객체 지향 프로그래밍

객체 지향 프로그래밍

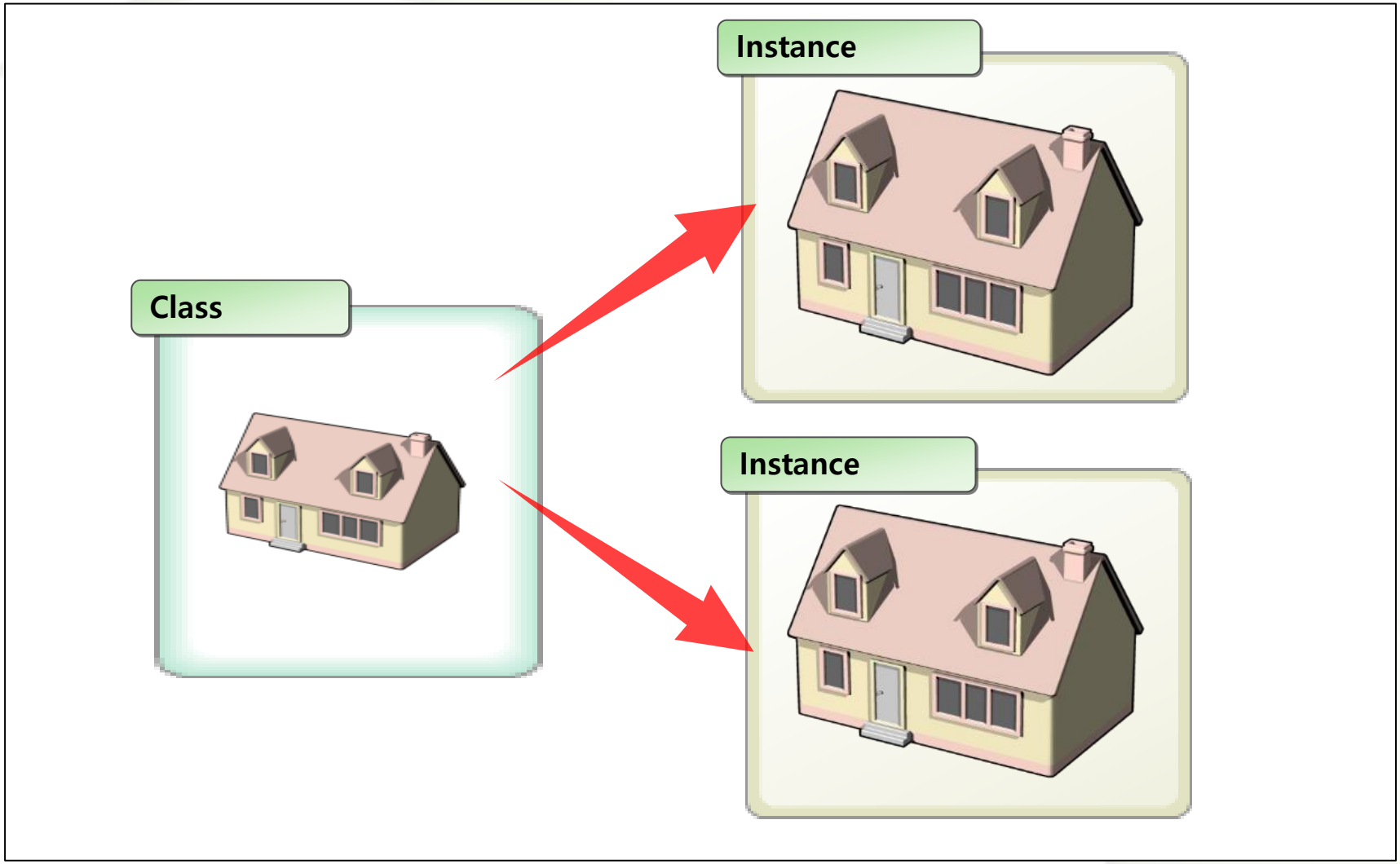
- 추상화 (Abstraction)
  - » 대상 세계의 처리 대상(객체)을 프로그래밍 영역의 표현 단위인 클래스 등으로 변환하는 과정
  - » 중요한 것과 중요하지 않은 것을 구분하고 선택적으로 재구성

- 객체, 클래스, 인스턴스
  - » 객체는 프로그램으로 다루고자 하는 모든 대상
  - » 클래스는 제어 대상을 프로그래밍 수준에서 정의한 사용자 정의 자료형
  - » 인스턴스는 클래스를 기반으로 메모리상에 생성된 데이터
  - » 하나의 클래스를 기반으로 여러 인스턴스를 생성하고 각 인스턴스는 서로 구분되는 독립적인 단위

