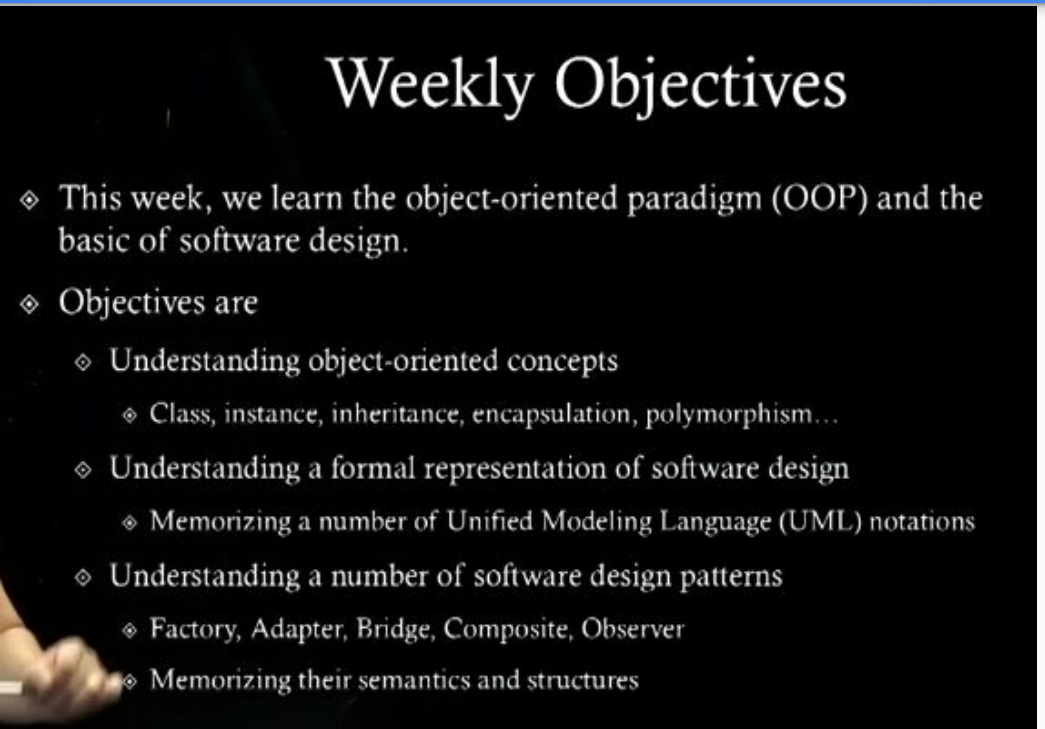
2장. 데이터 분석 및 알고리즘

Object-oriented paradigm and Software design

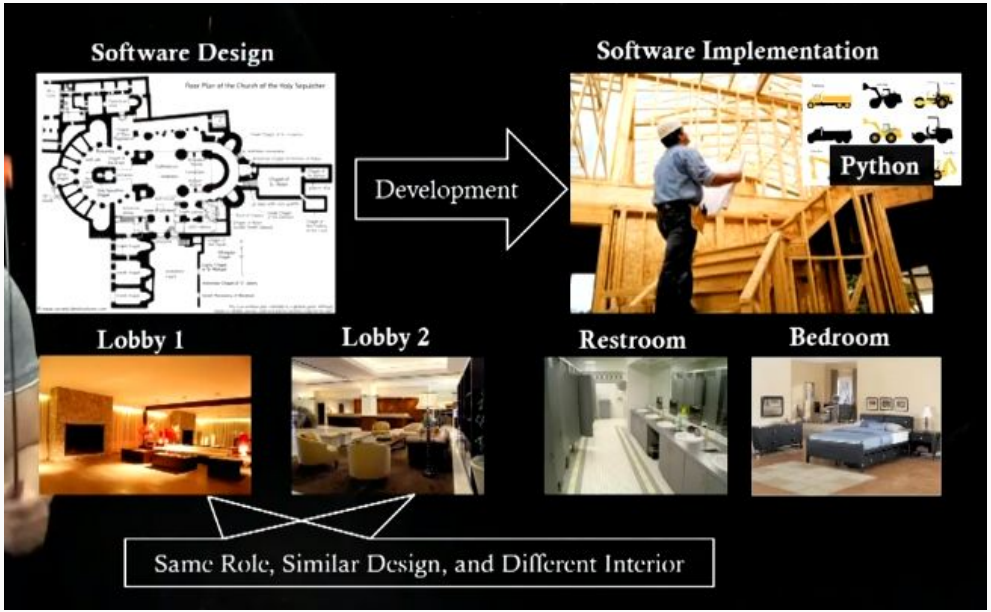


object-oriented paradigm(OOP), basic of software design

목표는

1. 설계의 기본 컨셉을 배우는 것
2. 설계도를 그리는 방법, 읽는 방법에 대해서 배우는 것
3. 사람들이 흔히 쓰는 패턴이 있는데 이것은 참고 자료로

