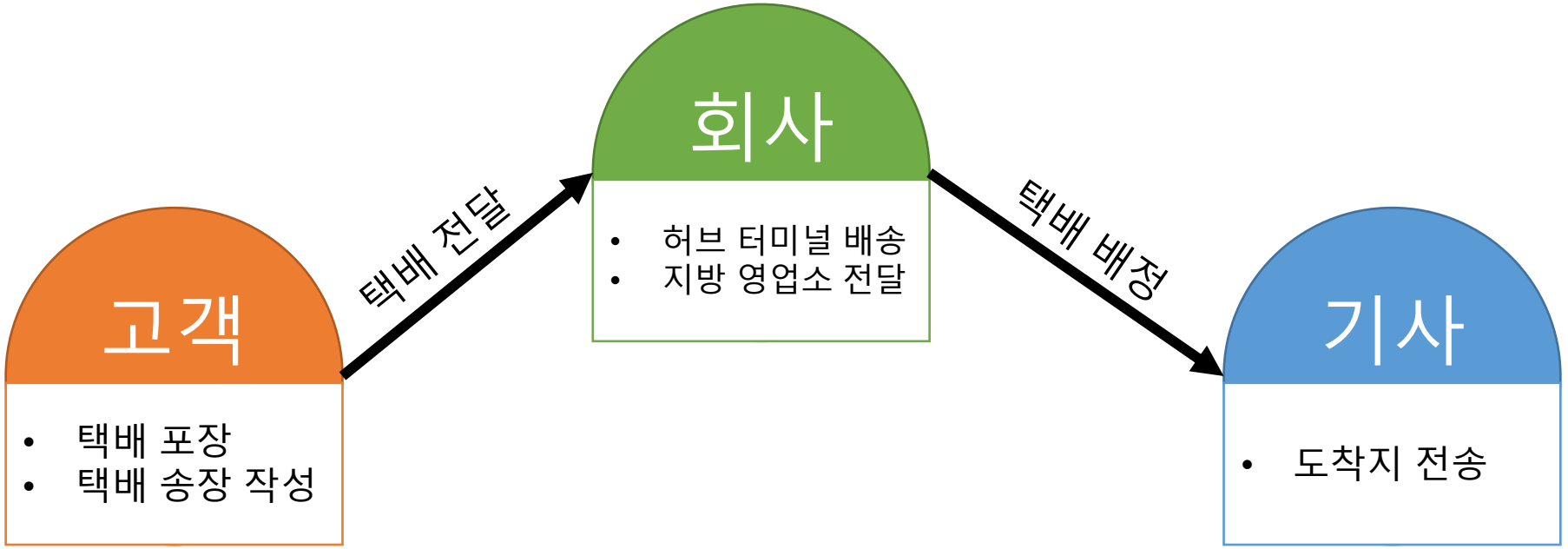
택배 기사

객체 단위 코드 작성 및 분업

→ 객체 지향 프로그래밍

Idea of Object-Oriented Programming

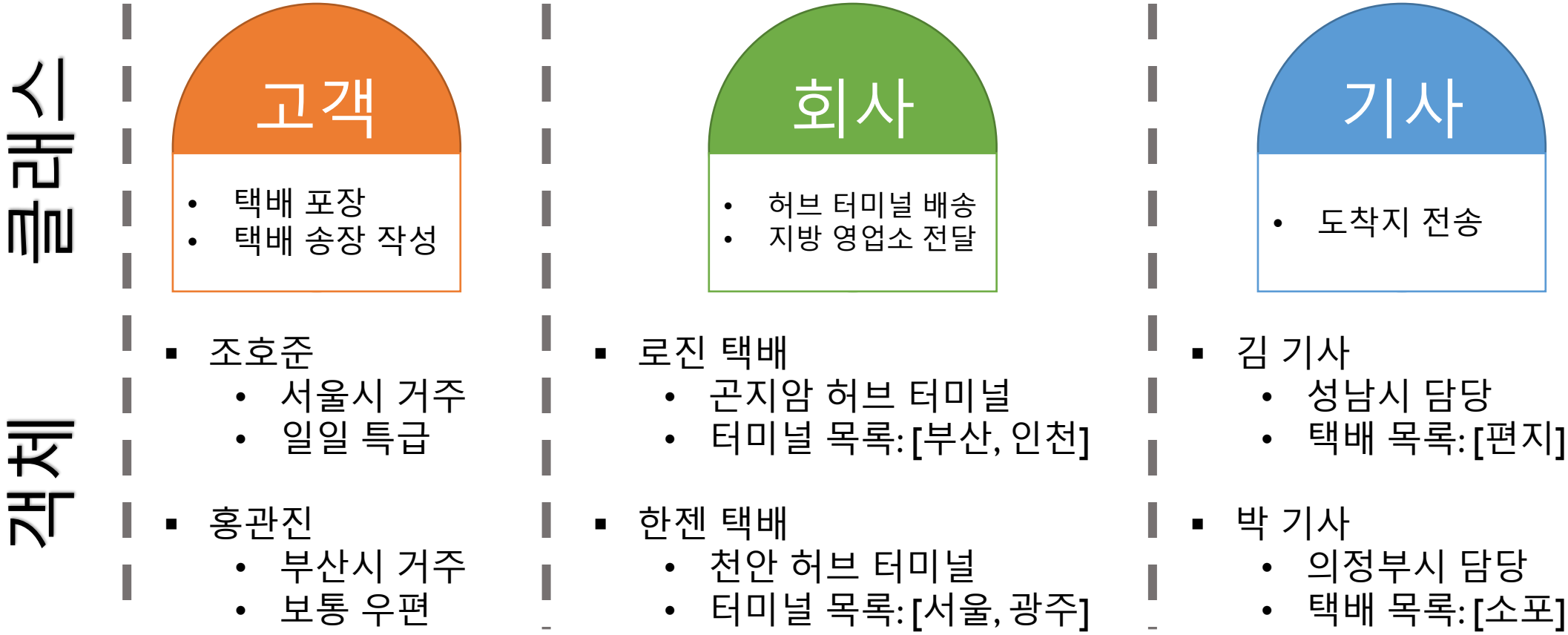
코드를 객체 단위로 나눌 필요가 있다!



객체 단위 코드 작성 및 분업

→ 객체 지향 프로그래밍

Class & Object



각 종류(Class)당 객체(Object)가 하나만 존재하진 않는다

그러나 각 객체의 데이터(Attribute)는 달라도 행동(Method)은 동일하다!

OOP Example

택배 기사의 객체를 만들어 보자

Class (설계도)

- Attribute (데이터, 속성)
  - 기사 이름
  - 담당 배송지
  - 현재 배달중인 택배
- Method (행동)
  - 택배 배정 받기
  - 택배 배송하기

Instance (객체)

- Attribute (데이터, 속성)
  - 김 기사
  - 경기도 성남시 분당구
  - [TV 상자, 서류 상자, 편지]
- Method (행동)
  - 택배 배정 받기
  - 택배 배송하기

Class Example

택배 기사의 틀(class)을 만들어 보자

```
class Courier(object):          # (object) 는 생략 가능
    NATIONALITY = 'KOR'         # 클래스 속성 (Class attribute)

    def __init__(self, name: str, address: str):      # 생성자
        self.name = name                              # 속성 (Attribute)
        self.address = address
        self.parcels = []

    def assign(self, parcel: str) -> None:
        self.parcels.append(parcel)

    def deliver(self) -> None:
        for parcel in self.parcels:
            print(parcel, "배달 중")
```

Class Example

택배 기사 객체(Instance)를 만들고 써보자

```
# 객체 생성
courier1 = Courier("김 기사", "경기도 성남시 정자동")

# 속성 출력
print(courier1.name, "-", courier1.address, "근무 중")

# 메소드 실행
courier1.assign("TV 상자")
courier1.assign("편지")
courier1.deliver()
```

