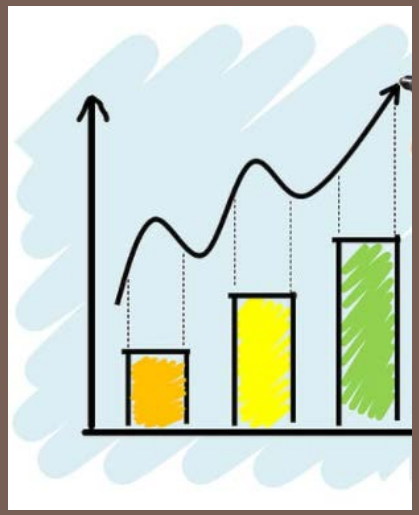
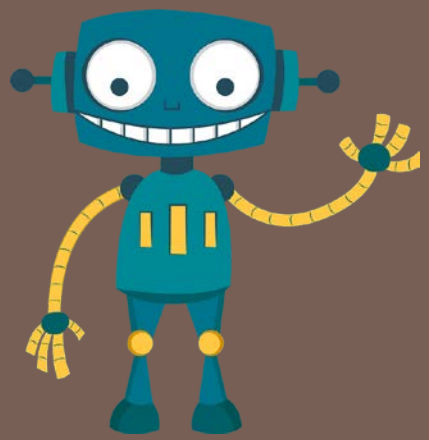
파이썬 익스프레스





12장 상속

학습 목표

- 부모 클래스를 **상속**받아서 자식 클래스를 정의할 수 있다
- 부모 클래스의 메소드를 자식 클래스에서 **재정의**할 수 있다.
- **Object 클래스**를 이해할 수 있다.
- **메소드 오버라이딩**을 이해하고 사용할 수 있다.
- **클래스 간의 관계**를 파악할 수 있다.

이번 장에서 만들 프로그램

1. 상속을 이용하여서 각 클래스에 **중복된 정보를 부모 클래스로** 모아 보자.
   - 구체적인 예로 Car 클래스와 ElectricCar 클래스를 작성해보자.

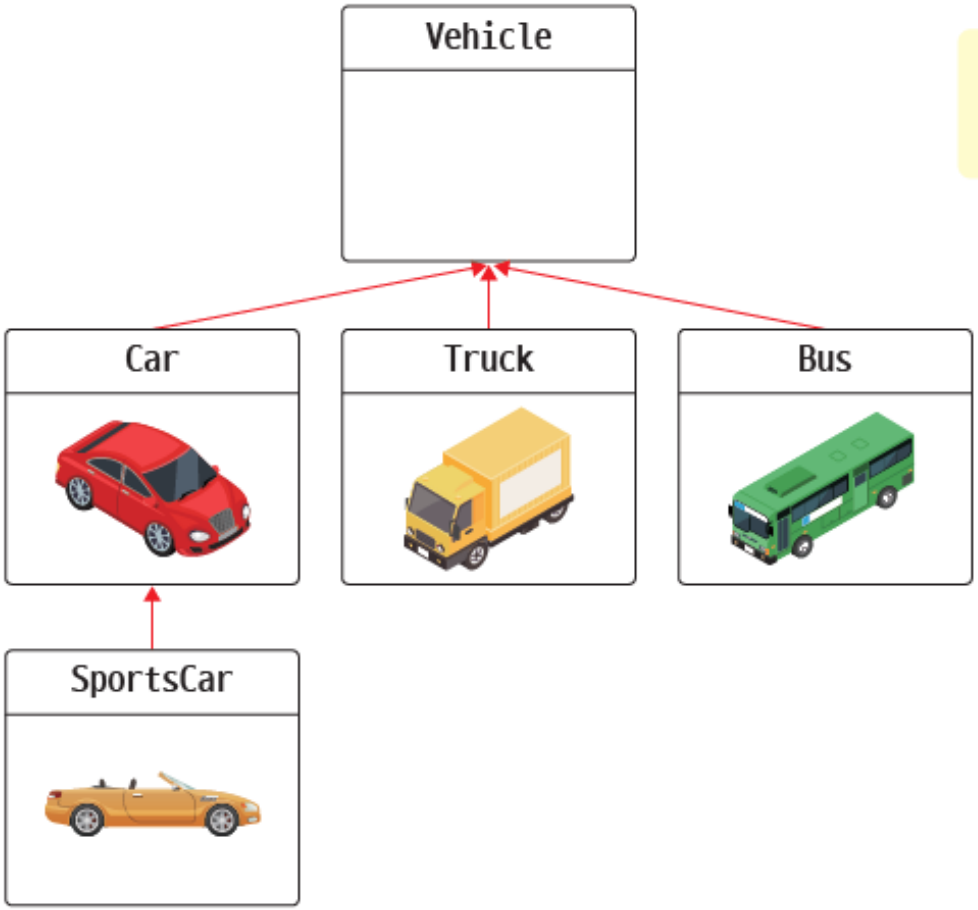
2. 상속을 사용할 때, **자식 클래스와 부모 클래스의 생성자가 호출되는 순서**를 살펴보자.

3. 부모 클래스의 함수를 **오버라이딩(재정의)**하여 자식 클래스의 기능을 강력하게 하는 기법을 살펴보자.

상속

- **상속(inheritance)**은 기존에 존재하는 클래스로부터 **코드와 데이터를 물려받고** 자신이 **필요한 기능을 추가**하는 기법이다.

상속의 예


Vehicle
Car
Truck
Bus
SportsCar


상속에서는 자식 클래스에서
부모 클래스로 화살표를
그립니다.

상속과 is-a 관계

- 객체 지향 프로그래밍에서는 상속이 클래스 간의 **"is-a" 관계**를 생성하는데 사용된다.
  - is-a 관계 : "***은 @@@이다."가 의미적으로 말이 되는 관계
    - *** : 자식 클래스
    - @@@ : 부모 클래스

- 예
  - **푸들**은 **강아지**이다.
  - **자동차**는 **차량**이다.
  - **꽃**은 **식물**이다.
  - **사각형**은 **모양**이다.

상속의 예

부모 클래스	자식 클래스
Animal(동물)	Lion(사자), Dog(개), Cat(고양이)
Bike(자전거)	MountainBike(산악자전거), RoadBike, TandemBike
Vehicle(탈것)	Car(자동차), Bus(버스), Truck(트럭), Boat(보트), Motocycle(오토바이), Bicycle(자전거)
Student(학생)	GraduateStudent(대학원생), UnderGraduate(학부생)
Person(사람)	Student(학생), Employee(직원)
Shape(도형)	Rectangle(사각형), Triangle(삼각형), Circle(원)

상속 구현하기

Syntax: 상속 정의

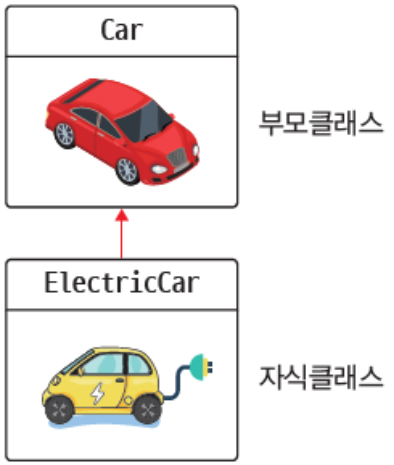
자식 클래스 또는 서브 클래스라고 한다.