Design and Programming



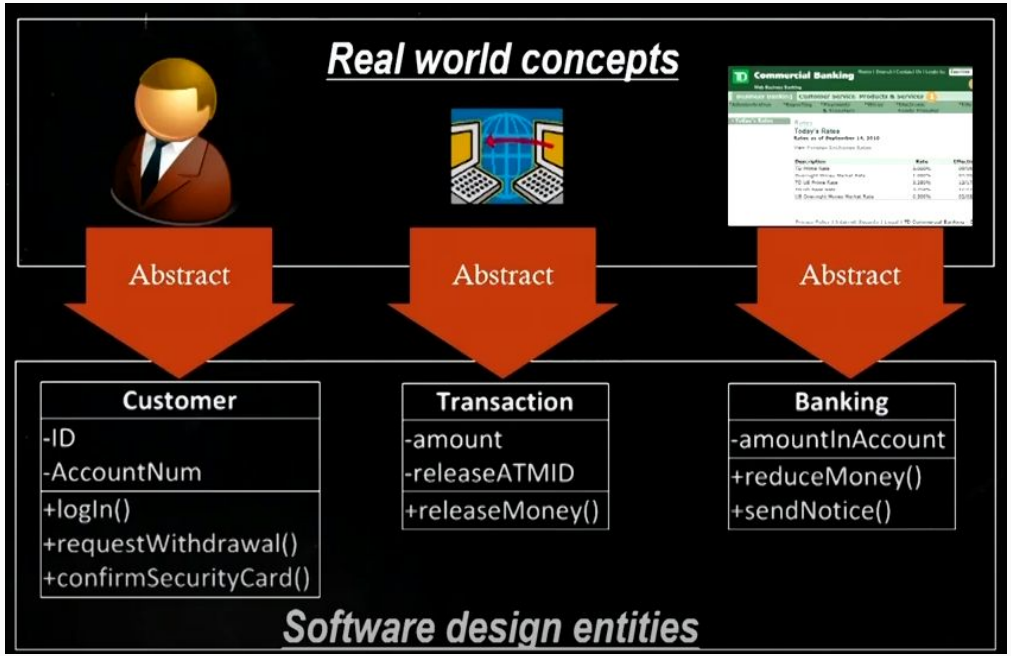
- 유사한 역할을 하는 로비1, 로비2는 비슷한 역할, 비슷한 디자인을 해야하고 하지만 둘은 실질적으로 다르다
- 그러면 로비1과 로비2는 완전 다르게 봐야하는가? 아니다. 설계를하기전 둘의 역할과 디자인은 비슷하지만 색상, 구체적인 기능은 다르다고 명시할 수 있다.

Good Software Design

	Building Design	Software Design
Correctness	• Meet the owner's purpose • Successful construction without faults	• Meet the client's purposes • Successful implementation without errors
Robustness	• Maintain integrity in a certain level of typhoons	• Execute under expected overloads
Flexibility	• Enable the future expansions and modifications of the structure	• Enable the future updates and expansions of functions
Usability and Reusability	• Good support for designed purposes • Easy to use for 1) other purposes and 2) other areas	• Good support for the designed • Easy to use for 1) other purposes and 2) other contexts
Efficiency	• Easy to build • Cover less area • Good mobility in the structure	• Easy to implement • Smaller size • Faster execution

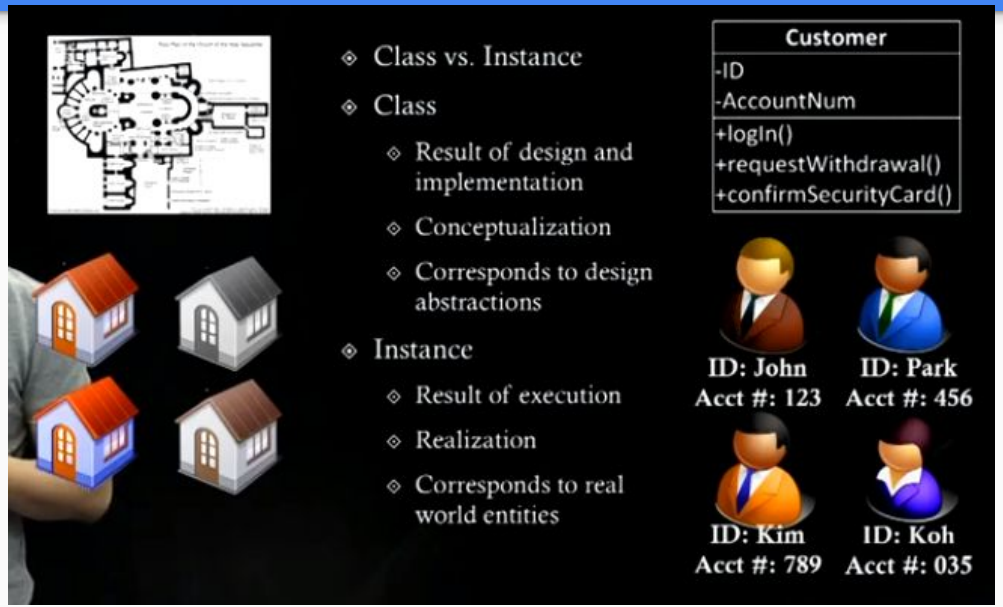
- 고객의 목적에 맞는 소프트웨어 디자인이어야한다
- 설계된 소프트웨어는 에러가 나지 않고 실행 되어야 한다.
- 과중한 입력이나 생각치 못했던 입력이 있을 경우에도 에러가 없이 실행 되어야 한다.
- 시간이 지나고 미래에 업데이트나 수정이 용이해야한다.
- 설계된 디잔인에 대한 좋은 지원이 있어야 한다.
- 고객이 다른 목적으로 사용하고 싶을때 용이하게 사용할 수 있어야 한다.
- 사용자가 실행 했을 때 빨라야하고 사이즈가 작아야 한다.
- 개발자 입장에서는 소프트웨어 설계가 복잡하면 구현하는데 시간이 많이 들고 비용도 많이 드니 복잡하게 하면 안된다.

