



01076105, 01076106

Object Oriented Programming

Object Oriented Programming Project

Encapsulation, Class Diagram, Inheritance



Encapsulation

- Encapsulation เป็นคุณสมบัติที่สำคัญหนึ่งของ Object Oriented โดย Encapsulation มาจากคำว่า capsule: (เป็นชิ้นเดียวกัน) กับ en = (ทำให้)
- ความหมาย คือ การนำข้อมูล (attribute) กับการทำงาน (method) มารวมไว้ด้วยกัน และป้องกันไม่ให้ภายนอกสามารถเข้าถึงข้อมูลที่ไม่ต้องการได้





Encapsulation

- ลักษณะอย่างหนึ่งที่เกิดขึ้นจาก Encapsulation คือ information hiding ซึ่งหมายถึง การซ่อนหรือจำกัดการเข้าถึง “ข้อมูล” บางส่วนที่ไม่ต้องการให้เข้าถึงจากภายนอก Object ได้โดยตรง
- ยกตัวอย่าง เช่น ความยาวของภาพยนตร์ จะต้องเป็นจำนวนเต็มบวกเสมอ ดังนั้น หากปล่อยให้มีการเข้าถึงข้อมูลจากภายนอก Object หรือ Instance ได้โดยตรง ก็อาจจะมีการใส่จำนวนลบในข้อมูลดังกล่าวได้
- การเข้าถึง attribute ของ Instance โดยใช้ dot notation จึงขัดกับหลัก information hiding เราจึงต้องหาทางจำกัดการเข้าถึงข้อมูล

```
<object>.<attribute> = <new_value>
```



Encapsulation

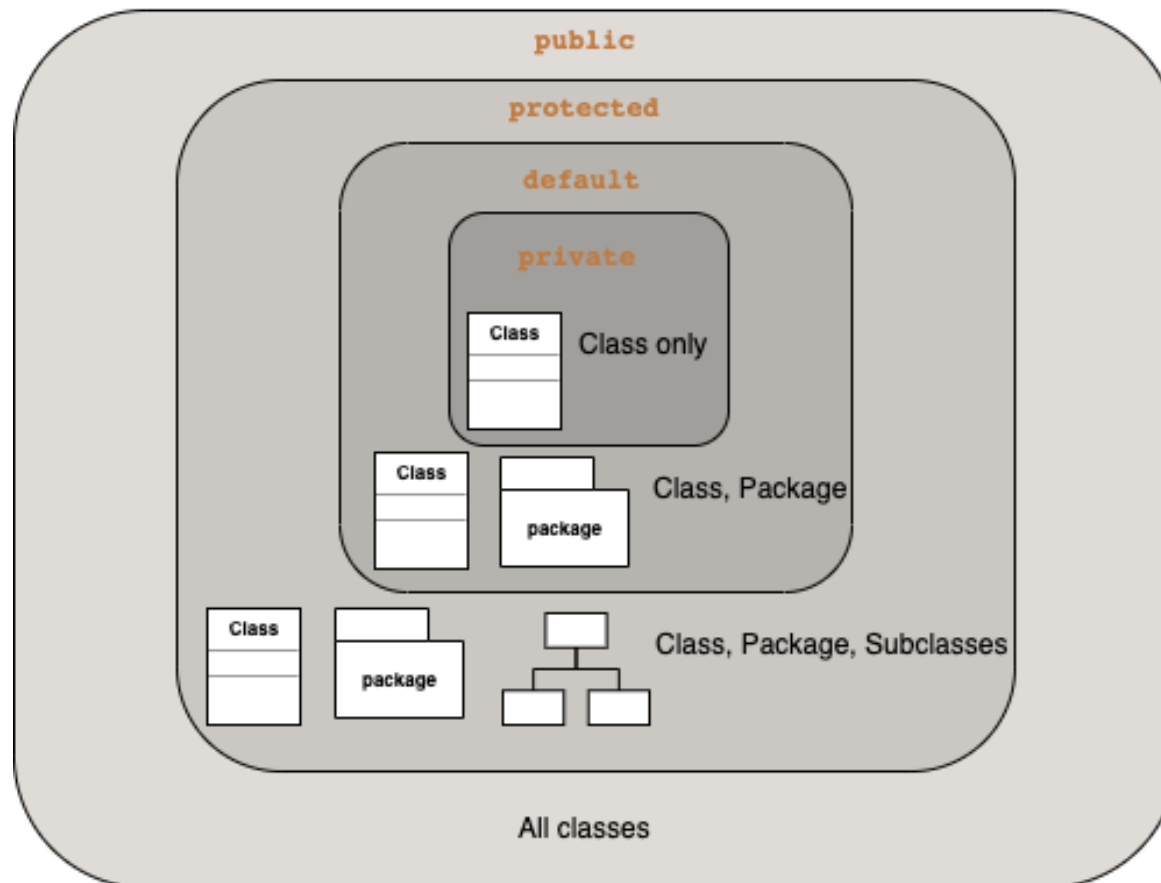
- ในภาษา Programming ที่เป็น Object Oriented โดยทั่วไปจะแบ่งประเภทข้อมูลของ Object เอาไว้ 2-3 ระดับ ในหลายภาษามีการแบ่งดังนี้
 - ข้อมูลระดับที่ 1 คือ Public หมายถึงข้อมูลที่อนุญาตให้เข้าถึงจากภายนอก Object ได้โดยตรง
 - ข้อมูลระดับที่ 2 คือ Protected หมายถึงข้อมูลที่อนุญาตให้เข้าถึง จากภายในคลาสของตนเองและคลาสที่สืบทอดไปเท่านั้น
 - ข้อมูลระดับที่ 3 คือ Private หมายถึงข้อมูลที่อนุญาตให้เข้าถึง จากภายในคลาสของตนเองเท่านั้น



Encapsulation



- Java access modifier





Encapsulation

- ในภาษา Java จะเข้มงวดกับ Access Modifier มาก โดยภาษาเองจะทำหน้าที่ป้องกันการเข้าถึงตามที่กล่าวมา
- แต่ภาษา Python ไม่ได้ใช้วิธี Access Modifier ในการควบคุมการเข้าถึง แต่ใช้วิธี Name Conventional โดยกำหนดให้แนวทางการตั้งชื่อให้เป็นไปตามตารางนี้
- Python ไม่ได้บังคับการเข้าถึง แต่ใช้แนวทางการตั้งชื่อเพื่อให้ทราบว่าต้องการแบบใด

Name	Notation	Behavior
name	Public	เข้าถึงได้จากภายในและภายนอก
_name	Protected	เหมือนกับ Public แต่ไม่ควรเข้าถึงจากภายนอก
__name	Private	ไม่สามารถเห็นได้จากภายนอก



Encapsulation

- จากตัวอย่างโปรแกรมนี้ จะเห็นว่ายังสามารถอ้างถึง `_year` ได้ แต่อ้างเป็น `year` ไม่ได้
- ดังนั้นการตั้งชื่อ attribute ในภาษา python ให้เป็น protected ให้ใช้ `_` นำหน้า

main.py

```
1 ▼ class Car:
2 ▼     def __init__(self, brand, model, year):
3         self._brand = brand
4         self._model = model
5         self._year = year
6
7 my_car = Car("Porsche", "911 Carrera", 2022)
8
9 print(my_car._year)
10 my_car._year = 2024
11 print(my_car._year)
12
13 #print(my_car.year)
```

2022

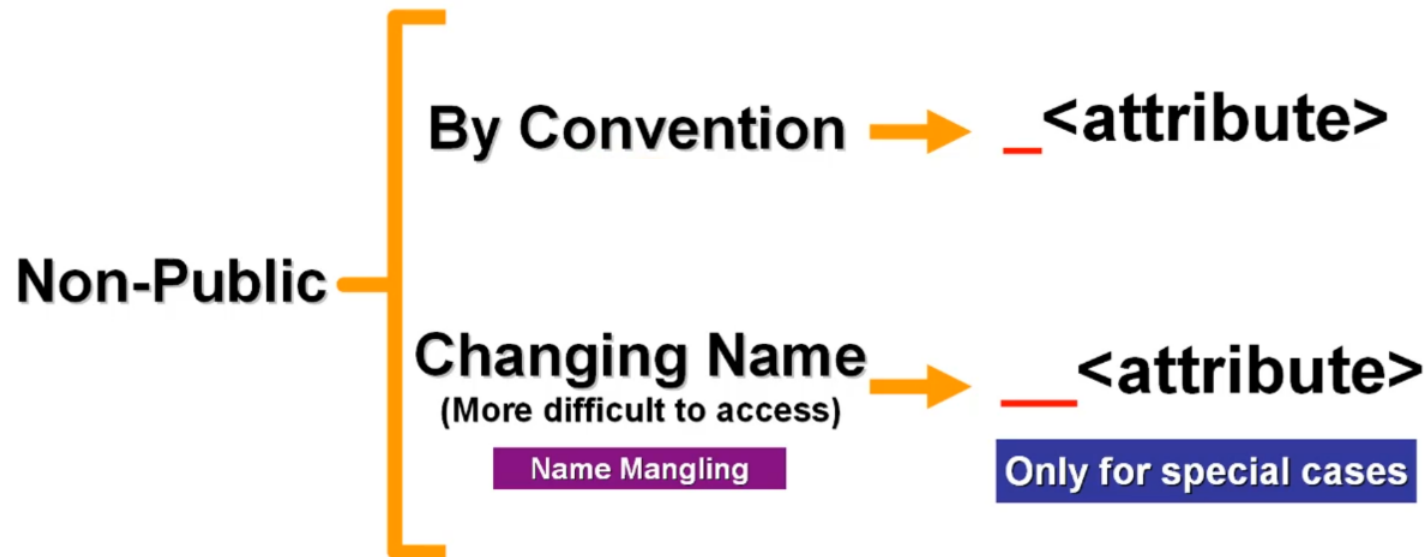
2024





Encapsulation

- ในบางแห่งจะไม่แยก attribute ของคลาสเป็น Public, Protected และ Private แต่จะเรียกเป็น Public กับ Non-Public
- จากนั้นจึงค่อยแยก Non-Public ออกเป็น 2 ประเภท ตามรูป
- และไม่ควรอ้างตัวแปรในคลาสโดยตรง โดยใช้ “_”





Class Diagram

- ส่วนประกอบของ Class Diagram
 - คลาสอาจจะเป็นตัวแทนของ คน สถานที่ เหตุการณ์ หรือสิ่งต่าง ๆ ซึ่งเป็นส่วนประกอบของระบบที่เรากำลังวิเคราะห์และออกแบบอยู่

แอตทริบิวต์ (Attribute)

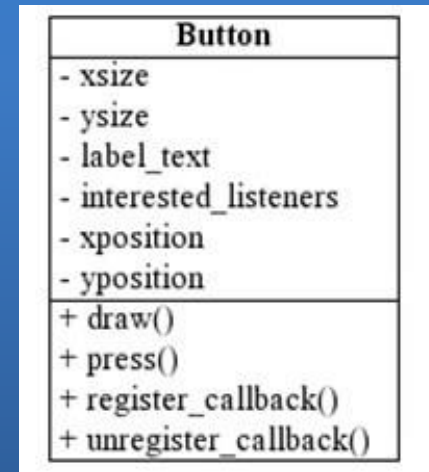
แอตทริบิวต์คือข้อมูลที่เป็นคุณสมบัติของคลาส คือข้อมูลที่เราสสนใจจะจัดเก็บและนำมาใช้ในระบบ

