01076105, 01076106

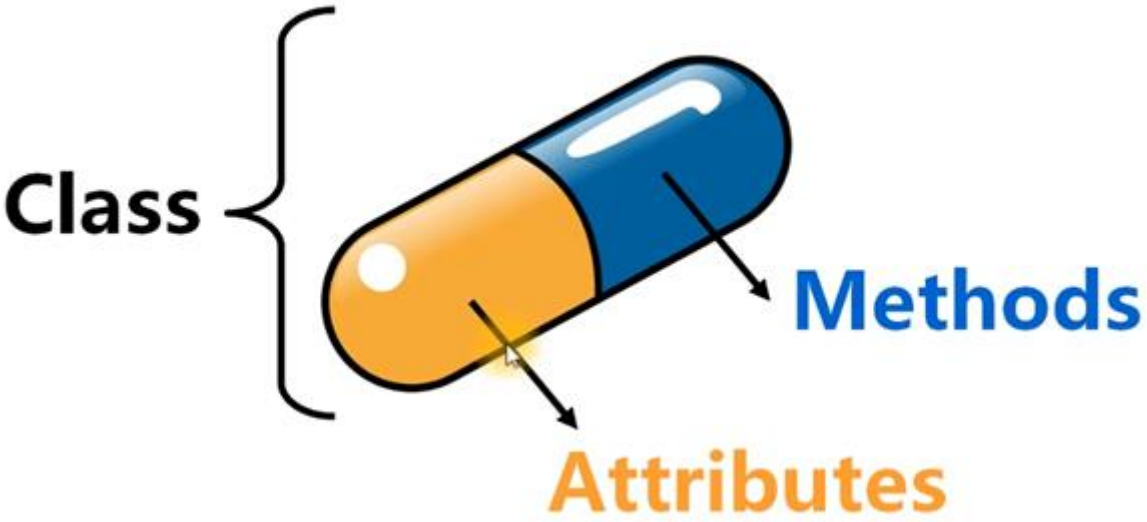
Object Oriented Programming

Object Oriented Programming Project

Encapsulation, Class Diagram

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

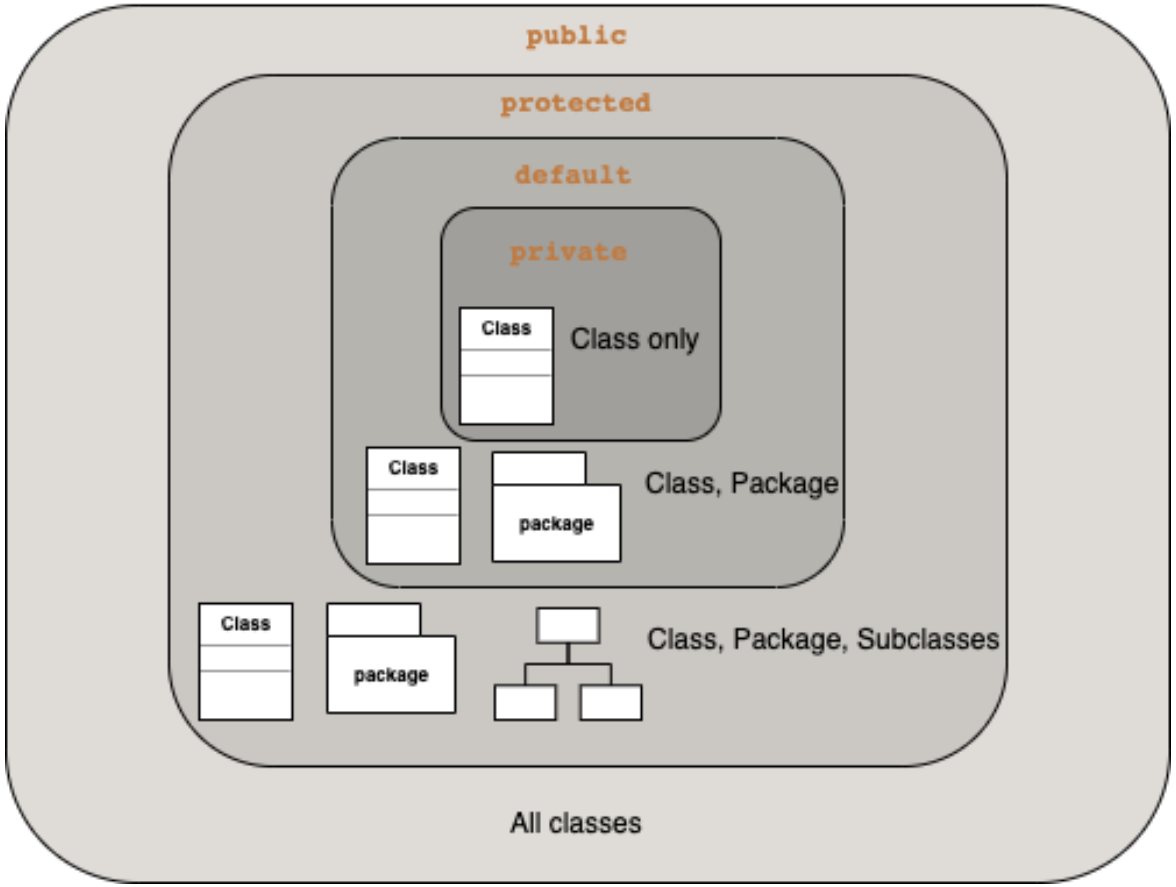
```
<object>.<attribute> = <new_value>
```

