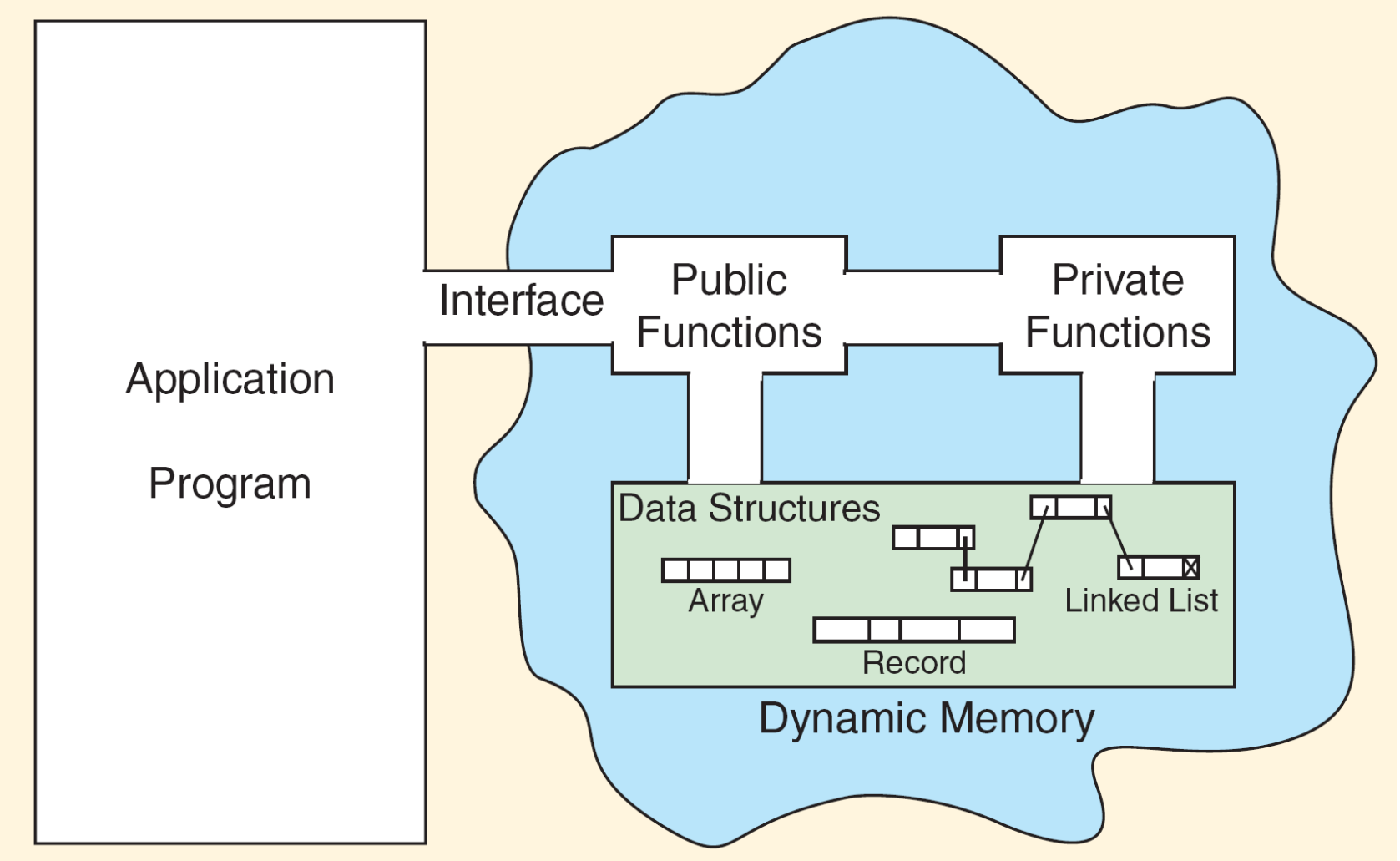
Object Oriented Programming

Contents

- Abstract Data Type: ADT
- Object Oriented Programming
- Encapsulation
  - Class, Object(instance)
- Inheritance
  - Method Overriding
  - Operator Overloading
- Polymorphism

추상 데이터 타입(Abstract Data Type: ADT)



[Abstract Data Type Model]

Procedural Programming vs Object Oriented Programming

- 프로시저 프로그래밍(Procedural programming)
  - 프로그램을 구성하는 데이터와 프로시저(함수)를 분리해서 접근
  - 명령어의 선언 순서와 함수 호출에 따라 프로그램이 작동하는 방식을 강조
  - 함수는 입력된 인수를 가지고 작동하고, 결과를 반환하며, 함수들이 모여서 프로그램을 구성
  - 프로그램은 주로 단순하고 구조화하기 쉬우며, 대부분의 컴퓨터 언어들이 이 방식을 지원

- 객체 지향 프로그래밍(Object-oriented programming)
  - 프로그램을 데이터와 함수를 하나의 단위로 묶은 객체(object)로 보는 관점
  - 객체는 각자의 특징을 가지고 있으며, 서로 다른 객체들끼리 상호작용
  - 모듈화가 쉽고 유지보수가 용이하며, 코드의 재사용성이 높아짐
  - 객체 지향적인 방식은 대규모 프로그램을 작성할 때 특히 유용

- 프로시저 프로그래밍은 함수 중심의 프로그래밍 방식이며, 객체 지향 프로그래밍은 객체 중심의 프로그래밍 방식
- 프로그래머는 프로그램의 목적과 특성에 따라서 적절한 방식을 선택하여 사용

Object Oriented Programming

- 객체 지향 프로그래밍(Object Oriented Programming, OOP)은 일상생활의 문제를 사람의 시각에서 사물을 바라보는 관점(데이터 + 기능)으로 프로그램을 설계하고자 하는 개념

- 어떤 기능을 함수 코드에 묶어 두는 것이 아니라, 데이터(변수)와 처리하는 기능(함수)을 묶은 하나의 단일 프로그램을 객체라고 하는 코드에 넣어 다른 프로그래머가 재사용할 수 있도록 하는, 컴퓨터 공학의 오래된 프로그래밍 기법 중 하나

Object Oriented Design 특징

- 추상화(Abstraction)
  - 복잡한 현실 세계를 객체로 모델링하여 단순화하는 과정
  - 추상화를 통해 핵심적인 특징을 추출하고, 이를 객체의 속성(Property)과 기능(Method)으로 표현
- 캡슐화(Encapsulation)
  - 객체의 상태(State)와 행위(Behavior)를 하나로 묶고, 외부에서의 접근을 제한하여 객체를 보호하는 과정
  - 캡슐화를 통해 객체의 내부 구조를 감추고, 외부에 제공하는 인터페이스만 노출
- 상속(Inheritance)
  - 이미 존재하는 클래스를 기반으로 새로운 클래스를 생성하는 과정
  - 상속을 통해 기존 클래스의 속성과 기능을 재사용하고, 새로운 기능을 추가하여 코드를 간결하게 작성 가능
- 다형성(Polymorphism)
  - 같은 이름의 메서드가 서로 다른 객체에서 다른 방식으로 동작하는 특성
  - 다형성을 통해 코드의 재사용성과 유연성을 높일 수 있으며, 객체 간의 상호작용을 쉽게 구현

