Object-oriented paradigm and Software design

Il-Chul Moon
Dept. of Industrial and Systems Engineering
KAIST

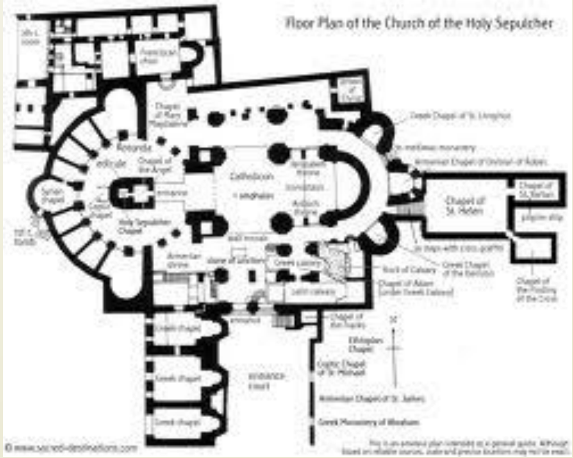
icmoon@kaist.ac.kr

Weekly Objectives

- This week, we learn the object-oriented paradigm (OOP) and the basic of software design.
- Objectives are
  - Understanding object-oriented concepts
    - Class, instance, inheritance, encapsulation, polymorphism…
  - Understanding a formal representation of software design
    - Memorizing a number of Unified Modeling Language (UML) notations
  - Understanding a number of software design patterns
    - Factory, Adapter, Bridge, Composite, Observer
    - Memorizing their semantics and structures

Design and Programming

Software Design


Floor Plan of the Church of the Holy Sepulcher
© www.sacred-destinations.com

Development

Software Implementation

Lobby 1

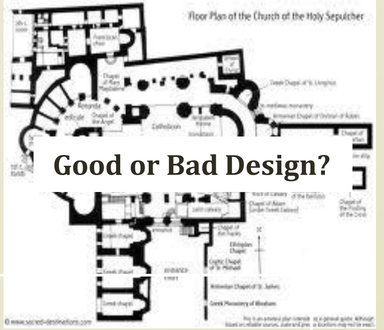
Lobby 2

Restroom

Bedroom

Same Role, Similar Design, and Different Interior

Good Software Design

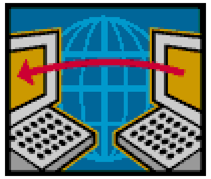


Good or Bad Design?

	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

Object-Oriented Design

Real world concepts



Rates

Today's Rates

Rates as of September 14, 2010

View Foreign Exchange Rates

Description	Rate	Effectiv
TD Prime Rate	3.000%	09/09,
Overnight Money Market Rate	1.000%	07/28,
TD US Prime Rate	3.250%	12/17,
TD US Base Rate	3.750%	12/17,
US Overnight Money Market Rate	0.300%	02/05,

Abstract

Abstract

Abstract

Customer

-ID
-AccountNum

+logIn()
+requestWithdrawal()
+confirmSecurityCard()

Transaction

-amount
-releaseATMID

+releaseMoney()

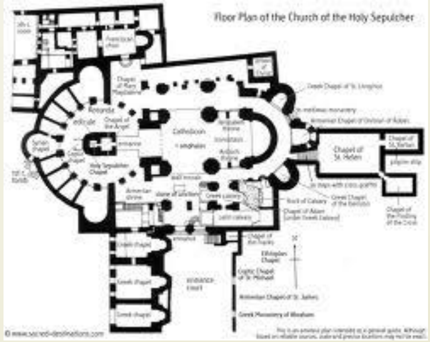
Banking

-amountInAccount

+reduceMoney()
+sendNotice()

Software design entities

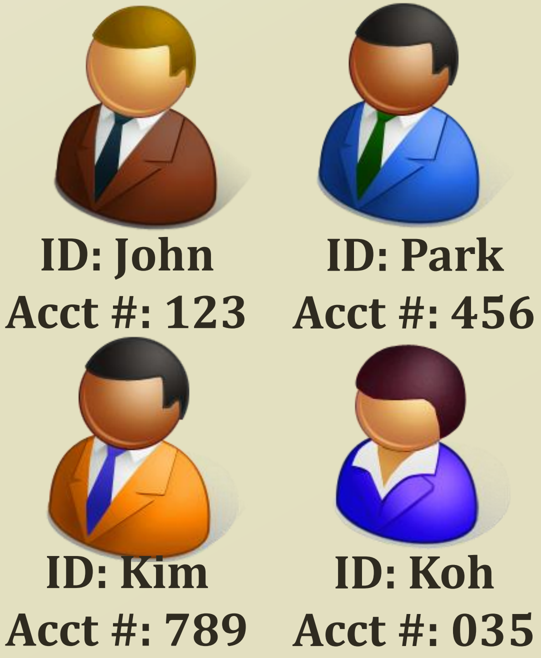
What are Class and Instance?



- Class vs. Instance
- Class
  - Result of design and implementation
  - Conceptualization
  - Corresponds to design abstractions
- Instance
  - Result of execution
  - Realization
  - Corresponds to real world entities

Customer
-ID -AccountNum

+logIn()
+requestWithdrawal()
+confirmSecurityCard()



ID: John
Acct #: 123

ID: Park
Acct #: 456

ID: Kim
Acct #: 789

ID: Koh
Acct #: 035

Software Design as House Floorplan

- After your graduation, some of you will be constructors of software
  - Mainly design
  - Some coding
- Need to learn how to communicate your colleagues
  - Learn standard
  - Learn how to represent your design to your boss
- In software engineering,
  - UML is the standard

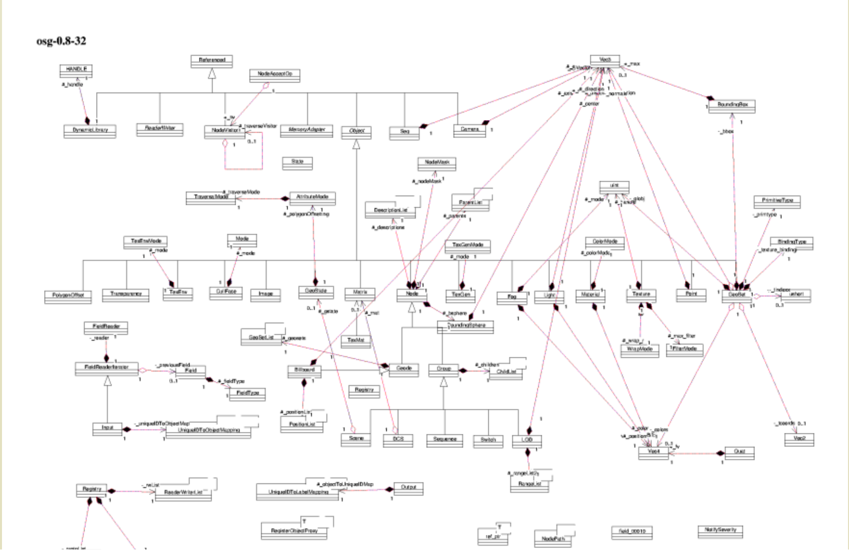
Floor Plan of the Church of the Holy Sepulcher