Class Declaration

Class 선언부

```
class Courier(object):
```

예약어 — 클래스 이름 — 부모 클래스

- 클래스 이름은 CamelCase가 관습적으로 사용됨
- 부모 클래스가 지정되지 않았을 시 object가 자동 상속 (python3)

클래스 속성

```
class Courier(object):
    NATIONALITY = 'KOR'     # 클래스 속성 (Class attribute)
```

- 클래스 전체가 공유하는 속성 값
- 모든 객체 (instance)가 같은 값을 참조
- 남용하면 스파게티 코드의 원인이 됨

- *클래스.attribute* 형태로 접근 (*객체.attribute* 형태로도 접근 가능)

```
# 속성 출력
print(Courier.NATIONALITY)      # courier1.NATIONALITY
```

클래스 함수 (Method)

```
class Courier:
    # 클래스 함수 (Method)
    def assign(self, parcel: str) -> None:
        self.parcels.append(parcel)
```

- 각 객체에 적용이 가능함 함수
- 현재 수정하고자 하는 객체를 "self"로 지칭 (관습)
  - C와 Java에서의 "this"
  - **파이썬은 "self"를 첫번째 파라미터로 명시적으로 받음**
- Class.method(instance, args, …) 혹은 **instance.method(args, …)**

```
# 메소드 실행
courier1.assign("TV 상자")
```

Class Attribute

객체 속성

```
class Courier(object):
    def __init__(self, name: str, address: str):      # 생성자
        self.name = name                              # 속성 (Attribute)
```

- 각각의 객체가 개인적으로 가지는 값
- instance.attr 의 형태로 접근
- class 형태로 선언되어 나온 객체는 언제 어디서든 attribute 수정 가능

```
courier1 = Courier("김 기사", "경기도 성남시 정자동")
courier1.value = 10      # 존재하지 않는 속성에 값 부여가 가능
print(courier1.value)    # 그러나 권장하지 않으므로 가능하면 생성자에서 설정
```

Magic Method: Constructor

- 메소드 이름이 "*__METHOD__*" 형태일 경우 특별한 Magic Method

생성자 (__init__)

```
class Courier(object):
    def __init__(self, name: str, address: str):      # 생성자
        self.name = name                              # 속성 (Attribute)
        self.address = address
        self.parcels = []
```

- 객체를 생성할 때 호출됨
- 일반적으로 객체의 속성을 초기화 하는데 사용
- Class(args, …) 형태로 호출하여 객체 생성
- 거의 유이하게 정해진 Argument format이 없는 Magic Method

```
# 클래스 생성
courier1 = Courier("김 기사", "경기도 성남시 정자동")
```

Magic Method: Destructor

소멸자 (__del__)

```
class Courier(object):
    def __del__(self):                                  # 소멸자
        self.parcels.clear()
```