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 โดยภาษาเองจะทำหน้าที่ป้องกันการเข้าถึงตามที่กล่าวมาก
- แต่ภาษา Python ไม่ได้ใช่วิธี 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

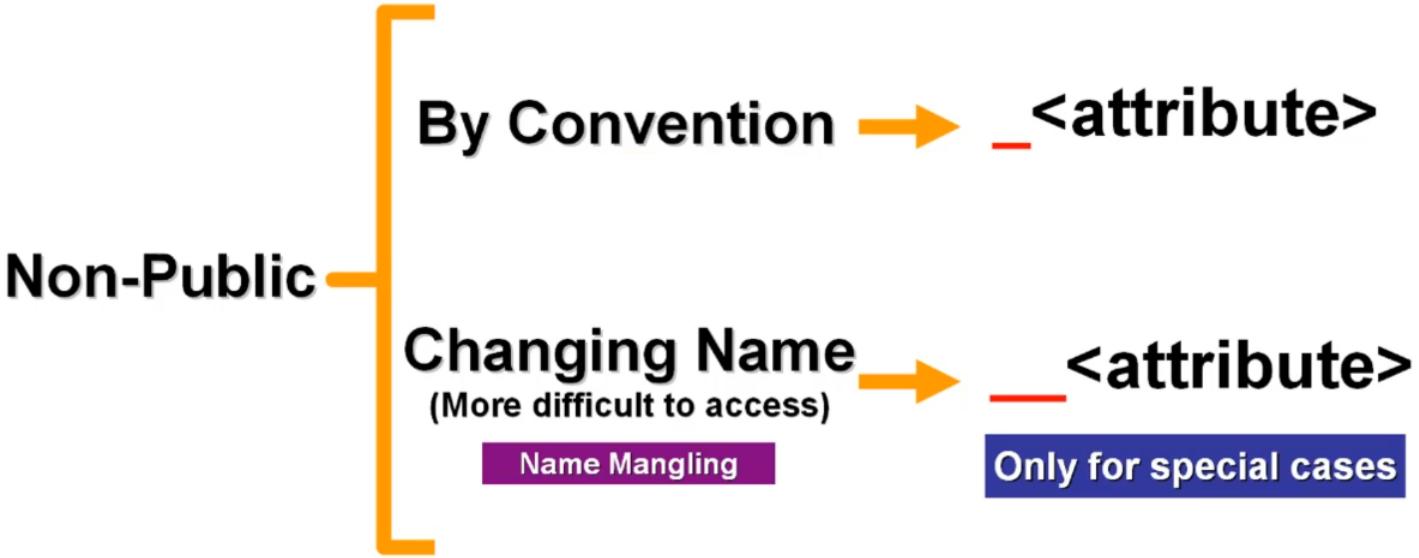
main.py

```
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 คือข้อมูลที่เราสนใจจะจัดเก็บและนำมาใช้ในระบบ

เมธอด (Method)

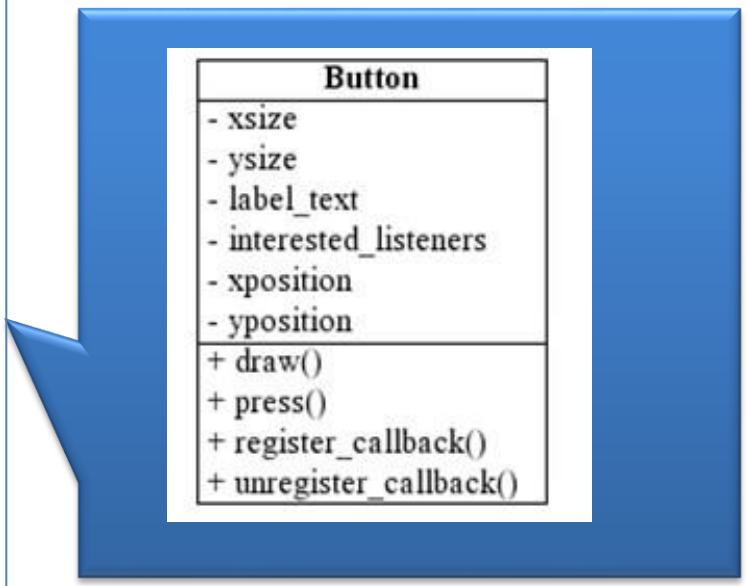