- 객체지향 설계는 이러한 특징들을 통해 모듈화와 추상화 수준이 높은 소프트웨어를 만들 수 있으며, 유지보수와 확장성이 뛰어난 코드를 작성

User-defined data type

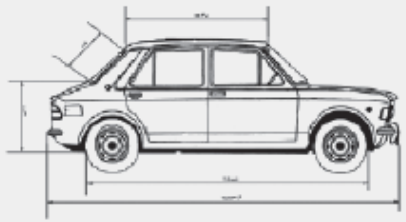
```
// C - struct

struct  car{
char color[10];
int speed;
};
main()
{struct car mycar1; //구조체변수 생성
}
upSpeed()
{}
downSpeed()
{}
```

```
#Python - Class

class  Car :
    color = ""
    speed = 0

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

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

mycar1=Car() #객체 생성
```

Class

- Class 개념
  - 파이썬은 Object Oriented 개념을 적용할 수 있는 프로그래밍 언어
  - Class의 모양과 생성

```
class 클래스이름 :
    # 이 부분에 관련 코드 구현
```



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

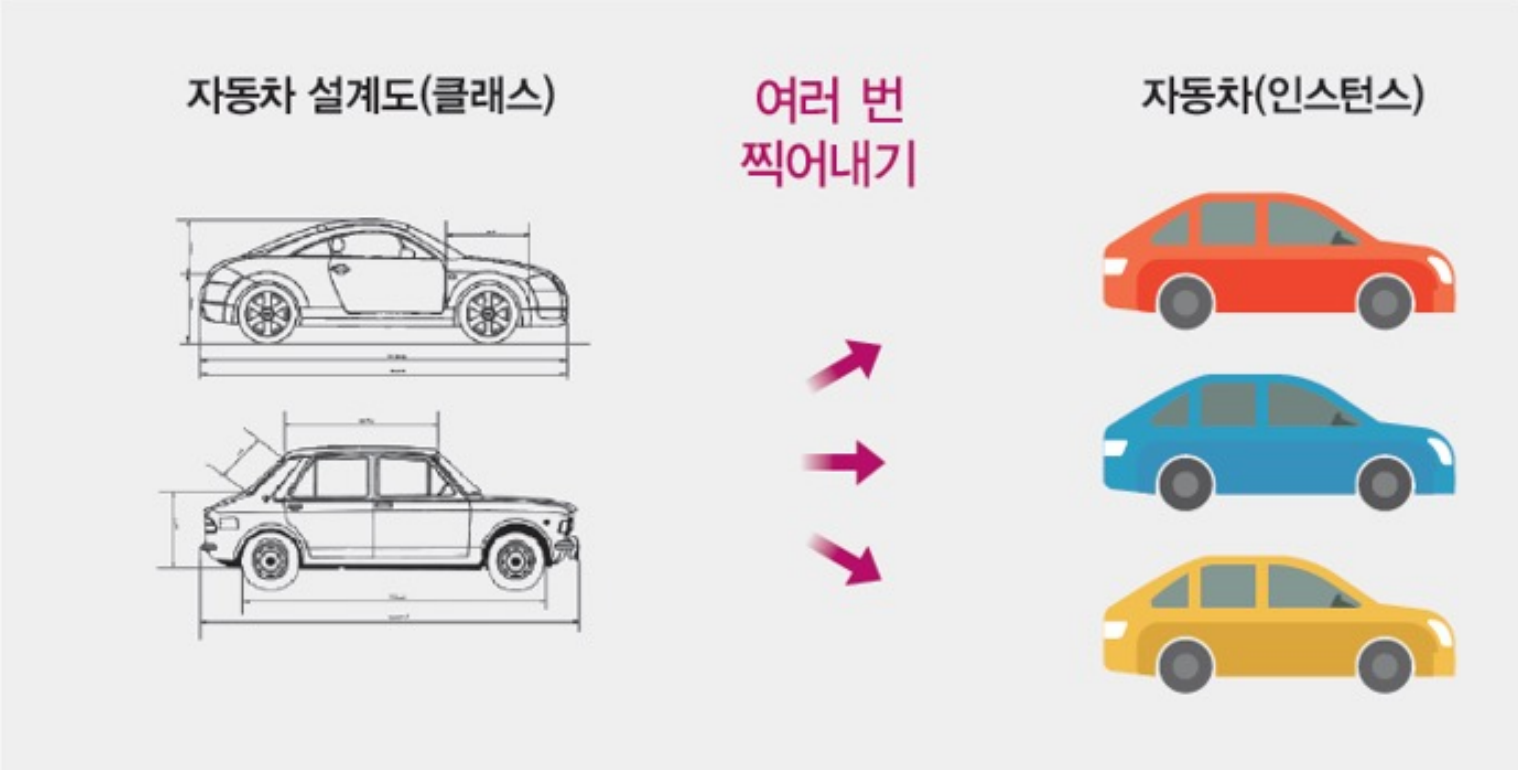
Class

- 자동차의 속성은 필드(Field)라 함. 자동차의 기능은 함수(Function)형태로 구현. Class 안에서 구현된 함수는 메소드(Method)라 부름.

