01076105, 01076106

Object Oriented Programming

Object Oriented Programming Project

Class Diagram, Inheritance

Class Diagram

- ในงานที่มีความซับซ้อนมากขึ้น การเขียนโปรแกรม object oriented ไปเลย อาจทำความเข้าใจได้ยาก จึงมีการแทนโครงสร้างของระบบในรูปแบบ diagram ซึ่งจะมองเห็นภาพของงานได้ดีขึ้น (visualize)
- Diagram ที่แทนโครงสร้างของระบบงาน ที่พัฒนาด้วย object oriented จะเรียกว่า Class Diagram โดยส่วนประกอบของ Class Diagram ที่สำคัญมี 2 ส่วน
  - Class ต่างๆ
  - ความสัมพันธ์ระหว่างคลาสเหล่านั้น

Class Diagram

- Class Notation คือ ภาพที่แสดงถึงส่วนประกอบของ Class ซึ่งมี 2 ส่วน
  - attribute เป็นส่วนที่ทำหน้าที่เก็บข้อมูลที่แสดงคุณสมบัติ
  - Mathod เป็นส่วนที่ทำหน้าที่แสดงฟังก์ชันหรือการทำงานของคลาสนั้น

แอตทริบิวต์ (Attribute)

แอตทริบิวต์คือข้อมูลที่เป็นคุณสมบัติของคลาส คือ ข้อมูลที่เราสนใจจะจัดเก็บและนำมาใช้ในระบบ

เมธอด (Method)