건설CALS/EC 전자도면 작성표준

Korea Standard of the CAD Drawing
in Construction CALS/EC

V1.3

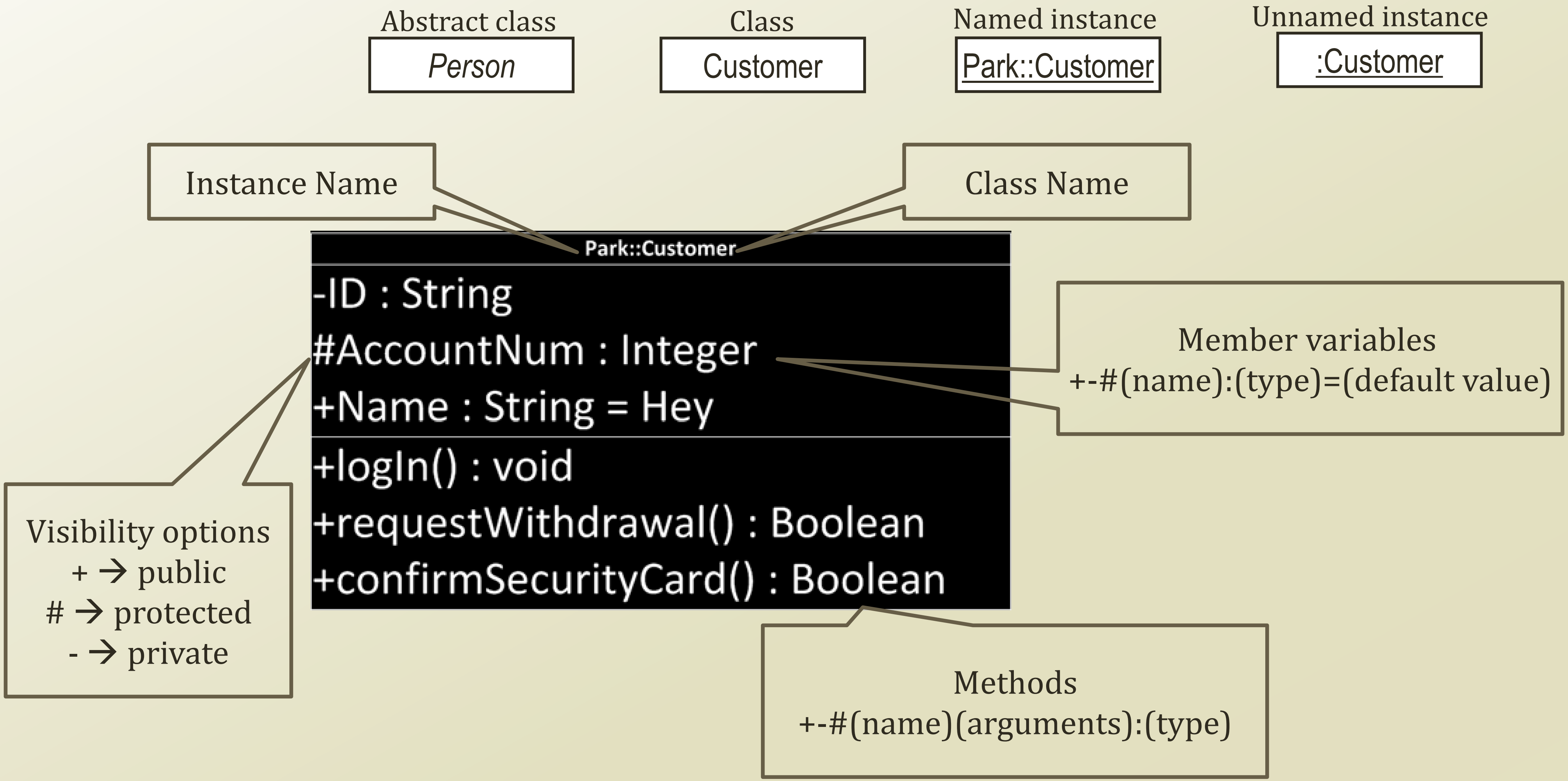
제정일 : 2004년 8월
개정일 : 2010년 2월



OMG WE SET THE STANDARD

UNIFIED MODELING LANGUAGE™

UML notation : Class and Instance



```
Park::Customer
-ID : String
#AccountNum : Integer
+Name : String = Hey
+logIn() : void
+requestWithdrawal() : Boolean
+confirmSecurityCard() : Boolean
```

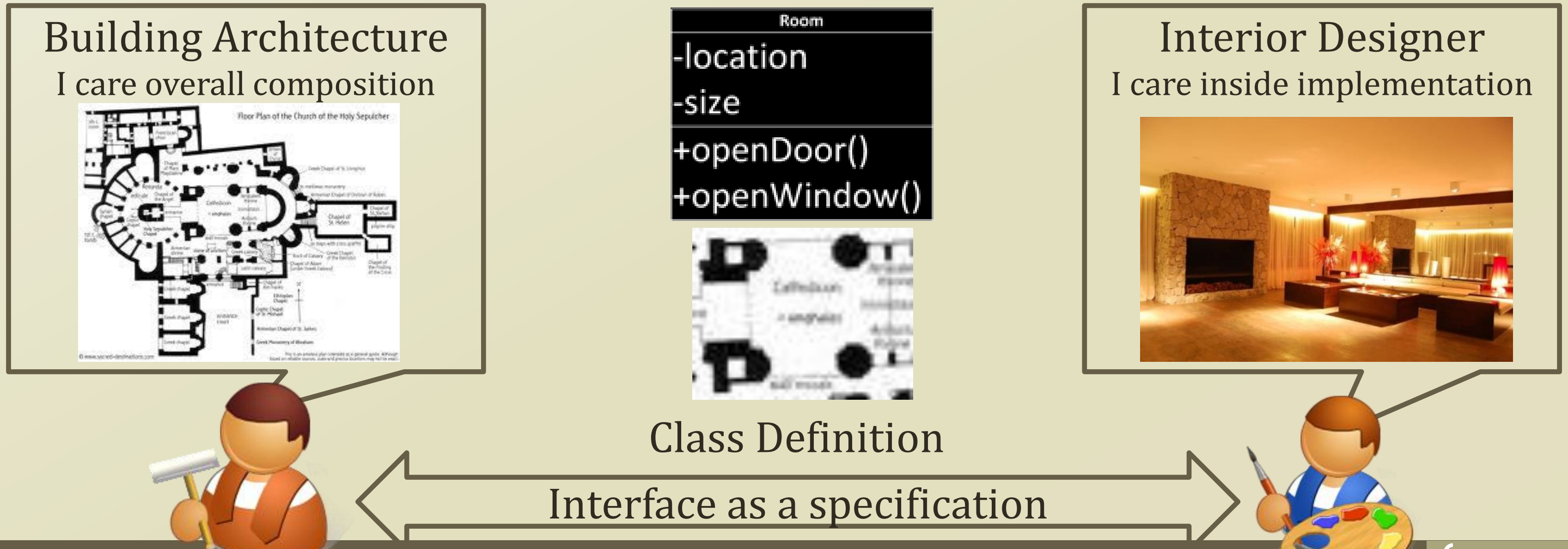
Member variables
+-#(name):(type)=(default value)

Visibility options
+ → public
# → protected
- → private

Methods
+-#(name)(arguments):(type)

Encapsulation

- Object = Data + Behavior
  - Data : field, member variable, attribute
  - Behavior : method, member function, operation
- Delegating the implementation responsibility!
  - Bring me a sausage, and I don’t care how you made it
- Utilizing the visibility
  - private: seen only within the class
  - protected: seen only within the class and its descendants
  - public: seen everywhere
- Python does not support the visibility options!



Building Architecture

I care overall composition

Room

-location

-size

+openDoor()

+openWindow()

Class Definition

Interface as a specification

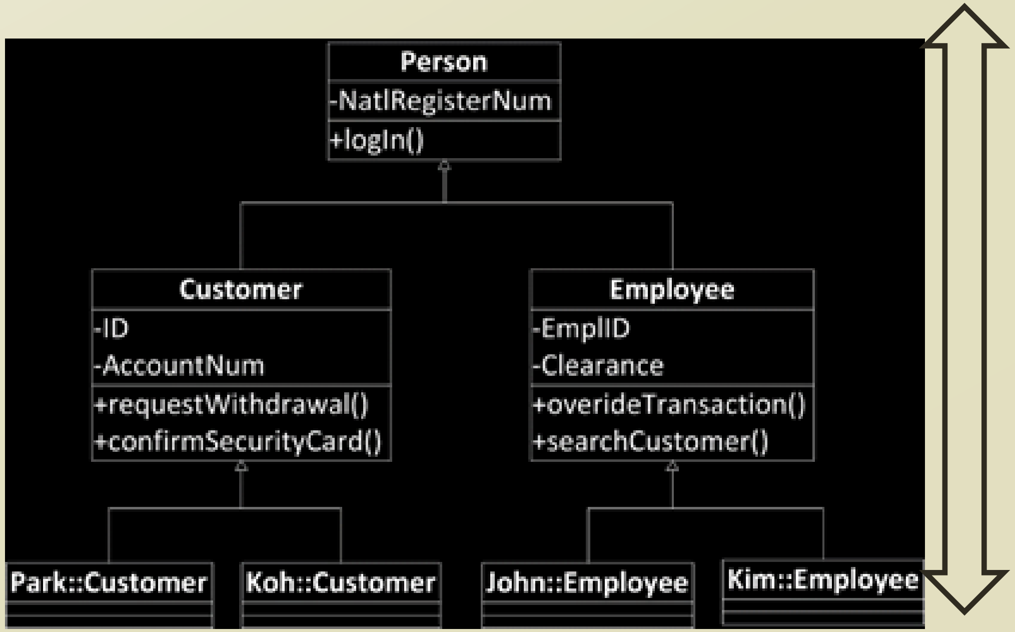
Interior Designer

I care inside implementation

Inheritance

- Inheritance
  - Giving my attributes to my descendants
    - My attributes include
      - Member variables
      - Methods
  - My descendants may have new attributes of their own
  - My descendants may mask the received attributes
  - But, if not specified, sons follow their father
- Superclass
  - My ancestors, specifically my father
  - Generalized from the conceptual view
- Subclass
  - My descendants, specifically my son
  - Specialized from the conceptual view
- How about having a mother?
  - Yes. It is possible in Python