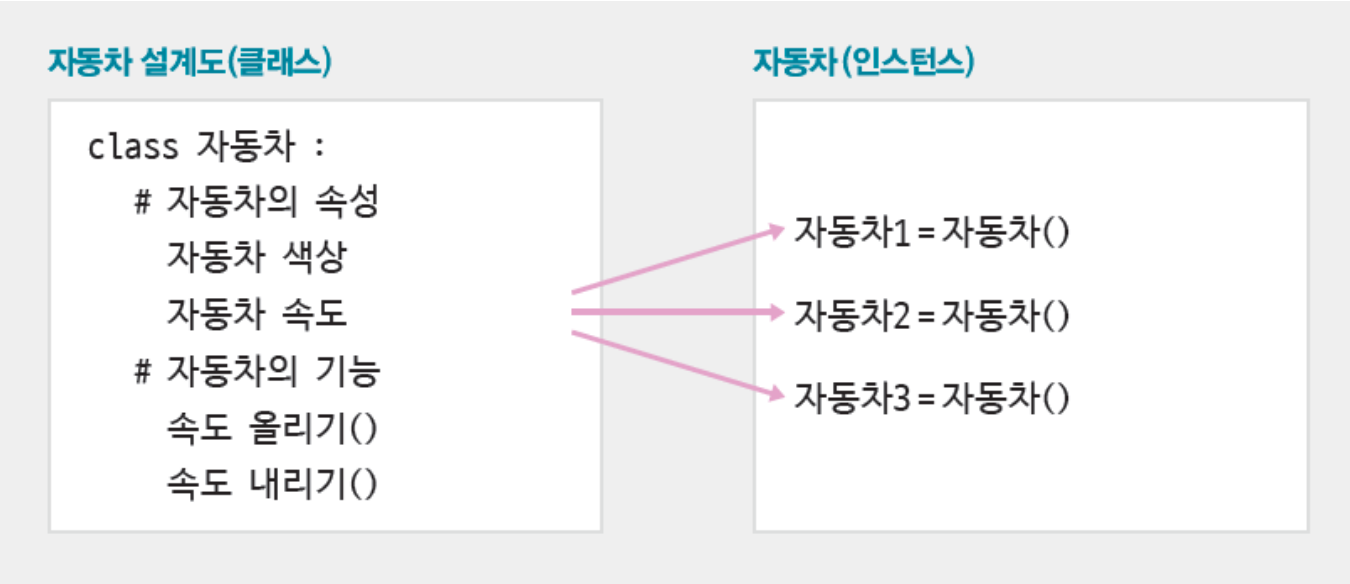
```
class Car :
    # 자동차의 필드
    색상 = ""
    현재속도 = 0

    # 자동차의 메소드
    def upSpeed(증가할_속도량) :
        # 현재 속도에서 증가할_속도량만큼 속도를 올리는 코드

    def downSpeed(감소할_속도량) :
        # 현재 속도에서 증가할_속도량만큼 속도를 내리는 코드
```

Class

- instance 생성하기



Class



- 세 대의 자동차 instance 생성을 실제 코드로 구현

```
myCar1=Car()
myCar2=Car()
myCar3=Car()
```

- 3개 instance는 각각 자동차의 색상(color), 속도(speed) 필드를 가짐

Class

- 필드에 값 대입하기
  - 각 instance에는 별도의 필드가 존재, 각각에 별도의 값 대입 가능함


```
myCar1.color = "빨강"
myCar1.speed = 0
myCar2.color = "파랑"
myCar2.speed = 0
myCar3.color = "노랑"
myCar3.speed = 0
```

Class

- 메소드 호출하기
  - Car Class에서 메소드는 upSpeed()와 downSpeed() 2개임

```
myCar1.upSpeed(30)
myCar2.upSpeed(60)
```

- Class의 완전한 작동 코딩

```
1  # 클래스 정의 부분
2  class  Car :
3      color = ""
4      speed = 0
5
6      def  upSpeed(self, value) :
7          self.speed += value
8
9      def  downSpeed(self, value) :
10         self.speed -= value
11
```

Class

```
12  # 메인 코드 부분
13  myCar1=Car()
14  myCar1.color="빨강"
15  myCar1.speed=0
16
17  myCar2=Car()
18  myCar2.color="파랑"
19  myCar2.speed=0
20
21  myCar3=Car()
22  myCar3.color="노랑"
23  myCar3.speed=0
24
25  myCar1.upSpeed(30)
26  print("자동차1의 색상은 %s이며, 현재속도는 %d km 입니다." % (myCar1.
    color, myCar1.speed))
27
28  myCar2.upSpeed(60)
29  print("자동차2의 색상은 %s이며, 현재속도는 %d km 입니다." % (myCar2.
    color, myCar2.speed))
30
```

Class

```
31  myCar3.upSpeed(0)
32  print("자동차3의 색상은 %s이며, 현재속도는 %d km 입니다." % (myCar3.
    color, myCar3.speed))
```

출력 결과

```
자동차1의 색상은 빨강이며, 현재속도는 30 km 입니다.
자동차2의 색상은 파랑이며, 현재속도는 60 km 입니다.
자동차3의 색상은 노랑이며, 현재속도는 0 km 입니다.
```

단계	작업	형식	예
1단계	클래스 생성	class 클래스이름 : // 필드 선언 // 메소드 선언	class Car : color="" def upSpeed (self, value) : ~~~
2단계	인스턴스 생성	인스턴스=클래스이름()	myCar1=Car()
3단계	필드나 메소드 사용	인스턴스.필드이름=값 인스턴스.메소드()	myCar1.color="빨강" myCar1.upSpeed(30)

Constructor

- Constructor의 의미
  - Constructor는 instance를 생성하면 무조건 호출되는 메소드

```
13행: myCar1=Car()
14행: myCar1.color="빨강"
15행: myCar1.speed=0
```

    - 13행에서 instance 생성 후 14,15행에서 따로 초기화를 하였는데, 13행 자체에서 instance 생성과 동시에 필드값을 초기화 할 수 있음. 이 함수를 Constructor라 함

- Constructor의 기본
  - Constructor는 __init__( )라는 이름을 가짐

```
class 클래스이름 :
    def __init__(self) :
      // 이 부분에 초기화할 코드를 입력
```

Constructor

- Car Class의 Constructor

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

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

```
myCar1 = Car()
```

- Instance를 생성하면 자동으로 Constructor가 호출

Constructor

- 기본 Constructor - 매개변수가 self만 있는 Constructor

```
1  # 클래스 정의 부분
2  class  Car :
3      color = ""
4      speed = 0
5
6      def __init__(self) :
7          self.color = "빨강"
8          self.speed = 0
9
10     def  upSpeed(self, value) :
11         self.speed += value
12
13     def  downSpeed(self, value) :
14         self.speed -= value
15
16 # 메인 코드 부분
17 myCar1 = Car()
18 myCar2 = Car()
```

Constructor

```
19
20  print("자동차1의 색상은 %s이며, 현재속도는 %d km 입니다." % (myCar1.
    color, myCar1.speed))
21
22  print("자동차2의 색상은 %s이며, 현재속도는 %d km 입니다." % (myCar2.
    color, myCar2.speed))
```

출력 결과

```
자동차1의 색상은 빨강이며, 현재속도는 0 km 입니다.
자동차2의 색상은 빨강이며, 현재속도는 0 km 입니다.
```

- 매개변수가 있는 Constructor
  - instance를 만들 때 초기값을 매개변수로 넘기는 방법을 사용해봄

Constructor

```
1  # 클래스 정의 부분
2  class  Car :
3      color = ""
4      speed = 0
5
6      def __init__(self, value1, value2) :
7          self.color = value1
8          self.speed = value2
9
10     ## [소스코드 11-3]의 upSpeed(), downSpeed() 함수와 동일
11
12 # 메인 코드 부분
13 myCar1 = Car("빨강", 30)
14 myCar2 = Car("파랑", 60)
15
16 ## [소스코드 11-3]의 20~22행과 동일
```

```
자동차1의 색상은 빨강이며, 현재속도는 30 km 입니다.
자동차2의 색상은 파랑이며, 현재속도는 60 km 입니다.
```