Object-Oriented design



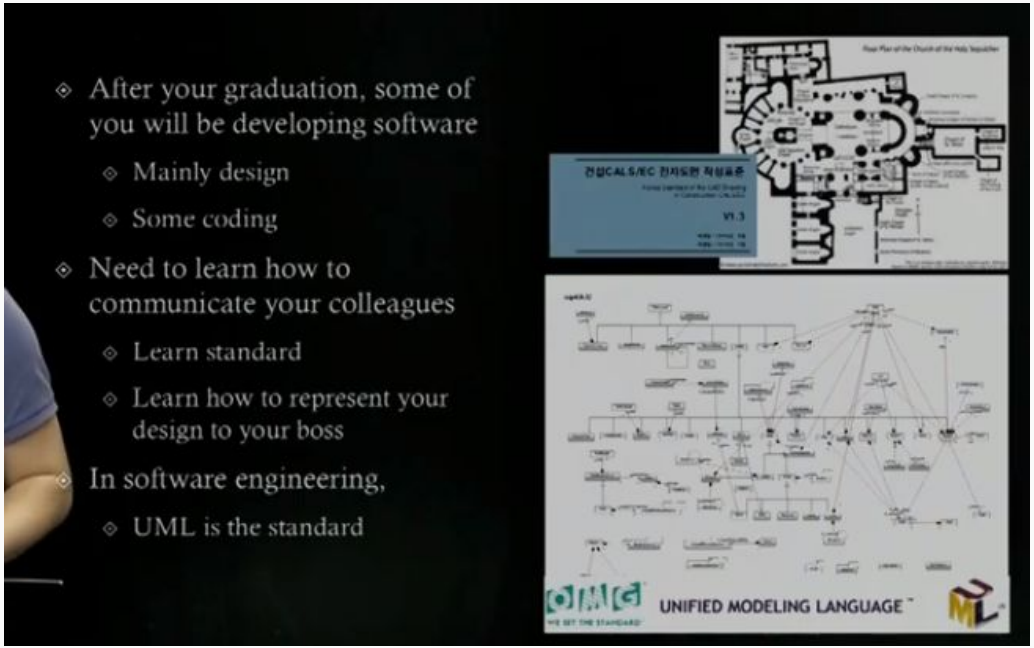
- object-oriented design은 real world concepts을 abstract를 통해서 software design entities 전환하는 것이다.
- 현실에 존재하는 개념을 어떻게 프로그래밍을 할까?
- 먼저 고객의 경우 개념의 이름을 customer를 부여한다.
- 다음 이 개념이 특성을 가지고 있어야 하는 개념 특성을 부여한다.
- 마지막으로 고객이 할 수 있는 행동을 부여한다.
- 고객의 이름, 고객의 특성, 고객의 행동은 사람이 주체라 가능하다
- 또한 거래도 추상적으로 세가지로 나누어 이름과 특성, 행동을 부여할 수 있다.
- 추상화는 목적에 맞게 간략화 하는 것이다.

What are Class and Instance?