- 파이썬은 객체 지향 프로그래밍을 지원하며 쉽게 클래스를 만들고 사용할 수 있도록 지원

객체 지향 프로그래밍

- 클래스, 인스턴스



클래스 만들기

- 클래스를 정의하기 위해 class 구문 사용

- 형식

```
class ClassName:
   'Optional class documentation string'
   class_suite
```

  - » class_suite는 멤버 변수, 메서드 등 클래스 내부에 포함되는 모든 요소

클래스 만들기

- 클래스 만들기 예제

```
class Employee:
   'Common base class for all employees’

   empCount = 0

   def __init__(self, name, salary):
      self.name = name
      self.salary = salary
      Employee.empCount += 1

   def displayCount(self):
     print("Total Employee %d" % Employee.empCount)

   def displayEmployee(self):
      print("Name : ", self.name,  ", Salary: ", self.salary)
```

  - empCount는 모든 인스턴스에 의해 공유되는 클래스 멤버
  - __init__ 함수는 생성자 또는 초기화 함수로 불리는 특별한 함수 (인스턴스 생성시 자동 호출)
  - 메서드의 첫 번째 전달인자인 self는 객체 자신을 참조하는 특별한 변수

클래스 사용

- 인스턴스 만들기
  - » 정의된 클래스를 기반으로 메모리에 공간을 할당

```
# This would create first object of Employee class
emp1 = Employee("Zara", 2000)
# This would create second object of Employee class
emp2 = Employee("Manni", 5000)
```

- 멤버 접근
  - » 객체의 멤버는 .(dot) 연산자를 사용해서 접근

```
emp1.displayEmployee()
emp2.displayEmployee()
print ("Total Employee %d" % Employee.empCount)
```

```
Name :  Zara ,Salary:  2000
Name :  Manni ,Salary:  5000
Total Employee 2
```

클래스 사용

- 객체 변경
  - » 객체 생성 후에도 멤버 추가 및 삭제 가능

```
emp1.salary = 7000  # Add an 'salary' attribute.
emp1.name = 'xyz'  # Modify 'age' attribute.
del emp1.salary  # Delete 'age' attribute.
```

생성자, 소멸자

- 생성자 메서드
  - » 인스턴스 생성 시점에 자동으로 호출되는 인스턴스 초기화 메서드
  - » 이름은 __init__

- 소멸자 메서드
  - » 인스턴스에 대한 참조 카운트가 0이 될 때 자동으로 호출

```
class MyClass:
        def __init__(self, value):
                self.Value = value
                print("Class is created! Value = ", value)
        def __del__(self):
                print("Class is deleted!")

def foo():
        d = MyClass(10)

foo()
```

클래스 메서드와 정적 메서드

- 인스턴스를 만들지 않고 클래스를 통해 사용할 수 있는 메서드
- 인스턴스의 메서드가 아니기 때문에 첫 번째 전달인자로 self를 사용할 필요 없음
- 클래스 메서드는 첫 번째 전달인자로 클래스 객체를 수신

```
class Test:
    @classmethod
    def class_method(cls):
        print('class method')

    @staticmethod
    def static_method():
        print('static method')

    def method(self):
        print("method")
```

상속

- 이미 만들어진 클래스의 내용을 재사용해서 새로운 클래스를 만드는 기법
  - » 클래스 수준의 재사용 원리

- 상속 받은 클래스는 부모 클래스의 멤버를 자동으로 포함하게 됨
  - » 부모 클래스의 멤버를 재정의 할 수 있음

- 형식

```
class SubClassName (ParentClass1[, ParentClass2, ...]):
   'Optional class documentation string'
   class_suite
```

상속

- 상속 예제

```
class Parent:        # define parent class
   parentAttr = 100
   def __init__(self):
      print ("Calling parent constructor")

   def parentMethod(self):
      print ('Calling parent method')

   def setAttr(self, attr):
      Parent.parentAttr = attr

   def getAttr(self):
      print ("Parent attribute :", Parent.parentAttr)

class Child(Parent): # define child class
   def __init__(self):
      print ("Calling child constructor")

   def childMethod(self):
      print ('Calling child method')
```

상속

- 상속 예제 (계속)

```
c = Child()          # instance of child
c.childMethod()      # child calls its method
c.parentMethod()     # calls parent's method
c.setAttr(200)       # again call parent's method
c.getAttr()          # again call parent's method
```

```
Calling child constructor
Calling child method
Calling parent method
Parent attribute : 200
```

클래스 간의 관계 확인

- issubclass(자식클래스, 부모클래스) → 상속 관계 확인

```
issubclass(Child, Parent) # True
issubclass(Parent, Child) # False
issubclass(Parent, Parent) # True
issubclass(Parent, object) # True
issubclass(Child, object) # True
```

