06

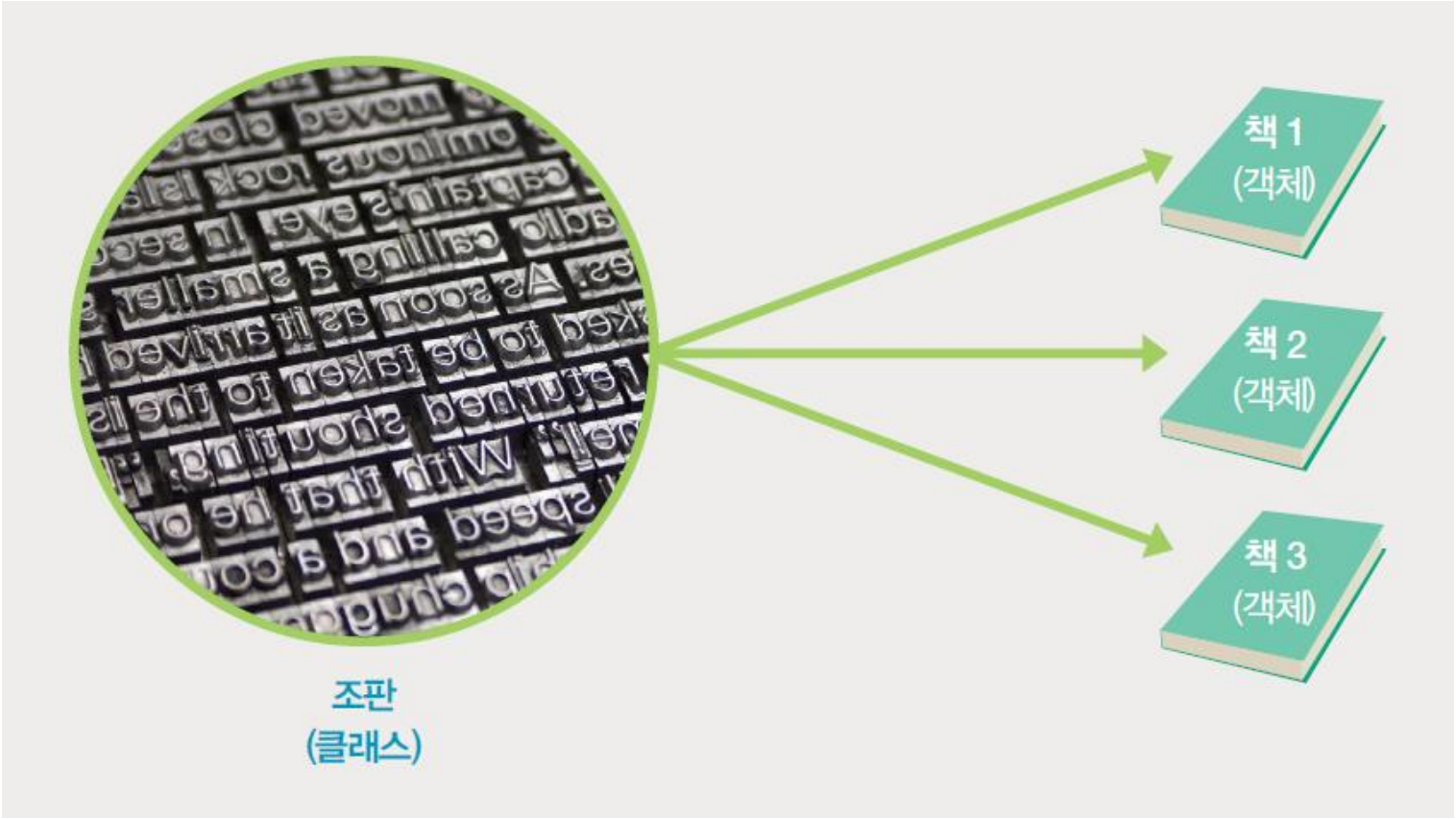
객체 지향

01

클래스

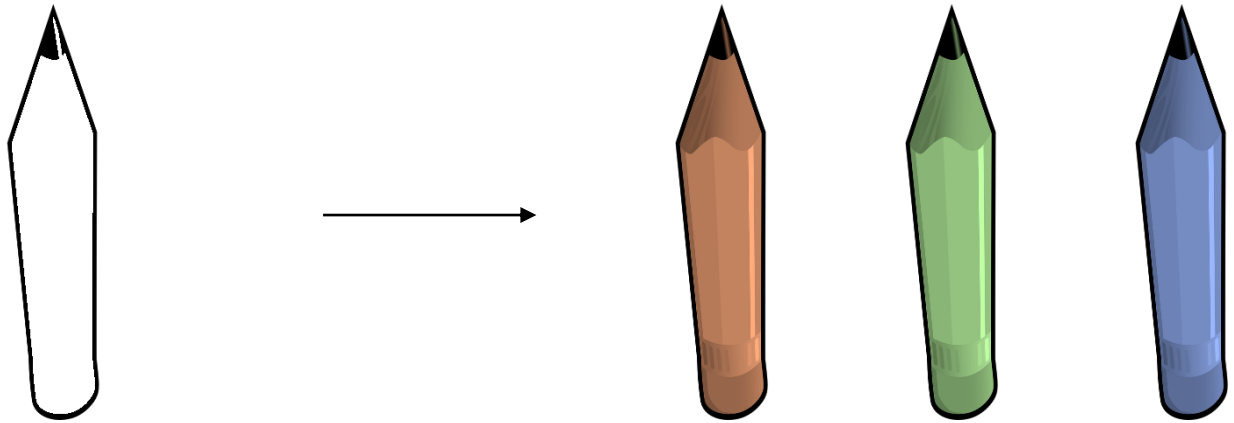
1. 클래스

■ 클래스 이해하기



1. 클래스

클래스 이해하기



클래스(Class)

객체(object)들

1. 클래스

클래스 이해하기

```
클래스 만들기
    class 클래스명:
        클래스의 특징이나 성격을 정의
        함수 등을 정의                    # 클래스에 정의된 함수를 메소드(method)라고 함

인스턴스 만들기
    인스턴스명 = 클래스명()

클래스 메소드 사용하기
    인스턴스.메소드()
```