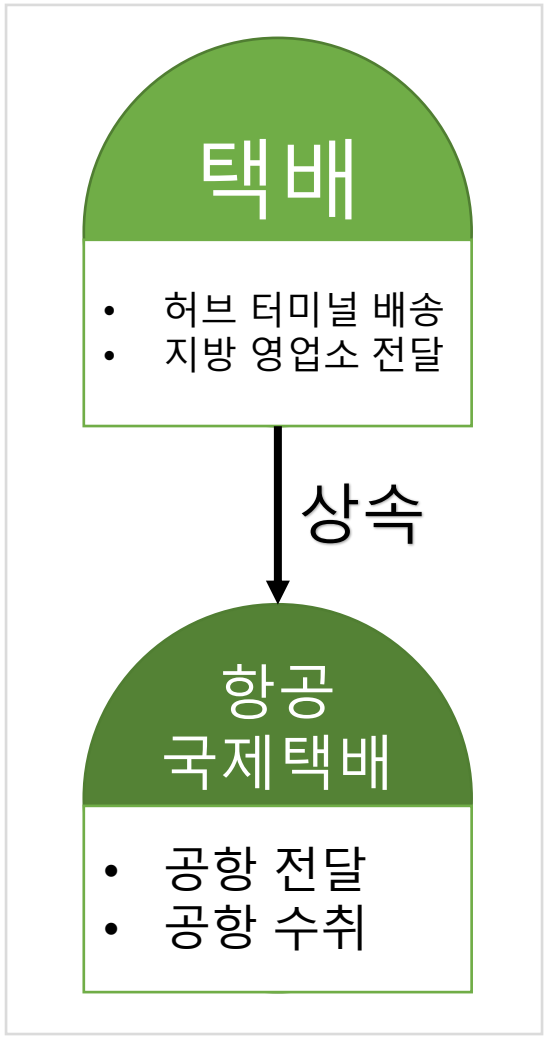
- 객체를 소멸할 때 호출됨
- 파이썬은 쓰레기 수거(Garbage Collection)로 메모리 관리
  - 객체가 어디에서도 참조되지 않을 때 객체가 소멸
  - 소멸 타이밍을 잡기 어려워 잘 사용되지 않음
- del 명령어
  - 변수 이름을 명시적으로 없애기 가능
  - 참조를 명시적으로 삭제하는 것이지 객체를 명시적으로 삭제하는 것이 아님

```
del courier1    # 명시적으로 이름을 지우더라도 GC되지 않으면 사라지지 않는다!
```

Three Elements of OOP

파이썬의 3요소 구현방식을 알아보자

Inheritance



기존에 구현틀 상속 → 새로운 틀 제작

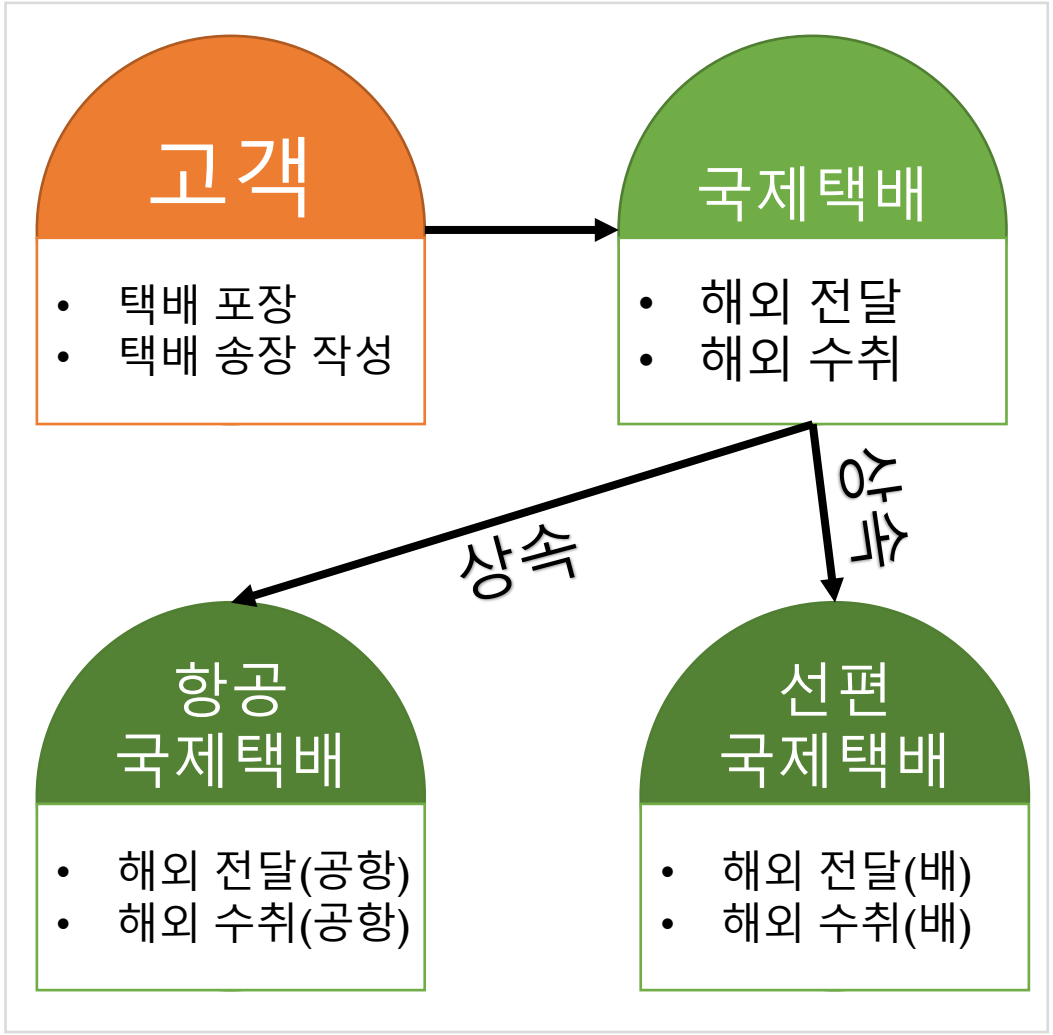
- 기존의 틀: 부모 Class, 새로운 틀: 자식 Class
- 자식 Class에서는 부모의 기능을 이용 가능
- **같은 기능을 재작성할 필요가 없음**

```
class Courier(object):
```