Generalized



Person
-NatlRegisterNum
+logIn()

Customer
-ID
-AccountNum
+requestWithdrawal()
+confirmSecurityCard()

Employee
-EmplID
-Clearance
+overideTransaction()
+searchCustomer()

Park::Customer

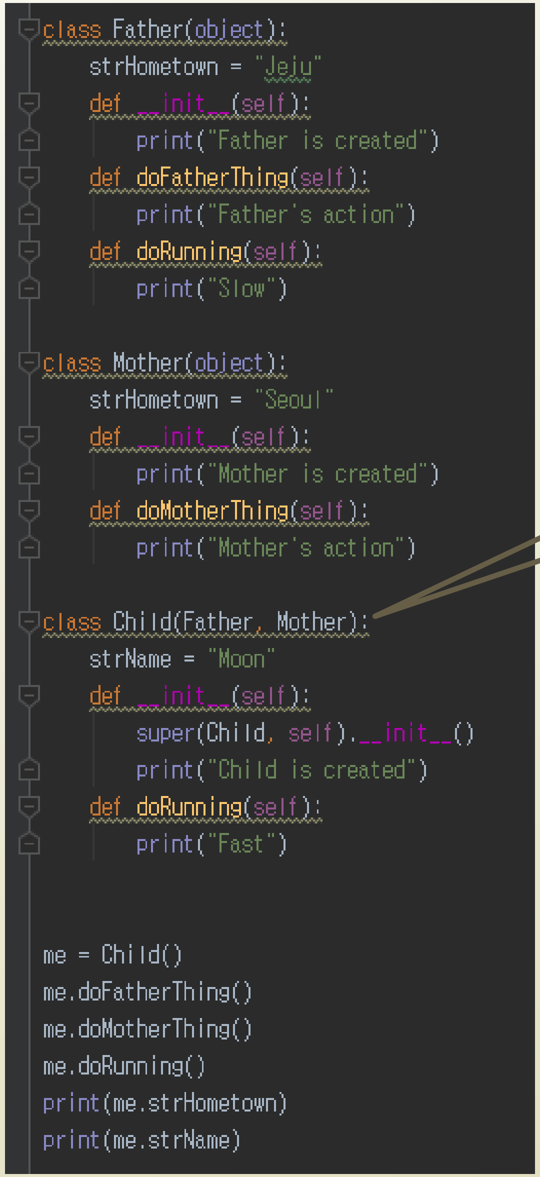
Koh::Customer

John::Employee

Kim::Employee

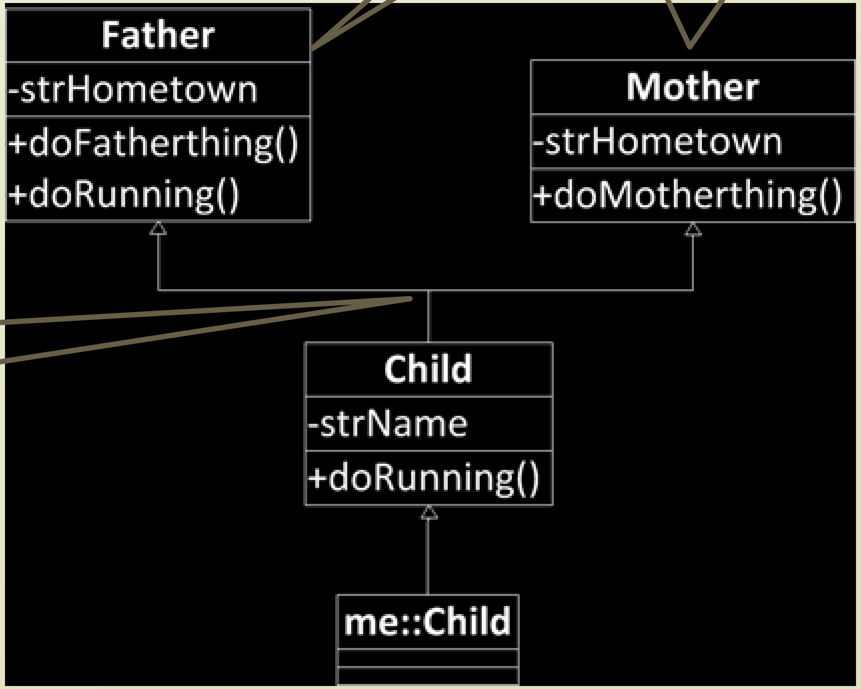
Specialized

Inheritance in Python

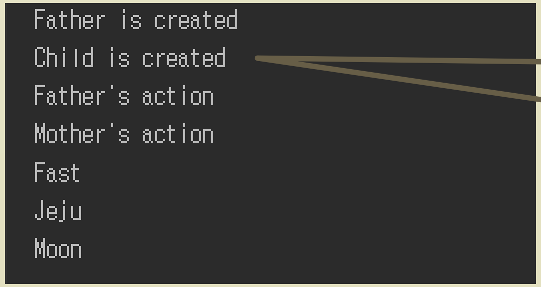


```
class Father(object):
    strHometown = "Jeju"
    def __init__(self):
        print("Father is created")
    def doFatherThing(self):
        print("Father's action")
    def doRunning(self):
        print("Slow")

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.strHometown)
print(me.strName)
```

Base Class
(Super Class)



Multiple
Inheritance

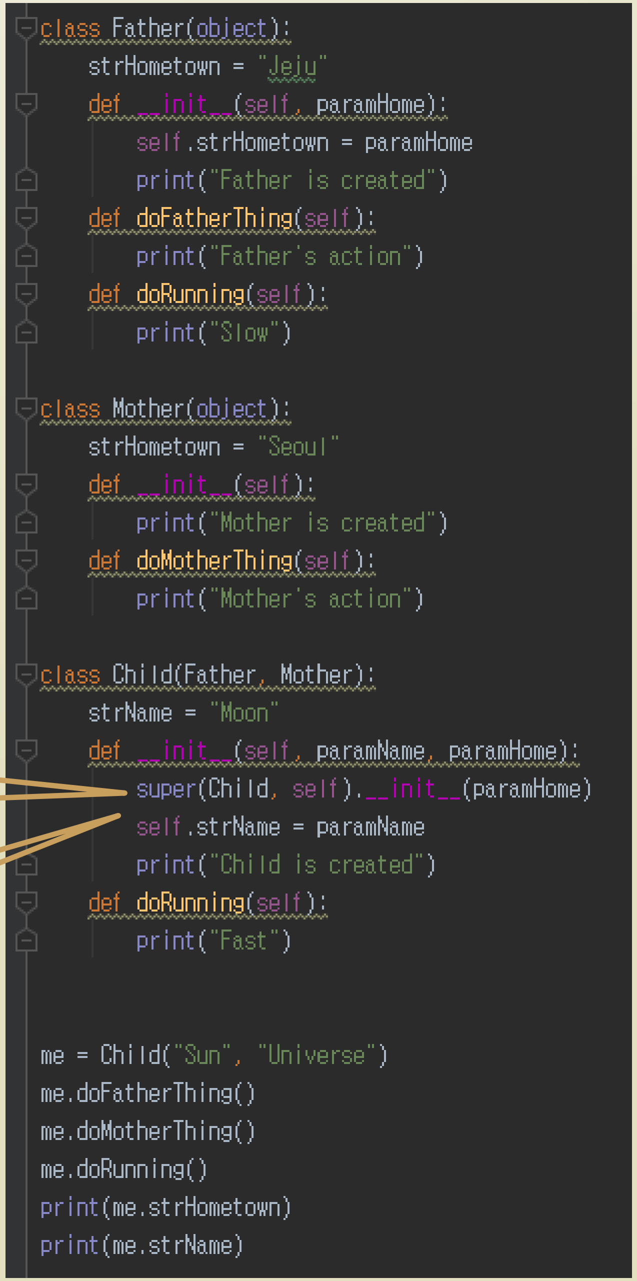


```
Father is created
Child is created
Father's action
Mother's action
Fast
Jeju
Moon
```