1. 클래스

클래스 이름, 메소드, 속성 정의하기

- 클래스 이름: 책 읽는 사람 → BookReader
- 속성: 이름 → name
- 메소드: 책 읽기 → read_book

2. 클래스 만들기

■ 속성 및 메소드 선언하기

```
class BookReader:                                    ← 클래스 BookReader 선언
    name = str()                                     ← 문자열형 변수 name 선언
    def read_book(self):                             ← 함수 read_book 선언
        print(name + 'is reading Book!!')            ← 출력
```

2. 클래스 만들기

■ 클래스 호출하기

```
01 class BookReader:
02     name = str()
03     def read_book(self):
04         print(name + ' is reading Book!!')
05
06 reader = BookReader()                    ← 인스턴스 생성
07 print(type(reader))                      ← 변수 reader타입확인
08 reader.name = 'Chris'                    ← 속성값 세팅
09 reader.read_book()                       ← 메소드호출(에러발생)
```

```
<class '__main__.BookReader'>
Traceback (most recent call last):
  File "test.py", line 9, in <module>
    reader.read_book()
  File "test.py", line 4, in read_book
    print(name + 'is reading Book!!')
NameError: name 'name' is not defined
```

2. 클래스 만들기

■ 에러 메시지 확인하기

```
class BookReader:                                          ← 클래스 BookReader 선언
    name = str()                                           ← 문자열형 변수 name 선언
    def read_book(self):                                   ← 함수 read_book 선언(인수 self추가)
        print(self.name + ' is reading Book!!')

reader = BookReader()                                      ← 인스턴스 재생성
reader.name = 'Chris'                                      ← 속성값 세팅
reader.read_book()                                         ← 메소드호출
```

```
Chris is reading Book!!
```

2. 클래스 만들기

■ 특수 내장 메소드 __init()__

```
class BookReader:                                    ← 클래스 BookReader 선언
    def __init__(self, name):                        ← 초기화 함수 재정의
        self.name = name

    def read_book(self):                             ← 함수선언
        print(self.name + ' is reading Book!!')
```

```
reader = BookReader()
```

```
Traceback (most recent call last):
  File "test.py", line 1, in <module>
    reader = BookReader()
TypeError: __init__() missing 1 required positional argument: 'name'
```

```
reader = BookReader('Chris')
reader.read_book()
```

```
Chris is reading Book!!
```

2. 클래스 만들기

■ 클래스변수와 인스턴스 변수 이해하기

```
class BookReader:                                    ← 클래스 변수 country 선언
    country = 'South Korea'

                                                     ← 초기화 함수 재정의
    def __init__(self, name):                        ← 인스턴스변수 name 선언
        self.name = name

    def read_book(self):
        print(self.name + ' is reading Book!!')
```

2. 클래스 만들기


```
class BookReader:
    country = 'South Korea'
    def __init__(self, name):
        self.name = name
    def read_book(self):
        print(self.name + ' is reading Book!!')

reader1 = BookReader('Tom')
print(reader1.country)
reader1.read_book()

reader2 = BookReader('Anna')
print(reader2.country)
reader2.read_book()
```

```
South Korea
Tom is reading Book!!
South Korea
Anna is reading Book!!
```

2. 클래스 만들기


```
class Car:
    def drive(self):
        self.speed = 10

myCar = Car()
myCar.color = "blue"
myCar.model = "E-Class"

myCar.drive()              # 객체 안의 drive() 메소드가 호출된다.
print(myCar.speed)         # 10이 출력된다.
```

```
10
```

2. 클래스 만들기


```
class Car:
    def drive(self):
        self.speed = 10

myCar = Car()
myCar.color = "blue"
myCar.model = "E-Class"

myCar.drive()              # 객체 안의 drive() 메소드가 호출된다.
print(myCar.speed)         # 10이 출력된다.
```

```
10
```

2. 클래스 만들기


```
class Car:
    def drive(self):
        self.speed = 60

myCar = Car()
myCar.speed = 0
myCar.model = "E-Class"
myCar.color = "blue"
myCar.year = "2017"

print("자동차 객체를 생성하였습니다.")
print("자동차의 속도는", myCar.speed)
print("자동차의 색상은", myCar.color)
print("자동차의 모델은", myCar.model)
print("자동차를 주행합니다.")
myCar.drive()
print("자동차의 속도는", myCar.speed)
```

```
자동차 객체를 생성하였습니다.
자동차의 속도는 0
자동차의 색상은 blue
자동차의 모델은 E-Class
자동차를 주행합니다.
자동차의 속도는 60
```

2. 클래스 만들기


```
class Car:
    def __init__(self, speed, color, model):
        self.speed = speed
        self.color = color
        self.model = model
    def drive(self):
        self.speed = 60

myCar = Car(0, "blue", "E-class")
print("자동차 객체를 생성하였습니다.")
print("자동차의 속도는", myCar.speed)
print("자동차의 색상은", myCar.color)
print("자동차의 모델은", myCar.model)
print("자동차를 주행합니다.")
myCar.drive()
print("자동차의 속도는", myCar.speed)
```

```
자동차 객체를 생성하였습니다.
자동차의 속도는 0
자동차의 색상은 blue
자동차의 모델은 E-Class
자동차를 주행합니다.
자동차의 속도는 60
```

2. 클래스 만들기


```
class Car:
    def drive(self):
        self.speed = 60

myCar = Car()
myCar.speed = 0
myCar.model = "E-Class"
myCar.color = "blue"
myCar.year = "2017"

print("자동차 객체를 생성하였습니다.")
print("자동차의 속도는", myCar.speed)
print("자동차의 색상은", myCar.color)
print("자동차의 모델은", myCar.model)
print("자동차를 주행합니다.")
myCar.drive()
print("자동차의 속도는", myCar.speed)
```

```
class Car:
    def __init__(self, speed, color, model):
        self.speed = speed
        self.color = color
        self.model = model
    def drive(self):
        self.speed = 60

myCar = Car(0, "blue", "E-class")

print("자동차 객체를 생성하였습니다.")
print("자동차의 속도는", myCar.speed)
print("자동차의 색상은", myCar.color)
print("자동차의 모델은", myCar.model)
print("자동차를 주행합니다.")
myCar.drive()
print("자동차의 속도는", myCar.speed)
```