예약어 클래스 이름 **부모 클래스**

상속
(Inheritance)

Polymorphism



같은 이름의 메소드를 다르게 작성

- 각 자식 클래스가 다른 클래스와 차별
- 부모 메소드로 접근시 자식 메소드 실행
- 외부에서는 똑같은 API로 접근
- **고객 입장에서는 코드 수정이 없음**

파이썬에서는 동적 타이핑

- 고객 입장에서는 클래스 구분 X

→ 같은 이름의 메소드가 있으면 실행

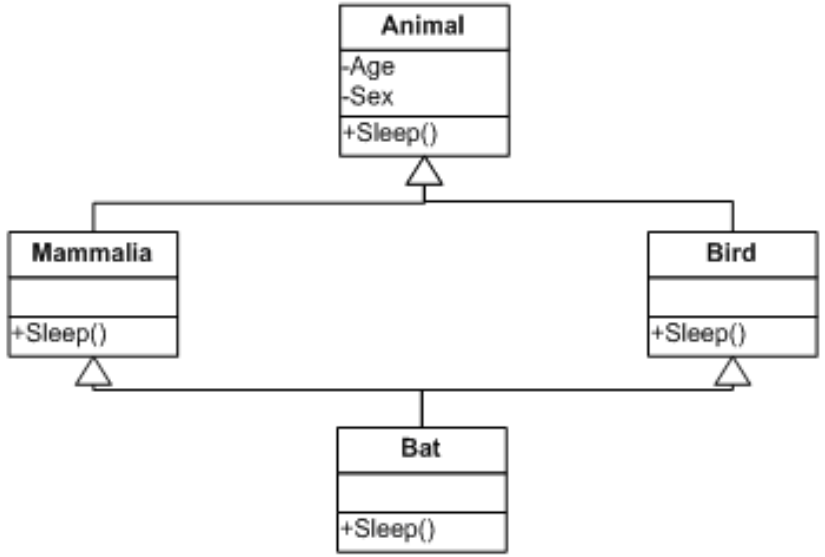
→ **Duck Typing**

다형성
(Polymorphism)