- 부모 클래스 확인

```
Parent.__bases__
Child.__bases__
```

부모 클래스 생성자 호출

- 부모클래스.__init__(…) 형식으로 부모 생성자 호출

```
class Person:
  " 부모 클래스 "

  def __init__(self, name, phoneNumber):
    self.Name = name
    self.PhoneNumber = phoneNumber

  def PrintInfo(self):
    print("Info(Name:{0}, Phone Number: {1})".format(self.Name, self.PhoneNumber))

  def PrintPersonData(self):
    print("Person(Name:{0}, Phone Number: {1})".format(self.Name, self.PhoneNumber))

class Student(Person):
  " 자식 클래스 "

  def __init__(self, name, phoneNumber, subject, studentID):
    # self.Name = name
    # self.PhoneNumber = phoneNumber
    Person.__init__(self, name, phoneNumber)
    self.Subject = subject
    self.StudentID = studentID
```

자식 클래스에 메서드 추가

- 자식클래스에 필요한 메서드 정의

```
class Person:
  " 부모 클래스 "

  def __init__(self, name, phoneNumber):
    self.Name = name
    self.PhoneNumber = phoneNumber

  def PrintInfo(self):
    print("Info(Name:{0}, Phone Number: {1})".format(self.Name, self.PhoneNumber))

  def PrintPersonData(self):
    print("Person(Name:{0}, Phone Number: {1})".format(self.Name, self.PhoneNumber))

class Student(Person):
  " 자식 클래스 "

  def __init__(self, name, phoneNumber, subject, studentID):
    # self.Name = name
    # self.PhoneNumber = phoneNumber
    Person.__init__(self, name, phoneNumber)
    self.Subject = subject
    self.StudentID = studentID

  def PrintStudentData(self):
    print("Student(Subject: {0}, Student ID: {1})".format(self.Subject, self.StudentID))
```

메서드 재정의

- 자식클래스에 부모클래스의 메서드와 동일한 이름의 메서드 정의

```
class Person:
  " 부모 클래스 "

  …

  def PrintInfo(self):
    print("Info(Name:{0}, Phone Number: {1})".format(self.Name, self.PhoneNumber))

  …

class Student(Person):
  " 자식 클래스 "

  …

  def PrintInfo(self):
    print("Info(Name:{0}, Phone Number:{1})".format(self.Name, self.PhoneNumber))
    print("Info(Subject:{0}, Student ID:{1})".format(self.Subject, self.StudentID))
```

부모 클래스의 메서드 확장

- 재정의 메서드에서 공통 코드는 부모 클래스의 메서드 호출

```
class Person:
  " 부모 클래스 "

  …

  def PrintInfo(self):
    print("Info(Name:{0}, Phone Number: {1})".format(self.Name, self.PhoneNumber))

  …

class Student(Person):
  " 자식 클래스 "

  …

  def PrintInfo(self):
    Person.PrintPersonData(self)
    print("Info(Subject:{0}, Student ID:{1})".format(self.Subject, self.StudentID))
```

다중 상속

- 두 개 이상의 부모 클래스로부터 상속

```
class Tiger:
  def Jump(self):
    print("호랑이처럼 멀리 점프하기")

class Lion:
  def Bite(self):
    print("사자처럼 한입에 꿀꺽하기")

class Liger(Tiger, Lion):
  def Play(self):
    print("라이거만의 사육사와 재미있게 놀기")
```

다중 상속 시 메서드 이름 검색

- 두 개 이상의 부모 클래스가 같은 이름의 메서드를 가진 경우 자식 클래스에서 어느 부모 클래스의 메서드를 호출하는지 기준 → 상속 표현 순서에 따라 결정

```
class Tiger:
  def Jump(self):
    print("호랑이처럼 멀리 점프하기")
  def Cry(self):
    print("호랑이: 어흥~")

class Lion:
  def Bite(self):
    print("사자처럼 한입에 꿀꺽하기")
  def Cry(self):
    print("사자: 으르렁~")

class Liger(Tiger, Lion):
  def Play(self):
    print("라이거만의 사육사와 재미있게 놀기")
```

상위 클래스의 생성자 메서드 호출

- super() 메서드를 사용해서 부모 클래스의 생성자 메서드 호출 가능

```
class Animal:
  def __init__(self):
    print("Animal __init__()")

class Tiger(Animal):
  def __init__(self):
    super().__init__()
    print("Tiger __init__()")

class Lion(Animal):
  def __init__(self):
    super().__init__()
    print("Lion __init__()")

class Liger(Tiger, Lion):
  def __init__(self):
    super().__init__()
    print("Liger __init__()")
```