เมธอด คือการทำงานที่คลาสสามารถทำงานได้

ระดับของการเข้าถึงข้อมูล

(+) public ให้คลาสอื่น ๆ ใช้งานข้อมูลนี้ได้อิสระ

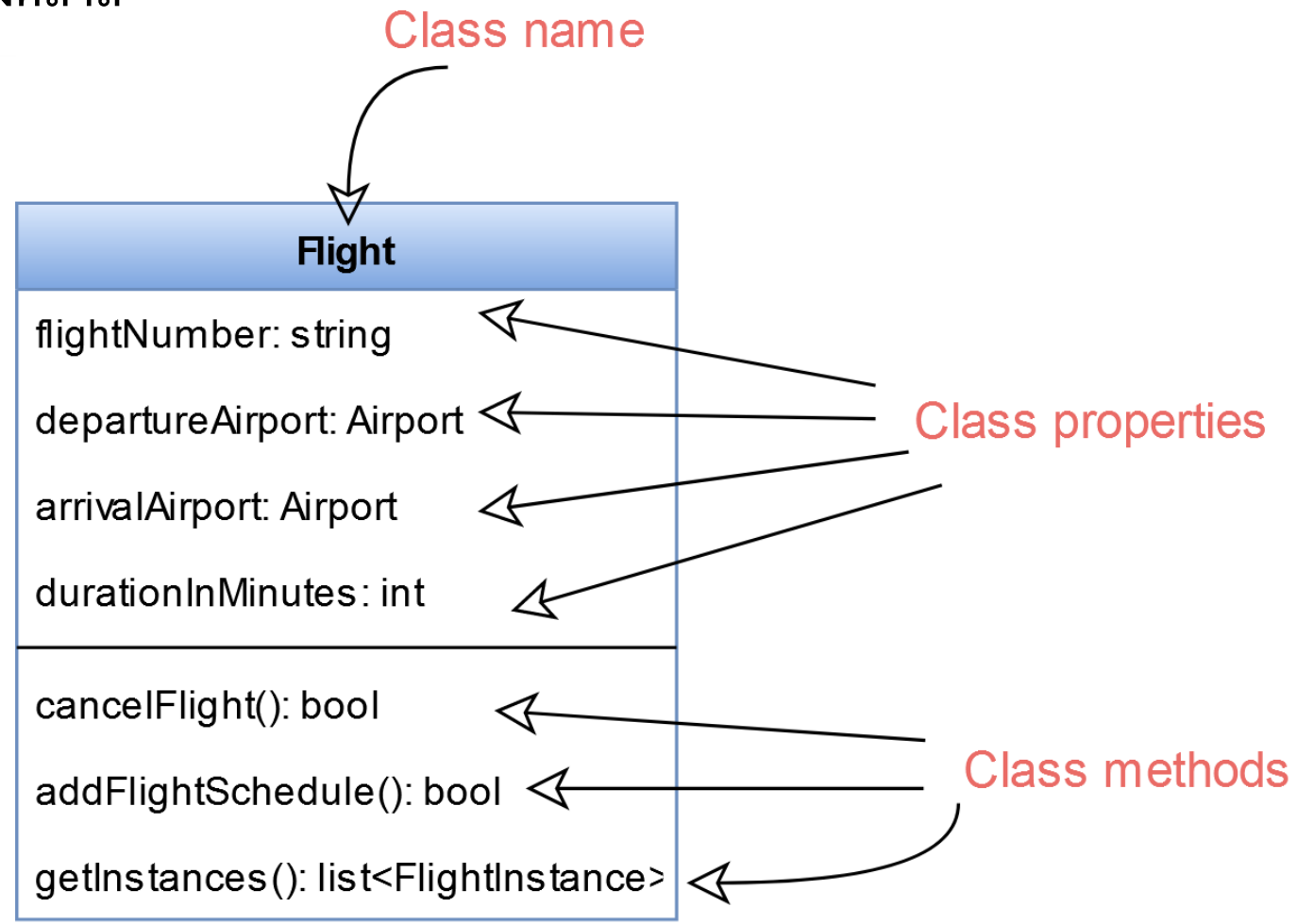
(#) protected ให้เฉพาะคลาสที่สืบทอดใช้งานได้

(-) private ไม่อนุญาติให้คลาสอื่นใช้งานได้



Class Diagram

- โครงสร้างคลาส

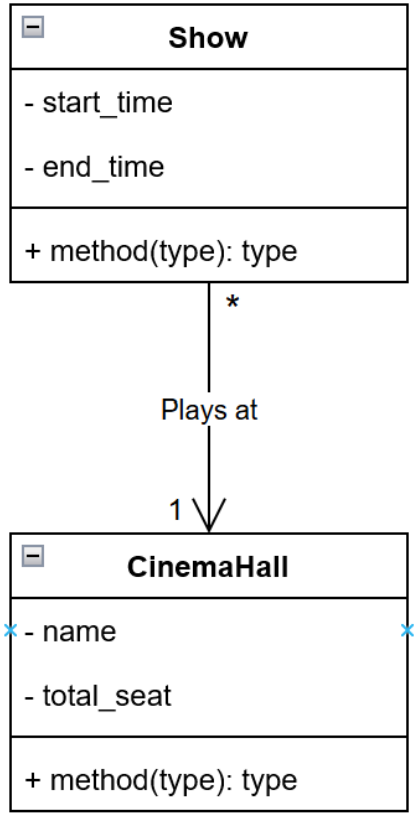


Class Diagram : ความสัมพันธ์

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