Syntax

```
class 자식클래스(부모클래스) :
        def 메소드1(self, ...) :
            ...
        def 메소드2(self, ...) :
            ...
```

부모 클래스 또는 슈퍼 클래스라고 한다.

```
class ElectricCar(Car) :
        def setBatterySize(self, size):
            ...
        def getBatterySize(self):
            ...
```

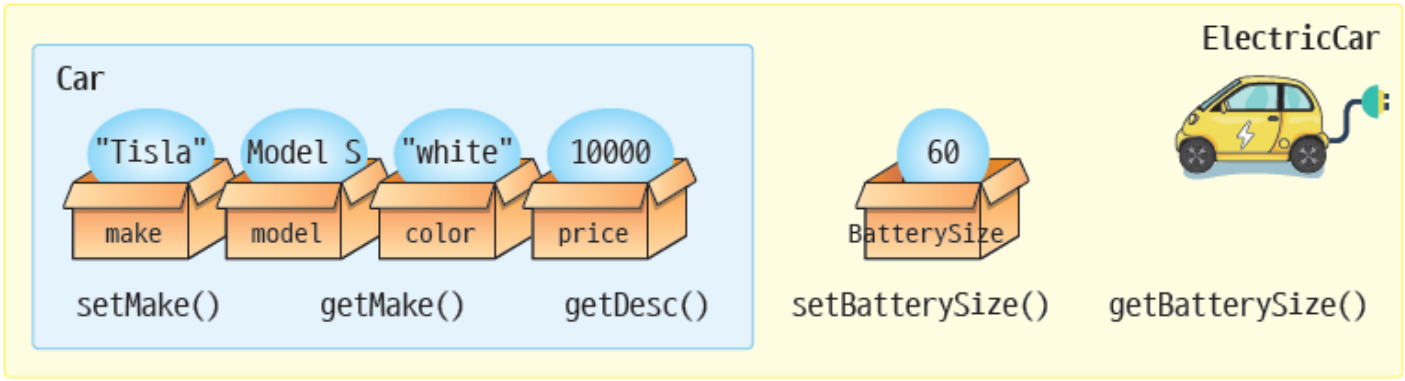


예제

```
# 일반적인 자동차를 나타내는 클래스이다.
class Car:
        def __init__(self, make, model, color,  price):
                self.make = make                # 메이커
                self.model = model              # 모델
                self.color = color              # 자동차의 색상
                self.price = price              # 자동차의 가격

        def setMake(self, make):                # 설정자 메소드
                self.make = make

        def getMake(self):                      # 접근자 메소드
                return self.make

        # 차량에 대한 정보를 문자열로 요약하여서 반환한다.
        def getDesc(self):
                return "차량 =("+str(self.make)+","+\
                        str(self.model)+","+\
                        str(self.color)+","+\
                        str(self.price)+")"
```

예제

```
class ElectricCar(Car) :                                    # ①
        def __init__(self, make, model, color, price, batterySize):
                super().__init__(make, model, color, price)   # ②
                self.batterySize=batterySize                  # ③

        def setBatterySize(self,  batterySize):          # 설정자 메소드
                self.batterySize=batterySize

        def getBatterySize(self):                        # 접근자 메소드
                return self.batterySize
```