เมธอด (Method)

เมธอด คือการทำงานที่คลาสสามารถทำงานได้

ระดับของการเข้าถึงข้อมูล

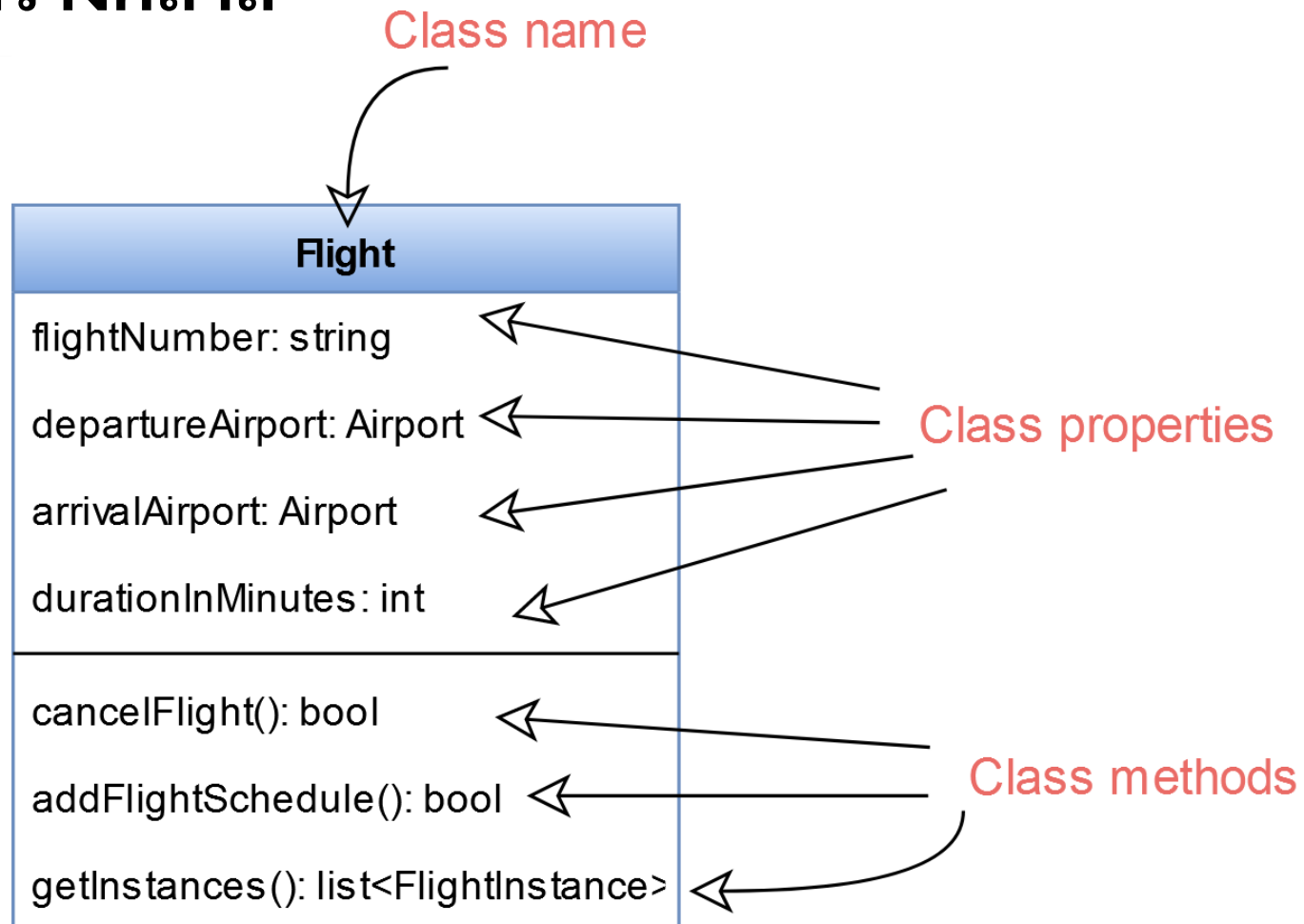
- (+) public ให้คลาสอื่น ๆ ใช้งานข้อมูลนี้ได้อิสระ
- (#) protected ให้เฉพาะคลาสที่สืบทอดใช้งานได้
- (-) private ไม่อนุญาตให้คลาสอื่นใช้งานได้





Class Diagram

- โครงสร้างคลาส



Class Diagram : ความสัมพันธ์

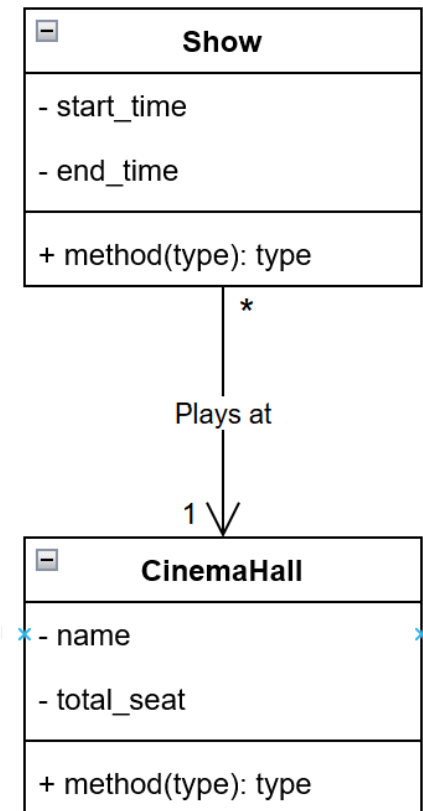


- ระหว่างคลาส อาจมีความสัมพันธ์กันในรูปแบบใดรูปแบบหนึ่ง โดยความสัมพันธ์ระหว่าง Class มีดังนี้
 - Association
 - Dependency
 - Aggregation
 - Composition

Class Diagram : ความสัมพันธ์



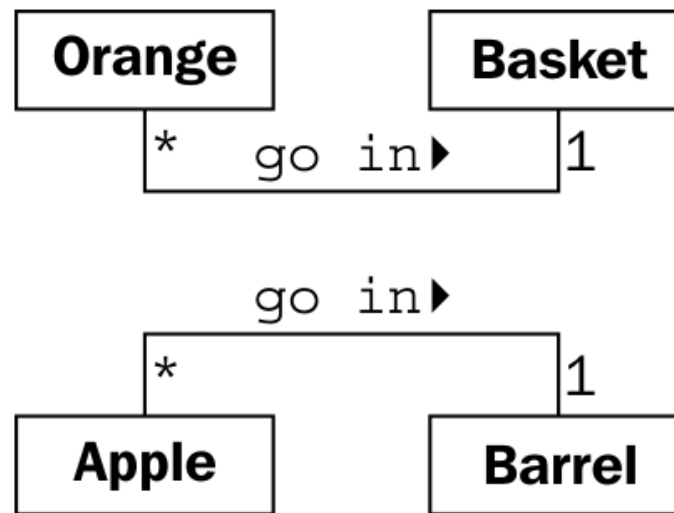
- ความสัมพันธ์แบบ Association เป็นความสัมพันธ์ระหว่างคลาส ที่แสดงถึงการกระทำที่ object หนึ่ง กระทำกับอีก object หนึ่ง แบ่งออกเป็น
- **Unary Association** หมายถึง object หนึ่งกระทำกับอีก object หนึ่งในทิศทางเดียว จากรูป คือ ความสัมพันธ์ รอบฉายกับโรงภาพยนตร์
- ให้สังเกตว่าในเส้นที่แสดงความสัมพันธ์จะเขียนชื่อการกระทำเอาไว้ด้วย
- อาจกล่าวได้ว่ารอบฉาย “ฉายที่” โรงภาพยนตร์หนึ่ง แต่โรงภาพยนตร์สามารถฉายภาพยนตร์หลายเรื่องก็ได้





Class Diagram : ความสัมพันธ์

- ตัวอย่างความสัมพันธ์ในลักษณะ Association ที่ชัดเจน เช่น



- ตัวอย่างข้างต้นจะเห็นได้อย่างชัดเจนว่าความสัมพันธ์แบบนี้ แสดงว่าจะต้องมีส้มในตะกร้า หรือ แอปเปิ้ลในถัง และ ส้ม กับ แอปเปิ้ล ก็จะต้องมีข้อมูลว่าอยู่ในถังใด



Class Diagram : ความสัมพันธ์

- ดังนั้นในกรณีนี้ ในคลาส CinemaHall ก็จำเป็นต้องเก็บข้อมูลรอบฉายเอาไว้
- จึงจำเป็นต้องแก้ไขโค้ดในคลาส CinemaHall โดยเพิ่ม List ของ Show (รอบฉาย) เข้าไป (เนื่องจาก 1 โรงจะมีรอบฉายหลายรอบ)
- จะเห็นว่าได้เปลี่ยน attribute เป็น protected ทั้งหมดแล้ว

```
class CinemaHall:
    def __init__(self, name, total_seat, seats, shows):
        self._name = name
        self._total_seat = total_seat
        self._shows = shows # List of Show
```



Class Diagram : ความสัมพันธ์

- และคลาส Show ก็ต้องบอกว่าฉายที่ไหน ดังนั้นคลาส show ก็จะต้องแก้ไขดังนี้

```
class Show:
    def __init__(self, played_at, start_time, end_time):
        self._show_id = id
        self._start_time = start_time
        self._end_time = end_time
        self._played_at = played_at
```

- จะเห็นว่าคลาส Show ยังไม่มีข้อมูลเกี่ยวกับภาพยนตร์ เพราะยังไม่ได้เขียนความสัมพันธ์ระหว่าง Movie กับ Show ก็จะมีการเก็บข้อมูลภาพยนตร์ที่ฉายขึ้นมา



Class Diagram : ความสัมพันธ์

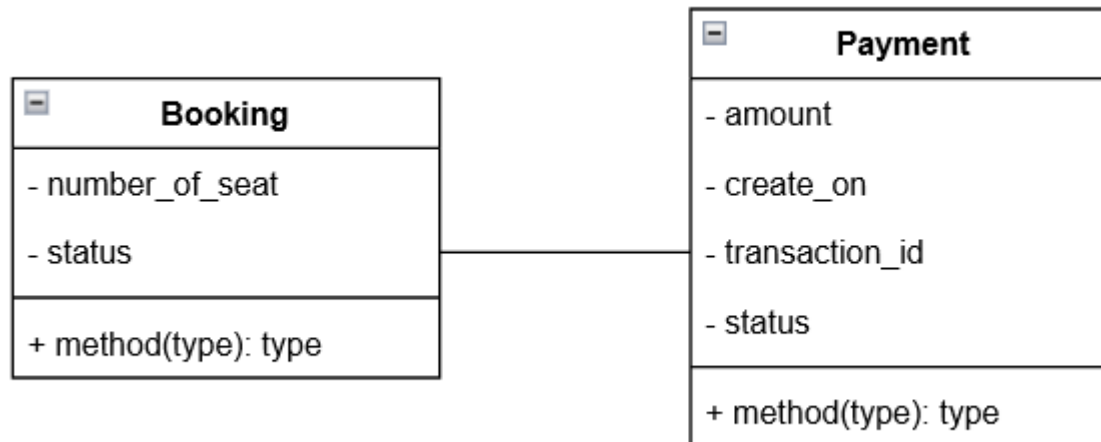
- หากเพิ่มข้อมูลภาพยนตร์ ก็จะทำให้ข้อมูลการฉายภาพยนตร์ครบถ้วน คือ รอบฉายนี้ฉายภาพยนตร์เรื่องอะไร เริ่มต้นเวลาใด สิ้นสุดเวลาใด และ ฉายที่โรงภาพยนตร์ใด

```
class Show:
    def __init__(self, movie, played_at, start_time, end_time):
        self._show_id = id
        self._start_time = start_time
        self._end_time = end_time
        self._played_at = played_at
        self._movie = movie
```