1. See Child has Father’s and Mother’s attributes
2. See Child overwrite Father’s method by his own

self and *super*

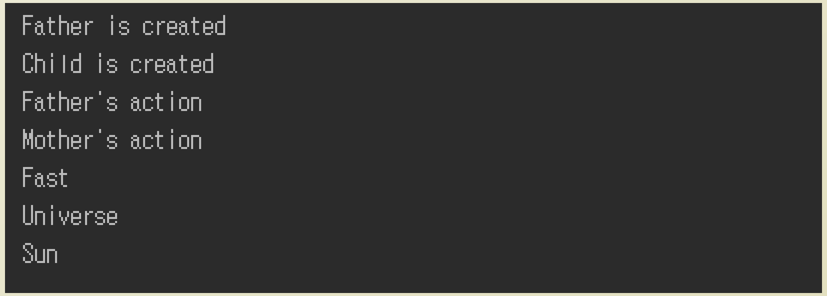
- *self* : reference variable pointing the instance itself
- *super* : reference variable pointing the base class instance
  - super is used to call the base class methods.



```python
class Father(object):
    strHometown = "Jeju"
    def __init__(self, paramHome):
        self.strHometown = paramHome
        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, paramName, paramHome):
        super(Child, self).__init__(paramHome)
        self.strName = paramName
        print("Child is created")
    def doRunning(self):
        print("Fast")


me = Child("Sun", "Universe")
me.doFatherThing()
me.doMotherThing()
me.doRunning()
print(me.strHometown)
print(me.strName)
```

Referring Father to point Father's attributes

Referring itself to point its attributes



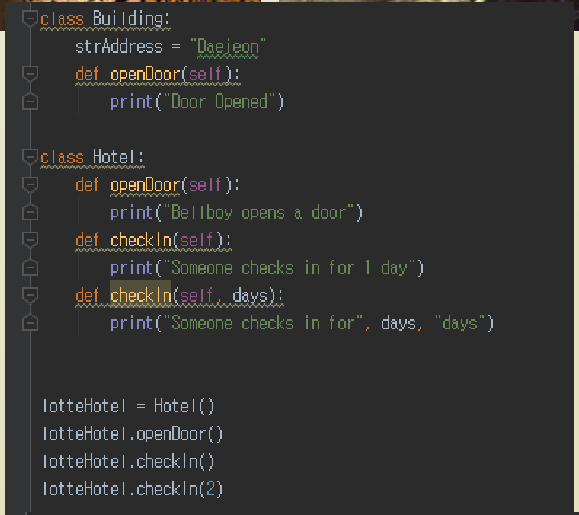
```
Father is created
Child is created
Father's action
Mother's action
Fast
Universe
Sun
```

Polymorphism

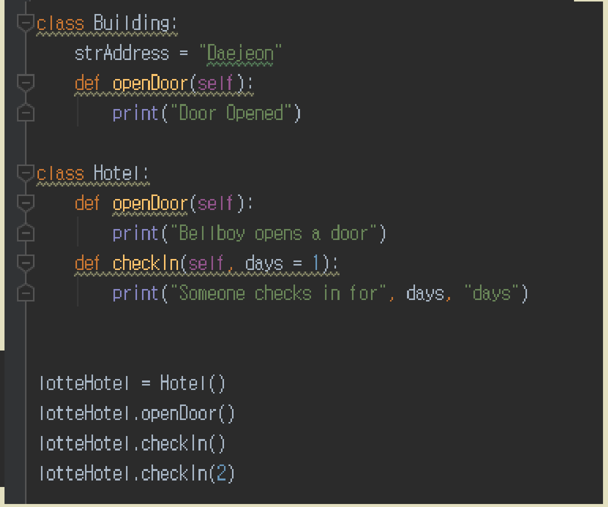
- Polymorphism
  - Poly: Many
  - Morph: Shape
  - Different behaviors with similar signature
    - Signature
      = Method name + Parameter list
  - Method Overriding
    - Base class has a method A(num), and its derived class has a method A(num)
  - Method Overloading
    - A class has a method A(num), A(num, name), and A(num, name, home)

Light Morph

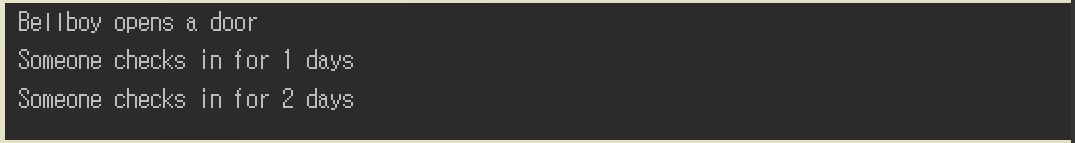
Black Morph



```
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")


lotteHotel = Hotel()
lotteHotel.openDoor()
lotteHotel.checkIn()
lotteHotel.checkIn(2)
```



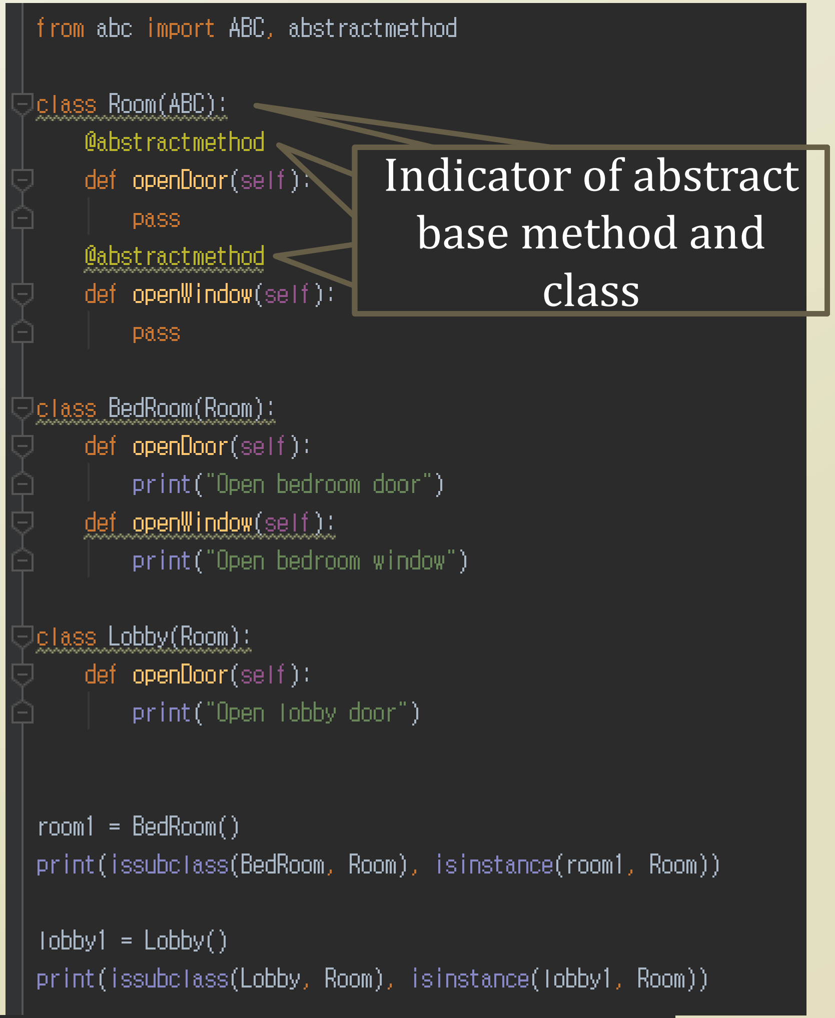
```
class Building:
    strAddress = "Daejeon"
    def openDoor(self):
        print("Door Opened")

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(2)
```



```
Bellboy opens a door
Someone checks in for 1 days
Someone checks in for 2 days
```