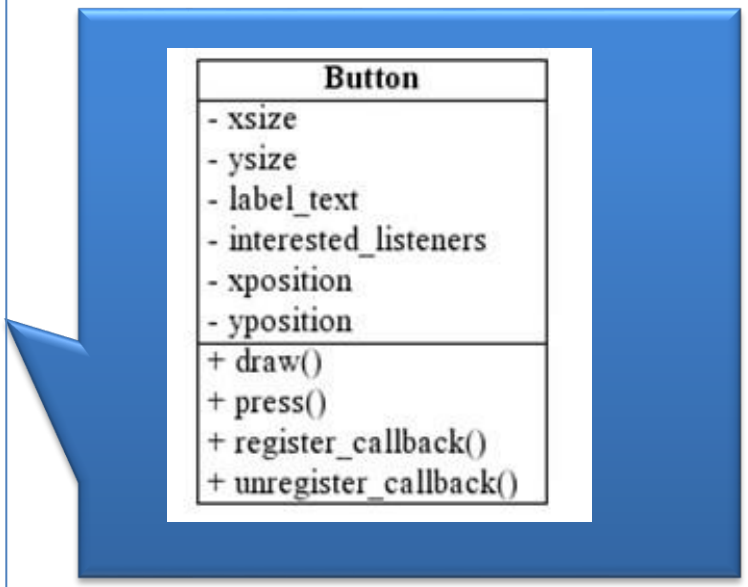
เมธอด คือการทำงานที่คลาสสามารถทำงานได้

ระดับของการเข้าถึงข้อมูล

(+) public ให้คลาสอื่น ๆ ใช้งานข้อมูลนี้ได้อิสระ

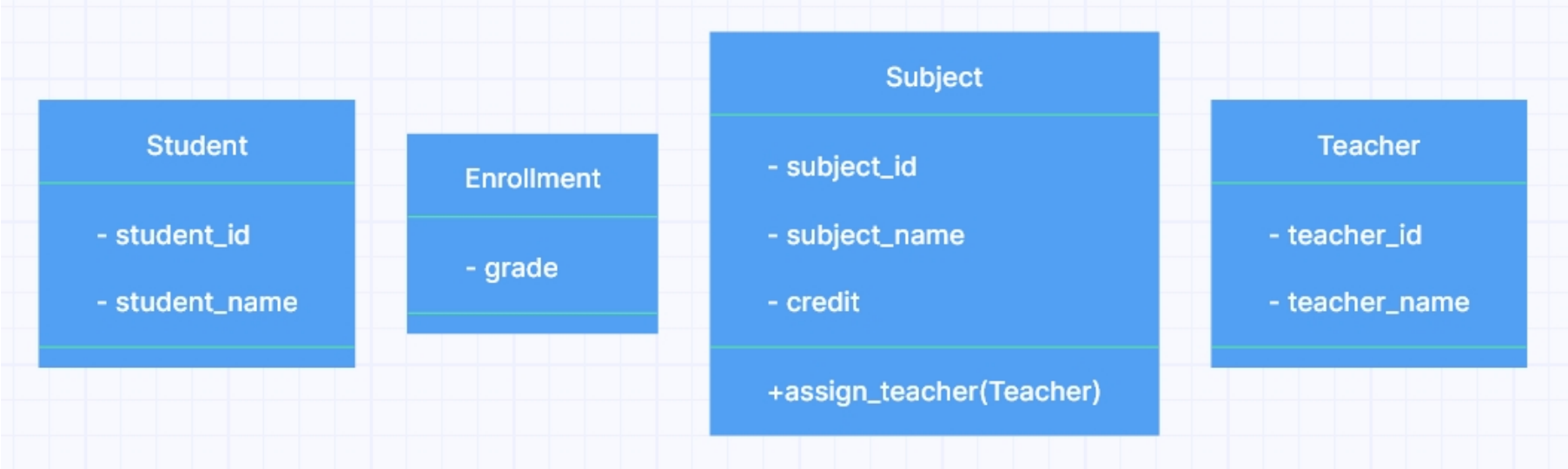
(#) protected ให้เฉพาะคลาสที่สืบทอดใช้งานได้

(-) private ไม่อนุญาติให้คลาสอื่นใช้งานได้



Class Diagram

- โดยระบบข้อมูลการเรียน สามารถเขียนเป็นรูปคลาสได้ดังนี้
- จะเห็นได้ว่าการเขียนเป็น class diagram ทำให้มองเห็นโครงสร้างคลาสได้ดีขึ้น สามารถเห็น attribute ของ class และ method ของคลาส
- โดยทั่วไปมักจะเขียน class diagram ก่อนจึงค่อยเขียนโปรแกรม