객체를 생성하고 정보출력


```
class block_factory:
    def __init__(self, company): #객체 생성시 company만 초기화
        self.company = company

newblock = block_factory("Gole") #객체 생성(초기화 company는 인자 전달)
newblock.color = "blue"          #속성 지정(초기화 하지 않고 추후에 지정 가능)
newblock.shape = "long"          #속성 지정2

print("블록 정보 출력")
print("제조 회사 :", newblock.company)
print("블록 컬러 :", newblock.color)
print("블록 모양 :", newblock.shape)
```

```
블록 정보 출력
제조 회사 : Gole
블록 컬러 : blue
블록 모양 : long
```

객체를 생성하고 정보출력

초기화 메소드 생성해두고 사용하기

```
class block_factory:
    def __init__(self, company, color, shape):
        self.company = company
        self.color = color
        self.shape = shape

newblock = block_factory("Gole", "blue", "long")

print("블록 정보 출력")
print("제조 회사 :", newblock.company)
print("블록 컬러 :", newblock.color)
print("블록 모양 :", newblock.shape)
```

```
블록 정보 출력
제조 회사 : Gole
블록 컬러 : blue
블록 모양 : long
```

여러 객체를 생성하기


```
class block_factory:
    def __init__(self, company, color, shape):
        self.company = company                          #초기화
        self.color = color
        self.shape = shape

newblock = block_factory("Gole", "blue", "long")        #객체 생성
newblock2 = block_factory("Gole", "black", "short")
newblock3 = block_factory("Oxfold", "red", "big")

print("블록 정보 출력")
print("제조 회사 :", newblock.company, "블록 컬러 :",\
      newblock.color, "블록 모양 :", newblock.shape)
print("제조 회사 : %s 블록 컬러 : %s 블록 모양 : %s"\
      %(newblock2.company, newblock2.color, newblock2.shape))
print("제조 회사 : {0} 블록 컬러 : {1} 블록 모양 : {2}"\
      .format(newblock3.company, newblock3.color, newblock3.shape))
```

```
제조 회사 : Gole 블록 컬러 : blue 블록 모양 : long
제조 회사 : Gole 블록 컬러 : black 블록 모양 : short
제조 회사 : Oxfold 블록 컬러 : red 블록 모양 : big
```

여러 객체를 생성하기


```
블록 정보 출력
제조 회사 : Gole 블록 컬러 : blue 블록 모양 : long
제조 회사 : Gole 블록 컬러 : black 블록 모양 : short
제조 회사 : Oxfold 블록 컬러 : red 블록 모양 : big
```

```
class block _factory():

    def __init__(self, company, color, shape):

newblock = block_factory("Gole", "blue", "long")
```

클래스에 메소드(.method)추가하기

- 실제로 기능을 하는 함수를 클래스에 추가하기

```
class block_factory:
    def __init__(self, company, color, shape):
        self.company = company                          #초기화
        self.color = color
        self.shape = shape
    def make_pink(self):
        self.color = "pink"

newblock = block_factory("Gole", "blue", "long")        #객체 생성
print("블록 정보 출력")
print("제조 회사 :", newblock.company, "블록 컬러 :",\
      newblock.color, "블록 모양 :", newblock.shape)

newblock.make_pink()                          #.make_pink() 메소드 호출

print("블록 정보 재 출력")
print("제조 회사 :", newblock.company, "블록 컬러 :",\
      newblock.color, "블록 모양 :", newblock.shape)
```

클래스에 메소드(.method)추가하기

- **실제로 기능을 하는 함수를 클래스에 추가하기**

```
블록 정보 출력
제조 회사 : Gole 블록 컬러 : blue 블록 모양 : long
블록 정보 재 출력
제조 회사 : Gole 블록 컬러 : pink 블록 모양 : long
```

클래스에서 만든 make_pink() 함수를 객체에서 실행 → 객체의 속성이 변경

PRACTICE

블록 모양을 같게 만드는 메소드 make_short()을 만들어 호출하라

```
블록 정보 출력
제조 회사 : Gole 블록 컬러 : blue 블록 모양 : long
블록 정보 재 출력
제조 회사 : Gole 블록 컬러 : pink 블록 모양 : short
```

ANSWER

```
class block_factory:
    def __init__(self, company, color, shape):
        self.company = company                          #초기화
    self.color = color
        self.shape = shape
    def make_pink(self):
        self.color = "pink"
    def make_short(self):
        self.shape = "short"

newblock = block_factory("Gole", "blue", "long")        #객체 생성
print("블록 정보 출력")
print("제조 회사 :", newblock.company, "블록 컬러 :",\
      newblock.color, "블록 모양 :", newblock.shape)
newblock.make_pink()                                    #.make_pink() 메소드 호출
newblock.make_short()                                   #.make_short() 메소드 호출
print("블록 정보 재 출력")
print("제조 회사 :", newblock.company, "블록 컬러 :",\
      newblock.color, "블록 모양 :", newblock.shape)
```

클래스에 메소드(.method)추가하기

- 실제로 기능을 하는 함수를 클래스에 추가하기

클래스 이름 : Calculator
생성자(initializer) : 없음
메소드(method) : plus – x+y
minus – x-y
multiply – x*y
divide – x/y
객체 이름 : calc

```
print(calc.plus(10,5))
15
print(calc.minus(10,5))
5
print(calc.multiply(10,5))
50
print(calc.divide(10,5))
2.0
```

클래스에 메소드(.method)추가하기

- 실제로 기능을 하는 함수를 클래스에 추가하기

```
class Calculator:
    def plus(self, x, y):
        return x+y
    def minus(self, x, y):
        return x-y
    def multiply(self, x, y):
        return x*y
    def divide(self, x, y):
        return x/y

calc = Calculator()
print(calc.plus(10,5))
print(calc.minus(10,5))
print(calc.multiply(10,5))
print(calc.divide(10,5))
```

