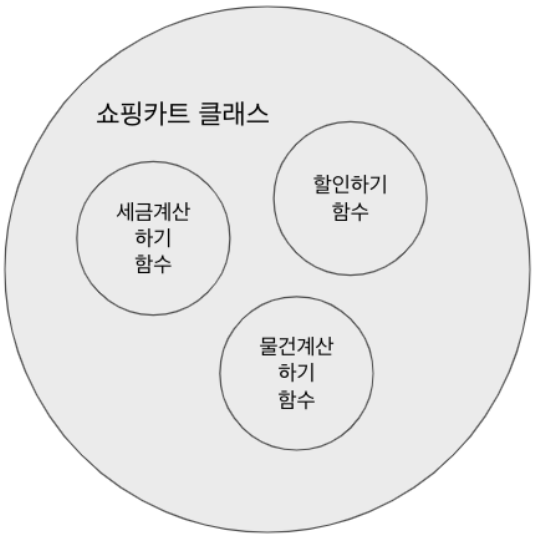
OOP의 구성

소요시간 : 11 min

> OOP의 구성을 보면서 OOP에 대해 가볍게 이해해봅니다.

OOP를 위한 구성을 파악한다.

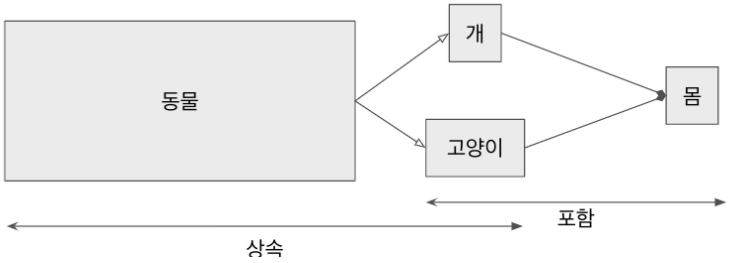
캡슐화



- 기본개념 : 내부 속성(변수)과 함수를 하나로 묶어서 클래스로 선언하는 일반적인 개념
    - 캡슐화형태로 코드를 작성하지 않으면 **특정 기능(함수, 변수)에 직접 접근**하게 되는 상황이 된다.
    - 기능이 많아질수록 재사용의 개념을 활용하기가 어려움
    - 해당 접근제어 관련해서는 **Note 04**에서 추가적으로 다룹니다.

```
class Encap:
  def __init__(self,value):
    self.value = value
    print('init :', self.value)

  def _set(self):
    print('set :', self.value)

  def printTest(self):
    print('printTest :', self.value)

  # def __printTest2(self):
  #   print('printTest :', self.value)

# object 생성
e = Encap(10)

# object 실행
# 케이스1
e.__init__(20)
e._set()
e.printTest()
#e.__printTest2()


print('\n')

# 케이스2
e.__init__(30)
e._set()
e.printTest()
```

상속과 포함(Inheritance & Composition)



- object의 종류는 현실 세계에 있는 대부분이기 때문에, 설계될 수 있는 다양한 object가 있다.

"개는 동물이다." 또는 "선생님은 직장인이다."

기본개념 : 상위 클래스의 모든 기능(함수, 변수

- 포함(Composition)

"개는 몸을 갖고 있다." 라는 관계로서 설명된다

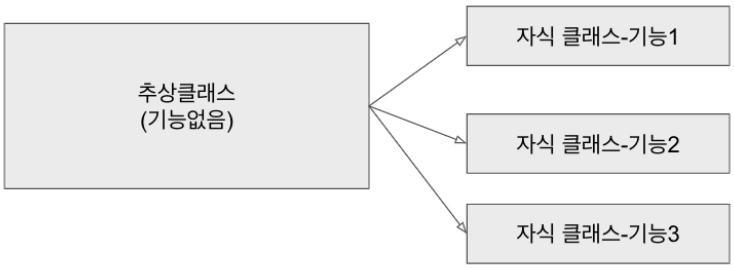
기본개념 : 다른 클래스의 일부 기능(함수)만을

```
# 상속코드

# 클래스 선언
class Person:
    def __init__(self, name):
        self.name = name

class Student(Person):      # Person 클래스 상
    def study(self):
        print (self.name + " studies hard")

class Employee(Person):     # Person 클래스 상
    def work(self):
        print (self.name + " works hard")

# object 생성
s = Student("Dave")
e = Employee("David")

# object 실행
s.study()
e.work()
```

```
# 클래스 선언
class Person:
    def __init__(self, age):
        self.age = age

    def printPerson(self):          # 포함을
        print('Person_printPerson')

class Student:
    def __init__(self, age):
        self.age = age
        self.p = Person(self)       # Studen

    def aging(self):
        return self.age

    def personComposit(self,age):
        return age, self.p.printPerson()  #

# object 생성
s = Student(10) # 한 번 생성된 object는 파라미
p = Person(20)

# object 실행
print("s.aging() :", s.aging())   # result :
print('\n')


print('test')
print('print with test', "s.personComposit()
# result 출력 순서 : Person.printPerson -> pr


print('\n')
print('test2')
print('print with test2', "p.printPerson() :
# result 출력 순서 : Person.printPerson -> pr
```

```
# 클래스 선언
class Bill():
    def __init__(self, description):
        self.description = description


class Tail():
    def __init__(self, length):
        self.length = length


class Duck():
    def __init__(self, bill, tail):
        self.bill = bill
        self.tail = tail

    def about(self):
        print(
            f"This duck has a {self.bill.des

# object 생성
duck = Duck(Bill('bill object'), Tail('tail

# object 실행
duck.about()
```