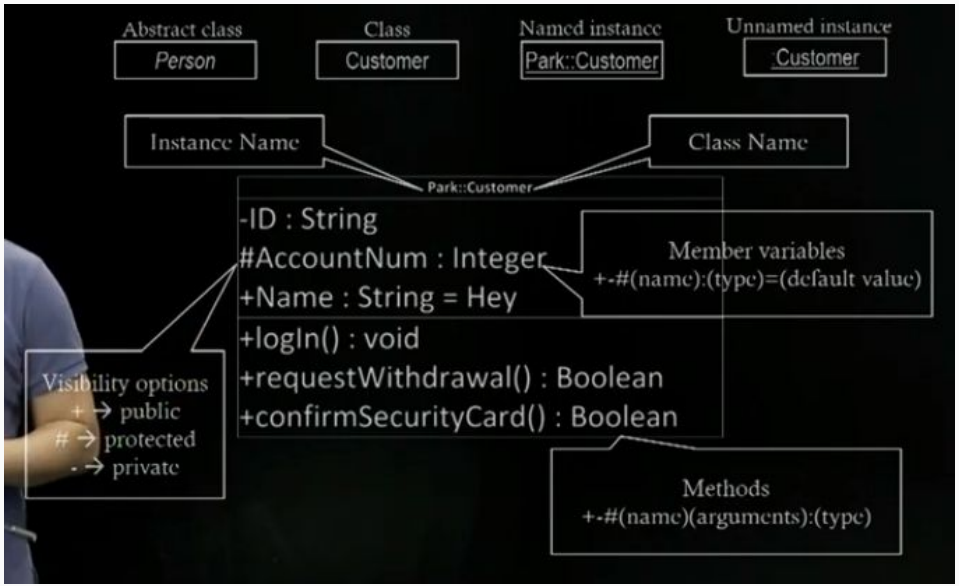
- 고객은 여러명인데 어떻게 하나의 entities로 표현할 수 있을까?
- 사실 entities가 아니라 여러개의 instance를 통해서 여러고객을 프로그래밍을 할 수 있다.
- class는 설계된 것이고 어떻게 컨셉화하는 것이고 어떻게 추상화하는것이다
- instance 설계된 메소드를 활용하는 것이다. 동일하게 설계된 여러개의 instance를 설정할 수 있다.

Software Design as House Floorplan



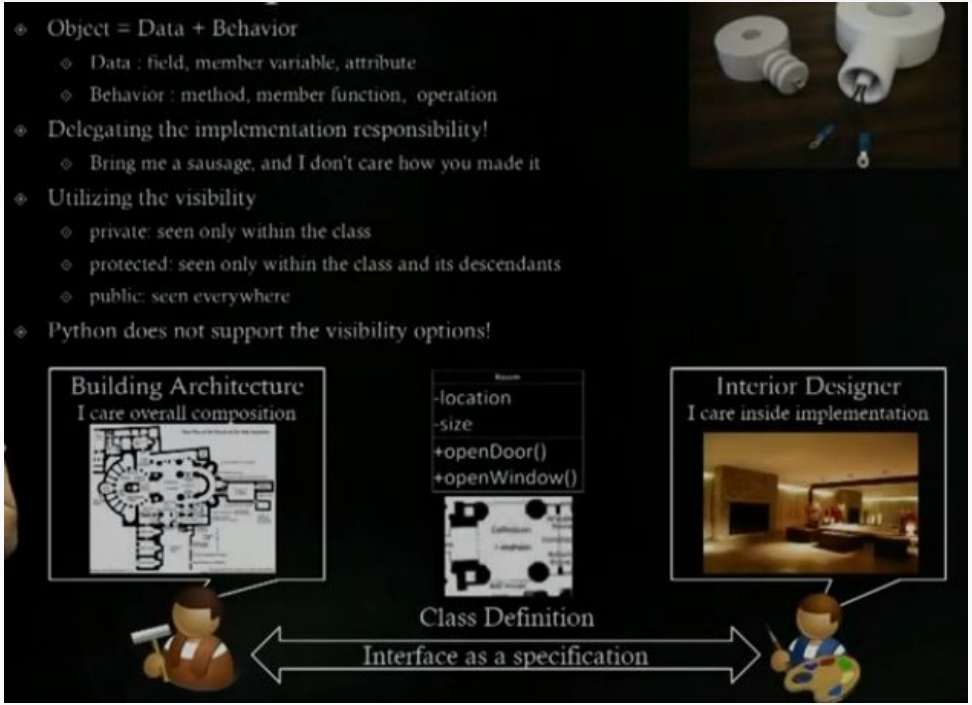
- 소프트웨어 설계는 평면도를 그리는것과 같은데 평면도를 그리는데에는 표준이라는게 있다.
- 소프트웨어 설계도 표준이라는게 있다.
- 소프트웨서 설계는 향후 다른 사람들과 협력도 해야하고 일도 같이 해야 하기 때문에 표준을 지키는것은 중요하다.
- UML은 소프트웨어 설계 표준이다.