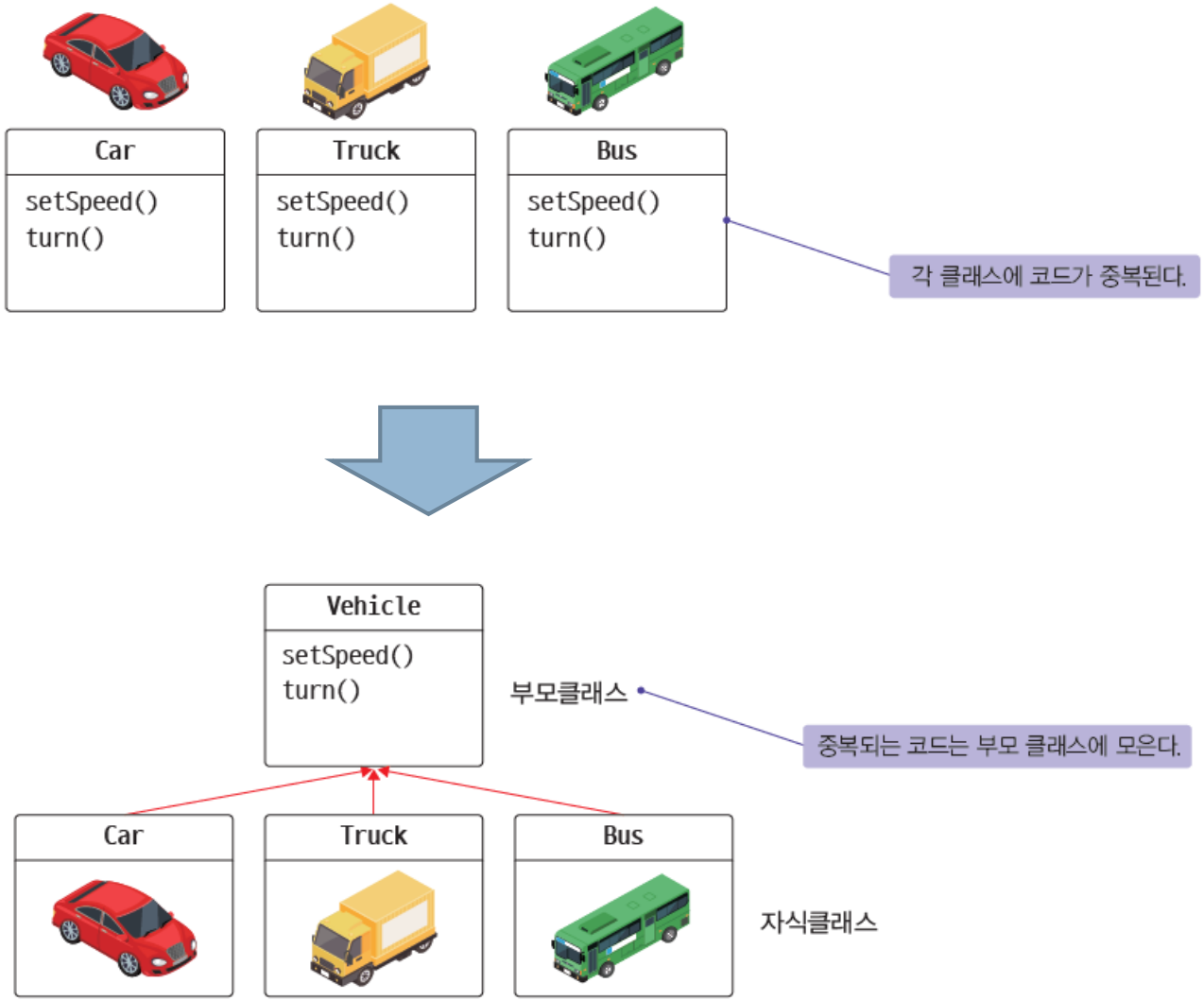


예제

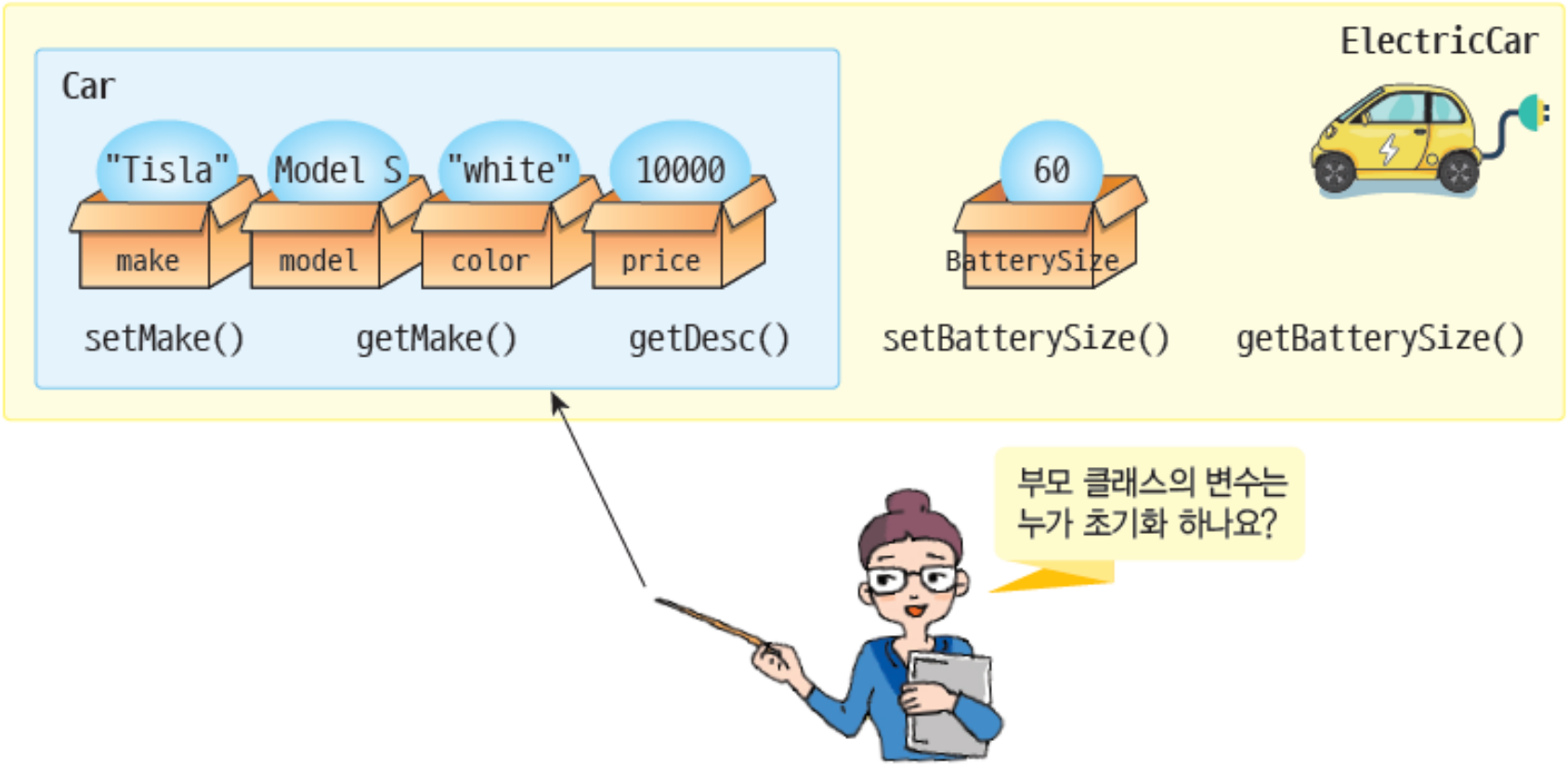
```
def main():
        myCar = ElectricCar("Tisla", "Model S", "white", 10000, 0)
        myCar.setMake("Tesla")                  # 설정자 메소드 호출
        myCar.setBatterySize(60)                # 설정자 메소드 호출
        print(myCar.getDesc())      # 전기차 객체를 문자열로 출력


main()
```

```
차량 = (Tesla,Model S,white,10000)
```

왜 상속을 사용하는가?


Car
setSpeed()
turn()
Truck
setSpeed()
turn()
Bus
setSpeed()
turn()
각 클래스에 코드가 중복된다.
Vehicle
setSpeed()
turn()
부모클래스
중복되는 코드는 부모 클래스에 모은다.
Car
Truck
Bus
자식클래스

부모 클래스의 생성자 호출



생성자를 호출하지 않으면 오류

```
class ElectricCar(Car) :
        def __init__(self, make, model, color, price, batterySize):
                super().__init__(make, model, color, price)
                self.batterySize=batterySize
```

생성자를 호출하지 않으면 오류

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

    def eat(self):
        print("동물이 먹고 있습니다. ")


# 부모 클래스의 생성자를 호출하지 않았다!
class Dog(Animal):
    def __init__(self, age=0, name=""):
        self.name=name

# 부모 클래스의 생성자가 호출되지 않아서 age 변수가 생성되지 않았다.
d = Dog()
print(d.age)
```

type()과 isinstance() 함수

```
...
x = Animal()
y = Dog()
print(type(x))     # 어떤 클래스의 객체인지 확인
print(type(y))
```

```
<class '__main__.Animal'>
<class '__main__.Dog'>
```

type()과 isinstance() 함수

```
...
x = Animal()
y = Dog()
print(isinstance(x, Animal))     # 객체를 만든 클래스인지 확인
print(isinstance(y, Animal))     # 부모클래스도 True
```

```
True True
```

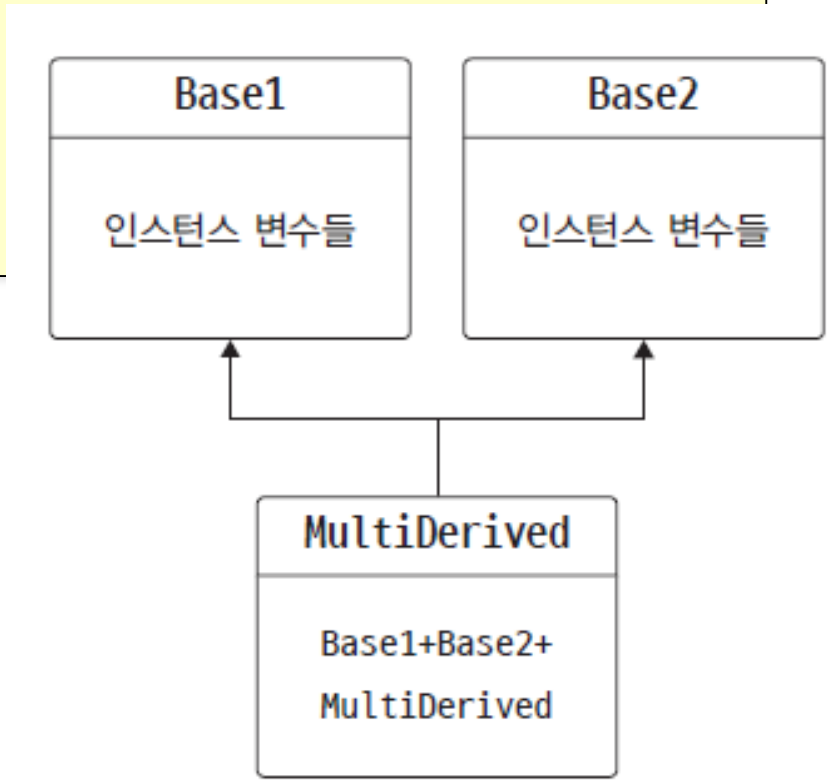
private 멤버 부모 클래스

```
class Parent(object):
    def __init__(self):
        self.__money = 100          # private한 인스턴스 변수