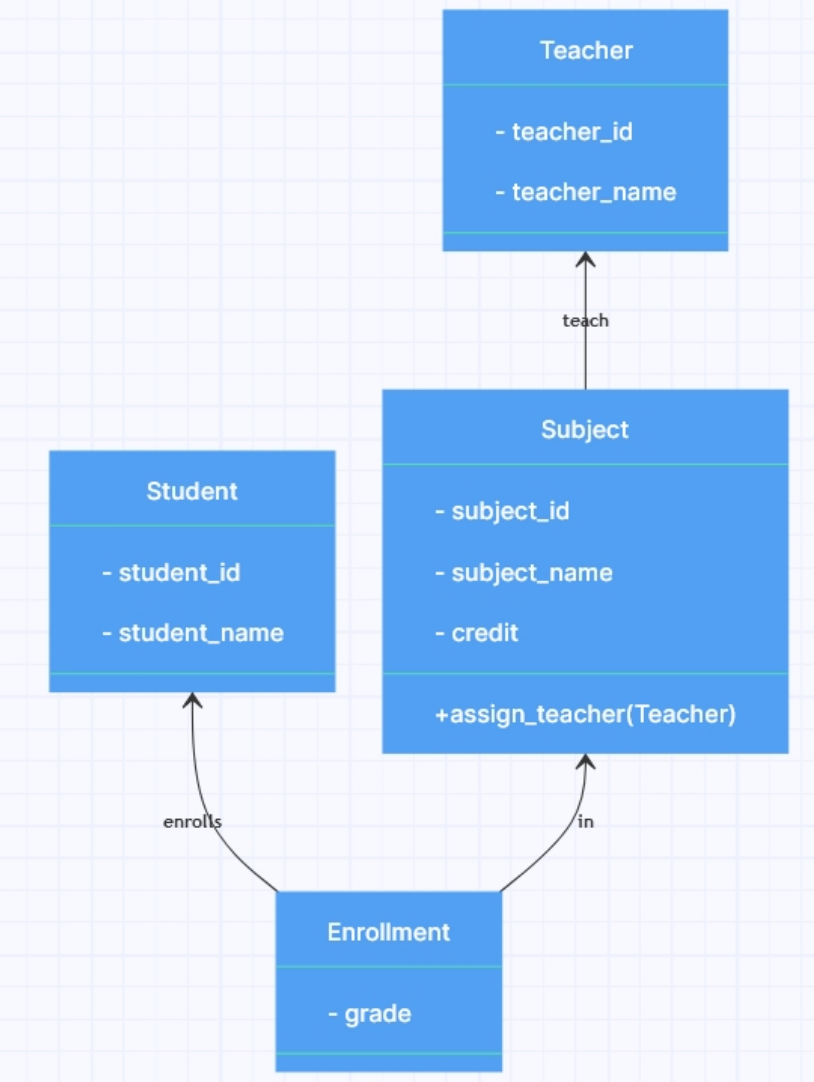


Class Diagram : ความสัมพันธ์

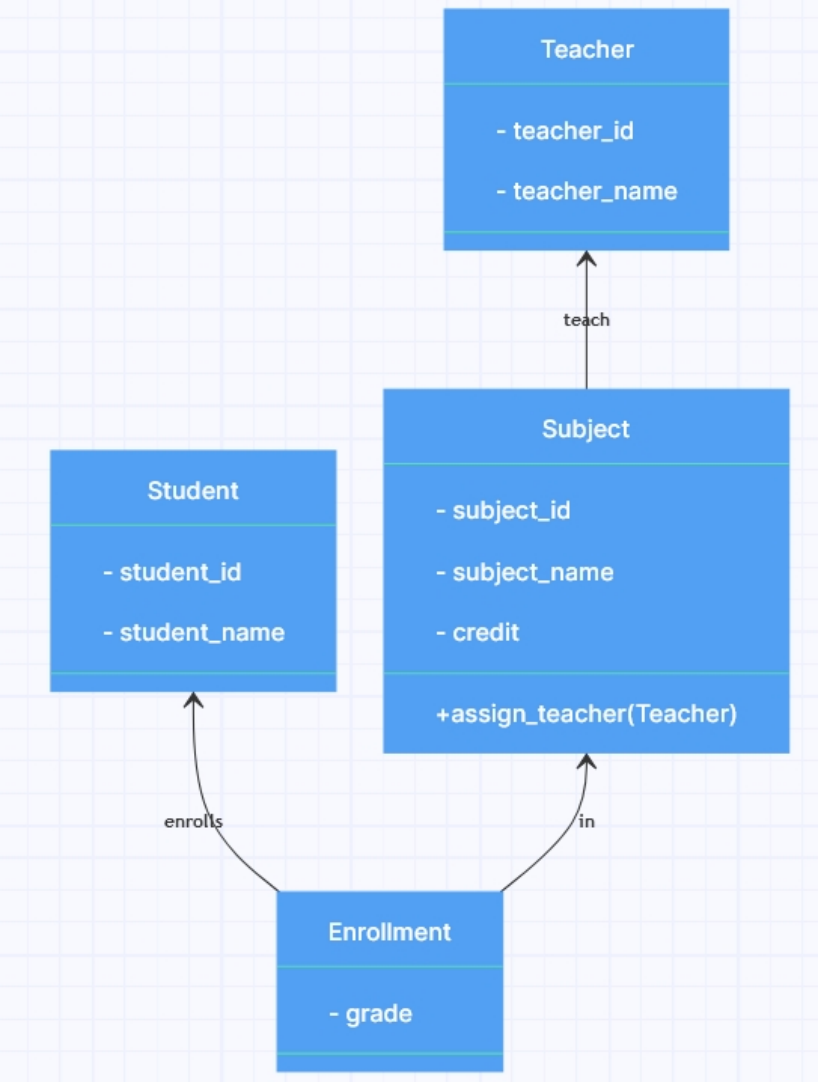
- ระหว่างคลาส อาจมีความสัมพันธ์กัน เช่น นักเรียนลงเรียนในรายวิชา อาจารย์สอนในรายวิชา อาจารย์ให้เกรดในวิชาที่เรียน ใน Class Diagram จะต้องแสดงความสัมพันธ์ระหว่างคลาสด้วย
- โดยความสัมพันธ์ระหว่าง Class มีดังนี้
  - Association
  - Dependency
  - Aggregation
  - Composition