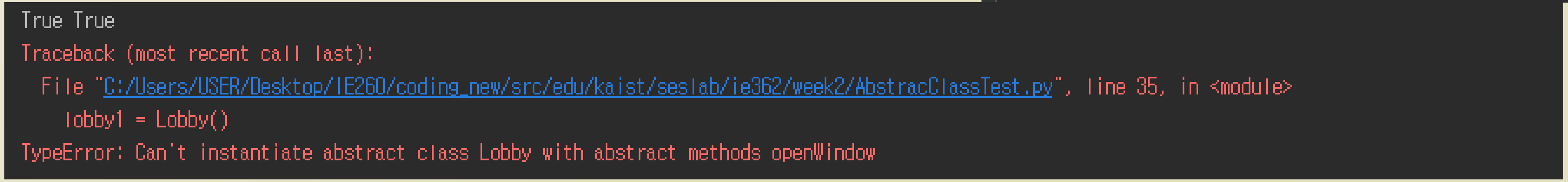
Abstract Class

- Abstract class, or Abstract Base Class in Python
  - A class with an abstract method
  - What is the abstract method?
    - Method with signature, but with no implementation
    - Why use it then?
    - I want to have a window here, but I don't know how it will look like, but you ***should*** have a window here!
  - Abstract class is not a complete implementation, it is more like a half-made produce
  - Therefore, you can't make an instance out of it
- The concrete class with full implementations and inheriting the abstract class will be a basis for instances



```
from abc import ABC, abstractmethod

class Room(ABC):
    @abstractmethod
    def openDoor(self):
        pass
    @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))

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

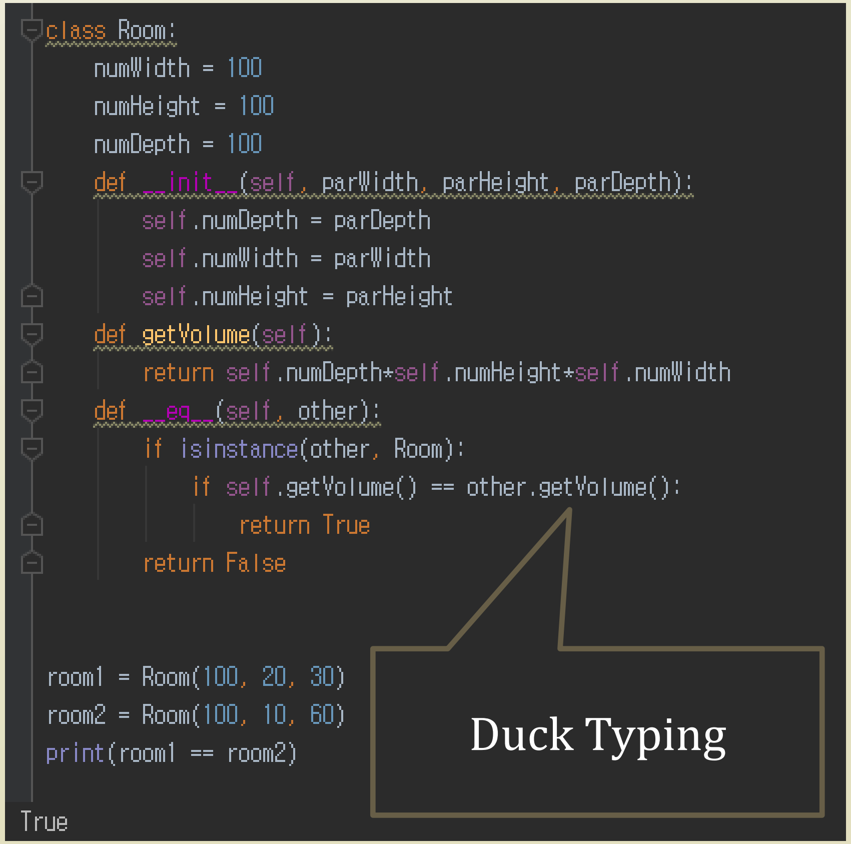
Indicator of abstract base method and class



```
True True
Traceback (most recent call last):
  File "C:/Users/USER/Desktop/IE260/coding_new/src/edu/kaist/seslab/ie362/week2/AbstracClassTest.py", line 35, in <module>
    lobby1 = Lobby()
TypeError: Can't instantiate abstract class Lobby with abstract methods openWindow
```

Overriding Methods in *object*

- All of Python classes are the descendants of *object*
  - If you don't specify the base class of your class, then your class is the direct derived class of *object*
- *object* has many hidden methods
  - __init__
  - __del__
  - __eq__
  - __cmp__
  - __add__
- You override them to make the methods behave as you please