클래스의 다양한 속성들 정의 하는법

객체가 시작될 때 또는 종료될 때

이름	설명	비고
__init__(cls, ...)	인스턴스가 생성되면 처음 하는 동작 지정	
__new__(self, ..)	인스턴스가 생성되면 처음 실행하는 동작 지정	return(obj) 필요
__del__(self)	객체가 소멸할 때 동작 지정	

객체를 표현

이름	설명	비고
__str__(self)	객체의 데이터를 문자열로 만들어서 반환, print(a)로 출력하면 해당 부분 호출(return)	__repr__(self)

__str__() 메소드


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

    def __str__(self):
        msg = "속도:"+ str(self.speed)+ " 색상:"+ self.color+ " 모델:"+ self.model
        return msg

myCar = Car(0, "blue", "E-class")
print(myCar)
```

```
속도:0 색상:blue 모델:E-class
```

클래스의 다양한 속성들 정의 하는법

객체의 속성과 관련된 부분

이름	설명
`__getattribute__(self, name, ..)`	객체의 속성(데이터)를 참조할 때 무조건 호출
`__getattr__(self, name, ..)`	참조 시, 속성이 존재하지 않을 때 호출
`__setattr__(self, name, ..)`	객체의 속성을 변경할 때 호출

다른 객체를 변경할 때

이름	설명
`__get__(self, instance, owner)`	특정 객체의 값을 참조할 때 호출
`__set__(self, instance, value)`	특정 객체의 값을 변경할 때 호출

3 생성자

- **생성자의 개념 : 인스턴스를 생성하면서 필드값을 초기화시키는 함수**
- **생성자의 기본**
  - 생성자의 기본 형태 : _ _ init _ _( )라는 이름

Tip • _ _ init _ _( )는 앞뒤에 언더바(_)가 2개씩, init 는 Initialize 의 약자로 초기화 의미
언더바가 2 개 붙은 것은 파이썬에서 예약해 놓은 것, 프로그램을 작성시 이 이름을 사용해서 새로운 함수나 변수명을 만들지 말 것

```
class 클래스명 :
    def __init__(self) :
        # 이 부분에 초기화할 코드 입력
```

```
class Car :
    color = ""
    speed = 0

    def __init__(self) :
        self.color = "빨강"
        self.speed = 0
```

생성자


```
class Car:
    def __init__(self, speed, color, model):
        self.speed = speed
        self.color = color
        self.model = model
    def drive(self):
        self.speed = 60

dadCar = Car(0, "silver", "A6")
momCar = Car(0, "white", "520d")
myCar = Car(0, "blue", "E-class")
```

생성자


```
class Car:
    def __init__(self, speed, color, model):
        self.speed = speed
        self.color = color
        self.model = model
    def drive(self):
        self.speed = 60

myCar = Car(0, "blue", "E-class")
yourCar = Car(0, "white", "S-class")

print(myCar.speed,yourCar.speed)
myCar.drive()
yourCar.drive()
print(myCar.speed,yourCar.speed)
```

4 인스턴스 변수와 클래스 변수

인스턴스 변수

- 예 : Car 클래스 2 개의 필드

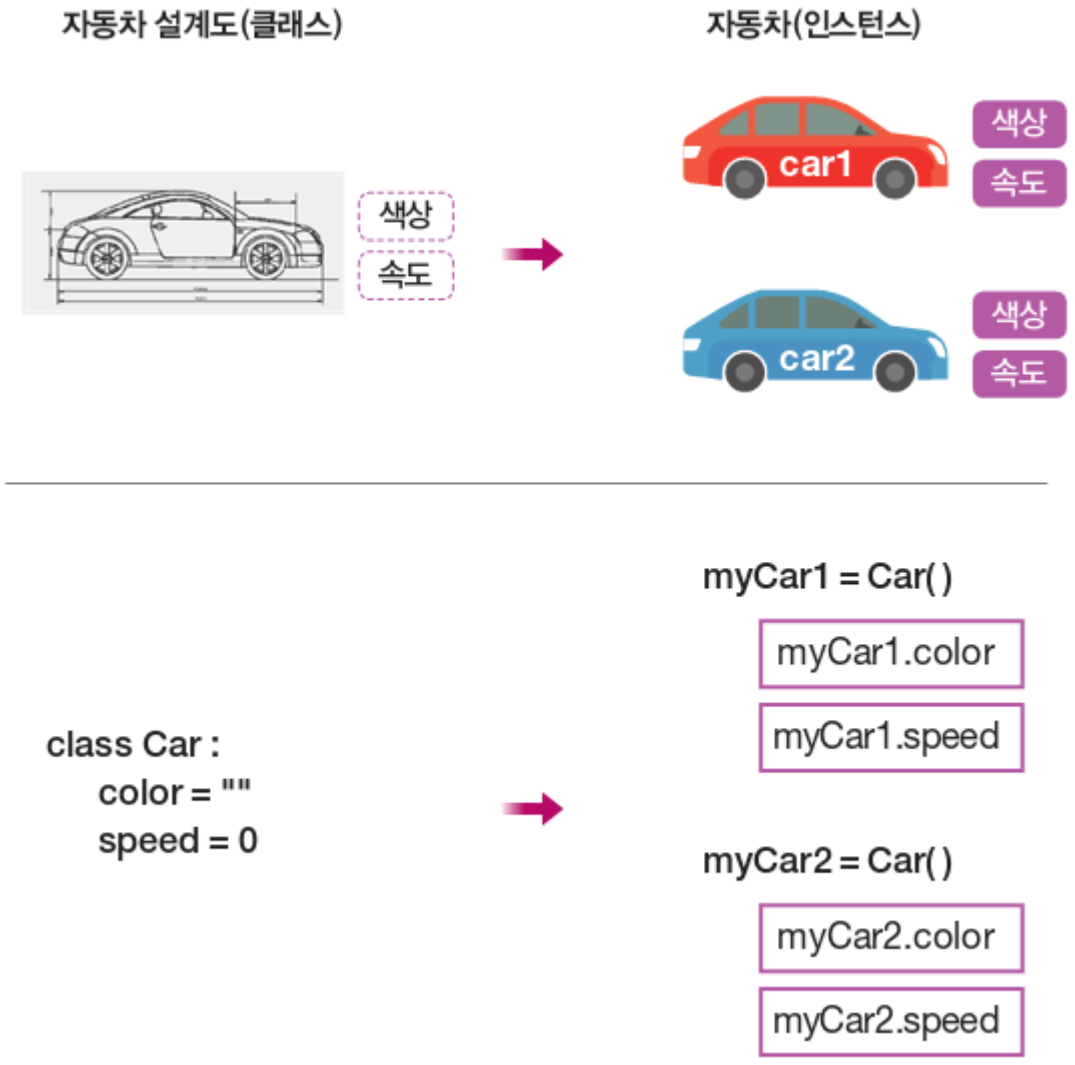
```
class Car :
    color = ""    # 필드 : 인스턴스 변수
    speed = 0     # 필드 : 인스턴스 변수
```

- 클래스를 이용해 메인 코드에서 인스턴스 만들기