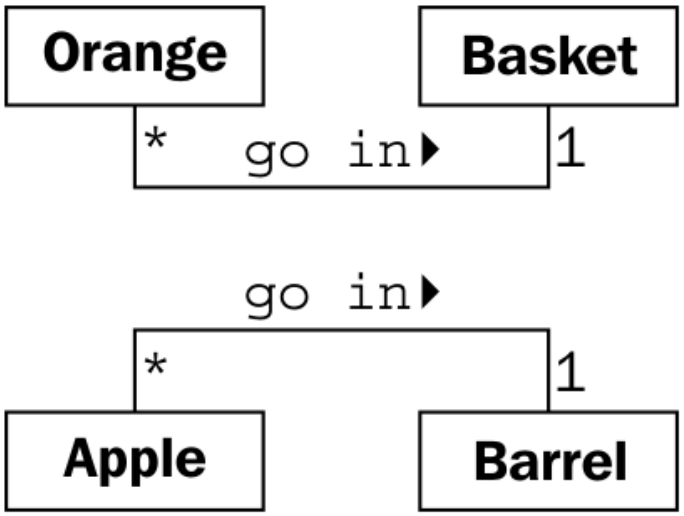
Class Diagram : ความสัมพันธ์

- 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 แอปเปิลในลัง และ ส้ม กับ แอปเปิล ก็จะต้องมีข้อมูลว่าอยู่ในลังใด

Class Diagram : ความสัมพันธ์

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

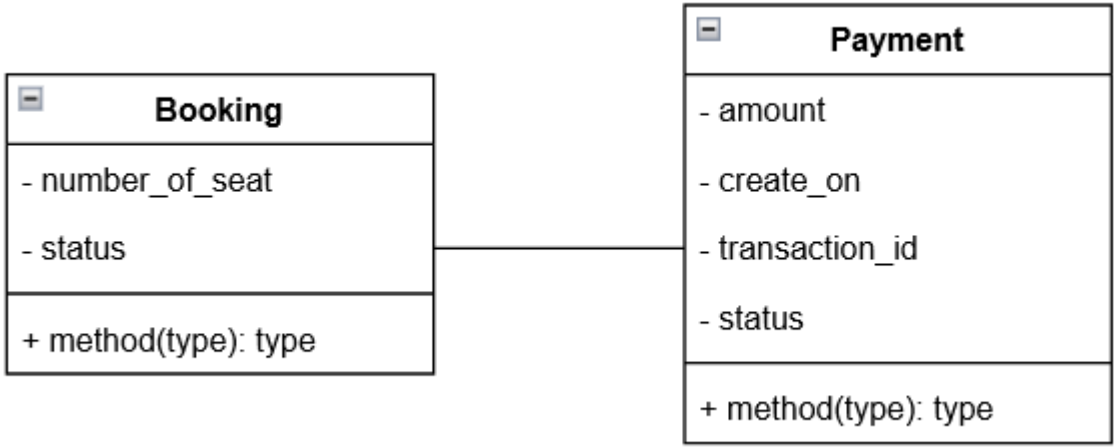
- และคลาส Show ก็ต้องบอกว่าฉายที่ไหน ดังนั้นคลาส show ก็จะต้องแก้ไขดังนี้