Constructor

- Object Oriented 기본 프로그램 완성

```
1  # 클래스 선언
2  class  Car :
3      name=""
4      speed=0
5
6      def __init__(self, name, speed):
7          self.name=name
8          self.speed=speed
9
10     def  getName(self) :
11         return self.name
12
13     def getSpeed(self) :
14         return self.speed
15
```

Constructor

```
16  # 변수 선언
17  car1, car2=None, None
18
19  # 메인 코드 부분
20  car1=Car("아우디", 0)
21  car2=Car("벤츠", 30)
22
23  print("%s의 현재 속도는 %d입니다." % (car1.getName(), car1.getSpeed() ))
24  print("%s의 현재 속도는 %d입니다." % (car2.getName(), car2.getSpeed() ))
```

- 10행,13행 : getName( )과 getSpeed( ) 메소드를 만들고 자동차의 이름과 현재 속도를 반환함
- 23행, 24행 : name이나 speed 필드를 사용하지 않고 getName( ), getSpeed( ) 메소드를 사용해서 값을 알아냄

Instance variable & Class variable

- instance 변수

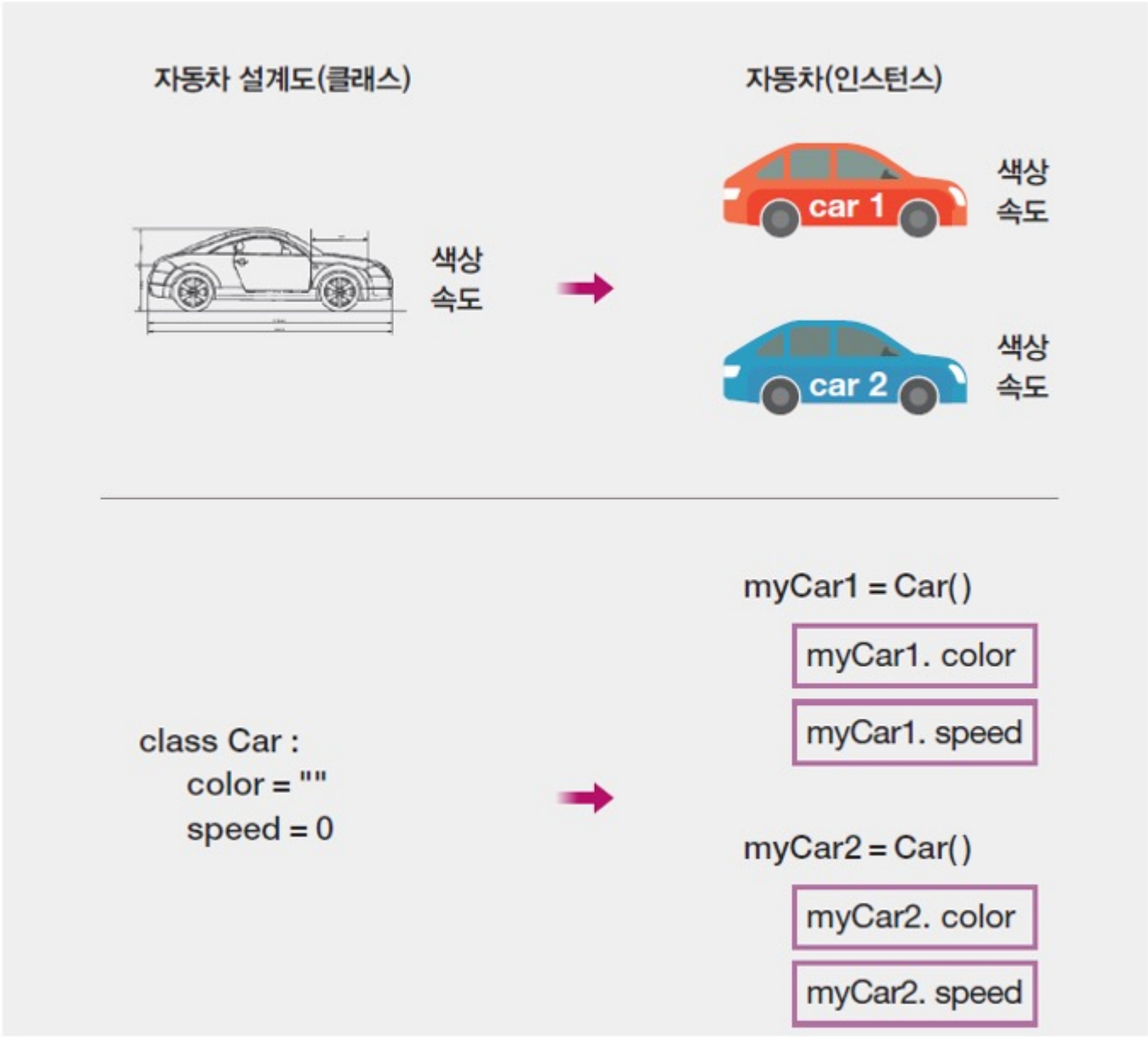
```
class Car :
   color=""  # 필드 : 인스턴스 변수
   speed=0    # 필드 : 인스턴스 변수
```