class Child(Parent):
    def __init__(self):
        super().__init__()

obj = Child()
print(obj.money)                    # 오류: 자식은 부모의 private한 변수 접근 불가능
```

```
...
AttributeError: 'Child' object has no attribute 'money'
```

다중 상속

```
class Base1:
    pass

class Base2:
    pass

class MultiDerived(Base1, Base2):
    pass
```

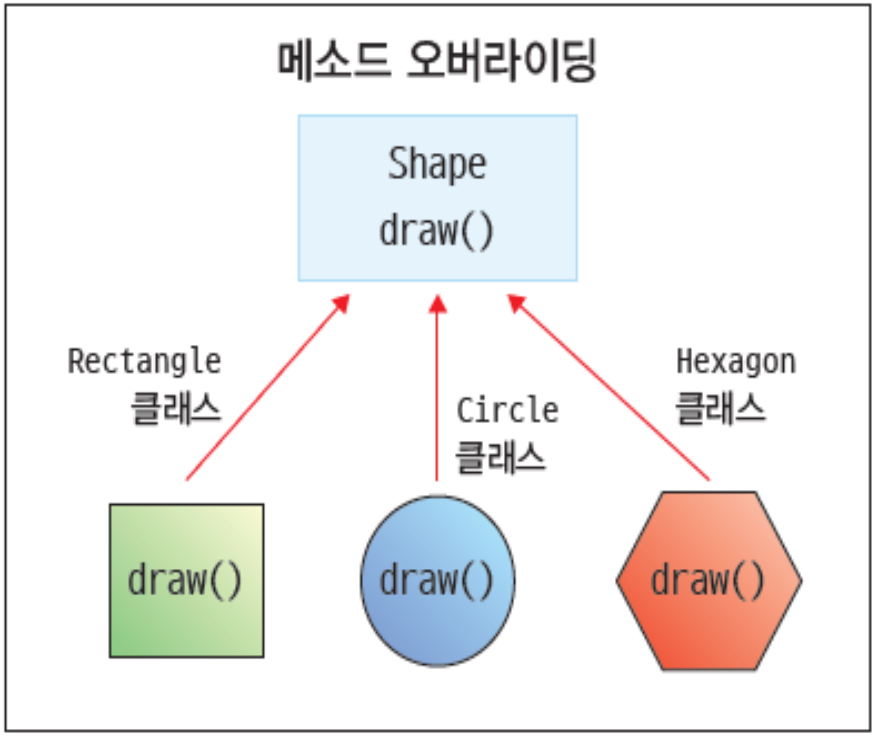


```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def show(self):
        print(self.name, self.age)

 class Student:
    def __init__(self, id):
        self.id = id

    def getId(self):
        return self.id

class CollegeStudent(Person, Student):
    def __init__(self, name, age, id):
        Person.__init__(self, name, age)
        Student.__init__(self, id)

obj = CollegeStudent('Kim', 22, '100036')
obj.show()
print(obj.getId())
```

```
Kim 22
100036
```

메소드 오버라이딩

- 자식 클래스의 메소드가 부모 클래스의 메소드를 오버라이드(재정의) 할 수 있다.



```
import math

class Shape:
    def __init__(self):
        pass

    def draw(self):
        print("draw()가 호출됨")

    def get_area(self):
        print("get_area()가 호출됨")

class Circle(Shape):
    def __init__(self, radius=0):
        super().__init__()
        self.radius = radius

    def draw(self):                          # 메소드 오버라이딩
        print("원을 그립니다.")

    def get_area(self):                      # 메소드 오버라이딩
        return math.pi * self.radius ** 2

c = Circle(10)
c.draw()
print("원의 면적:", c.get_area())
```

```
원을 그립니다.
원의 면적:
314.1592653589793
```

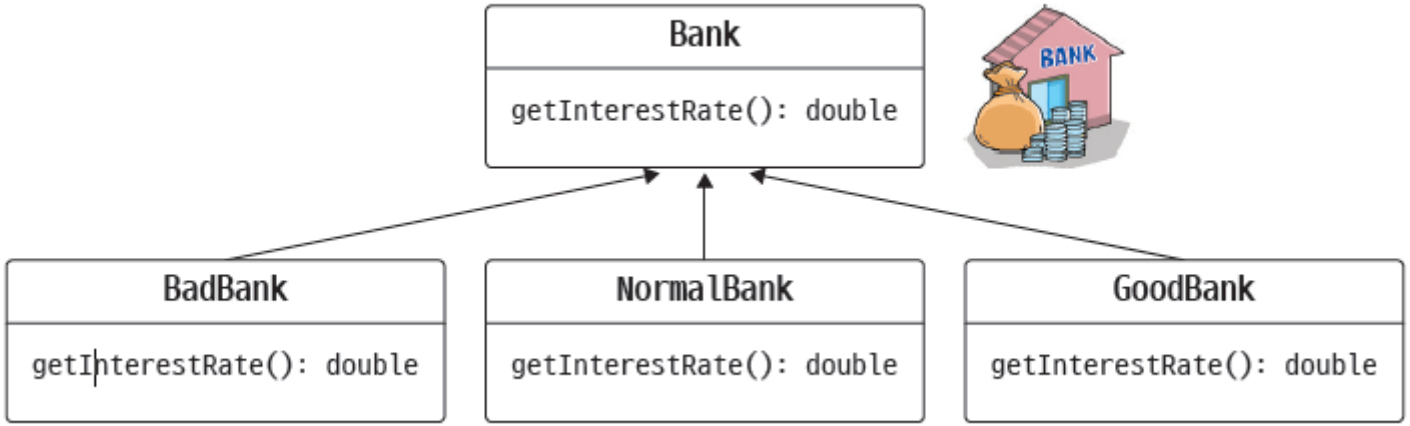
예제

```
class Circle(Shape):
        ...
    def draw(self):                     # 메소드 오버라이딩
        super().draw()                  # 부모 클래스의 draw() 호출
        print("원을 그립니다.")
```