Class Diagram : ความสัมพันธ์

- ความสัมพันธ์แบบ Association เป็นความสัมพันธ์ระหว่างคลาส ที่ object หนึ่ง กระทำกับอีก object หนึ่ง มี 2 แบบ คือ มีแบบทิศทางเดียว และ แบบ 2 ทิศทาง
- **Unary Association** เป็นความสัมพันธ์ทิศทางเดียว จากรูป คือ
  - นักศึกษา enroll การลงเรียน in รายวิชา
  - อาจารย์ สอน รายวิชา
- ให้สังเกตว่าเส้นที่แสดงความสัมพันธ์เขียนชื่อการกระทำเอาไว้ด้วย ทำให้ class diagram สามารถ “เล่าเรื่อง” ได้ในระดับหนึ่ง และใช้เป็นเครื่องมือในการสื่อสารในทีม หรือ กับผู้ใช้ได้
- เมื่อเขียนโปรแกรม จะมี attribute บางตัวของ class หนึ่งเข้าไปอยู่อีกคลาส (จะเพิ่มที่หางลูกศร)
- หัวลูกศรจะชี้บอกว่าคลาสใดที่ถูกใช้



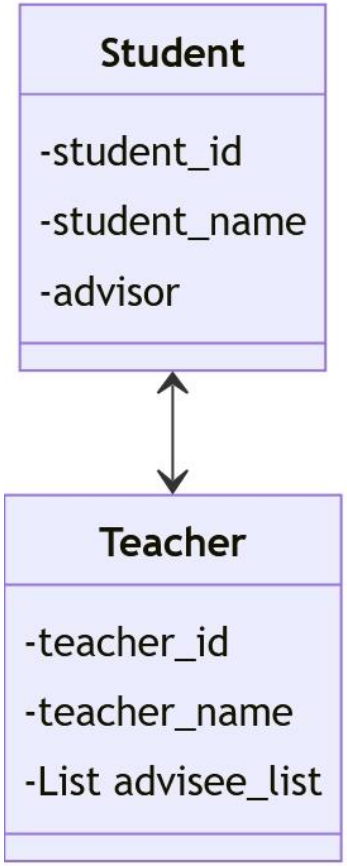
Class Diagram : ความสัมพันธ์




```python
class Student:
    def __init__(self, student_id, student_name):
        self.student_id = student_id
        self.student_name = student_name

class Subject:
    def __init__(self, subject_id, subject_name, credit):
        self.subject_id = subject_id
        self.subject_name = subject_name
        self.credit = credit
        self.teacher = None

    def assign_teacher(self,  teacher):
        self.teacher= teacher

class Teacher:
    def __init__(self, teacher_id, teacher_name):
        self.teacher_id = teacher_id
        self.teacher_name = teacher_name

class Enrollment:
    def __init__(self, student, subject):
        self.student = student
        self.subject = subject
        self.grade = None
```