  - Class를 이용하여 메인 코드 부분에서 instance 만들기

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

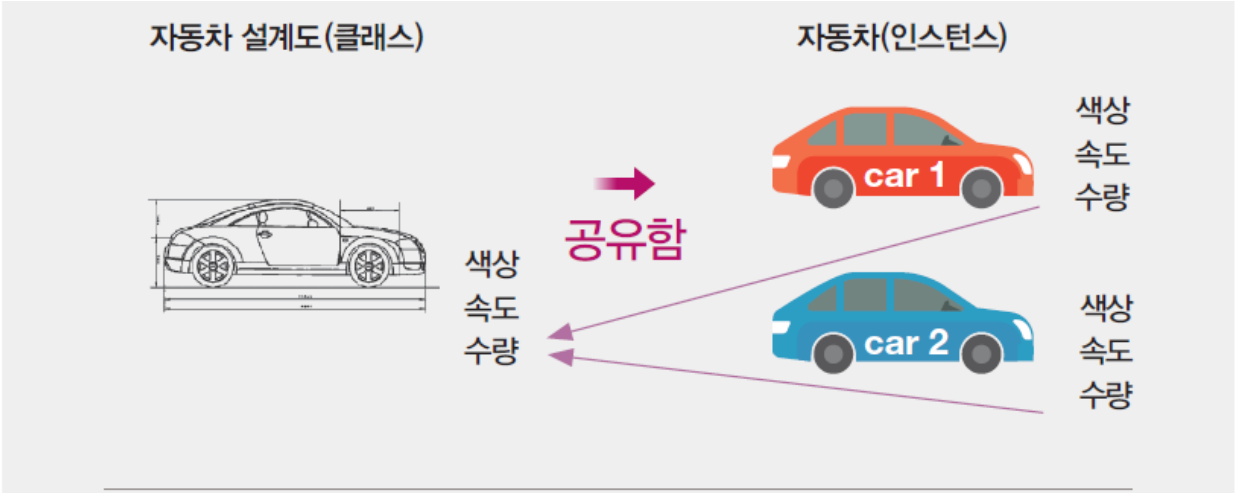
Instance variable & Class variable

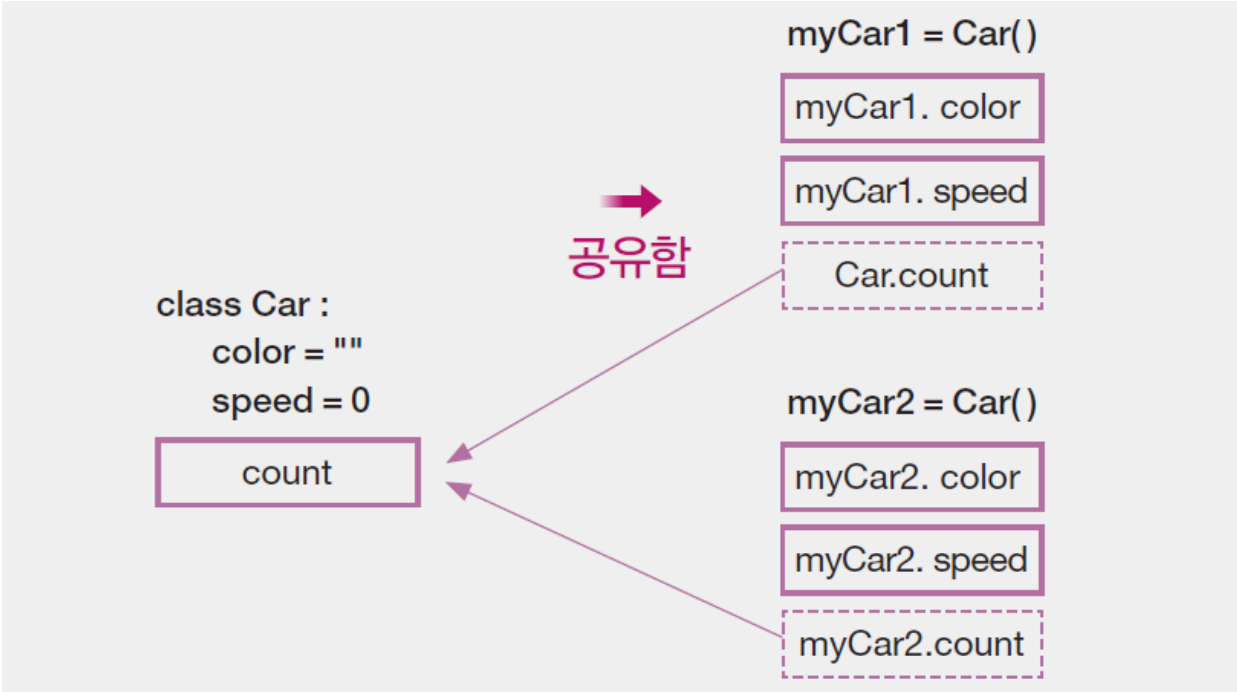
- 인스턴스 변수



Instance variable & Class variable

- Class 변수
  - Class 안에 공간이 할당된 변수





Instance variable & Class variable

- Class 변수에 접근시 'Class이름.Class변수명' 또는 'instance.Class변수명' 방식으로 접근해야 Class에 이미 생성되어 있는 공간을 공유 할 수 있음.

```
1  ## 클래스 선언
2  class  Car :
3      color=""   # 인스턴스 변수
4      speed=0   # 인스턴스 변수
5      count=0   # 클래스 변수
6
7      def __init__(self):
8          self.speed=0
9          Car.count+=1
10
11 # 변수 선언
12 myCar1, myCar2=None, None
13
14 # 메인 코드 부분
15 myCar1=Car()
16 myCar1.speed=30
17 print("자동차1의 현재 속도는 %dkm, 생산된 자동차 숫자는 총 %d대입니다."
   % (myCar1.speed, Car.count) )
```

Instance variable & Class variable

```
18
19  myCar2=Car()
20  myCar2.speed=60
21  print("자동차2의 현재 속도는 %dkm, 생산된 자동차 숫자는 총 %d대입니다."
    % (myCar2.speed, myCar2.count) )
```

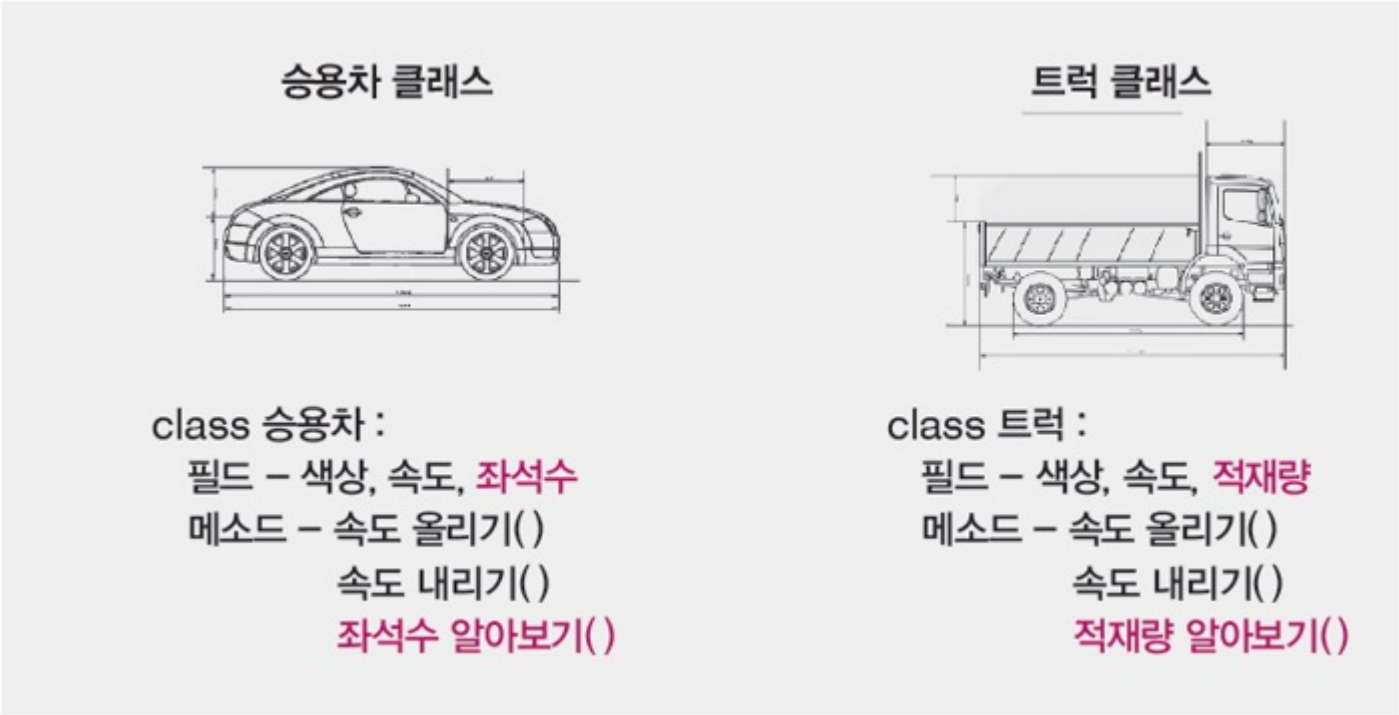
출력 결과

```
자동차1의 현재 속도는 30km, 생산된 자동차 숫자는 총 1대입니다.
자동차2의 현재 속도는 60km, 생산된 자동차 숫자는 총 2대입니다.
```