```
myCar1 = Car()
myCar2 = Car()
```

4 인스턴스 변수와 클래스 변수

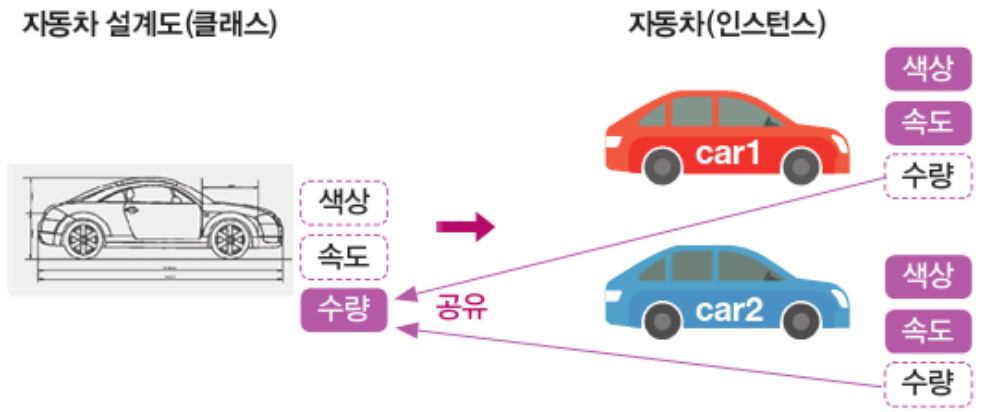
- 인스턴스 변수의 개념

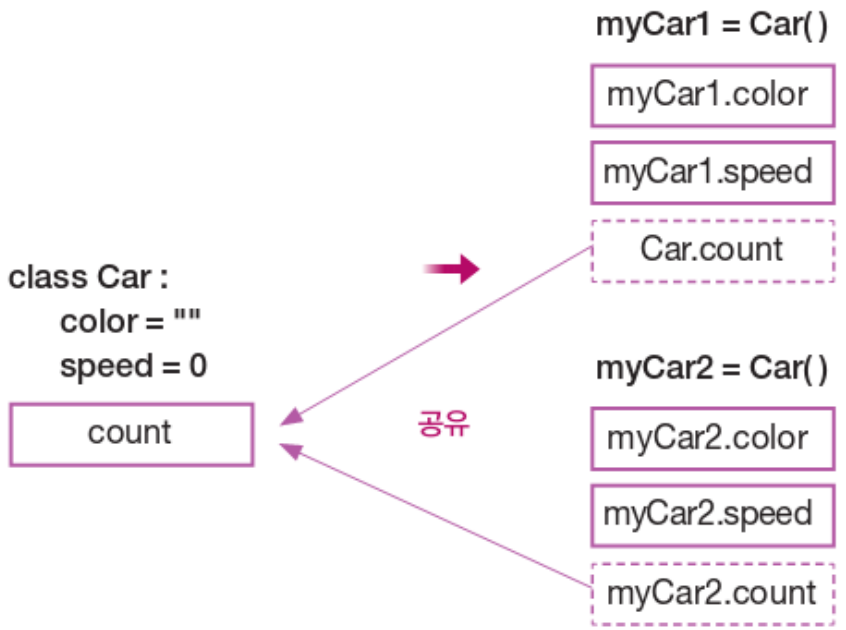


4 인스턴스 변수와 클래스 변수

■ 클래스 변수

- 클래스 안에 공간이 할당된 변수,
- 여러 인스턴스가 클래스 변수 공간 함께 사용





4 인스턴스 변수와 클래스 변수


```
class Car :
    color = ""
    speed = 0
    count = 0                  # 클래스 변수 count 를 선언하고 0으로 초기화
    def __init__(self):
        self.speed=0
        Car.count += 1         # 생성자 안에서 클래스 변수에 접근하려고 클래스명.count 를 1 증가