Class Diagram : ความสัมพันธ์

- **Binary Association** : เป็นความสัมพันธ์แบบ 2 ทิศทาง ตัวอย่างของความสัมพันธ์แบบนี้ ดังรูป
- จากรูปจะเห็นว่า เป็นข้อมูลคนละตัว แต่มีการใช้งานร่วมกัน คือ เมื่อจองแล้ว ก็ต้องมีการจ่ายเงิน และ ในส่วนของการจ่ายเงินก็ต้องระบุว่าเป็นการจ่ายเงินของรายการใด



Class Diagram : ความสัมพันธ์



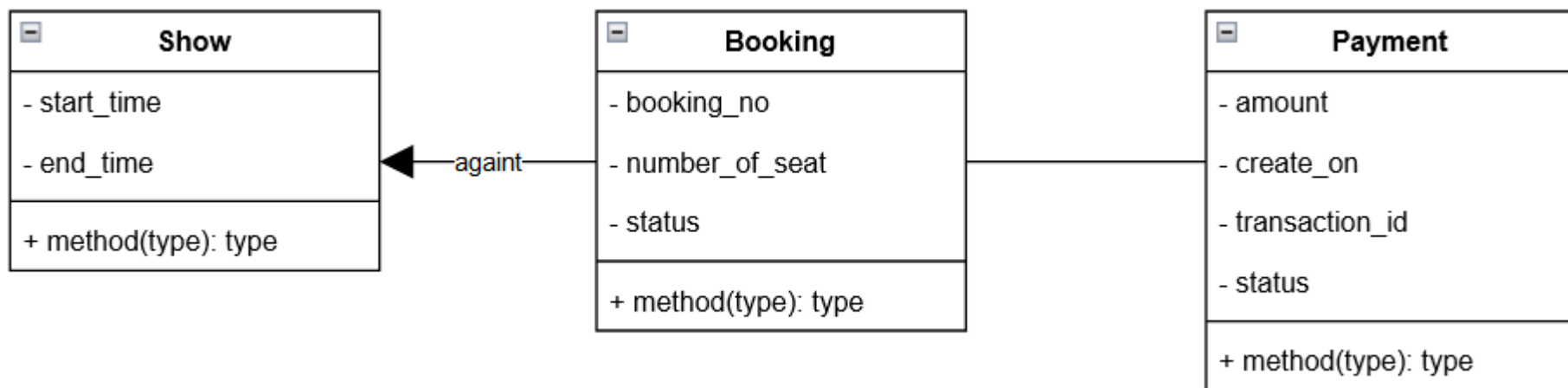
- ความสัมพันธ์แบบ Association แบบ 2 ทางก็ต้องมีข้อมูลความสัมพันธ์เก็บเอาไว้เช่นกัน โดยอาจเก็บที่คลาสใดคลาสหนึ่ง หรือ ทั้ง 2 คลาส ดังนั้นจากคลาสเดิม จะเก็บข้อมูล payment เก็บเอาไว้ในคลาส Booking เพื่อจะได้ทราบว่า Instance ที่สร้างจากคลาส Booking มีข้อมูล payment เป็นอย่างไร
- โดย payment ในคลาสนี้ คือ object ของ Payment ที่สัมพันธ์กับการจองนี้

```
class Booking:
    def __init__(self, booking_no, no_of_seat, status, payment):
        self._booking_no = booking_no
        self._no_of_seat = no_of_seat
        self._status = status
        self._payment = payment
```

Class Diagram : ความสัมพันธ์



- ในส่วนของคลาส Booking นั้น นอกเหนือจากมีความสัมพันธ์กับ Payment แล้ว ยังมีความสัมพันธ์กับ Show ตามรูป





Class Diagram : ความสัมพันธ์

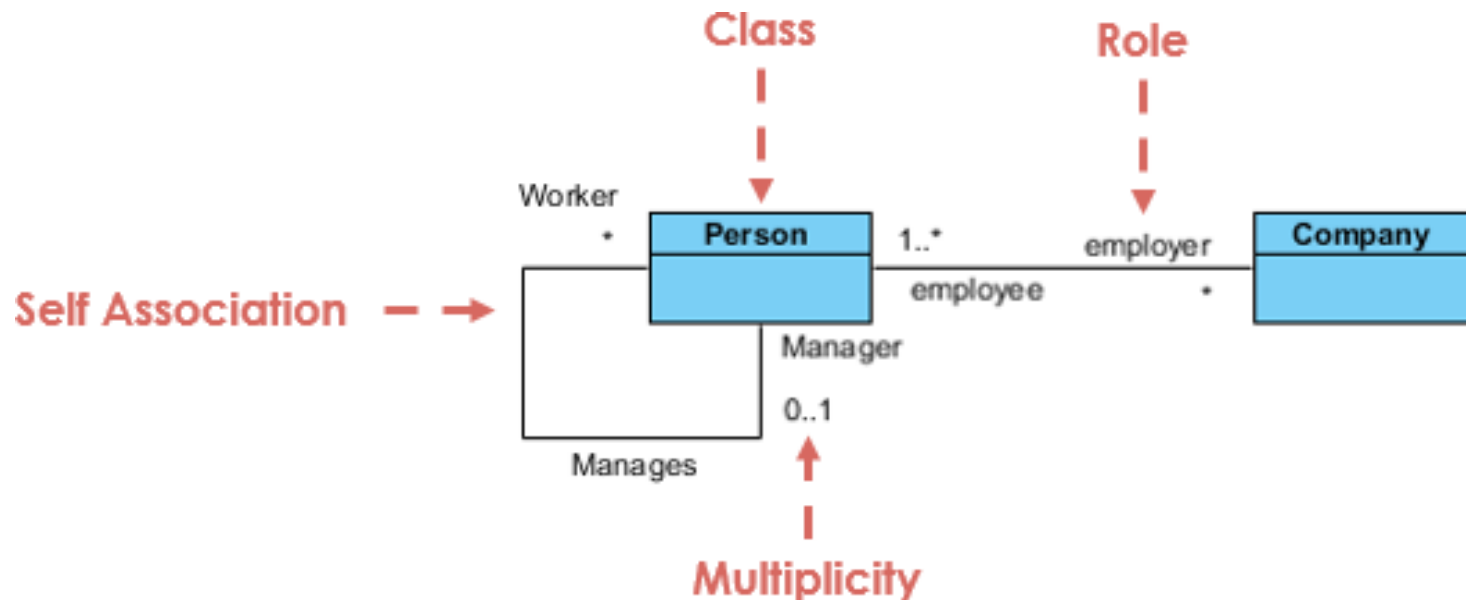
- ดังนั้นคลาส Booking จะต้องเพิ่มข้อมูลของ show เข้าไปด้วย
- Show ที่เป็นพารามิเตอร์ คือ Instance ของคลาส Show
- จะเห็นว่าคลาส Booking เก็บข้อมูลที่สมบูรณ์มากขึ้น คือ ข้อมูลจำนวนที่นั่ง ข้อมูลการชำระเงิน และ ข้อมูลของรอบฉาย

```
class Booking:
    def __init__(self, booking_no, no_of_seat, status, payment, show):
        self._booking_no = booking_no
        self._no_of_seat = no_of_seat
        self._status = status
        self._payment = payment
        self._show = show
```



Class Diagram : ความสัมพันธ์

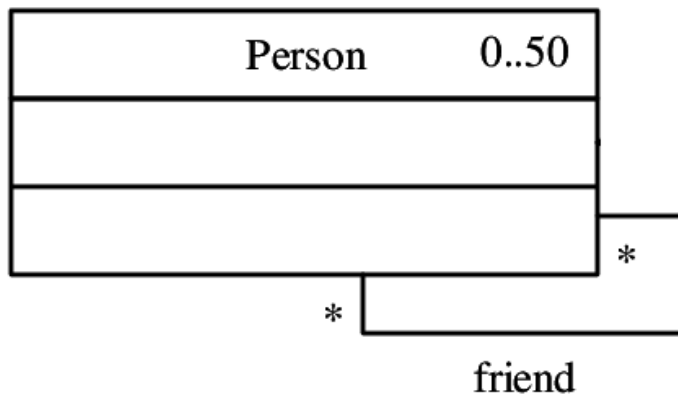
- **Self Association** เป็นความสัมพันธ์แบบ Association อีกแบบหนึ่ง แต่เป็นความสัมพันธ์กับคลาสตัวเอง เช่น หัวหน้ากับลูกน้อง
- จากตัวอย่างจะแสดงความสัมพันธ์ของ 2 object ที่ ทำงานภายใต้ หรือ เป็นผู้บังคับบัญชา





Class Diagram : ความสัมพันธ์

- ตัวอย่างของ Self Association
เช่น ความสัมพันธ์ของเพื่อน



- สามารถเขียน Code ได้เป็น

```
class Person:
    def __init__(self, name):
        self.name = name
        self.friend = []

    def add_friend(self, person):
        self.friend.append(person)
```

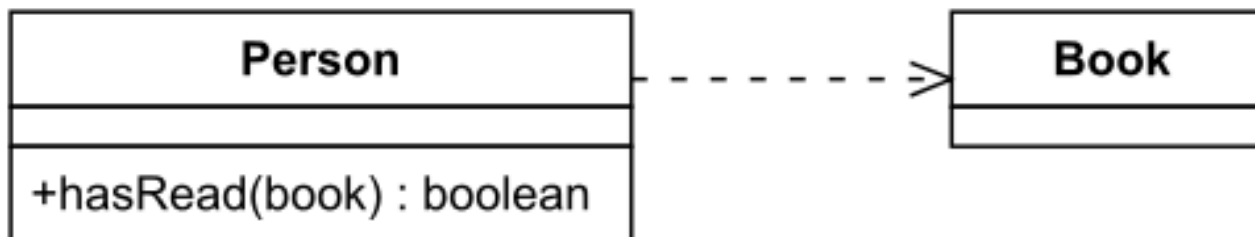
```
john = Person("john")
jane = Person("jane")
john.add_friend(jane)
print(john.friend[0].name)
```