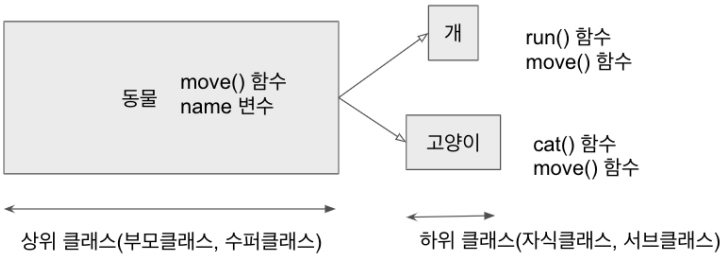
추상화



- 기본개념 : 추상화(abstraction)는 복잡한 내용에서 핵심적인 개념 및 기능을 요약하는 것을 말한다.
    - object의 기능에 따라 추상클래스(상위클래스)를 상속받아 개별적으로 클래스(하위클래스)를 생성한다.

- 실제 실행되는 기능은 선언된 추상클래스를 **상속받은 다른 클래스의 메소드**에서 확인할 수 있다.
- 추상클래스를 사용하는 이유
  - 대형 프로젝트를 진행하는 경우 또는 프로그램이 복잡해지는 경우 **1차적인 설계를 위해** 기능을 추상화시켜놓고, 활용여부는 차후 결정하기 위함이다.

```
from abc import *    # abc 모듈의 클래스와 메소

# 추상 클래스
class People(metaclass=ABCMeta):

# 추상 메소드
    @abstractmethod # 추상 메소드에는 @abstrac
    def charecter(self):
        pass        # 추상 메소드는 기능 내 실

# 상속받는 클래스
class Student(People):
    def charecter(self, pow, think):
        self.pow = pow
        self.think = think

        print('체력: {0}'.format(self.pow))
        print('생각: {0}'.format(self.think))

# 상속받는 클래스
class Driver(People):
    def charecter(self, pow, think):
        self.pow = pow
        self.think = think

        print('체력: {0}'.format(self.pow))
        print('생각: {0}'.format(self.think))

# Student object 생성
peo1 = Student()
print('Student : ')

# Student object 실행
peo1.charecter(30, 10)

print()

# Driver object 생성
peo2 = Driver()
print('Driver : ')

# Driver object 실행
peo2.charecter(10, 10)
```



- 다형성은 구현되는 **하위클래스에 따라 클래스를 다르게 처리**하는 기능이다.
    - 상속과 유사하다고 느껴질 수 있지만, 상속은 상위클래스의 기능(함수, 변수)을 재사용한다.
    - 위의 그림과 같이 다형성은 상위클래스의 기능을 변경하여 사용하는 것이다.(**그대로 재사용하지 않는다.**)

```
    print('run')

  def play(self):
    print('play')

class Student(Person):
  def run(self):
    print('fast run')

  def play(self):
    print('play')

class teacher(Person):
  def teach(self):
    print('teach')

  def play(self):
    print('teach play')


# 리스트를 생성한다.
number = list()
# 생성한 리스트에 다형성 개념을 위해 다른 클래스(
number.append(Student())  # 리스트 끝에 서브 클
number.append(teacher())  # 다시 리스트 끝에 서

print("=========")
for Student in number:
    Student.run()     # 상위클래스인 Person의


print("=========")
for teacher in number:
    teacher.play()    # 상위클래스인 Person의
```

🔍 자세히보기 : 헷갈리지 않도록 코드를 작성하자.
연산자를 효율적으로 활용하자.

- 연산자 1개(is not) : A는 **B가 아니다.**
- 연산자 2개(not, is) : A가 **아닌 것은 None이다.**

```
foo = ''

# 가독성 좋음
if foo is not None: # 한 번만 해석하면 된다. (A
  print('가독성 좋음')

# 가독성 좋지 않음
if not foo is None: # 두 번 해석해야된다. (A가
  print('가독성 좋지 않음')
```

☐ Completed!

< Prev Next >