```
draw()가 호출됨
원을 그립니다.
```

Lab: Bank 클래스

- 은행에서 대출을 받을 때, 은행마다 대출 이자가 다르다.
- 이것을 **메소드 오버라이딩**으로 해결하여 보자.



BadBank의 이자율: 10.0

NormalBank의 이자율: 5.0

GoodBank의 이자율: 3.0

```
class Bank():
        def getInterestRate(self):
                return 0.0

class BadBank(Bank):
        def getInterestRate(self):          # 메소드 오버라이딩
                return 10.0;

class NormalBank(Bank):
        def getInterestRate(self):          # 메소드 오버라이딩
                return 5.0;

class GoodBank(Bank):
        def getInterestRate(self):          # 메소드 오버라이딩
                return 3.0;

b1 = BadBank()
b2 = NormalBank()
b3 = GoodBank()
print("BadBank의 이자율: " + str(b1.getInterestRate()))
print("NormalBank의 이자율: " + str(b2.getInterestRate()))
print("GoodBank의 이자율: " + str(b3.getInterestRate()))
```

Lab: 직원과 매니저

- 회사에 **직원(Employee)**과 **매니저(Manager)**가 있다.
- 직원은 월급만 있지만 매니저는 월급 외에 보너스가 있다고 하자.
- Employee 클래스를 상속받아서 Manager 클래스를 작성한다.

  Employee 클래스의 getSalary()는 Manager 클래스에서 재정의된다.

```
이름: 김철수; 월급: 2000000; 보너스: 1000000
```

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

    def  getSalary(self):
        return self.salary

class Manager(Employee):
    def __init__(self, name, salary, bonus):
        super().__init__(name, salary)
        self.bonus = bonus

    def  getSalary(self):                    # 메소드 오버라이딩
        salary = super().getSalary()
        return salary + self.bonus

    def __repr__(self):
        return "이름: "+ self.name+ "; 월급: "+ str(self.salary) +\
               "; 보너스: " + str(self.bonus)

kim = Manager("김철수", 2000000, 1000000)
print(kim)
```

Lab: 은행 계좌

- **BankAccount** 클래스는 다음과 같은 인스턴스 변수와 메소드를 가진다.
  - **balance** : 잔액(정수형)
  - **name** : 소유자의 이름(문자열)
  - **name** : 통장 번호(정수형)
  - **withdraw**() : 출금 메소드
  - **deposit**() : 저금 메소드
- BankAccount 클래스를 상속받아서 **SavingsAccount** 클래스(저축예금)를 작성한다. SavingsAccount 클래스는 추가로 다음과 같은 인스턴스 변수와 메소드를 가진다.
  - **interest_rate** : 이자율(실수형)
  - **add_interest**() : 호출될 때마다 예금에 이자를 더하는 메소드
- BankAccount 클래스를 상속받아서 **CheckingAccount** 클래스(당좌예금)를 작성한다. CheckingAccount 클래스는 추가로 다음과 같은 인스턴스 변수와 메소드를 가진다.
  - **withdraw_charge** : 수표를 1회 발행할 때 수수료(정수형)
  - **withdraw**() : 찾을 금액에 수수료를 더해서 출금한다.

Solution

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

    def withdraw(self, amount):
        self.balance -= amount
        return self.balance

    def deposit(self, amount):
        self.balance += amount
        return self.balance
```

Solution

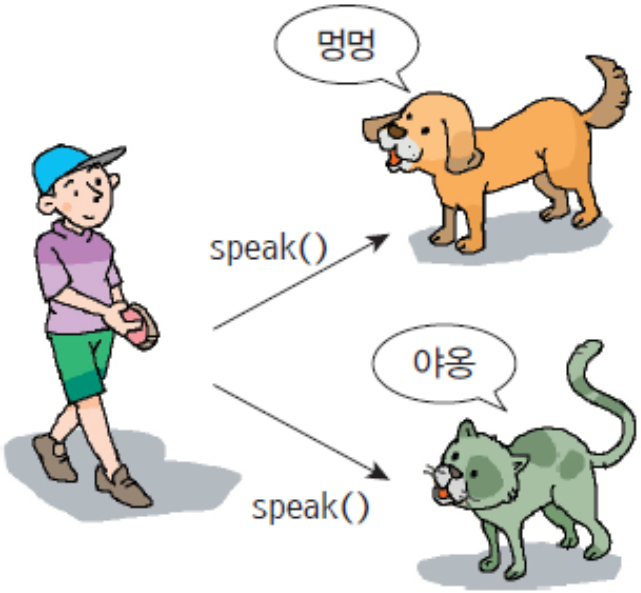
```
class SavingsAccount(BankAccount) :
    def __init__(self,  name, number, balance,  interest_rate):
        super().__init__( name, number, balance)
        self.interest_rate =interest_rate

    def set_interest_rate(self, interest_rate):
        self.interest_rate = interest_rate

    def get_interest_rate(self):
        return self.interest_rate

    def add_interest(self):                    # 예금에 이자를 더한다.
        self.balance += self.balance*self.interest_rate
```

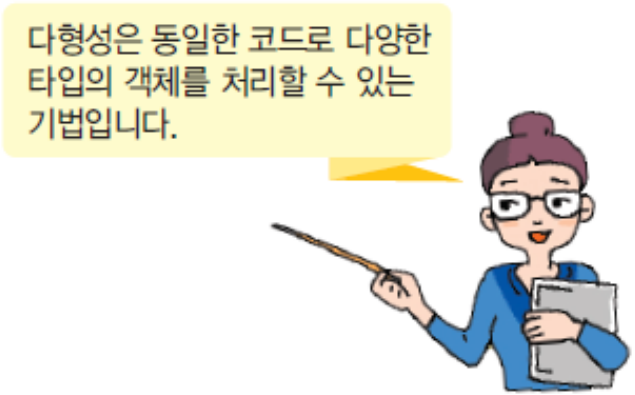
Solution

```
class CheckingAccount(BankAccount) :
    def __init__(self,  name, number, balance):
        super().__init__( name, number, balance)
        self.withdraw_charge = 10000            # 수표 발행 수수료

    def withdraw(self, amount):
        return BankAccount.withdraw(self, amount + self.withdraw_charge)

a1 = SavingsAccount("홍길동", 123456, 10000, 0.05)
a1.add_interest()
print("저축예금의 잔액=", a1.balance)