myCar1 = Car()
myCar1.speed = 30
print("자동차1의 현재속도: %dkm, 생산된 자동차는 총 %d대 입니다"\
      %(myCar1.speed,Car.count))
myCar2 = Car()
myCar2.speed = 60
print("자동차2의 현재속도: %dkm, 생산된 자동차는 총 %d대 입니다"\
      %(myCar2.speed,Car.count))
```

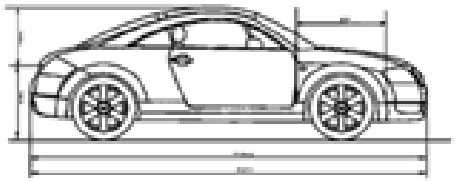
```
자동차1의 현재속도: 30km, 생산된 자동차는 총 1대 입니다
자동차2의 현재속도: 60km, 생산된 자동차는 총 2대 입니다
```

5 클래스의 상속

상속의 개념

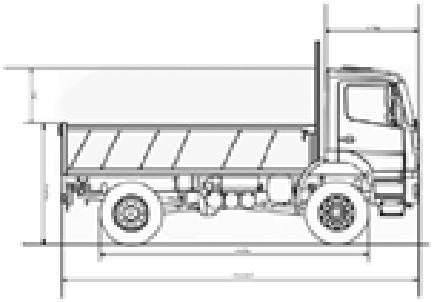
- 클래스의 상속(Inheritance) : 기존 클래스에 있는 필드와 메소드를 그대로 물려받는 새로운 클래스를 만드는 것

승용차 클래스



class 승용차 :
필드 – 색상, 속도, 좌석 수
메서드 – 속도 올리기( )
속도 내리기( )
좌석 수 알아보기( )

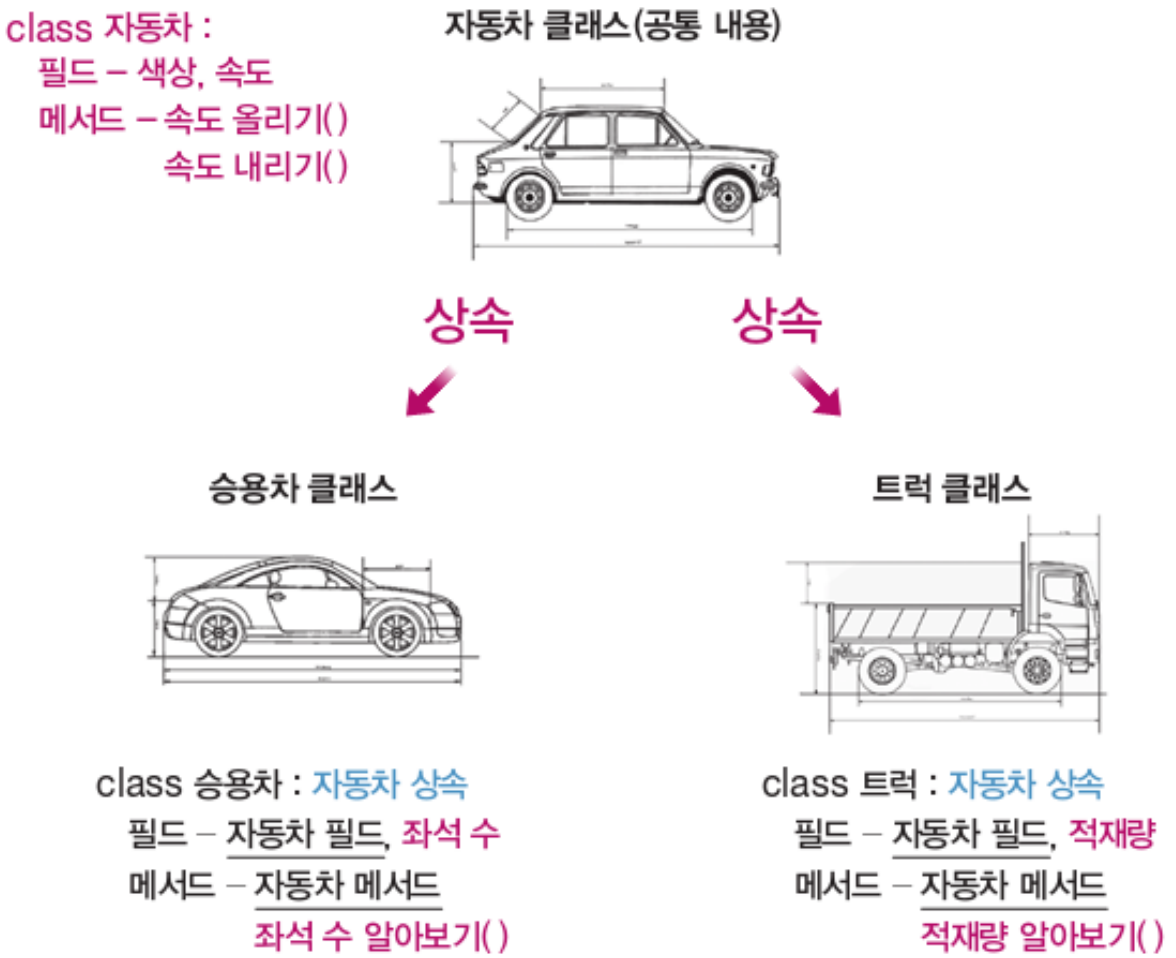
트럭 클래스



class 트럭 :
필드 – 색상, 속도, 적재량
메서드 – 속도 올리기( )
속도 내리기( )
적재량 알아보기( )

5 클래스의 상속

- 상속의 개념
  - 공통된 내용을 자동차 클래스에 두고 상속을 받음으로써 일관되고 효율적인 프로그래밍 가능
  - 상위 클래스인 자동차 클래스를 슈퍼 클래스 또는 부모 클래스, 하위의 승용차와 트럭 클래스는 서브 클래스 또는 자식 클래스



5 클래스의 상속

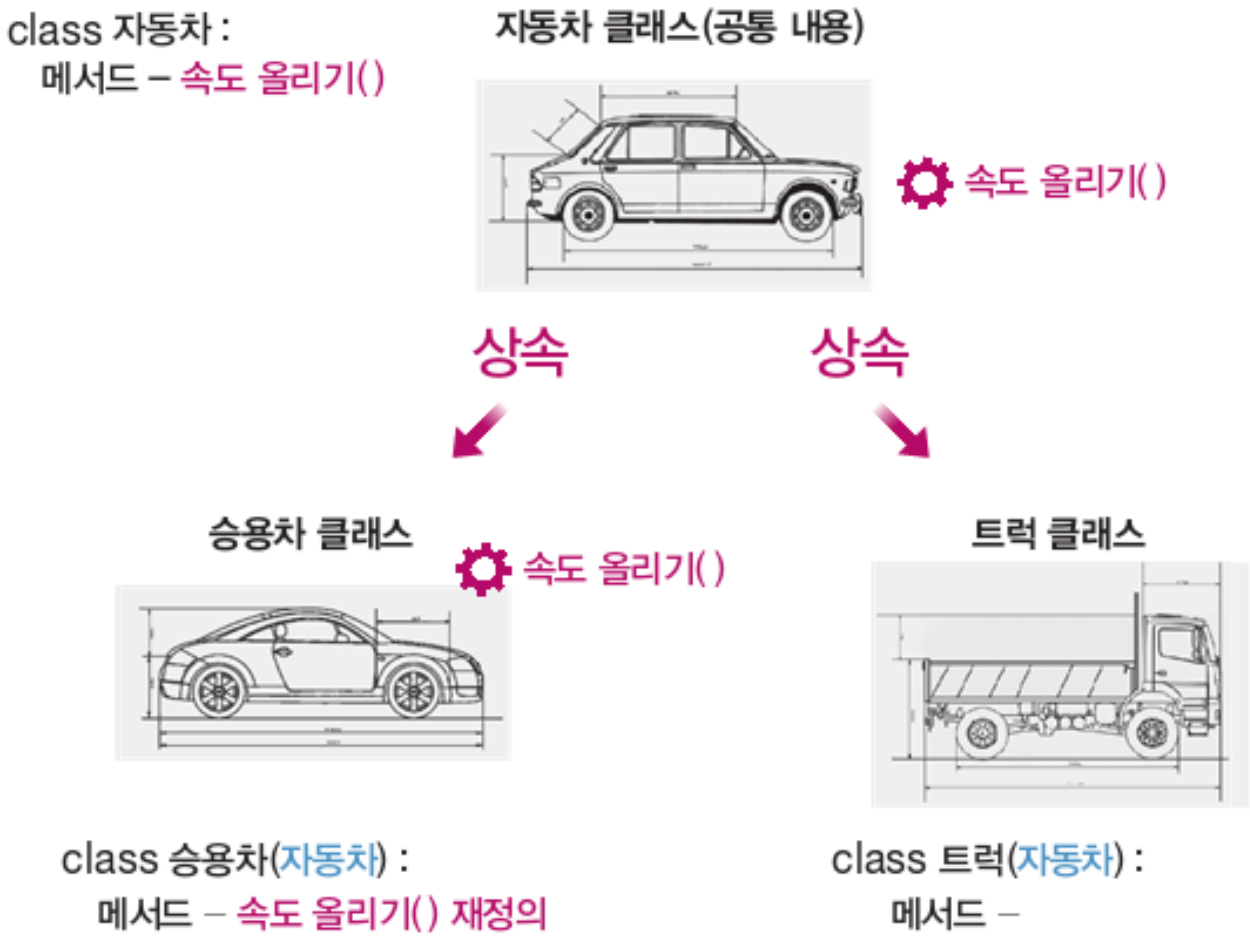
- 상속을 구현하는 문법