Inheritance & Polymorphism

- **Python에서의 상속과 다형성**
  - 다중 상속 지원

  → 죽음의 다이아몬드
    - 메서드 탐색 순서를 따름 (mro)
  - super 내장 함수를 이용하여 상위 클래스 접근 가능



Inheritance & Polymorphism

```python
class Courier:
    def __init__(self, name: str):
        self.name = name
        self.parcels = []

    def assign(self, parcel: str) -> None:
        self.parcels.append(parcel)

    def deliver(self) -> None:
        for parcel in self.parcels:
            print(parcel, "배달 중")

class JejuCourier(Courier):                          # Courier 상속
    def __init__(self, name: str, ticket: int):
        super().__init__(name)                       # 부모 클래스 생성자 접근, 정해진 부르는 타이밍은 없다.
        self.ticket = ticket

    def deliver(self) -> None:
        print(self.ticket, "티켓으로 제주도 이동")
        super().deliver()

courier = JejuCourier('김 기사', 15)
courier.assign('편지')
courier.deliver()
super(JejuCourier, courier).deliver()                # super로 언제나 원하는 상위 클래스로 변환 및 접근
```

Static & Class Method

```
class Number:
    Constant = 10

    @staticmethod
    def static_factory():
        obj = Number()
        obj.value = Number.Constant
        return obj

    @classmethod
    def class_factory(cls):
        obj = cls()
        obj.value = cls.Constant
        return obj

number_static = Number.static_factory()
number_class = Number.class_factory()
print(number_static.value, number_class.value)
```

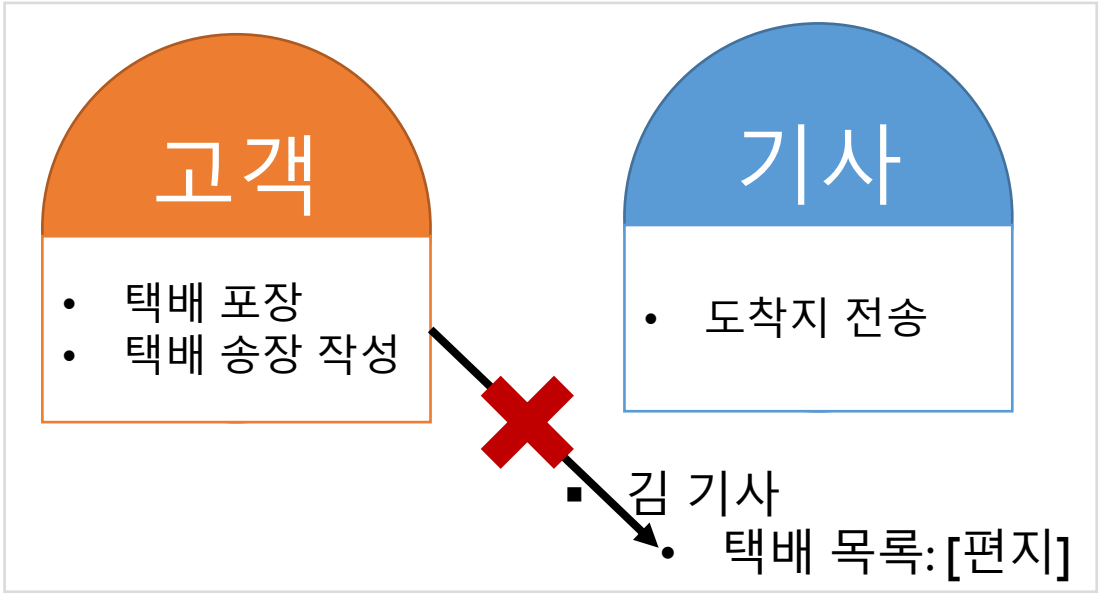
파이썬에는 2가지 정적 함수 존재

- instance.method 형태로도 접근 가능
- 일반적으로 Class.method 형태로 사용

- Static Method
  - staticmethod 꾸밈자 사용
  - 특별한 argument를 받지 않음
  - 일반적으로 class 내 유틸 함수로 사용
  - Class를 일종의 Namespace로 사용
- Class Method
  - Classmethod 꾸밈자 사용
  - 호출된 class인 **cls**를 받음 (self와 비슷)
  - factory 패턴에서 사용

상속하면 차이가 발생

Visibility



다른 클래스에게 객체의 내부를 감추기

- 캡슐화, 정보 은닉
- 클래스 간 간섭 최소화
- 최소한의 정보만을 지정된 API로 공개
- C나 Java에선 private & protected로 구현