UML notation: Class and Instance



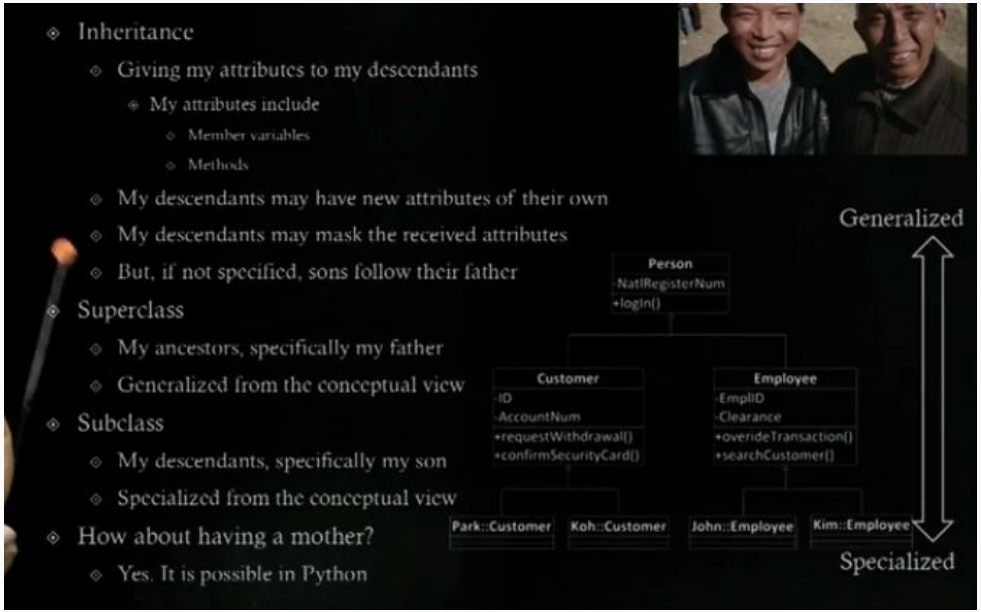
- 설계도가 다양한 설계가 있듯이 소프트웨어도 다양한 설계도 위엔 다양한 설계가 있다.
- 그중에서 가장 많이 사용하는 설계는 class를 설계하는 것이다.
- 그 중에 abstract class, class, Named instance, unnamed instance가 있다.
- instance name은 park이라는 네임에 class name을 붙힌다.
- instance의 네임이 없으면 :: 두개를 붙혀주면 된다.
- attribute, property, member variables 같은말
- methods는 member function이라고도 부른다.
- visibility opions는 퍼블릭은 다 볼수 있고 protected는 class간에 관계되어 있는 것만 볼 수 있고 private은 이 class에서만 볼수 있게 옵션으로 설정할 수 있다.
- 수업에서는 public으로 많이 사용할테지만 현실에는 private으로 많이 사용한다.
- name은 변수의 이름
- 밑에 name은 method의 name - 리턴을 정의해야한다

Encapsulation



- encapsulation은 클레스내부에 모든내용이 쌓여 있어야 하고 외부에서는 메서드를 통해서 접근해야 한다.
- object은 behavior를 통해서 data에 접근하는 것
- 그림에 하얀색 속에 어떻게 되어 있는지 모르지만 두 선을 통해서 속에 있는 것을 컨트롤 해야 한다.
- 속에 있는 data는 집적 건들지 못한다.
- data변형은 오르지 내가 한는 것이다.
- 구현에 대한 책임을 지우는 것이다.
- visibility 옵션으로 privata, protected, public
- private 온니 i can see, public mani can see
- 파이선은 visibility option을 제공하지는 않는다.
- 하지만 __var1 이런식으로 다른사람이 안썼으면 좋게다라는 메세지를 보낼수 있다.
- 구현한 사람들끼리 약속이다.
- 외부에서 볼수 있지만 보는것은 현명하지 않다.

Inheritance



- inheritance는
- 내 특성(attibute)을 자손(descendant)에게 물려주는 것이다.
- 내 자손은 새로운 특성을 가질수 있다.
- 내 자손은 물려받은 값을 mask하고 자기 자신의 특성으로 바꿀수 있다.
- 위의 두 가지 경우가 아니면 부모의 특성을 그대로 사용한다.
- 두개의 역할이 생겨나게 되는데 아버지나 조상역할을 하는 superclass역할을 하는것이다.
- 상위에는 새로운것이 없고 하위로 갈수록 새로운게 많아 지기 때문에 superclass는 generalize이고 하위로 갈수로 specialize가 된다.
- 자바는 하나의 class에서는 하나의 class에서만 상속이 되지만 파이썬은 여러 class에서 상속 받을 수 있다.
- person의 로그인 기능은 하위의 customer, employee에 표시는 안되어 있지만 로그닝 펑션이 있다.
- 그 외에는 새로 생긴 특성이다.