jane



Class Diagram : ความสัมพันธ์

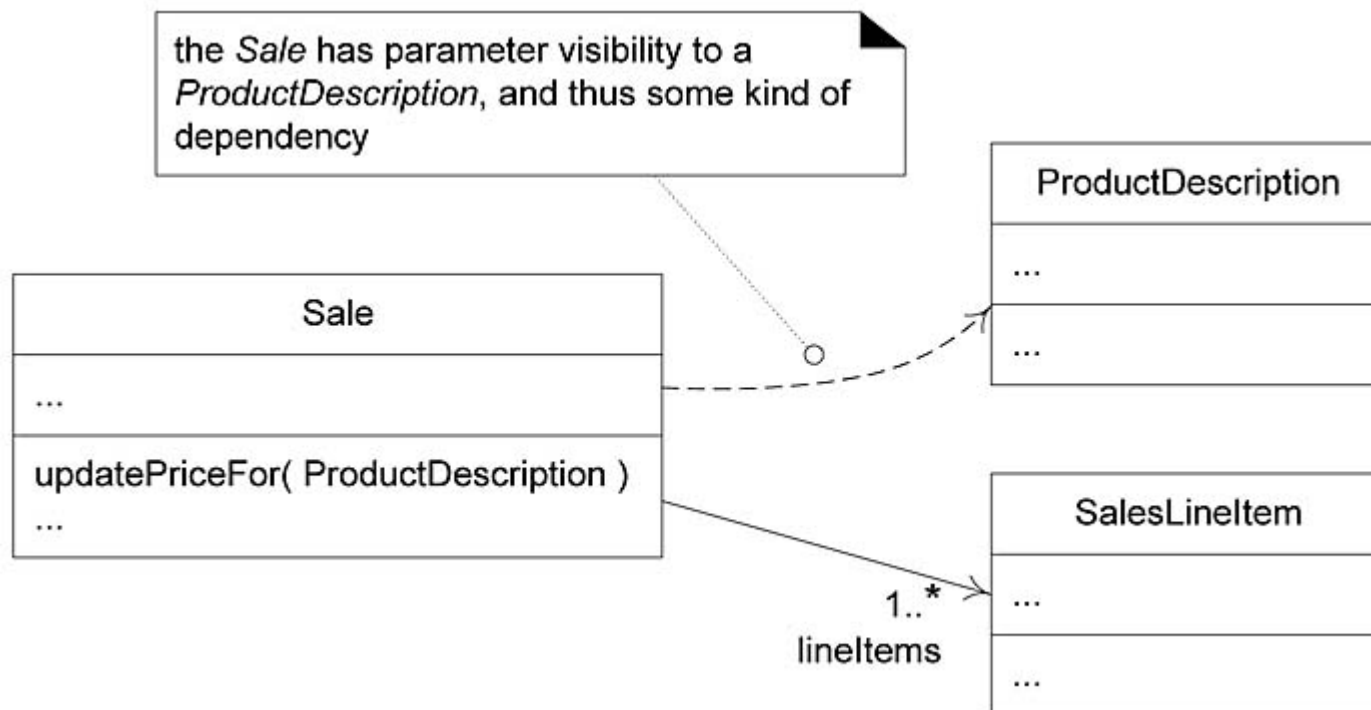
- **Dependency** : เป็นความสัมพันธ์ที่คล้ายกับ Association แต่ความสัมพันธ์ที่อ่อนกว่า คือ ในขณะที่ ความสัมพันธ์แบบ Association เป็นการสร้าง object และมีการกระทำระหว่างกัน โดยมีการเก็บความสัมพันธ์ลงในข้อมูลของ object แต่ Dependency จะไม่มีการเก็บความสัมพันธ์นั้น แต่จะมีการใช้อีก object เป็นพารามิเตอร์ของ method ใน object เท่านั้น
- ความสัมพันธ์แบบ Dependency จะแสดงโดยใช้เส้นประลูกศร
- ตัวอย่างจะเห็นว่า Book จะเป็นเพียงพารามิเตอร์ของฟังก์ชันเท่านั้น



Class Diagram : ความสัมพันธ์



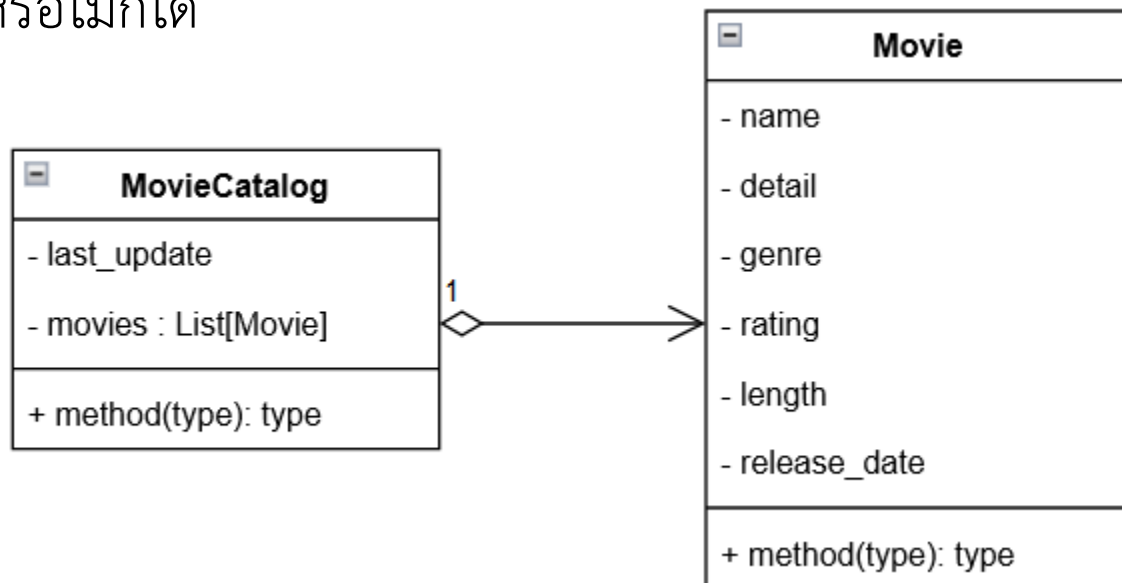
- อีกตัวอย่างของ Dependency
- จากรูปเป็นความสัมพันธ์ระหว่าง เซลล์ กับ ผลิตภัณฑ์ โดย เซลล์ทำหน้าที่ในการ update ราคาของผลิตภัณฑ์ แต่ไม่มีความสัมพันธ์กันโดยตรง





Class Diagram : ความสัมพันธ์

- **Aggregation** เป็นความสัมพันธ์ระหว่างคลาสที่ใกล้ชิดกว่า Association โดยคลาสที่มีความสัมพันธ์แบบ Aggregation ต่อกัน คือ คลาสที่เป็นองค์ประกอบของอีกคลาสหนึ่ง เช่น Movie เป็นองค์ประกอบของ MovieCatalog
- ความสัมพันธ์ **Aggregation** จะใช้สี่เหลี่ยมขนมเปียกปูนแบบโปร่ง ดังตัวอย่าง
- ลูกศรจะมีหรือไม่ก็ได้





Class Diagram : ความสัมพันธ์

- ความสัมพันธ์แบบ Aggregation นั้น จะต้องมีความสัมพันธ์เช่นเดียวกัน โดยจะเก็บไว้ที่คลาสหลัก (ที่มีสี่เหลี่ยม) ข้อควรพิจารณา คือ ข้อมูลที่เก็บจะเป็นลักษณะใด หากเป็นข้อมูลเดียว ก็เก็บเป็น attribute เดียว แต่หากเป็นข้อมูลชุด ก็ให้เก็บเป็น List
- เช่น MovieCatalog จะมี List ของ Movie อยู่
- ความสัมพันธ์แบบ Aggregation จะเห็นได้ว่า หากลบ Instance ของ MovieCatalog แต่ข้อมูล Movie ก็ยังคงอยู่ได้

```
class MovieCatalog:
    def __init__(self, last_update, movies):
        self.last_update = last_update
        self.movie = [] # list of Movie Object
```

Class Diagram : ความสัมพันธ์

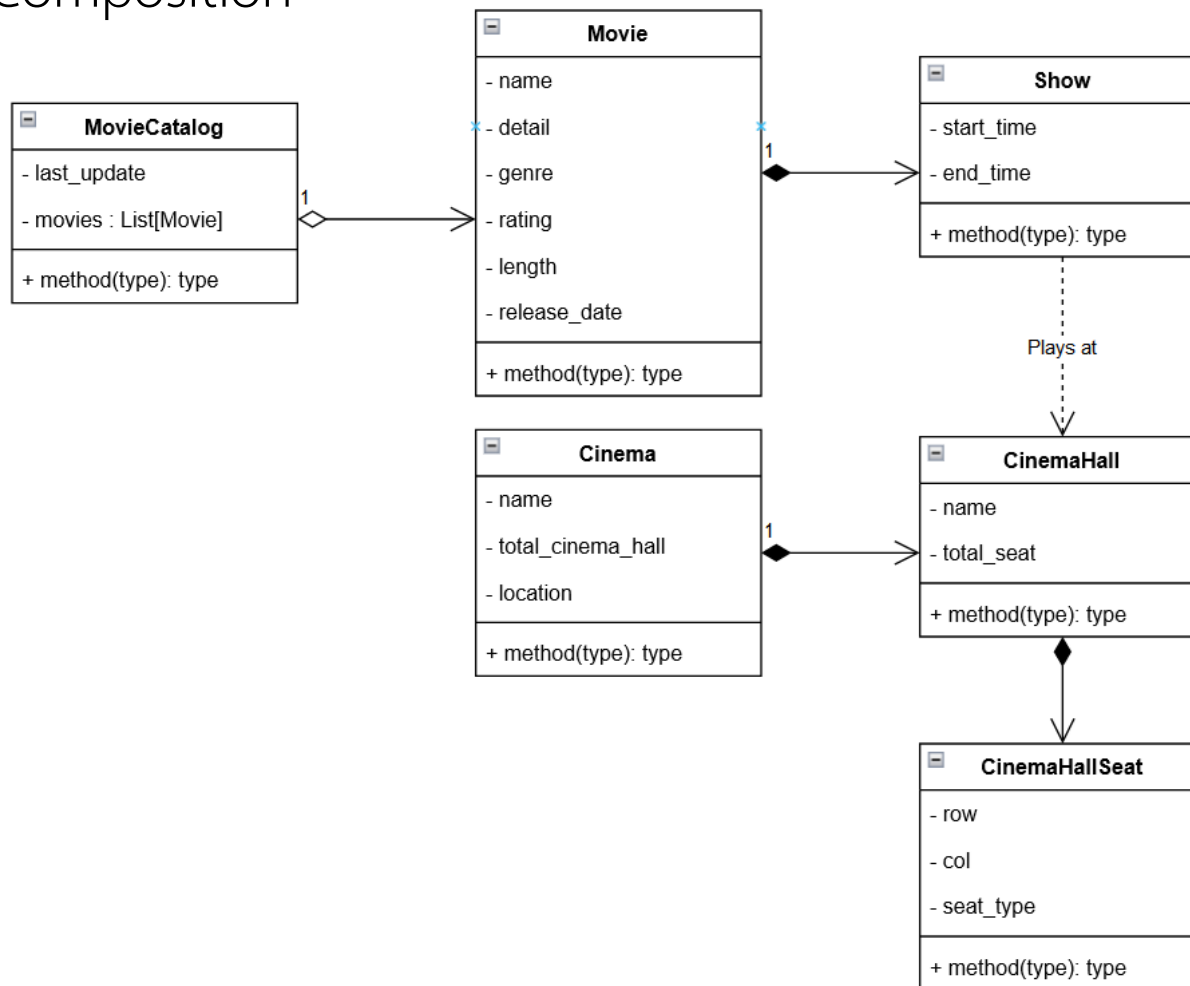


- **Composition** : เป็นความสัมพันธ์ที่คล้ายกับ Aggregation แต่จะใกล้ชิดมากขึ้นอีก โดยความหมาย คือ คลาสหนึ่งเป็นส่วนหนึ่งของอีกคลาสหนึ่ง เช่น คลาส Show เป็นส่วนหนึ่งของคลาส Movie เช่น
 - ภาพยนตร์เรื่อง Avartar และมีรอบฉาย หากลบภาพยนตร์เรื่อง Avatar ไป รอบฉายของ Avatar จะไม่มีความหมาย
 - คลาส CinemaHall เป็นส่วนหนึ่งของ Cineme หากลบ object Cinema ไป CinemaHall ก็จะไม่ได้อยู่ไม่ได้
 - คลาส CinemaHallSeat เป็นส่วนหนึ่งของ CinemaHall อีกที
- ความสัมพันธ์แบบ Composition จะใช้สี่เหลี่ยมขนมเปียกปูนแบบทึบ