```python
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
- ก็จะทำให้ข้อมูลการฉายภาพยนตร์ครบถ้วน คือ ภาพยนตร์ เวลา และ โรงฉาย

Class Diagram : ความสัมพันธ์

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



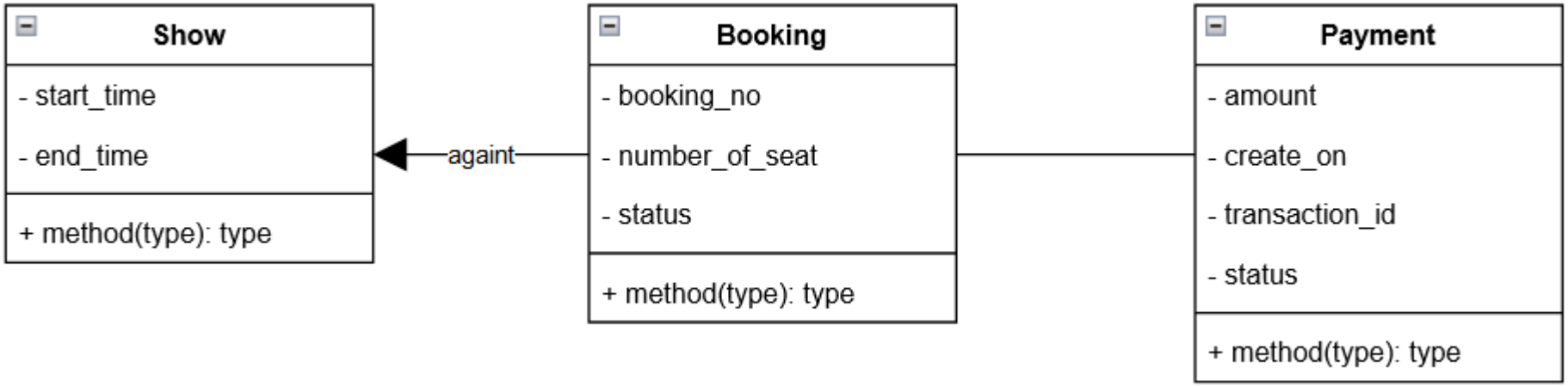
Class Diagram : ความสัมพันธ์

- 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



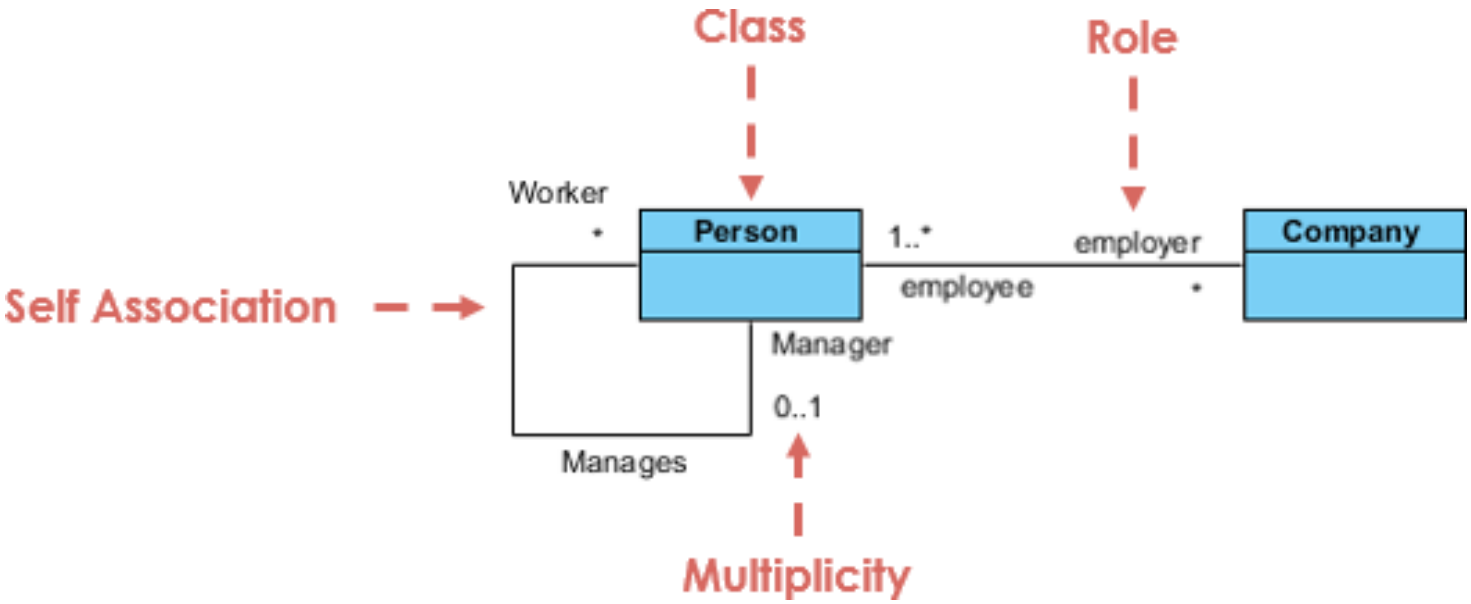
Class Diagram : ความสัมพันธ์

- 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
```

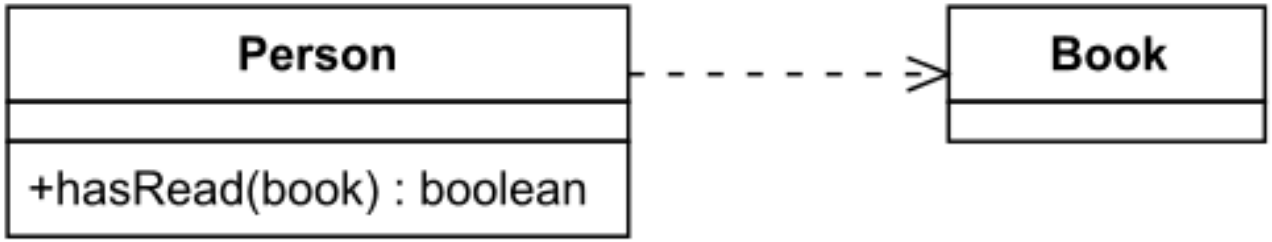
Class Diagram : ความสัมพันธ์

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



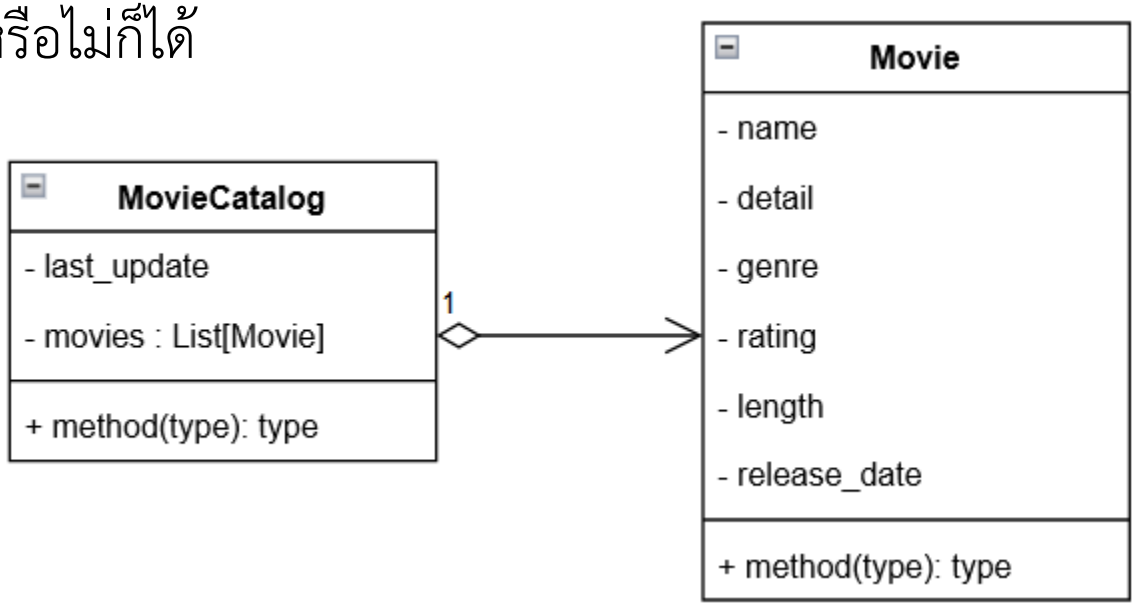
Class Diagram : ความสัมพันธ์