Inheritancs in Python

```
class Father(object):
    strHometown = 'Jeju'
    def __init__(self):
        print('Father is created')
    def doFatherThing(self):
        print("Father's action")
    def doRunning(self):
        print('Slow')

class Mother(object):
    strHometown = 'Seoul'
    def __Init__(self):
        print('Mother is created')
    def doMotherThing(self):
        print("Mother's action")

class Child(Father, Mother):
    StrName = 'Moon'
    def __init__(self):
        super(Child, self).__init__()
        print('Child is created')
    def doRunning(self):
        print('Fast')
```

```
me = Child()
me.doFatherThing()
me.doMotherThing()
me.doRunning()
print(me.StrName)
print(me.strHometown)
```

- object는 파이썬의 가장 상위에 있는 것이다.
- me = Child() 하게 되면 부모의 컨스트럭션은 자동적으로 실행하게 된다.
- me.doFatherThing()은 자식에 없지만 부모의 특성을 물려받아서 파덜스 액션이 실행이 된다.
- me.doMotherThing()은 자식에 없지만 부모의 특성을 물려받아서 마덜스 액션이 실행이 된다.
- doRunning()을 하게 되면 파더에도 있고 자식에도 있지만 자식의 특성을 실행하게 된다.
- me.StrName Moon을 출력하고 me.strHometown은 jeju를 출력한다.
- self는 자기 자신의 인스턴스를 말한다.

Polymorphism

poly = many

morph = shape

-> 다양한 모양이다

Different behaviors with similar signature

다른 행동이 일어난다

signiture = Method name + Parameter list

Polymorphism

```
class Building:
    strAddress = "Daejeon"
    def opendoor(self):
        print('Door Opened!')
```

```
class Hotel:
    def openDoor(self):
        print('Bellboy opens a
door')
    def checkIn(self):
        print('Someone checks in
for 1 day')
    def checkIn(self, days):
        print('Someone checks in
for', days, 'days')
```

```
motel = Building()
motel.strAddress
motel.opendoor()
```

```
motel.strAddress
Out[55]: 'Daejeon'
motel.opendoor()
Door Opened!
```

```
lotteHotel = Hotel()
lotteHotel.openDoor()
lotteHotel.checkIn()
lotteHotel.checkIn(3)
```

```
lotteHotel.openDoor()
Bellboy opens a door
```

```
lotteHotel.checkIn(3)
Someone checks in for 3 days
```

```
Traceback (most recent call last):
  File "C:\Users\hclee\anaconda3\lib\site-packages\IPython\core\int
    exec(code_obj, self.user_global_ns, self.user_ns)
  File "<ipython-input-61-7981db00e157>", line 1, in <module>
    lotteHotel.checkIn()
TypeError: checkIn() missing 1 required positional argument: 'days'
```

Polymorphism

```
class Building:
    strAddress = 'Daejeon'
    def openDoor(self):
        print('Door Opened')
```

```
class Hotel:
    def openDoor(self):
        print('Bellboy opens a
door')
    def checkIn(self, days = 1):
        print('Someone checks in
for', days, 'days')
```

```
lotteHotel = Hotel()
lotteHotel.openDoor()
lotteHotel.checkIn()
lotteHotel.checkIn(5)
```

```
lotteHotel.openDoor()
Bellboy opens a door
lotteHotel.checkIn()
Someone checks in for 1 days
lotteHotel.checkIn(5)
Someone checks in for 5 days
```

checkIn에 days의 값이 들어오지 않아도 자동적으로 days=1로 저장되어 있다.

Abstract Class

class인데 abstract method가 있는 class

method가 signiture만 정의 되어 있는 것

- 여러명이 작업을 할 때 유용함

object에 몇 가지 숨겨진 method가 있음

__init__ construct - 어떤 인스트럭스를 만들때마다 기존적인 이니셜라이즈를 셋팅

__del__ destructure - __init__과 같은 역할

__eq__ 값을 비교할 때

__cmp__ 값이 어느쪽이 큰지

__add__ 덧샘을 할 때 적용

Abstract Class

```
import abc
class Room(object):
    metaclass   = abc.ABCMeta
    @abc.abstractmethod
    def openDoor(self):
        pass
    @abc.abstractmethod
    def openWindow(self):
        pass
class BedRoom(Room):
    def openDoor(self):
        print('Open bedroom
door')
    def openWindow(self):
        print('Open bedroom
window')
class Lobby(Room):
    def OpenDoor(self):
        print('Open lobby door')
```

```
room1 = BedRoom()
print(issubclass(BedRoom, Room),
isinstance(room1, Room))
```

```
True True
```

```
lobby1 = Lobby()
print(issubclass(Lobby, Room),
isinstance(lobby1, Room))
```

```
True True
```