Class Diagram : ความสัมพันธ์



- ตัวอย่าง Composition





Class Diagram : ความสัมพันธ์

- จาก Class Diagram ใน slide ก่อนหน้า จะ refactor code โดยเพิ่มการเก็บข้อมูล object ในคลาสหลัก

```
class Cinema:
    def __init__(self, name, total_cinema_hall, location, halls):
        self._name = name
        self._total_cinema_hall = total_cinema_hall
        self._location = location
        self._halls = halls # List of CinemaHall
```

```
class CinemaHall:
    def __init__(self, name, total_seat, seats, shows):
        self._name = name
        self._total_seat = total_seat
        self.seats = seats # List of CinemaHallSeat
        self.shows = shows # List of Show
```



Class Diagram : ความสัมพันธ์

- คลาส Movie จะแตกต่างจากคลาสก่อนหน้านี้ ที่เก็บข้อมูลตั้งแต่สร้างคลาส โดยส่งเป็นพารามิเตอร์เข้ามาใน Constructor แต่คลาส Movie จะสร้างเป็น List ว่างๆ เอาไว้
- เรื่องนี้ไม่ตายตัว ขึ้นกับการออกแบบ และ การเขียนโปรแกรม คือ ถ้ามีข้อมูลอยู่แล้ว ก็ควรสร้างตั้งแต่แรก แต่หากจะเพิ่มเข้ามาภายหลัง ก็สร้างเป็นข้อมูลว่างไว้ก็ได้

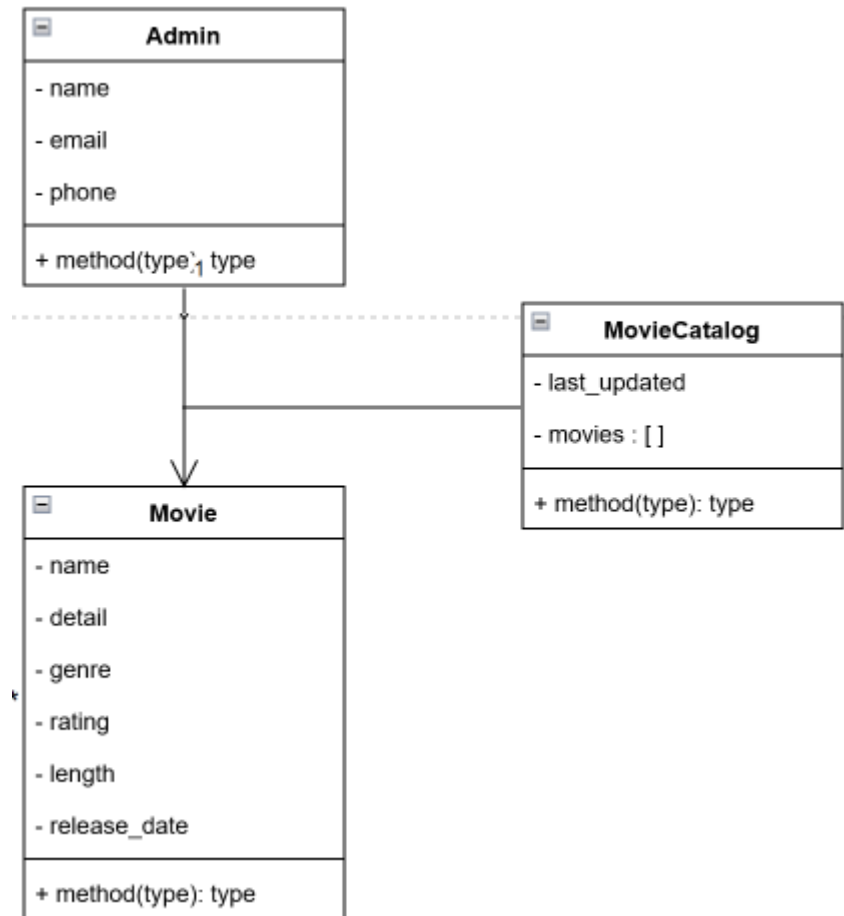
```
class Movie:
    def __init__(self, name, detail, genre, \
                  rating, length, release_date):
        self._name = name
        self._detail = detail
        self._genre = genre          # หมาดหมู
        self._rating = rating
        self._length = length
        self._release_date = release_date
        self._shows = []
```



Class Diagram : ความสัมพันธ์

- ในความสัมพันธ์บางแบบ เช่น ความสัมพันธ์ระหว่าง Admin กับ Movie ซึ่งเป็นความสัมพันธ์แบบ Association คือ Admin เป็นคน “เพิ่ม” ภาพยนตร์เข้าไปในระบบ
- จากที่ได้กล่าวมาว่าความสัมพันธ์แบบ Associative จะต้องมีการเก็บข้อมูลไว้ที่คลาสหนึ่ง แต่ในกรณีนี้ หากเก็บรายชื่อภาพยนตร์ไว้ในคลาส Admin อาจไม่เหมาะสม เพราะภาพยนตร์มีได้เกี่ยวกับ Admin มากนัก
- ในกรณีนี้ จะสร้างคลาสใหม่ขึ้นมาเพื่อเก็บรายชื่อภาพยนตร์ Class ลักษณะนี้เรียกว่า

Associative Class





Class Diagram : Multiplicities

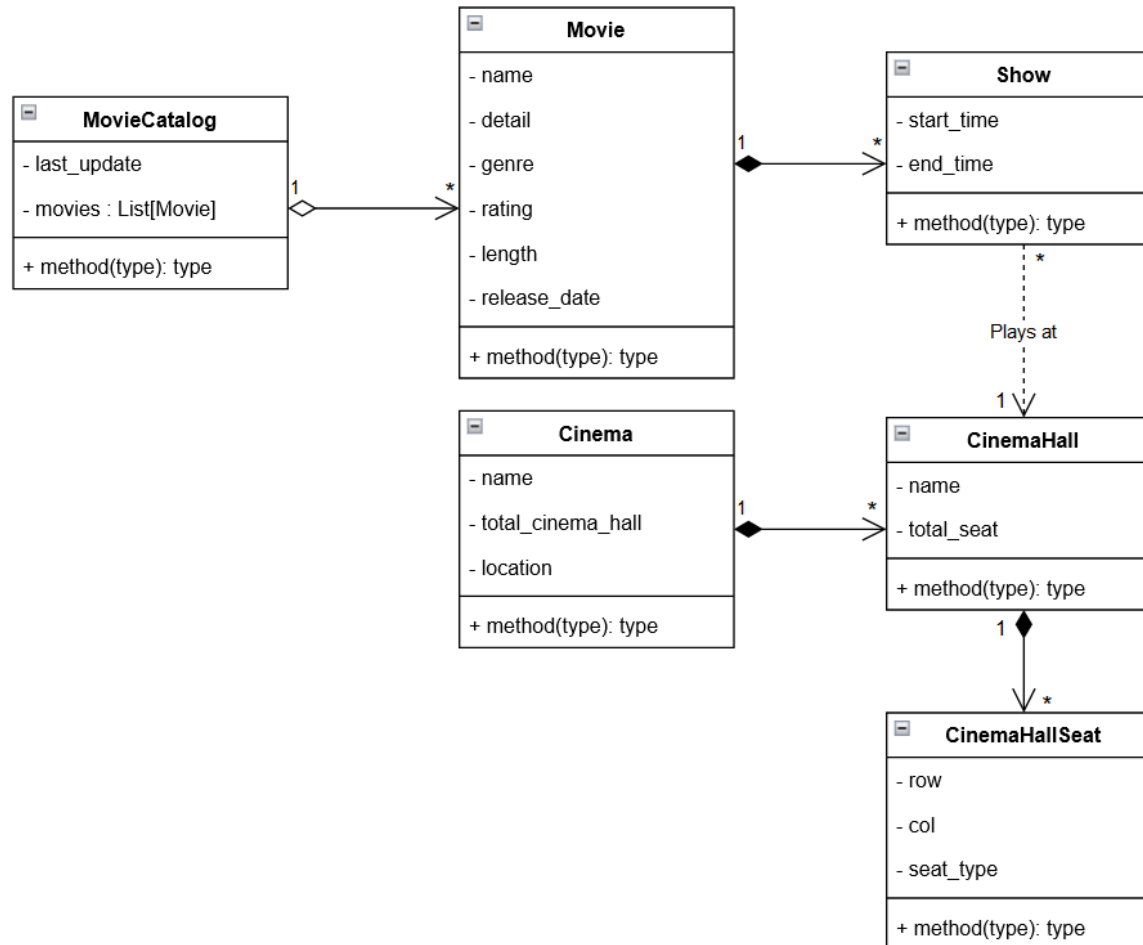
- หลังจากที่กำหนดความสัมพันธ์ระหว่างคลาสแล้ว ก็จะต้องกำหนด Multiplicities คือ ความสัมพันธ์นั้น ในแง่ของจำนวนแล้วเป็นแบบใด

Multiplicities	Meaning
0..1	zero or one instance. The notation <i>n</i> . . <i>m</i> indicates <i>n</i> to <i>m</i> instances.
0..* or *	no limit on the number of instances (including none).
1	exactly one instance
1..*	at least one instance



UML Class Example

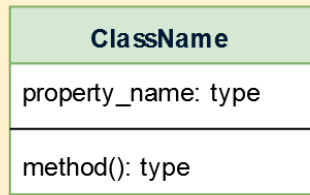
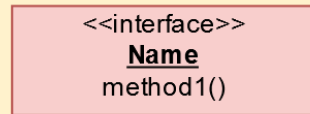
- จาก Class Diagram ก่อนหน้าเมื่อใส่ Multiplicities จะได้ดังนี้



Class Diagram

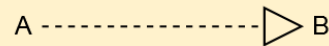


UML conventions

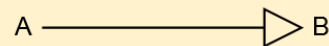


Interface: Classes implement interfaces, denoted by Generalization.

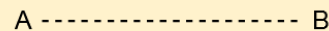
Class: Every class can have properties and methods.
Abstract classes are identified by their *Italic* names.



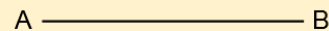
Generalization: A implements B.



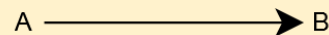
Inheritance: A inherits from B. A "is-a" B.



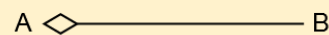
Use Interface: A uses interface B.



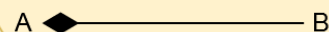
Association: A and B call each other.



Uni-directional Association: A can call B, but not vice versa.



Aggregation: A "has-an" instance of B. B can exist without A.



Composition: A "has-an" instance of B. B cannot exist without A.