- 5행 : Class 변수 count를 선언하고 0으로 초기화
- 7행~9행 : Constructor에서 Class 변수에 접근하기 위해 Class이름.count를 1 증가. 즉 Constructor가 작동할 때는 15행과 19행에서 instance를 생성할 때이므로, 자동차의 총 생산대수를 1씩 증가시킨 것임
- 메인 코드 부분에서 Class 변수를 사용하기 위해서 Car.count 또는 myCar2.count 모두 사용 가능. 즉 둘 다 Class 변수에 접근됨

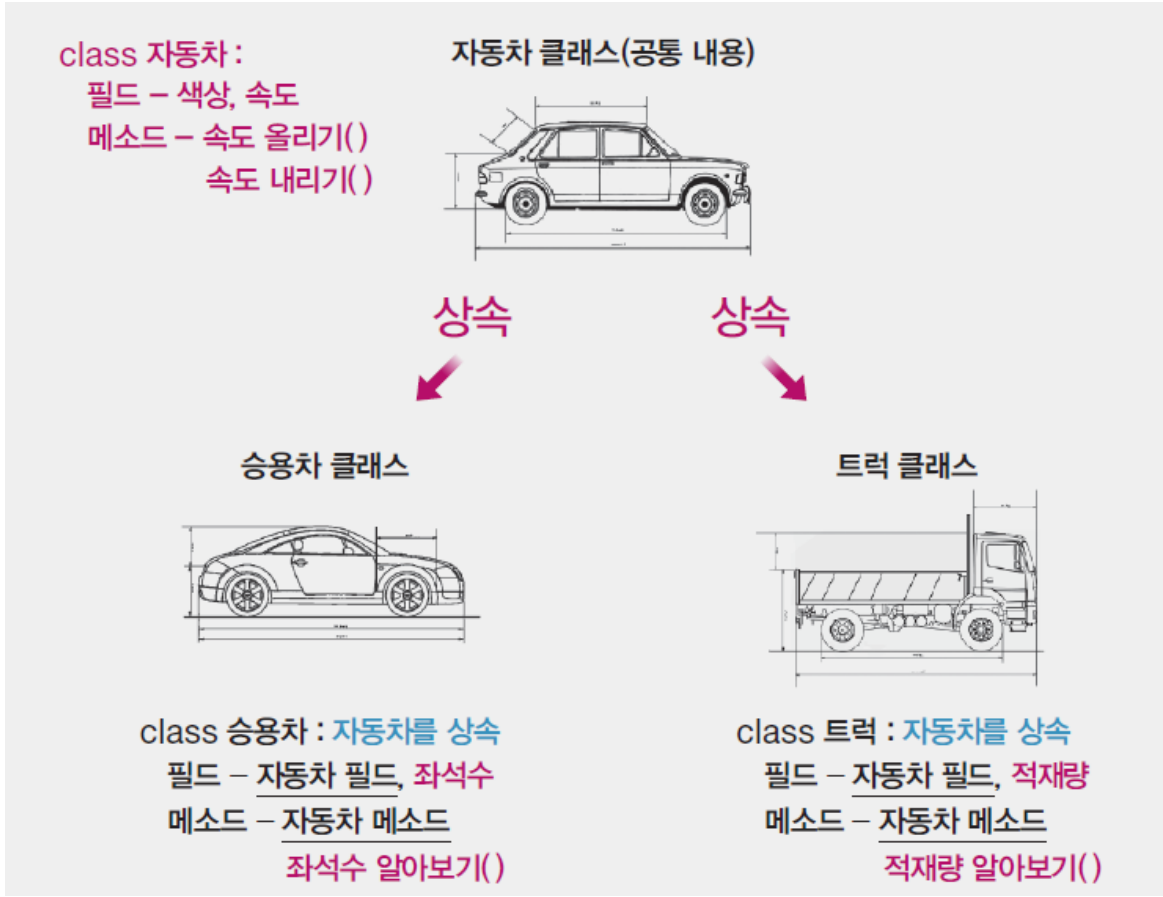
Inheritance

- Inheritance의 개념
  - 기존 Class의 필드와 메소드를 그대로 물려받는 새로운 Class를 만드는 것.



Inheritance

- 기존의 두 Class가 공통되는 것이 많음. 즉 공통된 특징을 '자동차' 라는 Class로 만들고 승용차와 트럭은 Class의 특징을 물려받아 각각에 필요한 필드와 메소드만 추가하면 효율적일 것임.



Inheritance

- 상위 Class인 자동차 Class를 '슈퍼 Class' 또는 '부모 Class'라 하며, 하위 Class인 승용차와 트럭 Class는 '서브 Class' 또는 '자식 Class'라 함

```
class 서브 클래스(슈퍼 클래스) :
   // 이곳에 서브 클래스의 내용을 코딩
```

Inheritance

- Object Oriented 활용 프로그램 완성
  - 자동차 Class를 만든 후, 승용차 Class와 트럭 Class가 자동차 Class를 상속 받음