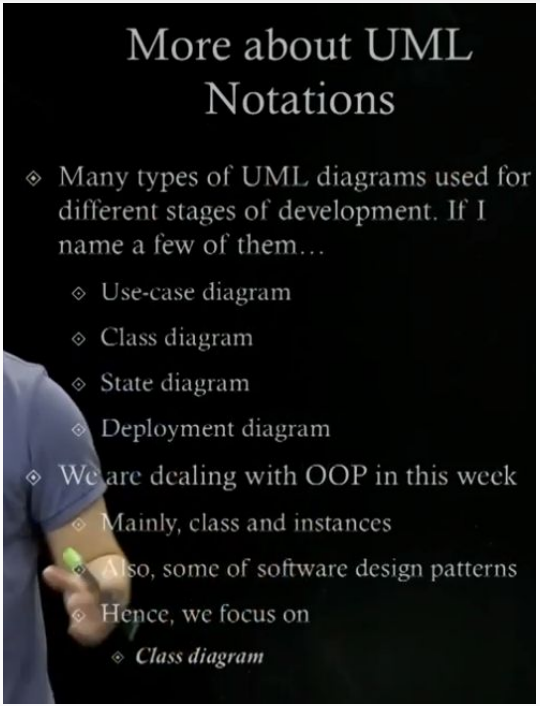
Abstract Class

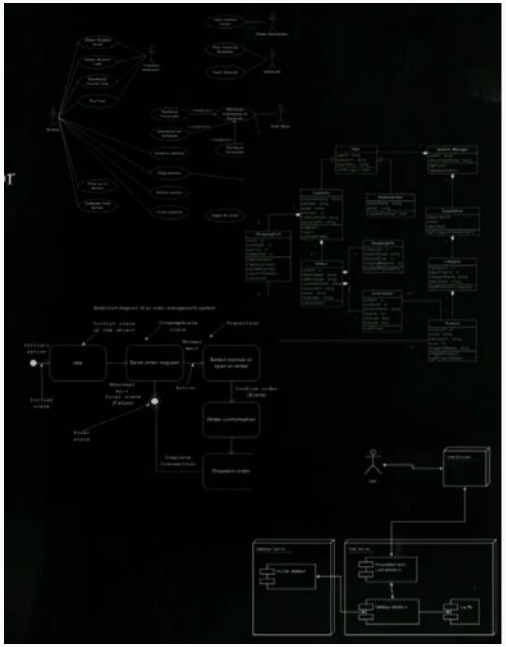
```
class Room:
    numWidth = 100
    numHeight = 100
    numDepth = 100
    def __init__(self, parWidth,
parHeight, parDepth):
        self.numDepth = parDepth
        self.numWidth = parWidth
        self.numDepth = parDepth
    def getVolume(self):
        return
self.numDepth*self.numWidth*self
.numHeight
    def __eq__(self, other):
        if isinstance(other,
Room):
            if self.getVolume()
== other.getVolume():
                return True
        return False
```

```
room1 = Room(100, 20, 30)
room2 = Room(100, 10, 60)
print(room1 == room2)
```

```
False
```