Class Attribute

- บางครั้งอาจต้องการเก็บข้อมูลส่วนกลางที่ใช้งานร่วมกันของทุก Instance ในคลาสนั้น
- สามารถทำได้ โดยเรียกว่า Class Attribute ซึ่งเป็น Attribute ของ Class ไม่ใช่ Attribute ของ Instance ใด Instance หนึ่ง
- ปกติเราจะกำหนด Class Attribute ไว้เหนือ `__init__()`

```
class ClassName:  
  
    # Class Attributes  
  
    # __init__()  
  
    # Methods
```



Class Attribute

- การกำหนด Class Attribute จะเหมือนกับกำหนดตัวแปรทั่วไป

```
<class_attribute> = <value>
```

- ตัวอย่าง จะเห็นว่าตัวแปร max_num_items ไม่จำเป็นต้องเขียน self. นำหน้า
- และการอ้างถึงก็ไม่จำเป็นต้องสร้าง Instance ก่อน

```
class Backpack:  
    max_num_items = 10  
  
    def __init__(self):  
        self.items = []
```



Class Attribute

- การอ้างถึง Class Attribute ใช้ dot notation โดยใช้ชื่อ class แล้วตามด้วย Class Attribute ดังรูป

```
<ClassName>.<class_attribute>
```

- เช่น จาก class Backpack ก็สามารถพิมพ์ออกมาได้เลย โดยไม่ต้องสร้าง Instance ก่อน

```
print(Backpack.max_num_items)
```

- แต่เมื่อมีการสร้างเป็น Instance ก็สามารถใช้ `<Instance>.<Class Attribute>` ได้เช่นกัน ได้ค่าเดียวกัน



Class Attribute

- เราสามารถใช้ Class Attribute ในการนับจำนวน Object ได้ เช่น

```
class Movie:
    id_counter = 1

    def __init__(self, title, rating):
        self.id = Movie.id_counter
        self.title = title
        self.rating = rating

        Movie.id_counter += 1

my_movie = Movie("Sense and Sensibility", 4.5)
your_movie = Movie("Legends of the Fall", 4.7)

print(my_movie.id)
print(your_movie.id)
```

```
1
2 _
```



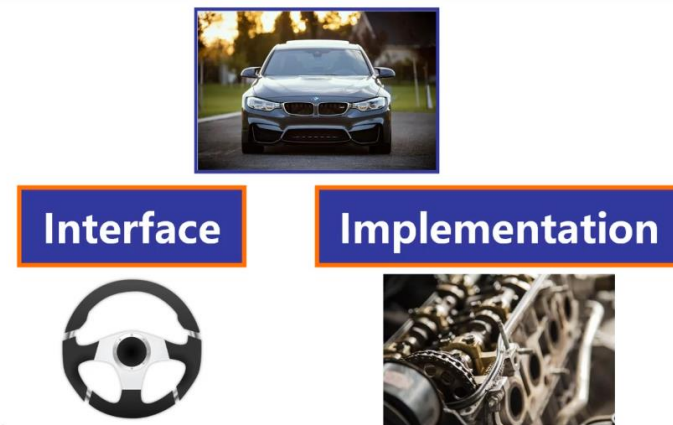
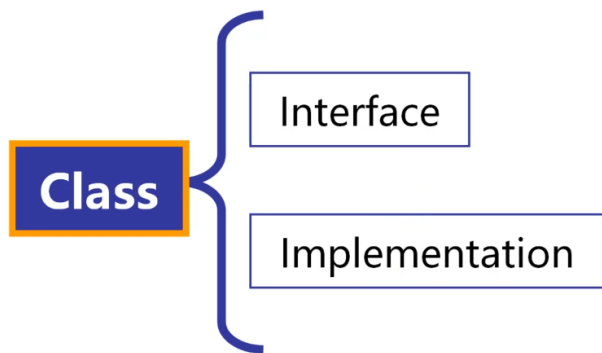
Class Attribute

- ความแตกต่างระหว่าง class attributes และ instance attributes คือ **class attributes** ใช้ร่วมกันในทุก instance ของ class ดังนั้นจึงมีค่าเดียวกัน ในขณะที่ **instance attributes** เป็นของ instance จึงมีค่าเป็นอิสระต่อกัน



Abstraction

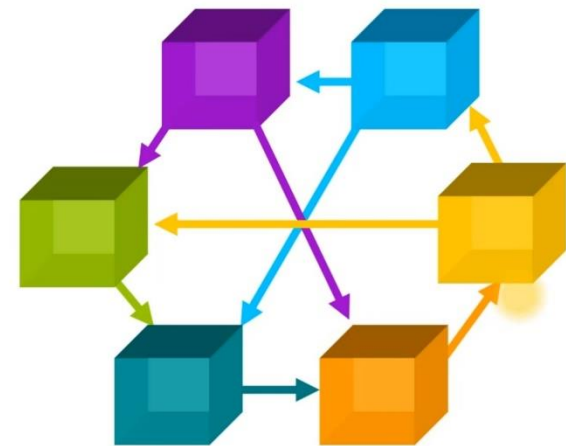
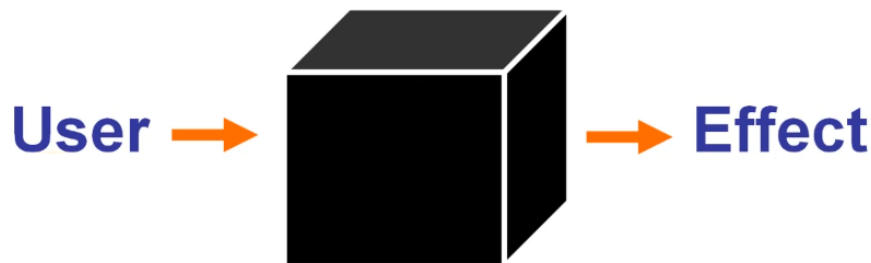
- คำว่า Abstraction มาจากรากศัพท์ Latin 2 คำคือ abs หมายถึง away from และ trahere หมายถึง draw ซึ่งเมื่อแปลรวมกันจะได้ เป็นกรรมวิธีที่กำจัดหรือเอา ลักษณะเฉพาะบางประการออกไป เพื่อให้เหลือเฉพาะลักษณะเฉพาะเท่าที่จำเป็นเท่านั้น
- Abstraction มีนัย 2 ความหมาย ความหมายแรก คือ การ Encapsulation ทำให้ภายนอกมองเห็นเพียงการใช้งานคลาส คือ เห็น Interface แต่ไม่เห็น Implementation





Abstraction

- เราอาจมอง Class หรือ Object เป็น กล่องดำ ที่เห็นเฉพาะผลการทำงาน เมื่อใส่ Input หรือ message เข้าไป
- การมองเช่นนี้ เป็นการแยกโปรแกรมออกเป็นส่วนๆ โดยแต่ละส่วนจะมีความทำงานเบ็ดเสร็จของตนเอง เมื่อเรียกใช้แต่ละส่วน (Module) ก็ทำผ่าน Interface โดยไม่จำเป็นต้องทราบวิธีการทำงานภายใน ดังนั้นการทำงานอาจมองได้ว่าเป็น Message ที่ส่งระหว่าง Object





Abstraction

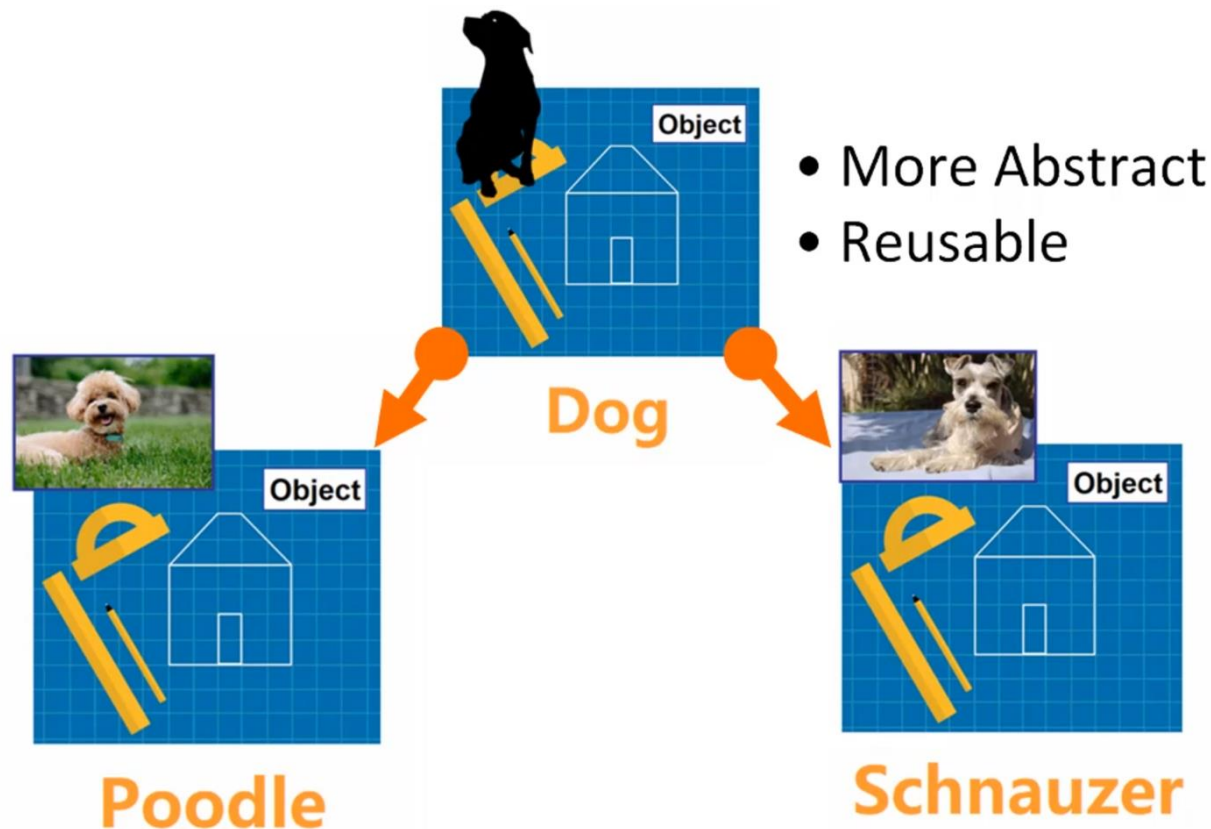
- ทำให้สามารถเปลี่ยนแปลงการทำงานภายใน ของแต่ละส่วนได้โดยไม่กระทบกับโปรแกรม ส่วนอื่น ตราบใดที่ Interface ยังคงเหมือนเดิม





Abstraction

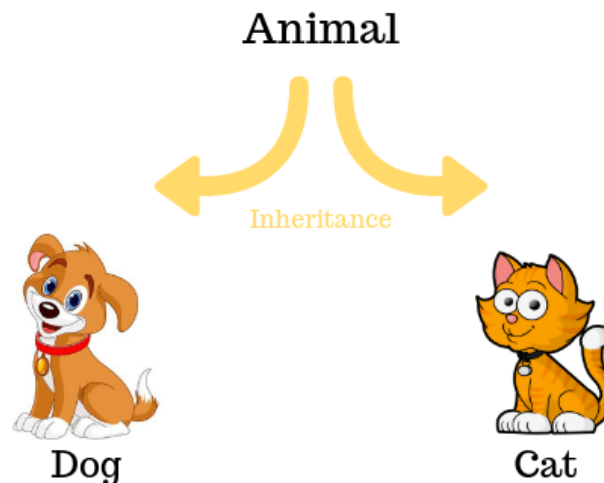
- นัยของคำว่า Abstraction ที่ 2 หมายถึง การนำสิ่งที่คล้ายกันมารวมกัน เพื่อลดความซ้ำซ้อน ซึ่งเป็นแนวคิดที่ทำให้เกิด Inheritance