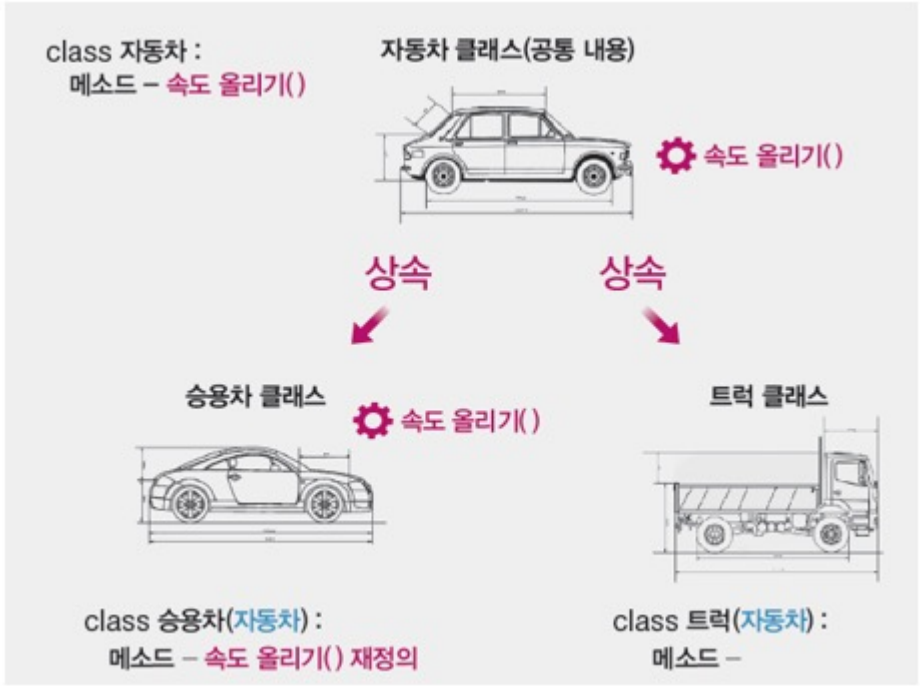
```
# 클래스 선언
class  Car :
    speed=0

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

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

class Sedan(Car) :
    seatNum=0

    def getSeatNum(self) :
        return self.seatNum

```

Inheritance

```
17 class Truck(Car) :
18     capacity=0
19 
20     def getCapacity(self) :
21         return self.capacity
22 
23 # 변수 선언
24 sedan1, truck1=None, None
25 
26 # 메인 코드 부분
27 sedan1=Sedan()
28 truck1=Truck()
29 
30 sedan1.upSpeed(100)
31 truck1.upSpeed(80)
32 
33 sedan1.seatNum=5
34 truck1.capacity=50
35 
36 print("승용차의 속도는  %d km, 좌석수는 %d개입니다." % (sedan1.speed,
   sedan1.getSeatNum() ))
37 print("트럭의 속도는  %d km, 총중량은 %d톤입니다." % (truck1.speed,
   truck1.getCapacity() ))
```

Method Overriding

- 슈퍼 Class의 메소드를 서브 Class에서 재정의하는 것. 다음의 그림에서 트럭은 속도에 제한이 없지만, 승용차는 안전상 속도가 최대 150km로 제한되어야 한다고 가정함.
- 슈퍼 Class(자동차)를 상속받은 서브 Class(승용차, 트럭)는 속도 올리기( ) 메소드를 상속 받았지만, 승용차의 경우 속도의 제한이 필요해서 자동차의 속도 올리기( )와 내용이 달라야 하므로 승용차 Class에서 속도 올리기( )를 다시 만들어서 사용함.



Method Overriding

```
1  ## 클래스 선언
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
16         print("현재 속도(서브 클래스) : %d" % self.speed)
17
18 class Truck(Car) :
19     pass
20
```

Method Overriding

```
21  # 변수 선언
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)
```

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

Operator Overloading

- 사용자 정의 객체에서 필요한 연산자를 내장 타입과 형태와 동작이 유사하도록 재정의
- Operator Overloading을 위하여 두 개의 밑줄 문자가 앞뒤로 있는 메소드를 미리 정의함
- 예제 ㅊ

```
>>> class NumBox:
    def __init__(self, num):
        self.Num = num

>>> n = NumBox(40)
>>> n + 100
Traceback (most recent call last):
  File "<pyshell#5>", line 1, in <module>
    n + 100
TypeError: unsupported operand type(s) for +: 'NumBox' and 'int'
```

Operator Overloading

- 수치 연산자

메소드	연산자	사용 예
__add__(self, other)	+ (이항)	A + B, A += B
__sub__(self, other)	- (이항)	A - B, A -= B
__mul__(self, other)	*	A * B, A *= B
__truediv__(self, other)	/	A / B, A /= B (3 이상 지원, 그 이하는 버전에서는 __div__가 사용)
__floordiv__(self, other)	//	A // B, A //= B
__mod__(self, other)	%	A % B, A %= B
__divmod__(self, other)	divmod()	divmod(A, B)
__pow__(self, other[, modulo])	pow(), **	pow(A, B), A ** B
__lshift__(self, other	<<	A << B, A <<= B

Operator Overloading

```
>>> class NumBox:
    def __init__(self, num):
        self.Num = num
    def __add__(self, num):
        self.Num += num
    def __sub__(self, num):
        self.Num -= num

>>> n = NumBox(40)
>>> n + 100
>>> n.Num
140
>>> n - 110
>>> n.Num
30
```

n + 100 →

```
n.__add__(100)
```

```
>>> 110 + n
Traceback (most recent call last):
  File "<pyshell#20>", line 1, in <module>
    110 + n
TypeError: unsupported operand type(s) for +: 'int' and 'NumBox'
```

Operator Overloading

100 + n : 피연산자의 순서가 바뀐 경우 r 을 붙인 __radd__ 메소드를 정의

```
__add__ = __radd__
__sub__ = __rsub__
__mul__ = __rmul__
```

```
>>> class NumBox:
    def __init__(self, num):
        self.Num = num
    def __add__(self, num):
        self.Num += num
    def __radd__(self, num):
        self.Num += num

>>> n = NumBox(100)
>>> 120 + n
>>> n.Num
220
>>> 300 + n
>>> n.Num
520
```

Polymorphism

- 다형성(Polymorphism)
  - inheritance 관계 내의 다른 Class들의 instance들이 같은 멤버 함수 호출(같은 방법)에 대해 각각 다르게 반응하도록 하는 기능
    - Method Overriding, Operator Overloading도 polymorphism을 지원하는 중요한 기술
  - 장점
    - 적은 코딩으로 다양한 객체들에게 유사한 작업을 수행 가능
    - 작성 코드량이 줄어들고 코드의 가독성을 높여줌
  - 파이썬에서 polymorphism의 장점
    - 형 선언이 없다는 점에서 파이썬에서는 polymorphism을 적용하기가 더욱 용이함
    - 실시간으로 객체의 형이 결정되므로 단 하나의 메소드에 의해 처리될 수 있는 객체의 종류에 제한이 없음

상속과 다형성 예시

```
class Animal:
    def __init__(self, name):
        self.name = name
    def speak(self):
        return '알 수 없음'

class Dog(Animal):
    def speak(self):
        return '멍멍!'

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

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

for a in animalList:
    print (a.name + ': ' + a.speak())
```

```
dog1: 멍멍!
dog2: 멍멍!
cat1: 야옹!
```