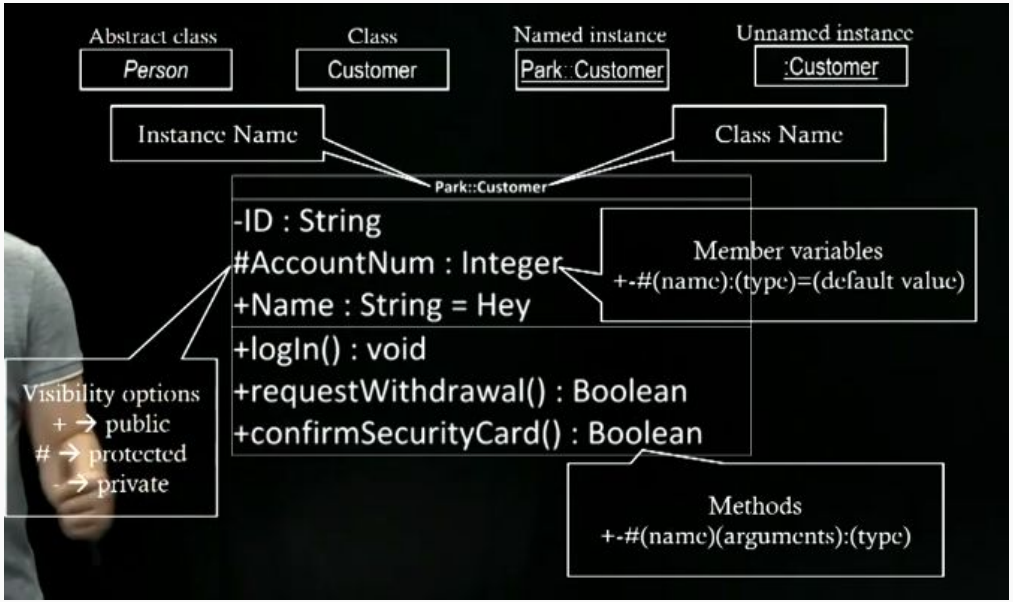
More about UML Notations





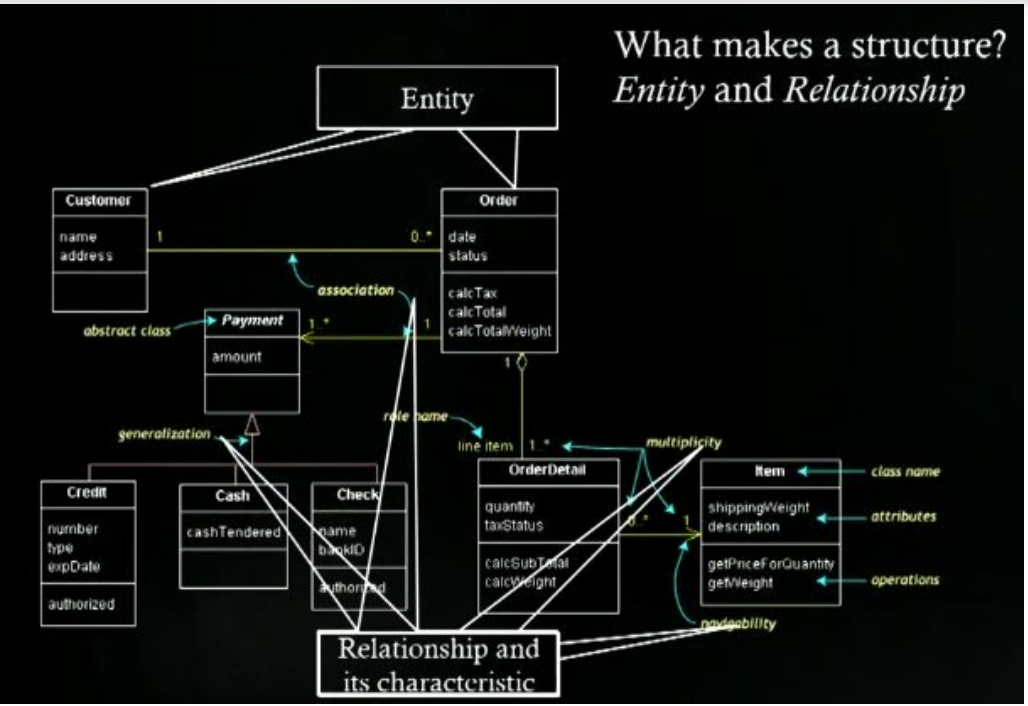
- usecase 다이어그램의 특성은 사람이 있다는 것이다.
- state 다이어그램은 메소드 속에 있는것이 점점 어떻게 개발되는지 순서도와 비슷한다.
- deplovment 다이어그램은 패키징을 어떻게하고 어떤 서버를 내릴것인지를 결정하는 것이다.
- 앞으로 class 다이어그램에 대해서 더 배울것이고 이거 외에서 다양한 설계 문서가 존재하다는 것을 알고 있어야한다.

UML notation: Class and Instance



- customer Class는 abstract class의 메소드를 override(=inheritance)해서 사용할 것이다.
- 비저빌러티 옵션은 인캡슐라이선을하는데 도움이 된다.
- 메소드라는것은오버라이드 오버로드가 가능하다
- login이 메소드 시그너처라고한다.
- 메소드 시그너처가 동일한 경우 메서드 오버라이드가 가능한다
- same 시니너처는 슈퍼 클래스와 자식 클래스 서브 클래스 사이에 same 시그너처가 돼서 자식 클래스의 메소드가 대표하게 된다.
- 시그너처가 동일한게 아니고 similar 할 경우 이름은 동일한데 파라미터가 다를경우 메소드 오버로드가 된다
- 메소드 오버로드는 다양한 파라미터를 가질 수 있는 여러 메소드가 있다. 하지만 그 메소드가 이른은 다 동일하다.
- 파이썬에서는 디폴드값을 넣어줌으로 인해 구현할 수 있습니다.

Structure of Classes in Class Diagram



- 큰 프로그램은 여러 클레스를 묶어서 사용함
- 여기서는 8개의 클레스가 있고 어떻게 관계를 가지고 있는지 보자
- association하는 것을 활용해 왼쪽과 오른쪽이 어떻게 관계를 가지고 있는지 읽어볼 수 있어야 한다.
- generalization은 inherite 할때 나왔는데 credit은 payment의 서브클레스이고 credit은 payment을 inherite하고 있다. Cash와 Check도 마찬가지고 payment를 상속받고 있다.
- 다시말하면 값을 내는데 3가지의 다른 방법이 있고 공통적인 내용은 얼마를 했냐라는 것인데 공통적인 속성 amount는 payment에서 받아 오겠지만 각각 서브클레스는 각자의 속성들다 있다.
- 화살표에 따라 generalization, association, aggregation으로 볼수있고 이것에 따라 서로가 위에따라 관계를 가지고 있다라고 볼수있다.