```
class 서브_클래스(슈퍼_클래스) :
    # 이 부분에 서브 클래스의 내용 코딩
```

5 클래스의 상속

메소드 오버라이딩

- 상위 클래스의 메소드를 서브 클래스에서 재정의



5 클래스의 상속

메소드 오버라이딩 구현 코드

```
1  ## 클래스 선언 부분 ##
2  class Car :
3      speed = 0
4      def upSpeed(self, value) :
5          self.speed += value
6
7          print("현재 속도(슈퍼 클래스) : %d" % self.speed)
8
9  class Sedan( Car ) :
10     def upSpeed(self, value) :
11         self.speed += value
12
13         if self.speed > 150 :
14             self.speed = 150
15
```

10 ~ 16행 : 서브 클래스(Sedan)의 upSpeed ( ) 메소드 다시 만듬

5 클래스의 상속


```
16          print("현재 속도(서브 클래스) : %d" % self.speed)
17
18  class Truck(Car) :
19      pass
20
21  ## 변수 선언 부분 ##
22  sedan1, truck1 = None, None
23
24  ## 메인 코드 부분 ##
25  truck1 = Truck()
26  sedan1 = Sedan()
27
28  print("트럭 --> ", end = "")
29  truck1.upSpeed(200)
30
31  print("승용차 --> ", end = "")
32  sedan1.upSpeed(200)
```

32행 : Sedan 인스턴스의 upSpeed ( ) 메소드 호출하면
10행에서 재정의된 upSpeed ( ) 메소드를 호출
18행 : 서브 클래스(Truck)에는 아무런 내용 없어
슈퍼 클래스(Car)의 메소드를 그대로 상속
29행 : Truck 인스턴스의 upSpeed ( ) 호출하면 4 ~ 7행
슈퍼 클래스(Car)의 upSpeed ( ) 메소드 호출

출력 결과

```
트럭 --> 현재 속도(슈퍼 클래스) : 200
승용차 --> 현재 속도(서브 클래스) : 150
```

출력 결과

트럭 --> 현재 속도(슈퍼 클래스) : 200
승용차 --> 현재 속도(서브 클래스) : 150

PRACTICE

위의 코드에 Sonata클래스를 추가해보자
단, Sonata클래스는 Car -> Sedan -> Sonata 순서로 상속을 받도록 하자
Sonata 클래스에는 특별히 추가하는 메소드가 없다.

```
트럭 --> 현재속도(수퍼클래스) : 200
승용차 --> 현재속도(서브클래스) : 150
소나타 --> 현재속도(서브클래스) : 150
```

클래스의 메소드 상속받기

클래스와 메소드 생성

```
class Korea:
    def say(self):
        print("I'm from Korea")
```

클래스를 추가

```
class Korea:
    def say(self):
        print("I'm from Korea")

Class South_Korea(Korea):    #괄호(Korea)는 Korea 클래스를 상속받았음을 의미
    pass
```

확인해보기

```
a = Korea()
b = South_Korea()
a.say()
I'm from Korea
b.say()
I'm from Korea
```

6 객체지향 프로그래밍의 심화 내용

클래스의 특별한 메소드

- __del__() 메소드
  - 소멸자(Destructor), 생성자와 반대로 인스턴스 삭제할 때 자동 호출.
- __repr__() 메소드
  - 인스턴스를 print() 문으로 출력할 때 실행
- __add__() 메소드
  - 인스턴스 사이에 덧셈 작업이 일어날 때 실행되는 메소드, 인스턴스 사이의 덧셈 작업 가능
- 비교 메소드 : __lt__(), __le__(), __gt__(), __ge__(), __eq__(), __ne__()
  - 인스턴스 사이의 비교 연산자(<, <=, >, >=, ==, != 등) 사용할 때 호출

6 객체지향 프로그래밍의 심화 내용