- **Python에서의 가시성**
  - 명시적인 private & protected 범위가 없음 → 모두 public
  - private 변수/함수 이름 앞에 "__"를 붙임 (밑줄 2개)
    - Ex) self.__name, self.__sid
  - protected 변수/함수 이름 앞에 "_"를 붙임 (밑줄 1개)
    - Ex) self._name, self._sid

가시성, 캡슐화 (Visibility)

Visibility

```
class TestClass(object):
    def __init__(self):
        self.attr = 1           # Public
        self._attr = 2          # Protected
        self.__attr = 3         # Private

instance = TestClass()
print(dir(instance))
```

['_TestClass__attr', '_attr', 'attr']를 포함

- "__"의 경우 변수명 앞에 Class 이름을 넣어 Mangling – 자식과 이름이 안 겹침
- Private와 Protected는 코드 완성 등에서 안 보임
- **그러나 둘 다 public과 기능적 차이는 없다 (밖에서 접근 가능함)**
- 굳이 구분하는 이유 → **가독성** 때문 (API를 이용하는 사람에게 알려주기 용)

Property

```python
class Circle(object):
    PI = 3.141592
    def __init__(self, raidus=3.):
        self.radius = raidus

    def get_area(self):
        return Circle.PI * self.radius ** 2

    def set_area(self, value):
        self.radius = (value / Circle.PI) ** .5

circle = Circle(5.)
print(circle.get_area())
circle.set_area(10)

print(circle.radius)
```

```python
class Circle(object):
    PI = 3.141592
    def __init__(self, raidus=3.):
        self.radius = raidus

    @property
    def area(self):
        return Circle.PI * self.radius ** 2

    @area.setter
    def area(self, value):
        self.radius = (value / Circle.PI) ** .5

circle = Circle(5.)
print(circle.area)

circle.area = 10.
print(circle.radius)
```

- Property를 통해 Getter, Setter를 명시적 설정 가능
- Encapsulation 등에 활용

Magic Methods

__str__	__nonzero__	__getitem__	__add__	__neg__
__repr__	__getattr__	__setitem__	__sub__	__abs__
__lt__	__setattr__	__delitem__	__mul__	__int__
__le__	__delattr__	__reversed__	__floordiv__	__long__
__eq__	__get__	__contains__	__mod__	__float__
__ne__	__set__	__setslice__	__divmod__	__oct__
__gt__	__delete__	__delslice__	__pow__	__hex__
__ge__	__call__	__lshift__	__and__	__index__
__cmp__	__len__	__rshit__	__xor__	__enter__
__hash__	__iter__	__div__	__or__	__exit__

파이썬에는 __init__, __del__외에도
다양한 매직 메소드가 존재

Magic Method: Indexing

Indexing 메소드 (__getitem__, __setitem__)

```
class DoubleMapper:
    def __init__(self):
        self.mapping = {}

    def __getitem__(self, index):              # Indexing get
        return self.mapping.get(index, index * 2)

    def __setitem__(self, index, item):        # Indexing set
        self.mapping[index] = item

mapper = DoubleMapper()
print(mapper[10], mapper[1, 2])
mapper[10] = 15
print(mapper[10], mapper[1, 2])
```

- [ ] indexing을 재정의

Magic Method: Length

Length 메소드 (__len__)

```
class Dataset:
    def __init__(self, data, times=3):
        self.data = data
        self.times = times

    def __len__(self):                    # len(instance) 호출될 시 호출
        return len(self.data) * self.times

    def __getitem__(self, index):
        if index > len(self):
            raise IndexError()

        return self.data[index % len(self.data)]

dataset = Dataset([10, 2, 5, 2], times=5)
print(len(dataset))
```

Magic Method: Typing

```
class Courier:
    def __init__(self, name: str, address: str):
        self.name = name
        self.address = address

    def __str__(self):                          # str 형변환
        return self.address + ' 담당 ' + self.name

courier = Courier("김 기사", "경기도 성남시 정자동")
text = str(courier)                             # str 형변환 호출
```