Inheritance

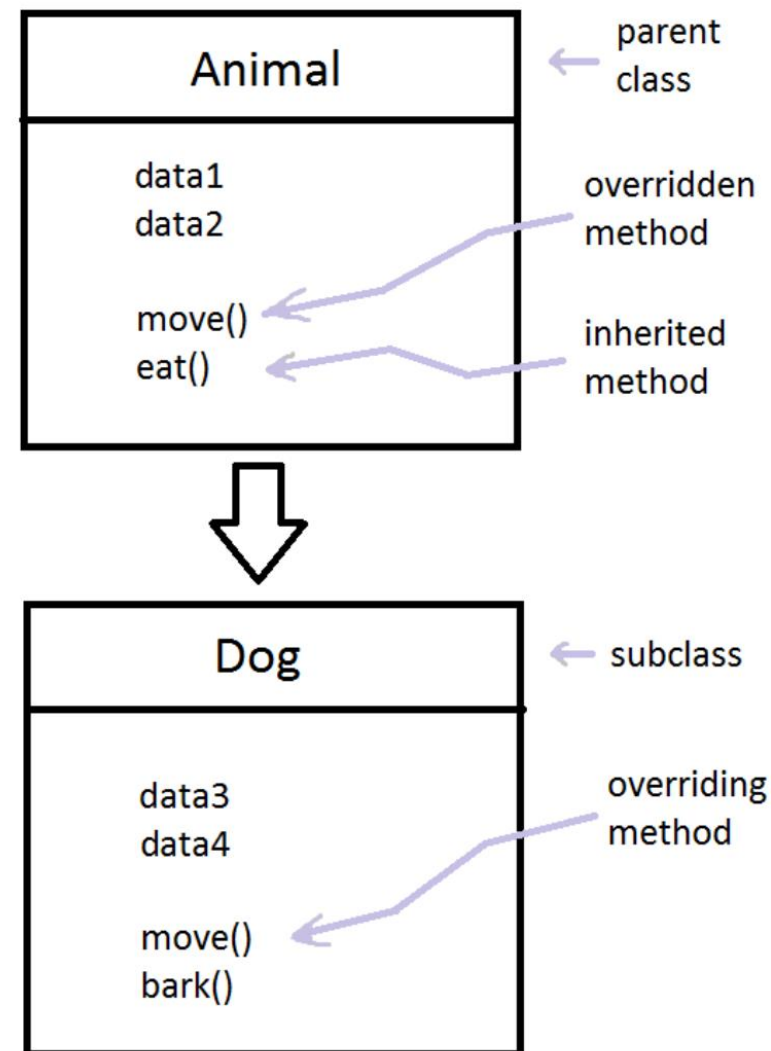
- Inheritance เป็น 1 ใน 4 คุณสมบัติหลักของ Object Oriented Programming
- Inheritance คือ ความสามารถในการสืบทอดคุณสมบัติจาก Class อื่น (เรียก Class ที่สืบทอดว่า Superclass และเรียกตัวเองว่า Subclass (บางครั้งเรียก Parent/Child))
- จากรูป Animal คือ Superclass และ Dog กับ Cat เป็น Subclass



Inheritance



- ประโยชน์ของ Inheritance
 - ลดความซ้ำซ้อนของ Code (หลักการเขียนโปรแกรม คือ เมื่อมี code ที่ซ้ำกันหรือคล้ายกัน ให้หาทางลด)
 - Reuse Code
 - ทำให้ Code อ่านได้ง่ายขึ้น
- จากรูป ถ้าเพิ่มสัตว์ชนิดอื่นๆ ก็จะทำให้ง่าย และกำหนดเฉพาะคุณลักษณะที่เพิ่มเติมเข้ามา





Inheritance

- Class ที่จะ Inherit จาก Class อื่น มีหลักดังนี้
 - ต้องเป็น (“is”) subset ของ Super Class เช่น ถ้า Super Class คือ Car แล้ว Subclass สามารถเป็น Trunk ได้ เพราะรถบรรทุก “เป็น” รถยนต์ประเภทหนึ่ง แต่มอเตอร์ไซค์ เป็น Subclass ไม่ได้
 - Subclass จะต้องมีการกำหนดลักษณะเฉพาะเพิ่มเติม เช่น รถบรรทุก อาจมี นน. บรรทุก พูดโดยรวม คือ Super Class จะมีลักษณะ “ทั่วไป” แต่ Subclass มีลักษณะ “เฉพาะ” เพิ่ม
- Class หนึ่ง อาจ Inherit จากหลาย Class ได้ เรียกว่า **Multiple Inheritance** (บางภาษาไม่มีคุณลักษณะนี้) และ Class ก็ถูก Inherit จากหลาย Class ได้เช่นกัน



Inheritance

- จากคลาสด้านล่าง จะเห็นว่ามีข้อมูลหลายข้อมูลที่ซ้ำ และเป็นข้อมูลพนักงานเช่นกัน

```
class Programmer:
    salary = 100000
    monthly_bonus = 500

    def __init__(self, name, age, address, phone, programming_languages):
        self.name = name
        self.age = age
        self.address = address
        self.phone = phone
        self.programming_languages = programming_languages

class Assistant:
    salary = 100000
    monthly_bonus = 500

    def __init__(self, name, age, address, phone, bilingual):
        self.name = name
        self.age = age
        self.address = address
        self.phone = phone
        self.bilingual = bilingual
```



Inheritance

- จะเห็นว่าเมื่อใช้ Inheritance จะทำให้ซ้ำซ้อนน้อยลง และ โครงสร้างดีขึ้น

```
# Superclass
class Employee:
    salary = 100000
    monthly_bonus = 500

    def __init__(self, name, age, address, phone):
        self.name = name
        self.age = age
        self.address = address
        self.phone = phone

class Programmer(Employee):
    def __init__(self, name, age, address, phone, programming_languages):
        Employee.__init__(self, name, age, address, phone)
        self.programming_languages = programming_languages

class Assistant(Employee):
    def __init__(self, name, age, address, phone, bilingual):
        Employee.__init__(self, name, age, address, phone)
        self.bilingual = bilingual
```


Inheritance



- รูปแบบการใช้งาน Inheritance

```
class Superclass:  
    pass  
  
class Subclass(SuperClass)  
    pass
```

- เมื่อ Inherit มาจากคลาสใด ให้ใส่วงเล็บต่อท้ายเอาไว้
- เนื่องจากทุกคลาสใน python จะ Inherit มาจากคลาส Object ดังนั้นใน Python เวอร์ชันเก่า จะวงเล็บ Object ต่อท้ายหมดทุกคลาสแต่ในเวอร์ชันหลังๆ ได้ตัดออกเพื่อให้ดูง่าย



Inheritance

- การ Inheritance มีข้อดีที่สามารถจะเพิ่ม Subclass ที่คล้ายกัน ได้โดย เช่น สมมติว่ามีคลาส Polygon และ Inherit โดยคลาส Triangle หากจะมีการเพิ่มคลาสอื่นๆ เช่น Square ก็ไม่ต้องไปแก้ไข Code ในส่วนคลาส Polygon และ Triangle
- ตัวอย่าง

```
class Polygon:  
    pass  
  
class Triangle(Polygon):  
    pass
```

- เพิ่ม Class

```
class Ractangle(Polygon):  
    pass
```

Inheritance

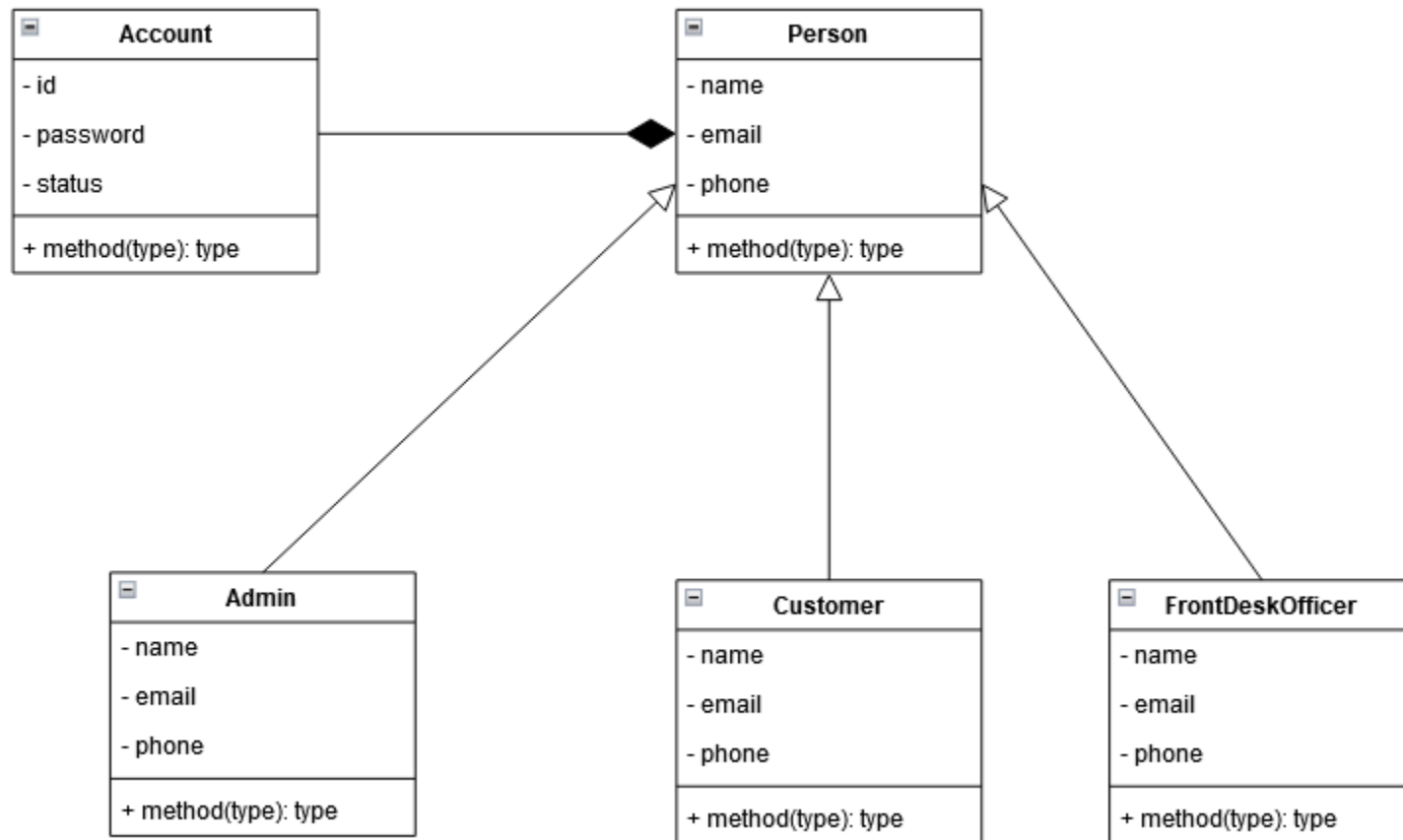


- มีหลักการออกแบบคลาสข้อหนึ่งมีชื่อว่า **O**pen-Closed Principle
- หลักการข้อนี้มีอยู่ว่า ส่วนประกอบของ Software ควรจะ Close สำหรับการแก้ไข แต่ Open สำหรับการเพิ่มเติม
- หมายความว่าหลังจากที่ Software เขียนเสร็จแล้ว ไม่ควรมีการแก้ไขใดๆ อีก กรณีของ Class คือ ไม่ไปแตะต้องคลาสนั้นอีก กรณีที่มีการเพิ่มเติม ก็ควรใช้วิธีการ Inheritance มากกว่าจะไปแก้ไขที่ Class เดิม
- หลักการข้อนี้ เป็นความพยายามในการหลีกเลี่ยงการแก้ไข Code เดิม โดยหากมีการแก้ไขใดๆ ก็ให้สืบทอดจากคลาส และเพิ่มเติมแทนการแก้ไขคลาสเดิม ทั้งนี้เพื่อให้การดูแลรักษาซอฟต์แวร์สามารถทำได้ง่ายขึ้น



Inheritance

- ตัวอย่าง ในระบบโรงพยาบาลส่วนที่สามารถใช้ Inheritance คือ Admin, Customer, Front Desk Officer ซึ่งสามารถ Inherit มาจาก Person





Inheritance

- มีหลักการออกแบบคลาสข้อหนึ่งมีชื่อว่า Dependency Inversion Principle
- เนื้อหาของหลักการข้อนี้ คือ

“High level modules should not depend upon low level modules. Both should depend upon abstractions.”