```
class Room:
    numWidth = 100
    numHeight = 100
    numDepth = 100
    def __init__(self, parWidth, parHeight, parDepth):
        self.numDepth = parDepth
        self.numWidth = parWidth
        self.numHeight = parHeight
    def getVolume(self):
        return self.numDepth*self.numHeight*self.numWidth
    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)
```

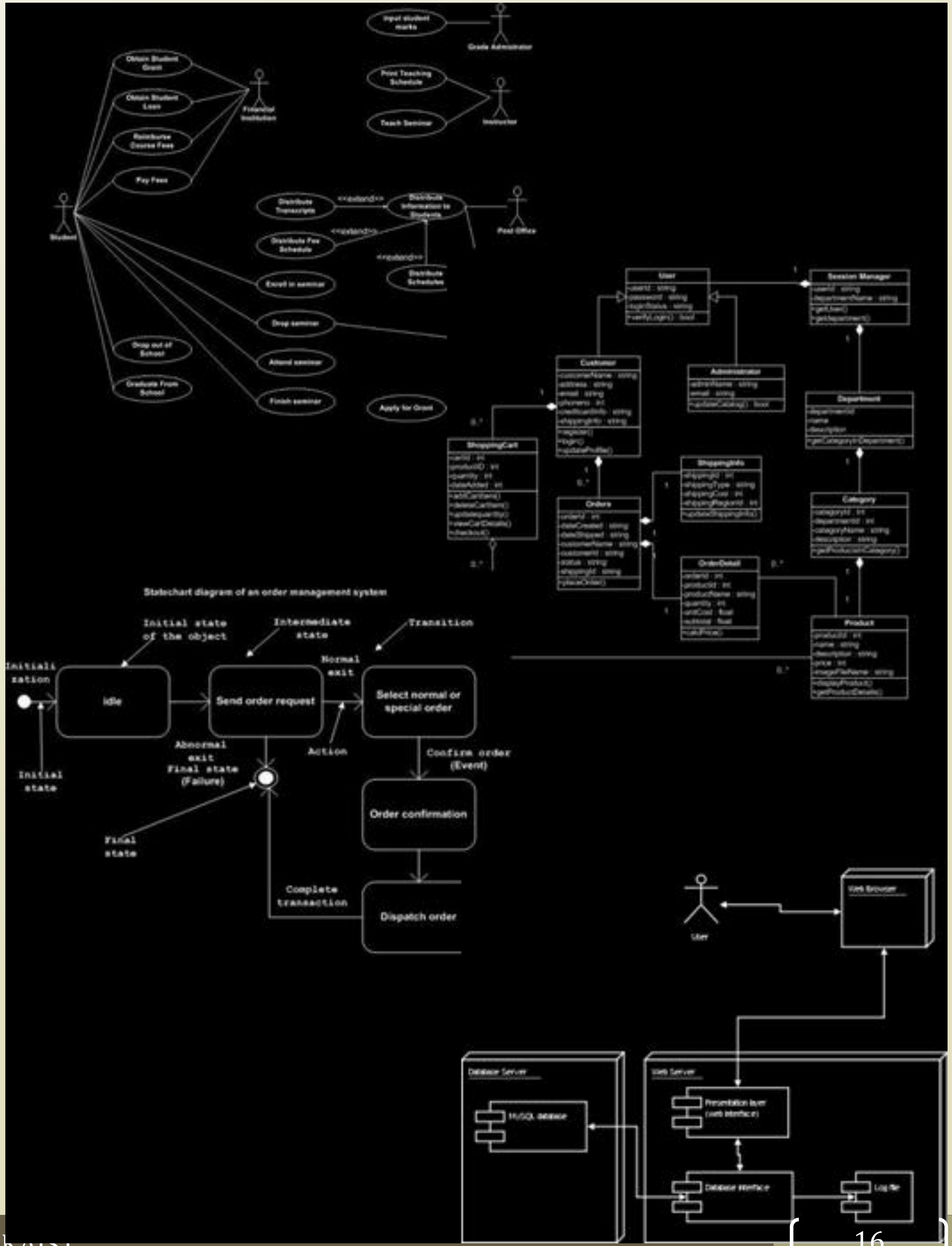
```
True
```

Duck Typing

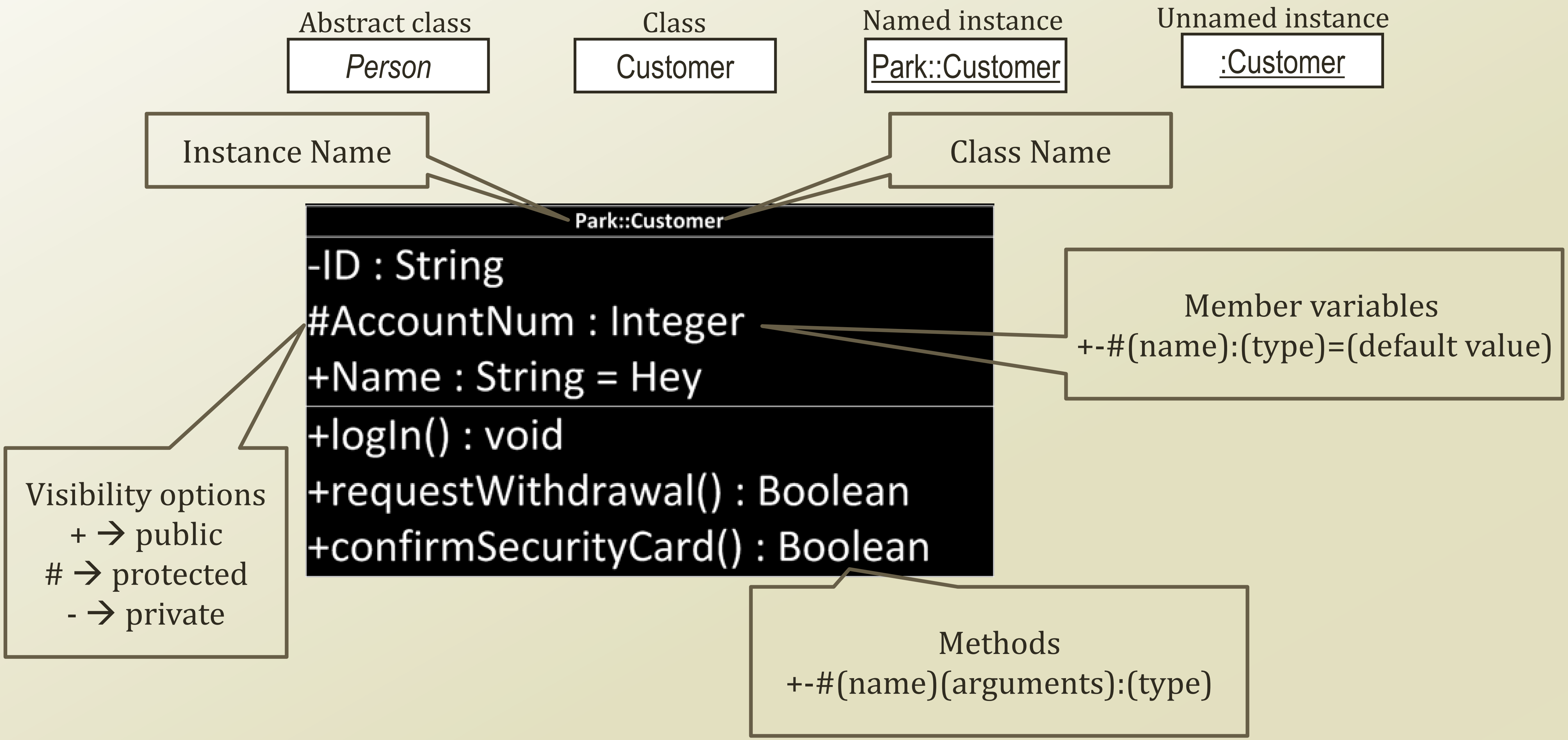
Easier to Ask for Forgiveness then Permission (EAFP)

More about UML Notations

- Many types of UML diagrams used for different stages of development. If I name a few of them…
  - Use-case diagram
  - Class diagram
  - State diagram
  - Deployment diagram
- We are dealing with OOP in this week
  - Mainly, class and instances
  - Also, some of software design patterns
  - Hence, we focus on
    - ***Class diagram***



UML notation : Class and Instance (one more time)



Abstract class

Person

Class

Customer

Named instance

Park::Customer

Unnamed instance

:Customer

Instance Name

Class Name

Park::Customer

-ID : String

#AccountNum : Integer

+Name : String = Hey

+logIn() : void

+requestWithdrawal() : Boolean

+confirmSecurityCard() : Boolean