Appendix

Appendix

- 파이썬에서 모든 데이터는 객체(Object)라는 개념을 사용하여 저장
- 기본 데이터 타입 : 숫자, 문자열, 리스트, 사전 등도 객체
  - 객체는 신원(identity), 타입(type(class)), 값을 가짐
  - a = 10은 10 이라는 값을 갖는 정수 객체 생성
  - 객체의 id는 객체가 메모리에 저장된 위치를 가리키는 포인
  - a는 그 위치를 가리키는 이름
- Class는 새로운 객체를 생성하는데 사용되는 메커
  - Class를 사용하여 사용자 정의 객체를 생성

```
a = 10
b = 10.5
c = 'A'
d = "Hello World!"
print(id(a))
print(id(b))
print(id(c))
print(id(d))
print(type(a))
print(type(b))
print(type(c))
print(type(d))
```

```
140714042454048
2042341725576
2042302229784
2042342148592
<class 'int'>
<class 'float'>
<class 'str'>
<class 'str'>
```

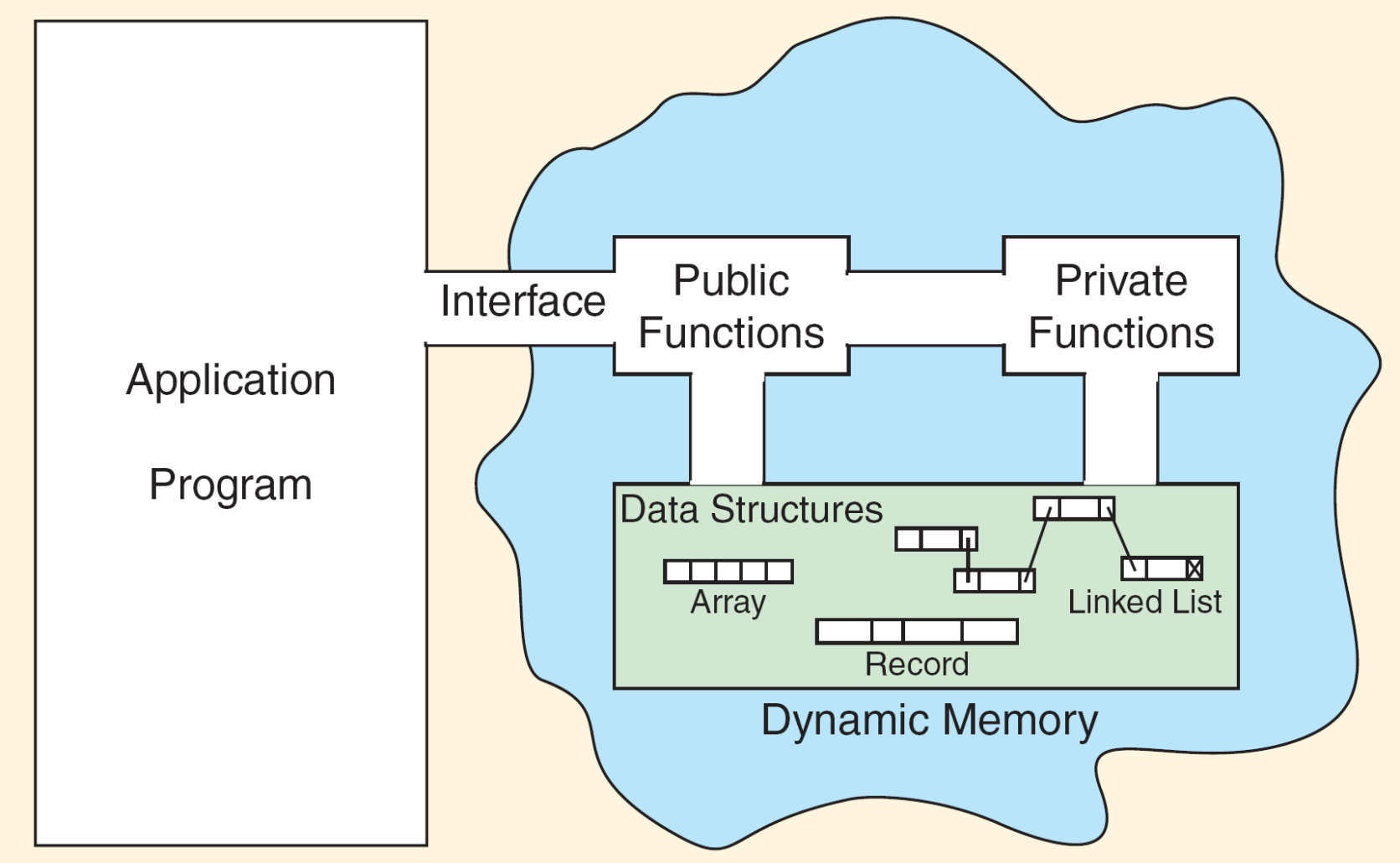
추상 데이터 타입(Abstract Data Type: ADT)

- 추상(abstract)
  - 명세(specification), 이름만 있는, 껍데기만 있는, 내용은 없는, 정의만 해놓은, 틀만 짜놓은, 구현은 안해 놓은, 요구사항...
  - 추상적이라는 것은 기능(function)들만 기술해 놓고, 그것의 내부적 표현이나 구현은 안 해 놓은 것을 의미

추상 데이터 타입(Abstract Data Type: ADT)

- ADT의 필요성
  - '이 데이터 타입은 이러이러한 객체들(objects)과 이러이러한 함수들(functions)로 구성되어 있다‘ 라는 것을 명시해 놓은 것이고 '사용설명서' 같은 것이기 때문에 외부에 공개적(public)
  - 구현 부분을 나타내지 않기 때문에 내부는 비공개적(private)
  - 사용자는 사용법만 알면, 내부는 몰라도 그 데이터 타입을 쉽게 쓸 수 있음
  - ADT를 통해 데이터나 연산이 무엇(what)인지는 알 수 있음
  - ADT를 통해 데이터나 연산이 어떻게(how) 동작하는지는 알 수 없음

추상 데이터 타입(Abstract Data Type: ADT)



[Abstract Data Type Model]

추상 데이터 타입(Abstract Data Type: ADT)

- ADT 개념은 데이터구조, 알고리즘 등에 보편적으로 사용되며 OOP 개념에 영향
- 캡슐화(Encapsulation)
  - 객체지향프로그래밍에서의 중요한 특징 중 하나로, 연관된 데이터와 함수를 논리적으로 묶어놓은 것이며, 데이터를 보호하기 위해 다른 객체의 접근을 제한하는 접근 제한 수식자(private, public)의 기능을 제공 : 정보은닉(information hiding)

- 객체지향언어는 ADT 개념을 Class 명령어를 사용하여 구현
  - Python에서 Class는 데이터와 함수를 묶어서 선언할 수 있도록 만들어진 사용자 정의 자료형(User defined data type)이라 할 수 있음
  - Python은 객체지향 캡슐화 개념을 도입하여 class 명령어를 사용하나 기본적으로 데이터

47