- **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

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



Class Diagram : ความสัมพันธ์

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

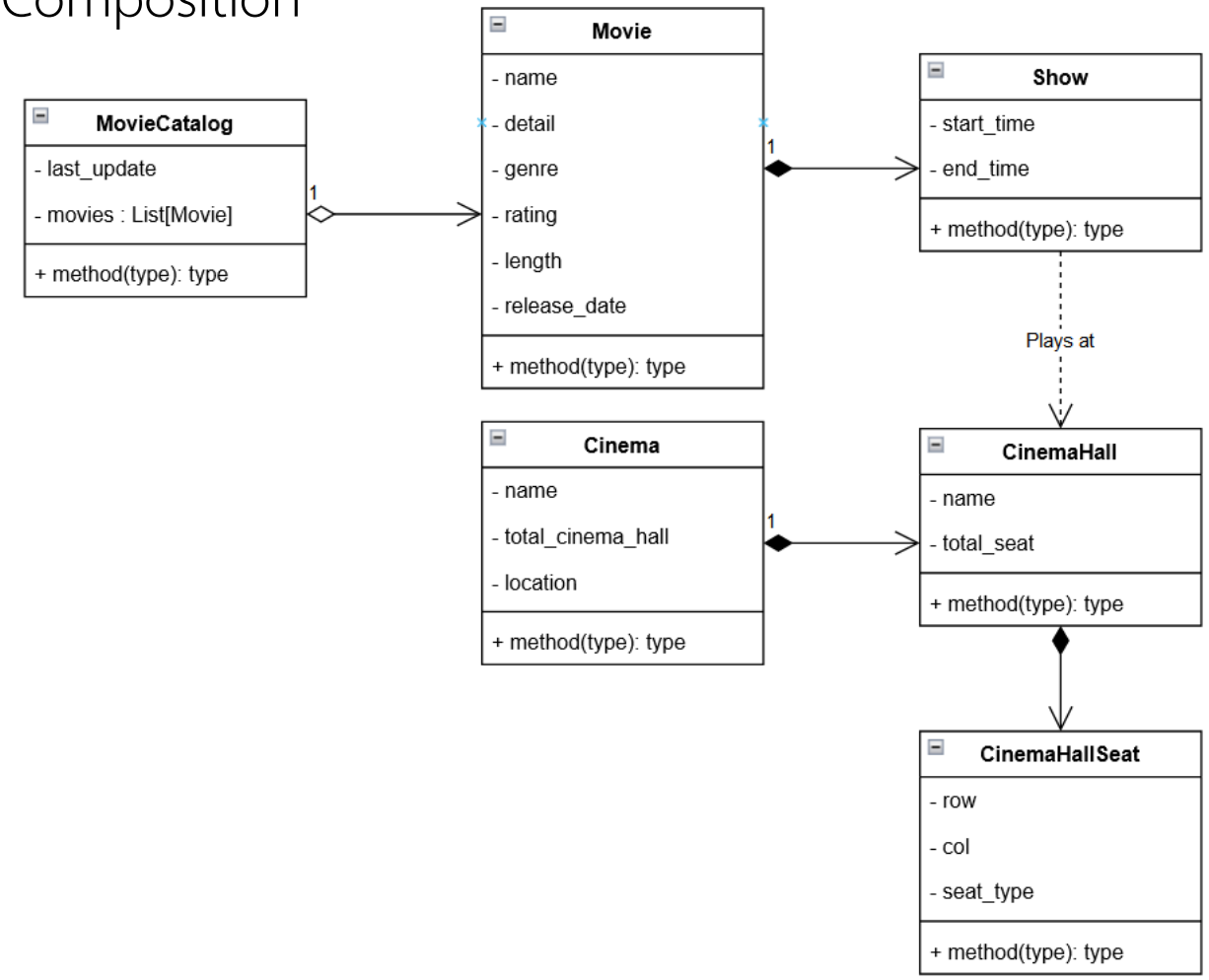
```
class MovieCatalog:
    def __init__(self, last_update, movies):
        self.last_update = last_update
        self.movie = []    # list of Movie Object
```

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 ก็จะอยู่ไม่ได้
  - 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_cenema_hall, location, halls):
        self._name = name
        self._total_cenema_hall = total_cenema_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
```