a2 = CheckingAccount("김철수", 123457, 2000000)
a2.withdraw(100000)
print("당좌예금의 잔액=", a2.balance)
```

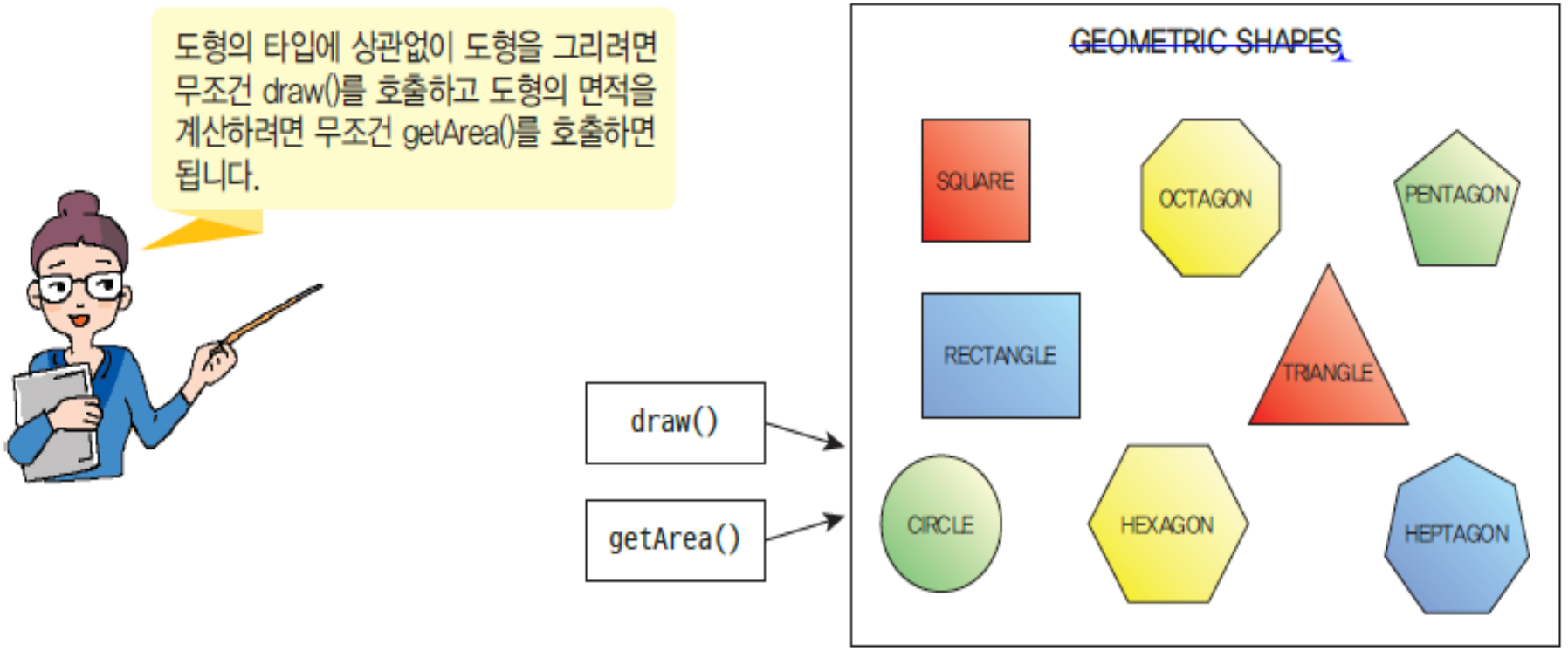
다형성

- 다형성(polymorphism)은 “**많은(poly)+모양(morph)**”이라는 의미
- 주로 프로그래밍 언어에서 **하나의 식별자로 다양한 타입(클래스)을 처리**하는 것을 의미





다형성의 예


도형의 타입에 상관없이 도형을 그리려면 무조건 draw()를 호출하고 도형의 면적을 계산하려면 무조건 getArea()를 호출하면 됩니다.
draw()
getArea()
GEOMETRIC SHAPES
SQUARE
OCTAGON
PENTAGON
RECTANGLE
TRIANGLE
CIRCLE
HEXAGON
HEPTAGON

상속과 다형성

```
class Shape:
    def __init__(self, name):
        self.name = name
    def getArea(self):
        raise NotImplementedError("이것은 추상메소드입니다. ")

class Circle(Shape):
    def __init__(self, name, radius):
        super().__init__(name)
        self.radius = radius

    def getArea(self):                          # 메소드 오버라이딩
        return  3.141592*self.radius**2

class Rectangle(Shape):
    def __init__(self, name, width, height):
        super().__init__(name)
        self.width = width
        self.height = height

    def getArea(self):                          # 메소드 오버라이딩
        return  self.width*self.height
```

상속과 다형성

```
shapeList = [ Circle("c1", 10), Rectangle("r1", 10, 10) ]
for s in shapeList:
        print(s.getArea())     # 다형성
```

```
314.1592
100
```

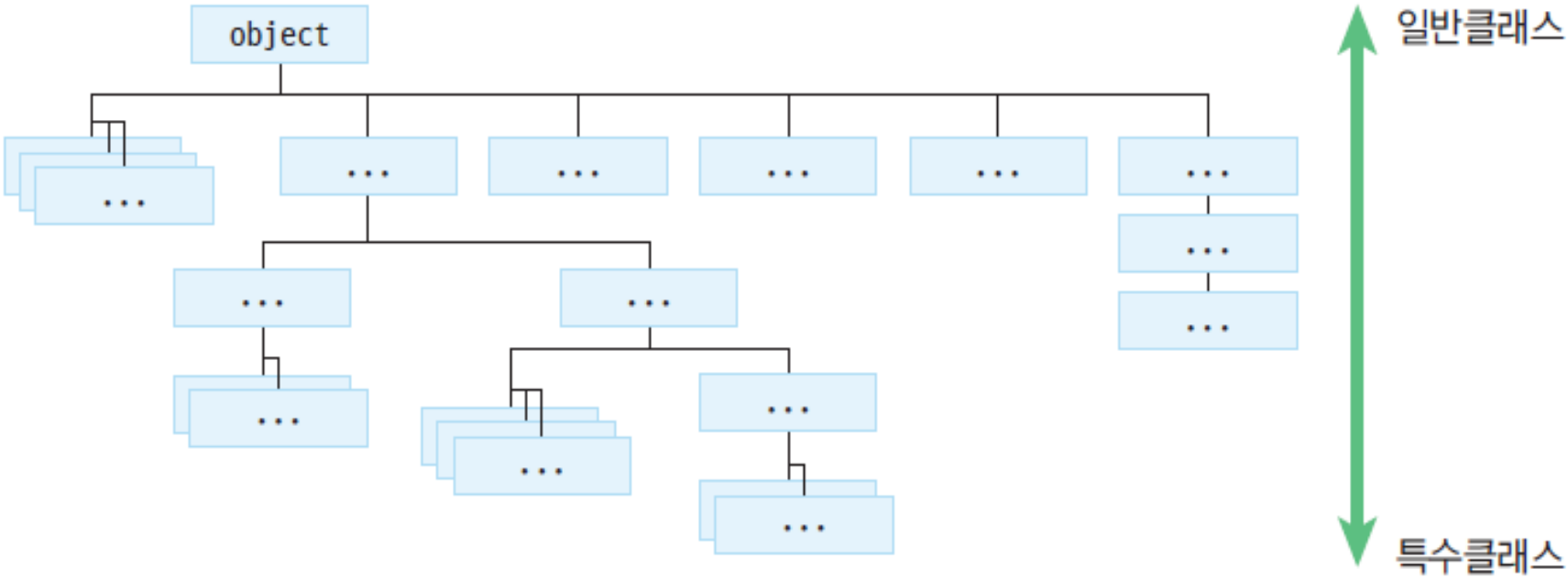
내장 함수와 다형성

```
mylist = [1, 2, 3]                  # 리스트
print("리스트의 길이=", len(mylist))


s = "This is a sentense"            # 문자열
print("문자열의 길이=", len(s))


d = {'aaa': 1, 'bbb': 2}            # 딕셔너리
print("딕셔너리의 길이=", len(d))
```

```
리스트의 길이= 3
문자열의 길이= 18
딕셔너리의 길이= 2
```

object 클래스

- 모든 클래스의 맨 위에는 **object 클래스**가 있다.



object 클래스의 메소드

메소드	
__init__ (self [,args...])	생성자 예 obj = className(args)
__del__(self)	소멸자 예 del obj
__repr__(self)	객체 표현 문자열 반환 객체를 구별하기 위한 출력 예 repr(obj)
__str__(self)	문자열 표현 반환 객체를 가독성 있게 문자열로 출력 예 str(obj)
__cmp__ (self, x)	객체 비교 예 cmp(obj, x)

__repr__() 메소드