“Abstractions should not depend upon details, details should depend upon abstractions.”

- “ของที่เป็น High level module ไม่ควรไปผูกติดกับ Low level module และทั้งสองควรรู้จักกันในรูปแบบ abstraction เท่านั้น” กับ “Abstraction ไม่ควรรู้รายละเอียดการทำงาน แต่โค้ดที่ทำงานที่แท้จริงต้องทำตาม Abstraction ที่วางไว้”
- จากรูปจะเห็นว่า Account มีความสัมพันธ์กับ Person แทนที่จะสัมพันธ์กับ Subclass ของ Person จึงเป็นไปตามหลักการนี้

Inheritance



- การแยกคลาส Person ออกมาจะทำให้ความซ้ำซ้อนของข้อมูลลดลง โดยข้อมูลเหมือนกันจะอยู่ในคลาส Person และข้อมูลที่แตกต่างกันจะอยู่ใน Subclass ของแต่ละประเภทย่อย
- การทำเช่นนี้ มีข้อดี ที่ทำให้การปรับเปลี่ยนในอนาคตสามารถทำได้โดยมีการแก้ไข Code เดิมน้อยลง หากมีการเปลี่ยนแปลงโดยรวมก็แก้ไขเพียงคลาส Person คลาสเดียว หรือ หากมีการเปลี่ยนแปลงย่อย ก็เพียงแต่สร้าง Subclass ใหม่ขึ้นมา
- ขอยกตัวอย่าง หากในอนาคตมีการเพิ่มผู้ใช้ประเภทใหม่ขึ้นมา เช่น โรงภาพยนตร์อาจกำหนดผู้ใช้แบบรายเดือน โดยใน 1 เดือนสามารถดูภาพยนตร์ได้ 10 เรื่อง หากใช้วิธี Inherit จะทำให้ไม่ต้องไปแก้ไข Code เดิม โดย Code ที่มีการเพิ่มเติม ก็จะอยู่ในคลาสที่สร้างเพิ่มเติมขึ้นมาใหม่



Inheritance

- ข้อมูลอีกส่วนในระบบโรงภาพยนตร์ที่ใช้กับ Inheritance ได้ คือ คลาส CinemaHallSeat กับ ShowSeat ซึ่งเป็นคลาสที่นิ่งเหมือนกัน แต่คลาสหนึ่งเป็นที่นั่งของโรง แต่อีกคลาสเป็นที่นั่งที่มีการจอง
- ซึ่งไม่ควรจะมี 2 คลาสที่มีข้อมูลเดียวกัน

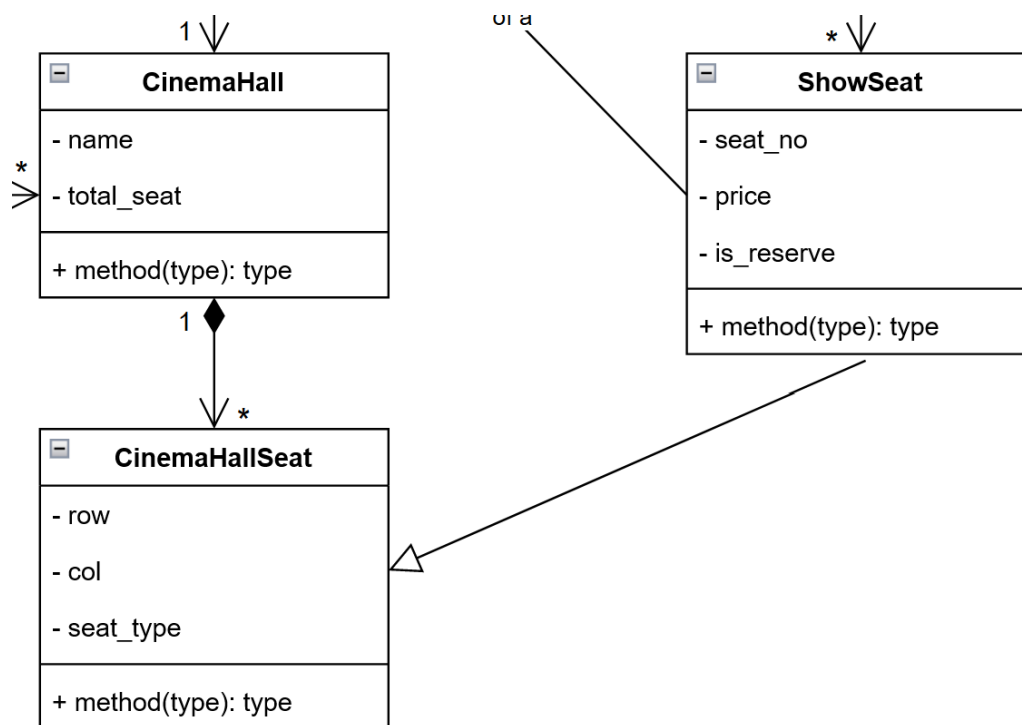
```
class CinemaHallSeat:  
    def __init__(self, seat_row, seat_col, seat_type):  
        self.seat_row = seat_row  
        self.seat_col = seat_col  
        self.seat_type = seat_type
```

```
class ShowSeat:  
    def __init__(self, seat_no, price):  
        self.seat_no = seat_no  
        self.price = price  
        self.is_reserve = None
```

Inheritance



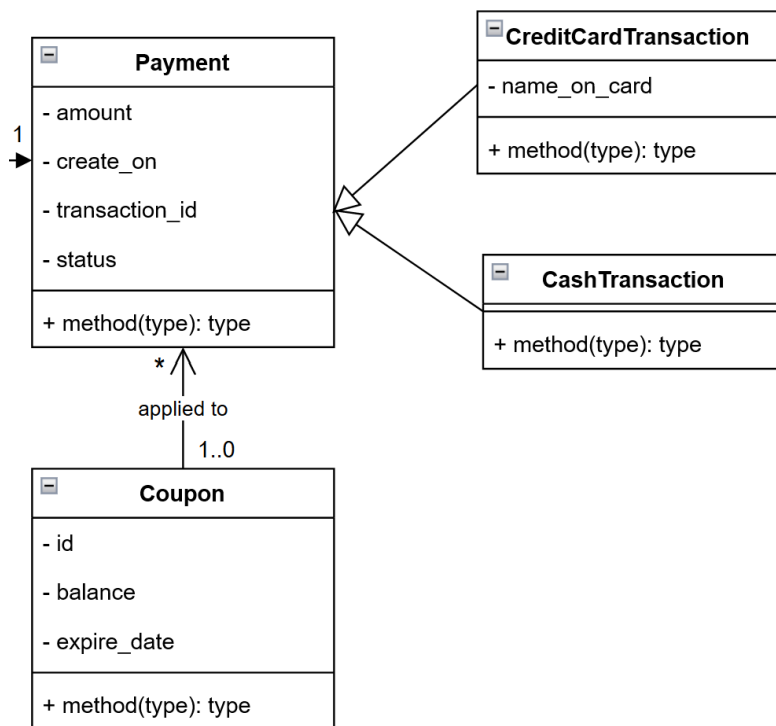
- แต่หากจะรวมเป็นข้อมูลคลาสเดียวกัน จะขัดแย้งในเรื่องของ cohesion เพราะข้อมูลการจอง ไม่ได้มีความเกี่ยวข้องกับที่นั่ง
- ดังนั้นในกรณีนี้ การใช้ Inheritance จะมีความเหมาะสมมากกว่า ดังนี้



Inheritance



- ข้อมูลอีกส่วนหนึ่งที่เราควรใช้ Inheritance คือ ข้อมูลการชำระเงิน ซึ่งมีวิธีชำระเงินหลายวิธี แต่ละวิธีก็มีรายละเอียดการทำงานในแบบของตัวเอง
- หากมีช่องทางการชำระเงินวิธีอื่นๆ เพิ่มในอนาคตก็เพียงแค่เพิ่ม Inherit Class เข้าไป





Inheritance

- คลาสที่ Inherit มากจากคลาสอื่น และใน Subclass ไม่มี Constructor จะใช้ Constructor ของ Superclass แทน

```
class Polygon:

    def __init__(self, num_sides, color):
        self.num_sides = num_sides
        self.color = color

class Triangle(Polygon):
    pass

my_triangle = Triangle(3, "Blue")

print(my_triangle.num_sides)
print(my_triangle.color)
```

3
Blue



Inheritance

- แต่หาก Subclass มี Constructor ของตนเอง ก็จะไม่ใช้ Constructor ของ Superclass

```
class Polygon:

    def __init__(self, num_sides, color):
        self.num_sides = num_sides
        self.color = color

class Triangle(Polygon):

    def __init__(self, base, height):
        self.base = base
        self.height = height

my_triangle = Triangle(3, "Blue")

print(my_triangle.num_sides)    # Error
print(my_triangle.color)       # Error
```



Inheritance

- แต่หากจะให้ subclass ไปเรียกใช้ Constructor ของ Superclass จากนั้นจึงเรียกใช้ Constructor ของคลาสตัวเองจะเขียนดังนี้

```
class Triangle(Polygon):  
  
    NUM_SIDES = 3  
  
    def __init__(self, base, height, color):  
        super().__init__(Triangle.NUM_SIDES, color)  
        self.base = base  
        self.height = height  
  
my_triangle = Triangle(5, 4, "blue")  
  
print(my_triangle.num_sides)  
print(my_triangle.color)  
print(my_triangle.base)  
print(my_triangle.height)
```



Inheritance

- ให้ subclass ไปเรียกใช้ Constructor ของ Superclass อีกวิธี (มี self)

```
class Triangle(Polygon):  
  
    NUM_SIDES = 3  
  
    def __init__(self, base, height, color):  
        Polygon.__init__(self, Triangle.NUM_SIDES, color)  
        self.base = base  
        self.height = height  
  
my_triangle = Triangle(5, 4, "blue")  
  
print(my_triangle.num_sides)  
print(my_triangle.color)  
print(my_triangle.base)  
print(my_triangle.height)
```



***For your
attention***