```
## 클래스 선언 부분 ##
class Line :
    length = 0
    def __init__(self, length) :
        self.length = length
        print(self.length, '길이의 선이 생성되었습니다.')

    def __del__(self) :
        print(self.length, '길이의 선이 삭제되었습니다.')

    def __repr__(self) :
        return '선의 길이 : ' + str(self.length)

    def __add__(self, other) :
        return self.length + other.length

    def __lt__(self, other) :
        return self.length < other.length

    def __eq__(self, other) :
        return self.length == other.length

## 메인 코드 부분 ##
myLine1 = Line(100)
myLine2 = Line(200)

print(myLine1)

print('두 선의 길이 합 : ', myLine1 + myLine2)

if myLine1 < myLine2 :
    print('선분 2가 더 기네요.')
elif myLine1 == myLine2 :
    print('두 선분이 같네요.')
else :
    print('모르겠네요.')

del(myLine1)
```

출력 결과

```
100 길이의 선이 생성되었습니다.
200 길이의 선이 생성되었습니다.
선의 길이 : 100
두 선의 길이 합 : 300
선분 2가 더 기네요.
100 길이의 선이 삭제되었습니다.
```

클래스의 다양한 속성 정의 하는법

연산자 재정의

이름	설명
`__neg__(self)`	-a(객체)를 정의한다.
`__gt__(self, other)`	x>y를 정의한다.
`__add__(self, other)`	x+y를 정의한다.
`__int__(self)`	int(a)를 정의한다.

PRACTICE

• 조건에 맞는 클래스 생성

어느 회사에서 직급에 따라 연봉을 다르게 준다. 직급은 연차가 쌓이면 자동으로 승진하게 되는 구조이다. 고용된 사람들은 이름과 연차 정보를 가지고 있다. 최근 회사에서 경력직을 대상으로 채용을 하고 있고 경력을 100% 인정한다.

클래스는 이름과 연차를 입력한 객체를 만들면 이름과 연차를 먼저 출력한다.
메소드 1 : 연차를 이용하여 이름과 연봉을 출력하라.
메소드 2 : 학위 소지 시 연봉에 1200만원을 더해 이름과 연봉을 출력한다.

연차 5년 이하 : 사원 - 3000만원 + (연차 * 100만원)
연차 10년 이하 : 대리 - 3500만원 + (연차 * 110만원)
연차 10년 이상 : 과장 - 4000만원 + (연차 * 130만원)

PRACTICE

```
a = Employ("파이썬", 7)
파이썬님의 연차는 7년차 입니다.

a.salary()
파이썬님의 연봉은 4270만원 입니다.

a.degree()
파이썬님의 연봉은 학위 소지로 인하여 5270만원 입니다.

print(a.money)                    #__getattr__
잘못된 값입니다.

print(a.name, a.salary_d)
('파이썬', 5270)
```

ANSWER

```
class Employ:
    def __init__(self, name, career):
        self.name = name
        self.career = career
        print(self.name+"님의 연차는", str(self.career)+"년차 입니다.")

    def salary(self):
        if self.career <= 5:
            self.salary = (self.career * 100) + 3000
        elif self.career <= 10:
            self.salary = (self.career * 110) + 3500
        elif self.career > 10:
            self.salary = (self.career * 130) + 4000

        print("%s님의 연봉은 %d만원 입니다."%(self.name, self.salary))

    def degree(self):
        self.salary_d = int(self.salary) + 1000
        print("%s님의 연봉은 학위 소지로 인하여 %d만원 입니다."%(self.name, self.salary_d))

    def __getattr__(self,anything):
        print("잘못된 값입니다.")
```

PRACTICE

- **한 클래스에서 두 객체를 만들어 서로 영향 주기**

make_character 라는 하나의 클래스를 생성하고 한클래스에서 두개의 객체를 생성하고 두객체가 서로 영향을 줄수 있는 메소드(attack, defence)를 생성하여 다음과 같은 결과가 나오도록 작성해보자

```
나 = 캐릭터생성("전사", 100)
적 = 캐릭터생성("흑마법사", 100)
나.공격(적)
상대 흑마법사에게 10 피해를 입혔습니다.
전사 : 100   흑마법사 : 90

나.방어()
전사 은(는) 5 피해를 입었습니다.
전사 : 95
```

ANSWER


```
class make_character:
    def __init__(self, job, hp):
        self.job = job
        self.hp = int(hp)

    def attack(self, enemy):
        enemy.hp = enemy.hp - 10
        print("상대 %s에게 10 피해를 입혔습니다."%enemy.job)
        print("%s : %d   %s : %d"%(self.job, self.hp, enemy.job, enemy.hp))

    def defense(self):
        self.hp = self.hp - 5
        print(self.job, "은(는) 5 피해를 입었습니다.")
        print("%s : %d"%(self.job, self.hp))

myChar = make_character("warrior",100)
enemyChar = make_character("darkSoccer",100)
myChar.attack(enemyChar)
myChar.defense()
enemyChar.attack(myChar)
```

ANSWER

```
import random
import time

class make_character:
    def __init__(self, job, hp):
        self.job = job
        self.hp = int(hp)

    def attack(self, enemy):
        damage = random.randint(1, 10)
        enemy.hp = enemy.hp - damage
        print("상대 %s에게 %d 피해를 입혔습니다."%(enemy.job, damage))
        print("%s : %d   %s : %d"%(self.job, self.hp, enemy.job, enemy.hp))

    def defense(self):
        damage = random.randint(1, 5)
        self.hp = self.hp - damage
        print("%s은(는) %d 피해를 입었습니다."%(self.job, damage))
        print("%s : %d"%(self.job, self.hp))

myChar = make_character("warrior",100)
enemyChar = make_character("darkSoccer",100)
while True:
    if myChar.hp <= 0 or enemyChar.hp <= 0 :
        break
    else:
        myChar.attack(enemyChar)
        myChar.defense()
        enemyChar.attack(myChar)
        enemyChar.defense()
    time.sleep(1)
```