```
class Book:
        def __init__(self, title, isbn):
                self.__title = title
                self.__isbn = isbn

        # 객체가 가진 정보를 문자열로 반환
        def __repr__(self):             # 메소드 오버라이딩
                return "ISBN: "+ self.__isbn+ "; TITLE: "+ self.__title

book = Book("The Python Tutorial", "0123456")
print(book)
```

```
ISBN: 0123456; TITLE: The Python Tutorial
```

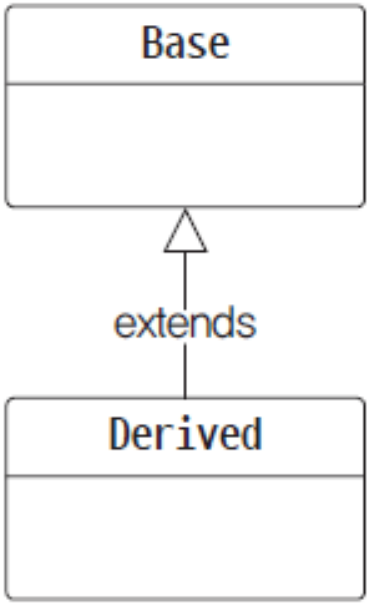
__str__() 메소드

```
class MyTime:
        def __init__(self, hour, minute, second=0):
                self.hour = hour
                self.minute = minute
                self.second = second
        def __str__(self):                  # 메소드 오버라이딩
                return '%.2d:%.2d:%.2d' % (self.hour, self.minute, self.second)

time = MyTime(10, 25)
print(time)
```

```
10:25:00
```

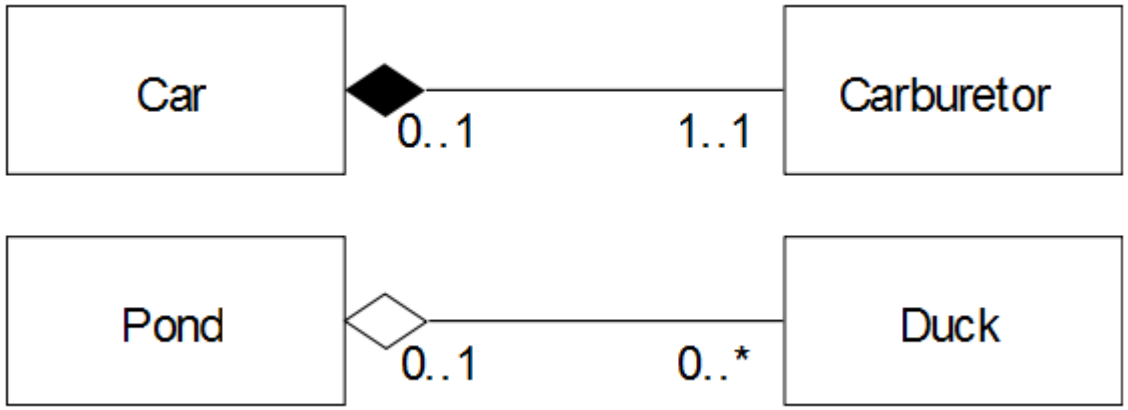
클래스 관계: 상속

- **is-a 관계**: 상속
  - 승용차는 차량의 일종이다(**Car** is a **Vehicle**).
  - 강아지는 동물의 일종이다(**Dog** is an **Animal**).
  - 원은 도형의 일종이다(**Circle** is a **Shape**).

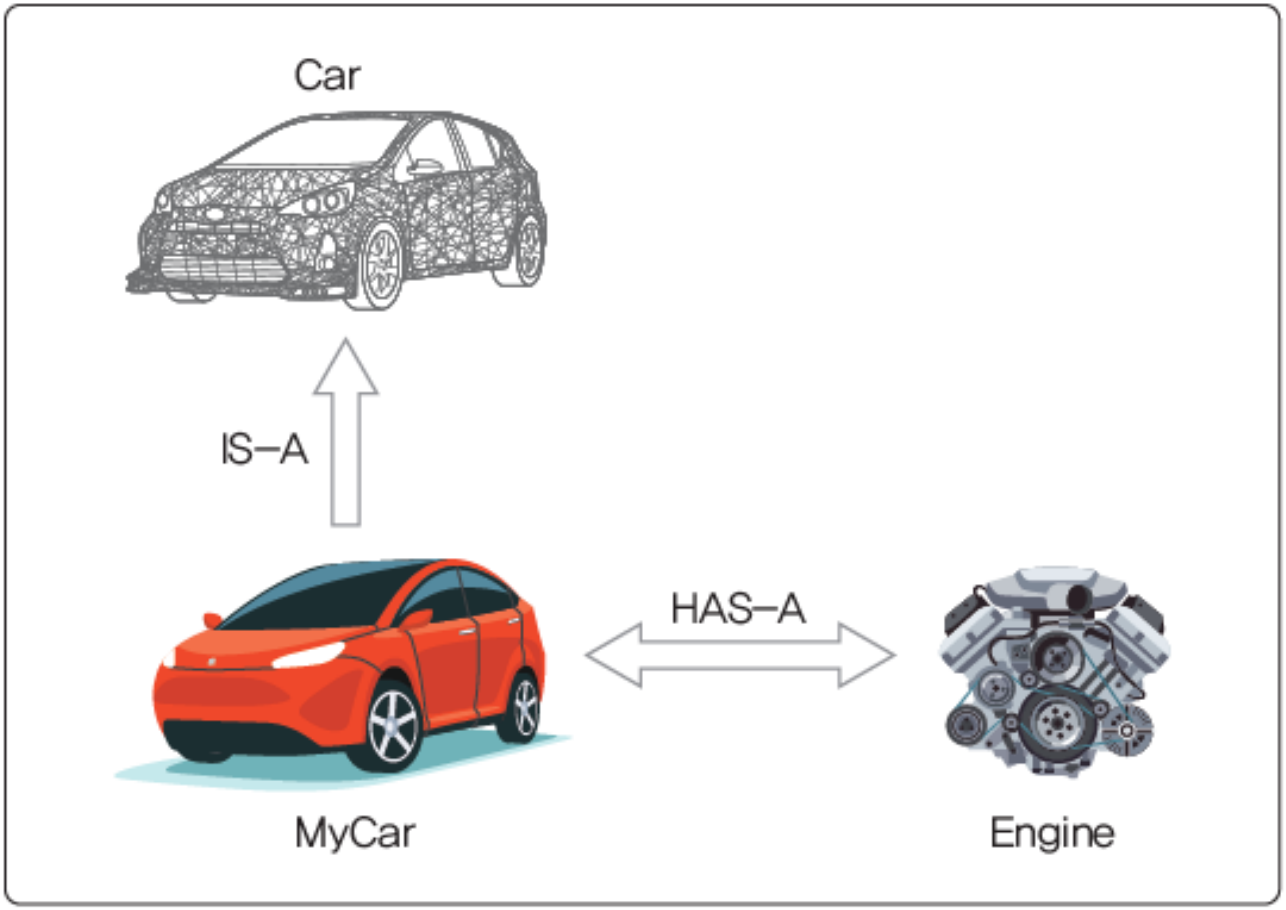


클래스 관계:구성

- **has-a 관계**: 구성
  - 도서관은 책을 가지고 있다(**Library** has a **Book**).
  - 거실은 소파를 가지고 있다(**Living room** has a **Sofa**).



is-a vs has-a



is-a & has-a 관계의 예