Member variables

+-#(name):(type)=(default value)

Visibility options

+ → public

# → protected

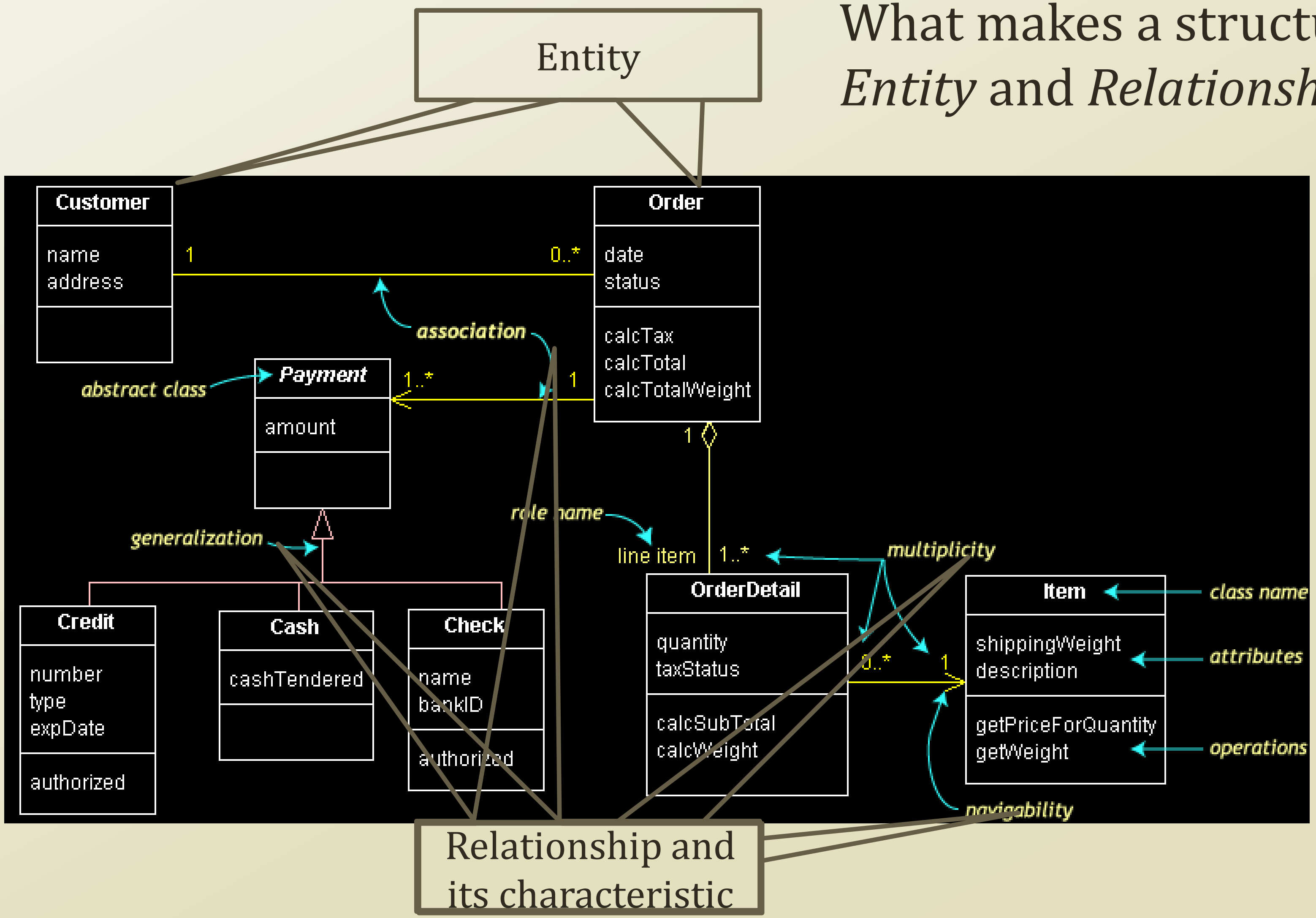
- → private

Methods

+-#(name)(arguments):(type)

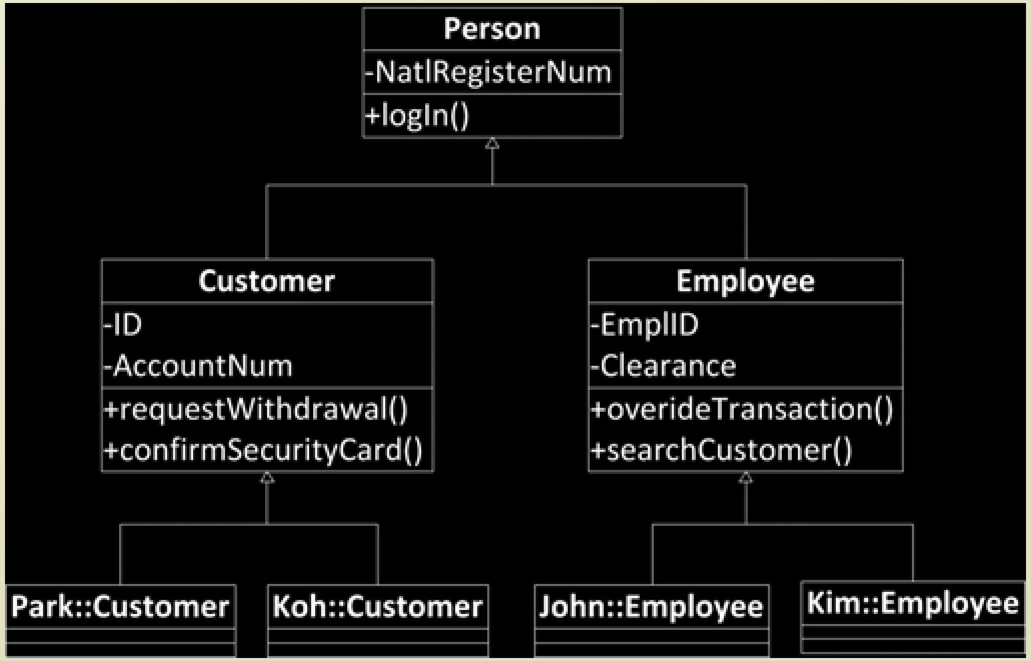
Structure of Classes in Class Diagram

What makes a structure?
Entity and *Relationship*



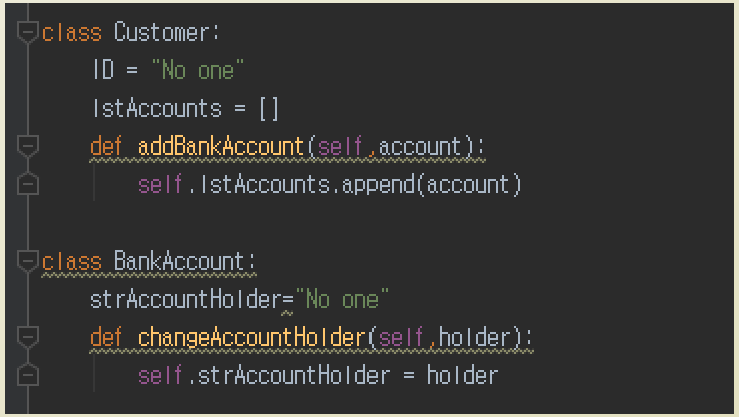
Generalization

- Generalization between classes
  - *is-a* relationship
  - Inheritance relationship
  - Customer → Person
    - From subclass
    - To superclass
    - Direction of generalization
  - Hollow triangle shape
- Base class
  - Person
- Leaf class
  - Park::Customer…

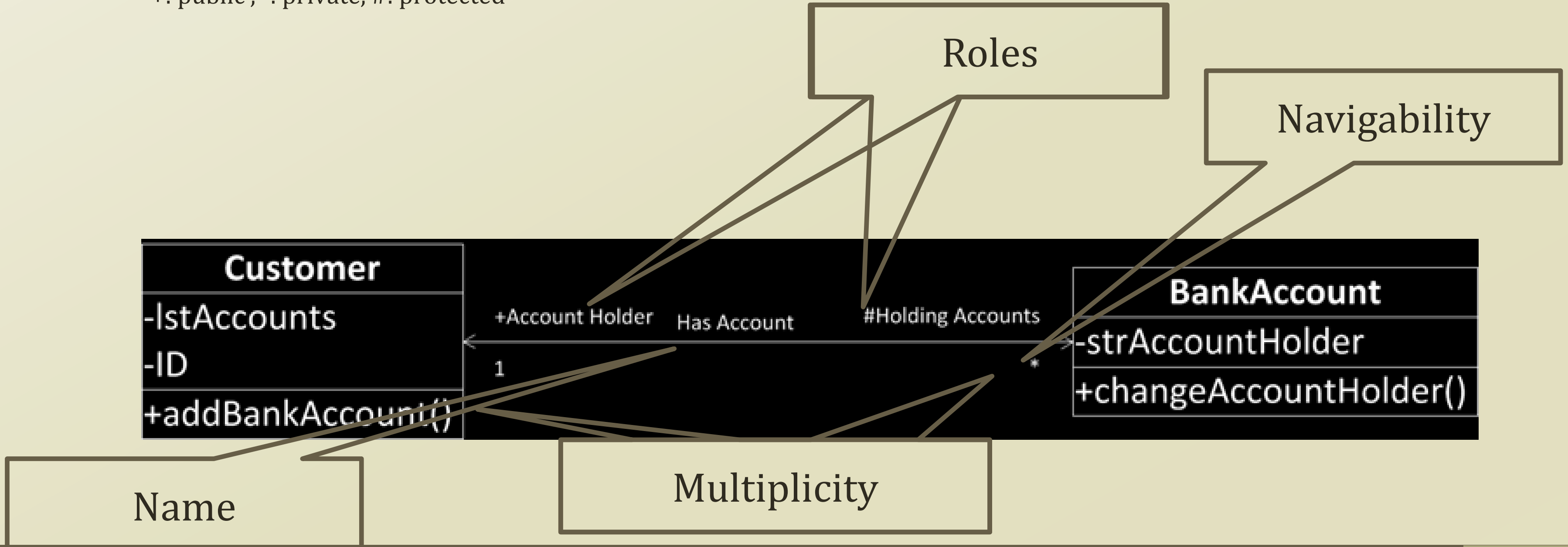


Association

- Association between classes
  - *has-a* relationship
  - Member variables
    - A customer has a number of holding accounts
    - An account has an account holder customer
  - Simple line
  - If a simple arrow is added
    - A customer has a reference to bank accounts
    - A bank account has a reference to a customer
    - Navigability
  - Line ends are tagged by roles
    - Account holder
    - Holding accounts
    - With prefix showing the visibiliy
      - +: public , -: private, #: protected

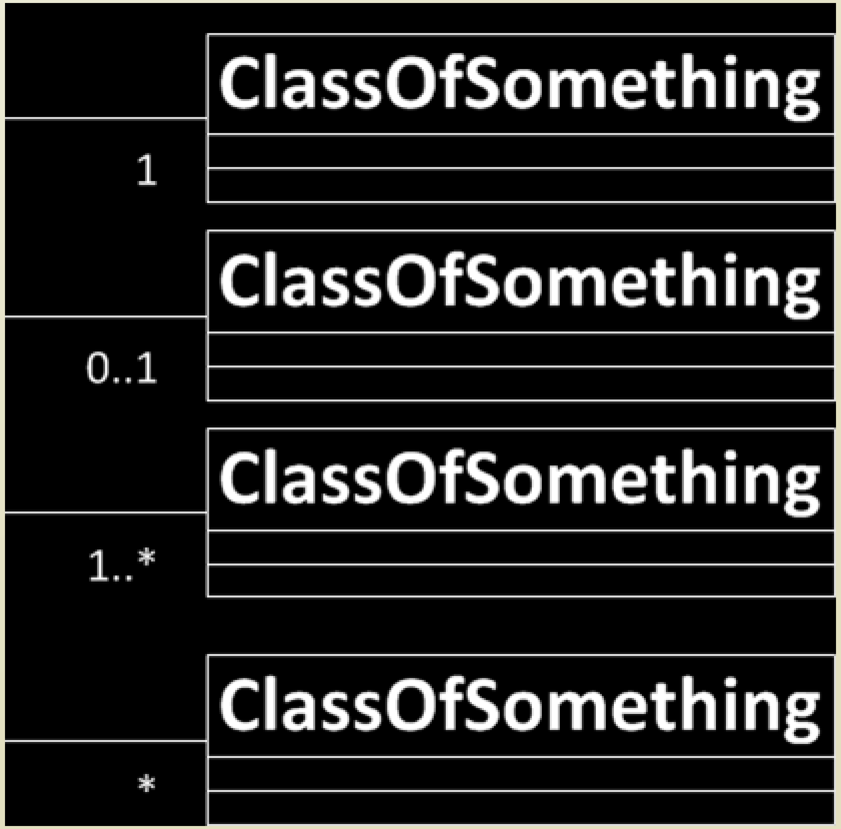