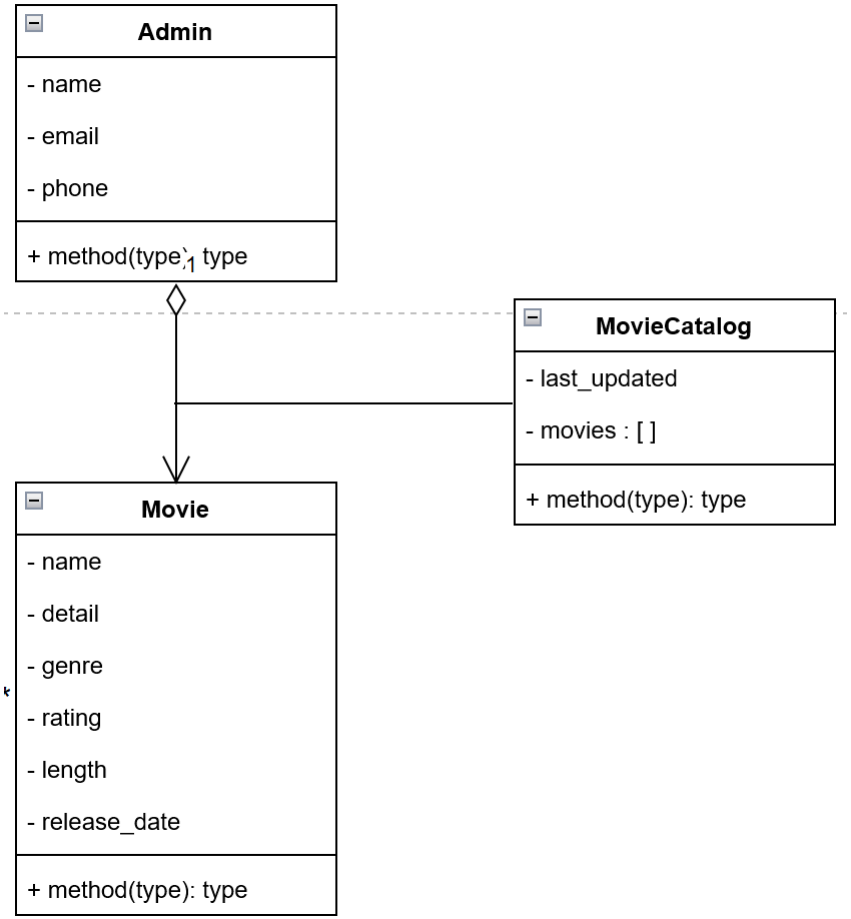
Class Diagram : ความสัมพันธ์

- คลาส Movie จะแตกต่างจากคลาสก่อนหน้า 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        # หมวดหมู่
        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 เพราะภาพยนตร์มิได้เกี่ยวกับ Admin มากนัก
- ในกรณีนี้ จะสร้างคลาสใหม่ขึ้นมาเพื่อเก็บรายชื่อภาพยนตร์ Class ลักษณะนี้เรียกว่า **Associative Class**



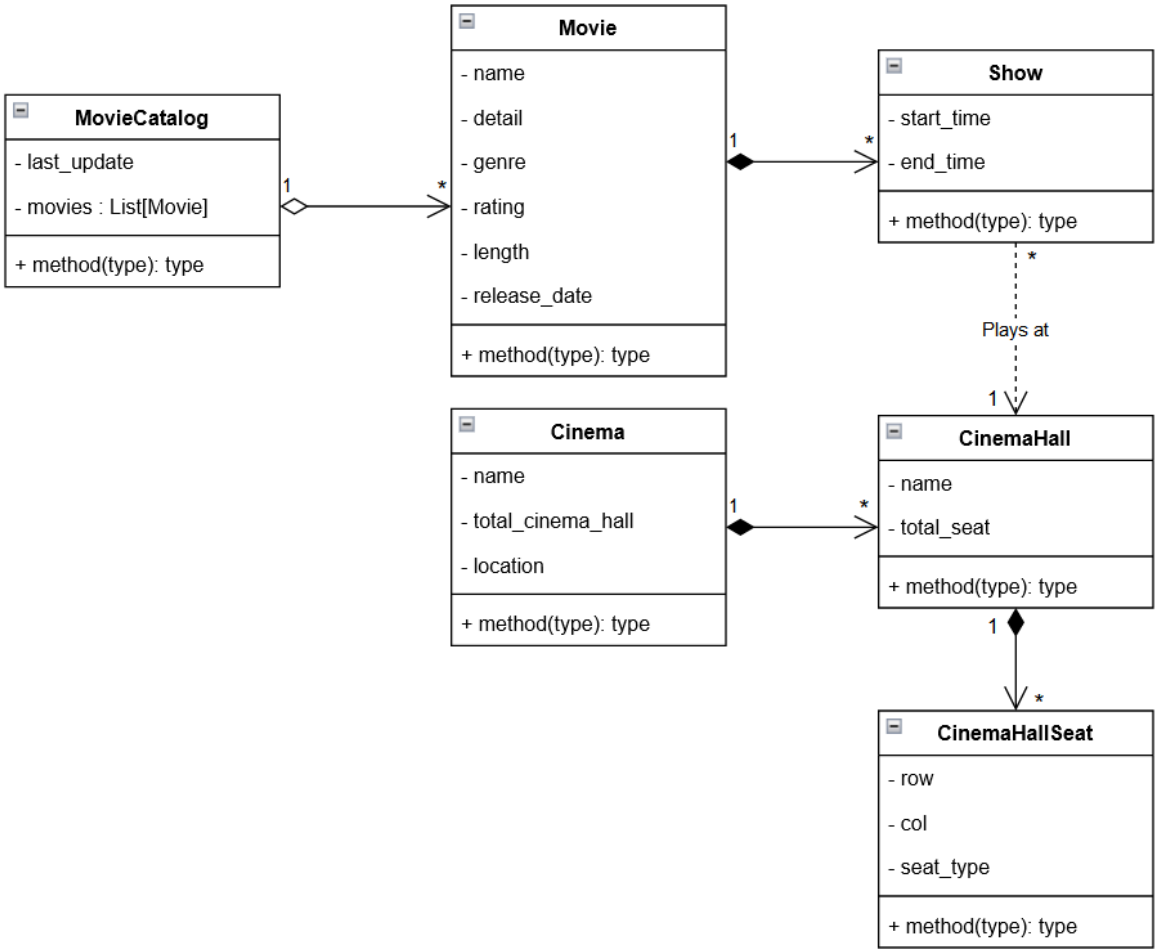
Class Diagram : Multiplicities

- หลังจากที่กำหนดความสัมพันธ์ระหว่างคลาสแล้ว ก็จะต้องกำหนด Multiplicities คือ ความสัมพันธ์นั้น ในแง่ของจำนวนแล้วเป็นแบบใด