```
class Animal(object):
    pass

class Dog(Animal):          # is-a 관계 설정
    def __init__(self, name):
        self.name = name

class Person(object):
    def __init__(self, name):
        self.name = name
        self.pet = None         # has-a 관계



dog1 = Dog("푸들")
person1 = Person("홍길동")
person1.pet = dog1          # has-a 관계 설정
```

Lab: Card와 Deck

- 카드를 나타내는 **Card 클래스**를 작성하고 52개의 Card 객체를 가지고 있는 **Deck 클래스**를 작성한다.
- 각 클래스의 **__str__() 메소드**를 구현하여서 deck 안에 들어 있는 카드를 다음과 같이 출력한다.

['클럽 에이스', '클럽 2', '클럽 3', '클럽 4', '클럽 5', '클럽 6', '클럽 7', '클럽 8', '클럽 9', '클럽 10', '클럽 잭', '클럽 퀸', '클럽 킹', '다이아몬드 에이스', '다이아몬드 2', '다이아몬드 3', '다이아몬드 4', '다이아몬드 5', '다이아몬드 6', '다이아몬드 7', '다이아몬드 8', '다이아몬드 9', '다이아몬드 10', '다이아몬드 잭', '다이아몬드 퀸', '다이아몬드 킹', '하트 에이스', '하트 2', '하트 3', '하트 4', '하트 5', '하트 6', '하트 7', '하트 8', '하트 9', '하트 10', '하트 잭', '하트 퀸', '하트 킹', '스페이드 에이스', '스페이드 2', '스페이드 3', '스페이드 4', '스페이드 5', '스페이드 6', '스페이드 7', '스페이드 8', '스페이드 9', '스페이드 10', '스페이드 잭', '스페이드 퀸', '스페이드 킹']

Solution

```
class Card:
    suitNames = ['클럽', '다이아몬드', '하트', '스페이드']
    rankNames = [None, '에이스', '2', '3', '4', '5', '6', '7', '8', '9', '10', '잭', '퀸', '킹']

    def __init__(self, suit, rank):
        self.suit = suit
        self.rank = rank

    def __str__(self):
        return Card.suitNames[self.suit]+" "+\
                        Card.rankNames[self.rank]
```

Solution

```
class Deck:
    def __init__(self):
        self.cards = []

        for suit in range(4):
            for rank in range(1, 14):
                card = Card(suit, rank)          # has-a 관계
                self.cards.append(card)

    def __str__(self):
        lst = [str(card) for card in self.cards]
        return str(lst)

deck = Deck()                    # Deck 객체를 생성한다.
print(deck)                      # Deck 객체를 출력한다. __str__()이 호출된다.
```

Lab: 학생과 강사

- 일반적인 사람을 나타내는 **Person 클래스**를 정의한다.
- Person 클래스를 상속받아서 학생을 나타내는 **클래스 Student**와 선생님을 나타내는 **클래스 Teacher**를 정의한다.

이름=**홍길동**

주민번호=12345678

수강과목=['자료구조']

평점=0

이름=**김철수**

주민번호=123456790

강의과목=['Python']

월급=2000000

Solution

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

class Student(Person):
    UNDERGRADUATE = 0
    POSTGRADUATE = 1

    def __init__(self, name, number, studentType ):
        super().__init__(name, number)
        self.studentType = studentType      # 학생 구분
        self.gpa=0                          # 성적
        self.classes = []                   # 수강 과목 목록

    def enrollCourse(self, course):         # 수강 신청하기
        self.classes.append(course)

    def __str__(self):
        return  "\n이름="+self.name+ "\n주민번호="+self.number+\
            "\n수강과목="+ str(self.classes)+ "\n평점="+str(self.gpa)
```

Solution

```
class Teacher(Person):
    def __init__(self, name, number):
        super().__init__(name, number)
        self.courses = []                    # 강의 과목 목록
        self.salary=2000000                  # 급여

    def assignTeaching(self, course):        # 강의를 배정
        self.courses.append(course)

    def __str__(self):
        return  "\n이름="+self.name+ "\n주민번호="+self.number+\
            "\n강의과목="+str(self.courses)+ "\n월급="+str(self.salary)

hong = Student("홍길동", "12345678", Student.UNDERGRADUATE )
hong.enrollCourse("자료구조")
print(hong)

kim  = Teacher("김철수", "123456790")
kim.assignTeaching("Python")
print(kim)
```

Lab: Vehicle, Car, Truck

- 일반적인 운송수단을 나타내는 **Vehicle 클래스**를 상속받아서 **Car 클래스**와 **Truck 클래스**를 작성해보자.

```
truck1: 트럭을 운전합니다.
truck2: 트럭을 운전합니다.
car1: 승용차를 운전합니다.
```

Solution

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

    def drive(self):
        raise NotImplementedError("이것은 추상메소드입니다. ")

    def stop(self):
        raise NotImplementedError("이것은 추상메소드입니다. ")
```

Solution

```
class Car(Vehicle):
    def drive(self):                    # 메소드 오버라이딩
        return '승용차를 운전합니다. '

    def stop(self):                     # 메소드 오버라이딩
        return '승용차를 정지합니다. '


class Truck(Vehicle):
    def drive(self):                    # 메소드 오버라이딩
        return '트럭을 운전합니다. '

    def stop(self):                     # 메소드 오버라이딩
        return '트럭을 정지합니다. '


cars = [Truck('truck1'), Truck('truck2'),  Car('car1')]

for car in cars:
    print( car.name + ': ' + car.drive())      # 다형성
```

이번 장에서 배운 것

- 상속은 다른 클래스를 **재사용**하는 탁월한 방법이다. 객체와 객체 간의 is-a 관계가 성립된다면 상속을 이용하도록 하자.
- 상속을 사용하면 **중복된 코드를 줄일 수** 있다. 공통적인 코드는 부모 클래스를 작성하여 한 곳으로 모으도록 하자.
- 상속에서는 **부모 클래스의 메소드를 자식 클래스가 재정의**할 수 있다. 이것을 메소드 오버라이딩이라고 한다.