- 객체를 다른 타입으로 형 변환할때 호출
- 이외에도 __int__, __float__, __bool__ 등이 존재

Magic Method: Comparison Operator

```python
class Courier:
    def __init__(self, name: str, cid: int):
        self.name = name
        self.cid = cid

    def __lt__(self, other):                    # < 연산자를 재정의
        return self.cid < other.cid

couriers = [
    Courier("김 기사", 56),
    Courier("박 기사", 72),
    Courier("정 기사", 62)
]

print(*[courier.name for courier in sorted(couriers)])    # sorted 사용 가능
```

- A < B를 호출 → A.__lt__(B)를 호출
- 이외에도 __le__, __gt__, __ge__, __eq__, __ne__ 가 존재

Magic Method: Arithmetic Operator

```
class MyComplex:
    def __init__(self, real, imaginary):
        self.real = real
        self.imaginary = imaginary

    def __str__(self):
        return str(self.real) + '+' + str(self.imaginary) + 'j'

    def __add__(self, other):                # + 연산자 재정의
        return MyComplex(                    # Out-place 연산
            self.real + other.real,
            self.imaginary + other.imaginary
        )

a = MyComplex(3, -5)
b = MyComplex(-6, 7)
print(a + b)
```

- 이외에도 __sub__, __mul__ 등이 존재
- In-place 버전인 __iadd__ 가 존재 (이 경우 self를 직접 수정 필요)

Magic Method: Callable

함수화 메소드 (__call__)

```
class AdditionNumber(object):
    def __init__(self, number: int):          # 생성자
        self.number = number

    def __call__(self, number: int):          # 함수화 메소드
        return number + self.number

addition_5 = AdditionNumber(5)
print(addition_5(10))                         # 객체를 함수처럼 사용
```

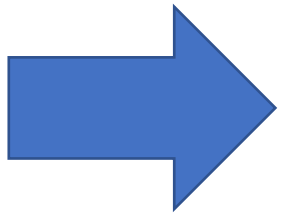
- 생성된 객체를 호출 가능하게 만듦
- instance(args, …)가 instance.__call__(args, …)를 호출

Magic Method: Iterable

실제 for 문에서 일어나는 일은?

```
seq = [1, 2, 3, 4, 5]

for elem in seq:
    print(elem)
```



```
seq = list([1, 2, 3, 4, 5])

iterator = iter(seq)
while True:
    try:
        elem = next(iterator)
    except StopIteration:
        break
    print(elem)
```

Magic Method: Iterable

실제 for 문에서 일어나는 일은?

```
seq = list([1, 2, 3, 4, 5])

iterator = iter(seq)
while True:
    try:
        elem = next(iterator)
    except StopIteration:
        break
    print(elem)
```

- iter 내장함수
  - 해당 객체의 순환자 반환
  - **__iter__** 호출
- next 내장함수
  - 해당 순환자를 진행
  - **__next__** 호출
- 끝에서 StopIteration Exception

- Generator는 자동으로 __iter__와 __next__가 구현

https://docs.python.org/ko/3/library/stdtypes.html#typeiter

Magic Method: Context Manager

```python
class Courier:
    def __init__(self, name: str):
        self.name = name

    def __enter__(self): # with 구문에 들어갈 때 사용, return 값이 as 이하로 할당
        self.parcels = []
        return self

    def __exit__(self, exc_type, exc_value, trace): # with 구문 나갈 때 적용
        for parcel in self.parcels:
            print(parcel, "배달 실패")
        self.parcels.clear()

courier = Courier("김 기사")
with courier:
    courier.parcels.append("소포")

with Courier("김 기사") as courier:
    courier.parcels.append("소포")
```

- 소멸자 대용으로 특정 Block 입장/종료 시 자동으로 호출
- File description 등을 자동으로 닫고자 할 때 사용

https://docs.python.org/ko/3/library/stdtypes.html#context-manager-types