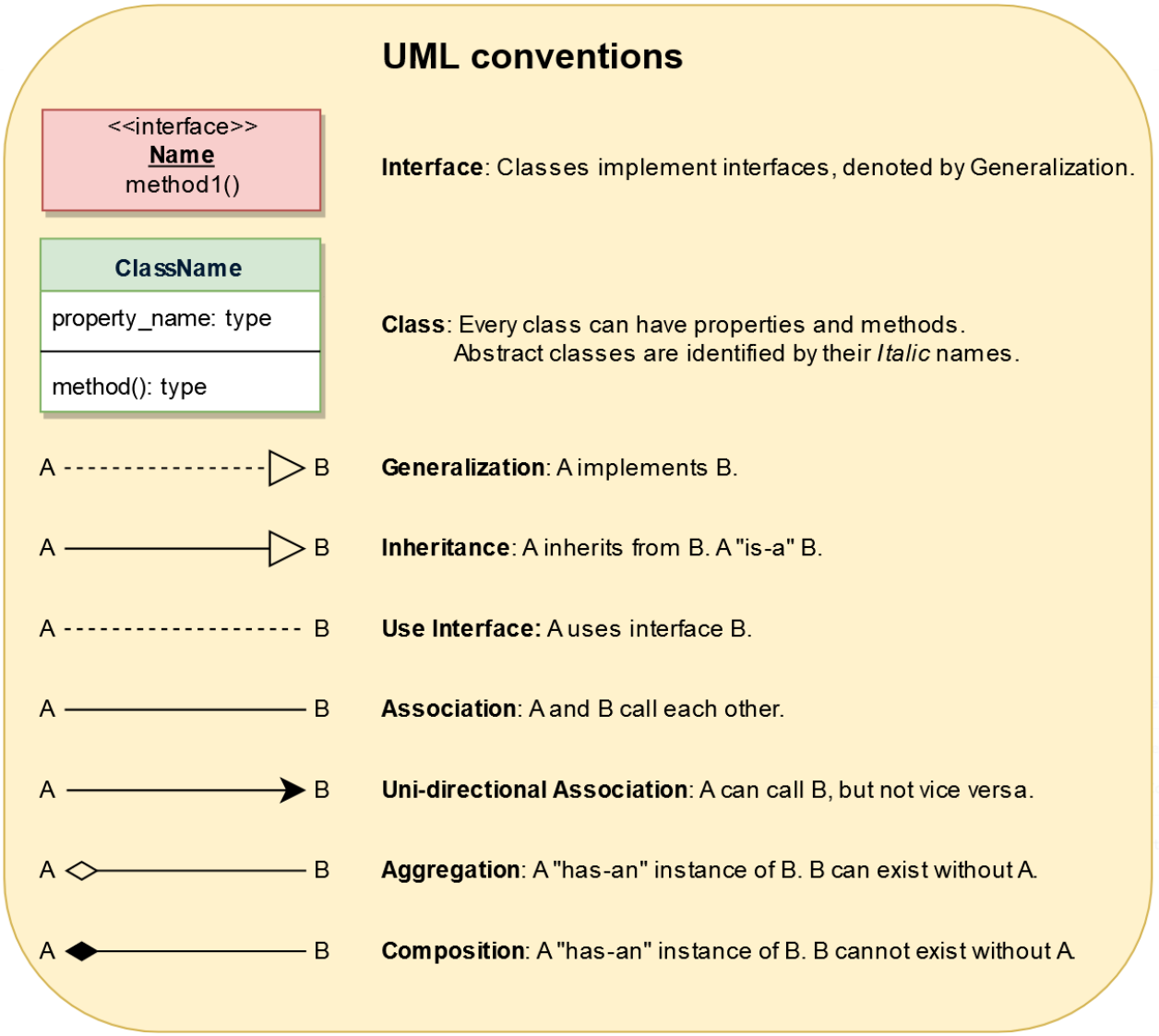
Multiplicities	**Meaning**
0..1	zero or one instance. The notation ***n . . m*** indicates ***n*** to ***m*** instances.
0..* *or* *	no limit on the number of instances (including none).
1	exactly one instance
1..*	at least one instance

UML Class Example

- จาก Class Diagram ก่อนหน้าเมื่อใส่ Multiplicities จะได้ดังนี้



Class Diagram



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
- และการอ้างถึงก็ไม่จำเป็นจะต้องสร้าง Instance ก่อน

```python
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

```python
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("Inception", 8.8)
your_movie = Movie("Legends of the fall", 7.5)
print(Movie.id_counter)
```

Class Attribute

- ความแตกต่างระหว่าง class attributes และ instance attributes คือ

  class attributes ใช้ร่วมกันในทุก instance ของ class ดังนั้นจึงมีค่าเดียวกัน ในขณะที่ **instance attributes** เป็นของ instance จึงมีค่าเป็นอิสระต่อกัน

Class Attribute : Quiz

- จากตัวอย่างนี้ อะไร คือ Class Attribute

```
1  class Bird:
2
3      num_wings = 2
4      flies = True
5
6      def __init__(self, name, species, color, sings):
7          self.name = name
8          self.species = species
9          self.color = color
10         self.sings = sings
```

Class Attribute : Quiz

- จากรูป ข้อใดถูก

```
1  class Rainbow(object):
2
3      num_colors = 5
4
5      def __init__(self, location, latitude, longitude, rating):
6          self.location = location
7          self.latitude = latitude
8          self.longitude = longitude
```

- ○ `Rainbow_num_colors = 6`
- ○ `num_colors = 6`
- ○ `Rainbow.num_colors = 6`

Class Attribute : Quiz

- การกำหนดค่า class attributes ถูกต้องหรือไม่

```
1  class A:
2
3      self.attr1 = 5
4
5      def __init__(self, param1, param2):
6          self.param1 = param1
7          self.param2 = param2
8
9      # Methods
```

Class Attribute : Quiz

- หลังจากโปรแกรมทำงานแล้ว ค่าของ class attributes attr1 เป็นเท่าไร

```
1  class A:
2
3      attr1 = 5
4
5      def __init__(self):
6          A.attr1 += 1
7
8  a1 = A()
9  a2 = A()
10 A.attr1 = 26
11 a3 = A()
12 print(A.attr1) # Output?
```

For your attention