```
class Customer:
    ID = "No one"
    lstAccounts = []
    def addBankAccount(self,account):
        self.lstAccounts.append(account)

class BankAccount:
    strAccountHolder="No one"
    def changeAccountHolder(self,holder):
        self.strAccountHolder = holder
```



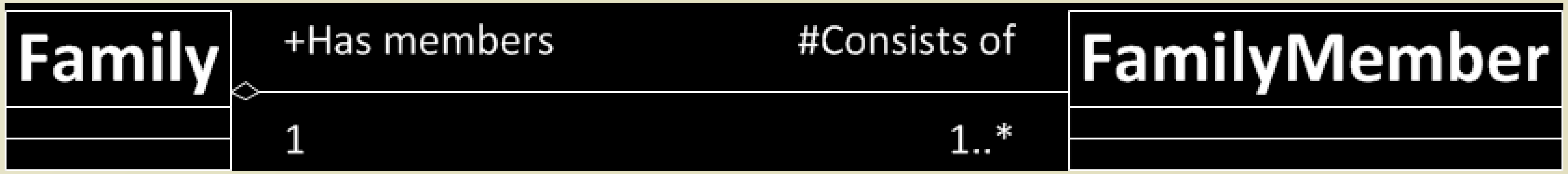
Multiplicity of Association

- In computer science and engineering
  - * often means many
  - Hence,
    - 1..*
      - 1 to Many
    - *
      - 0 to Many
  - Naturally
    - 1
      - Exactly one
    - 0..1
      - One or zero
- If not specified, it means one



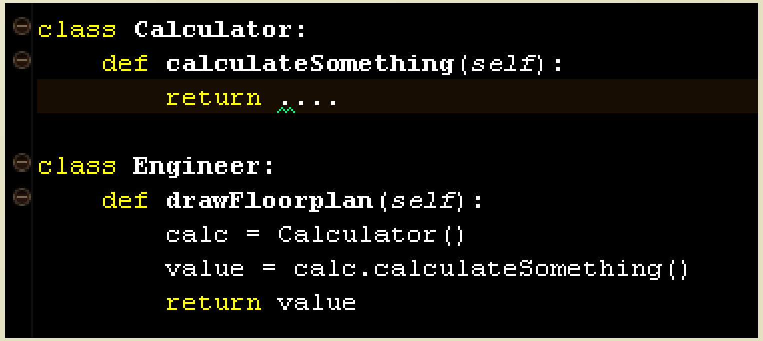
Aggregation

- Special case of association
  - Special *has-a* relationship
  - More like, *part-whole* or *part-of* relationship
  - A family member is a part of a family
    - The existence of the family depends on the aggregation of the family member
    - If nothing to aggregate, there is no family
  - Hollow diamond shape
- Aggregation often occur
  - when an aggregating class is a collection class
  - When the collection class's life cycle depends on the collected classes

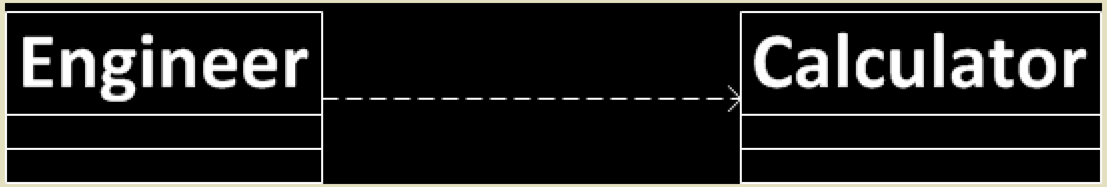


Dependency

- Dependency between classes
  - *use* relationship
  - An engineer uses a calculator
    - May use for
      - Local variables
      - Method signatures
        - Parameter types
      - Method return types
  - Something that you import for the implementation



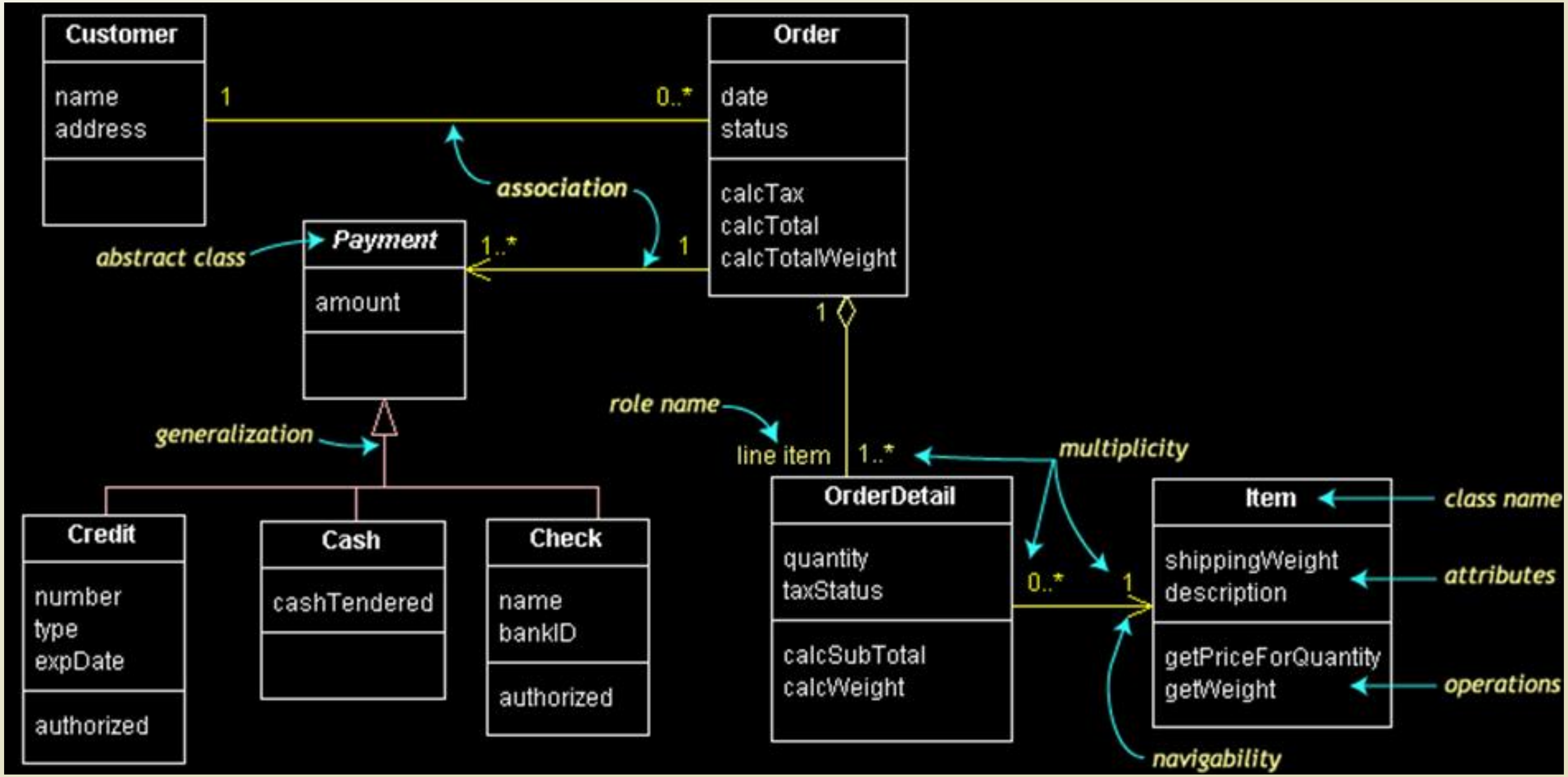
```
class Calculator:
    def calculateSomething(self):
        return ....

class Engineer:
    def drawFloorplan(self):
        calc = Calculator()
        value = calc.calculateSomething()
        return value
```



Engineer

Calculator

Let’s Practice



Customer
name
address
1
0..*
Order
date
status
calcTax
calcTotal
calcTotalWeight
association
abstract class
Payment
1..*
1
amount
generalization
Credit
number
type
expDate
authorized
Cash
cashTendered
Check
name
bankID
authorized
role name
line item
1
1..*
multiplicity
OrderDetail
quantity
taxStatus
calcSubTotal
calcWeight
0..*
1
Item
shippingWeight
description
getPriceForQuantity
getWeight
class name
attributes
operations
navigability