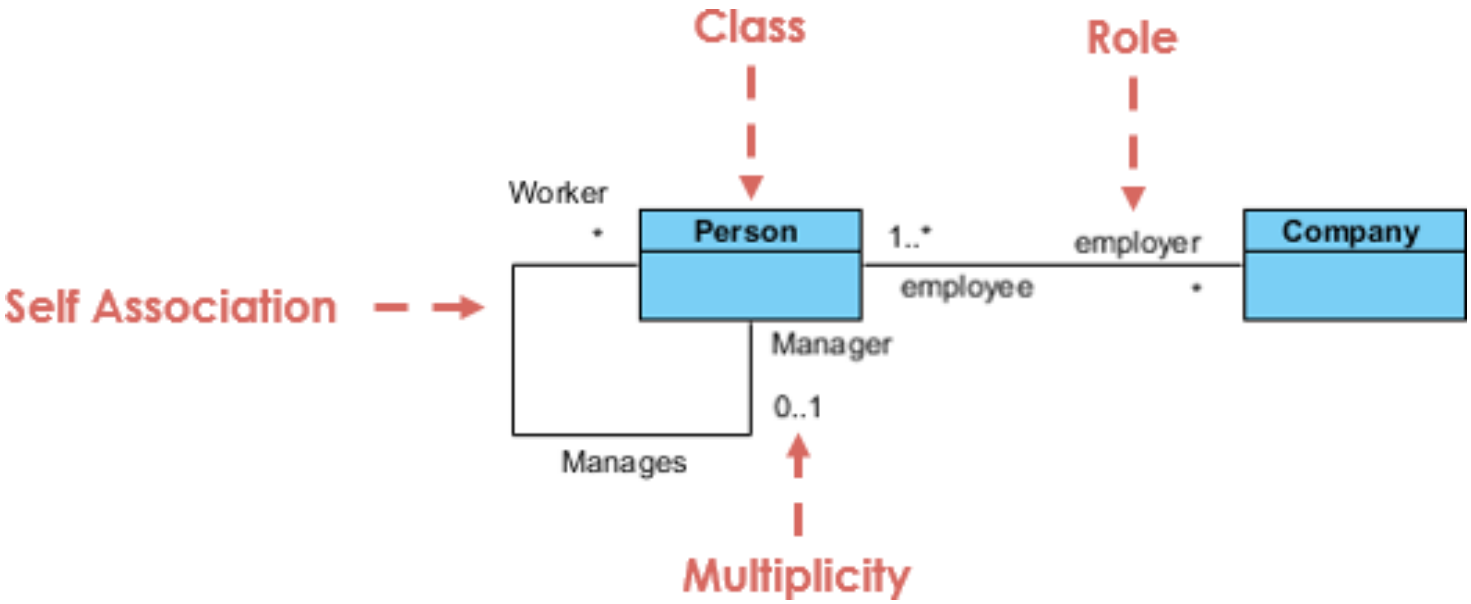
Class Diagram : ความสัมพันธ์

- **Binary Association** : เป็นความสัมพันธ์แบบ 2 ทิศทาง ตัวอย่างของความสัมพันธ์แบบนี้ คือ ความสัมพันธ์ระหว่างอาจารย์ที่ปรึกษากับนักศึกษา
- นักศึกษาจะมีอาจารย์ที่ปรึกษา 1 คน แต่อาจารย์ 1 คนอาจมีนักศึกษาที่ให้คำปรึกษาหลายคนก็ได้
- จากรูปจะเห็นได้ว่า ในคลาส Student จะมี attribute advisor ซึ่งทำหน้าที่เก็บว่า Teacher คนใดเป็นที่ปรึกษา และในคลาส Teacher จะมี attribute advisee_list ซึ่งเป็น List ของ Student (เขียนใน diagram เพื่อให้เห็น)
- การจะใช้ Unary หรือ Binary ขึ้นกับการจะอ้างอิง หรือ อ้างถึง เช่น ในรายวิชาจำเป็นต้องอ้างถึงผู้สอน แต่ผู้สอนไม่จำเป็น



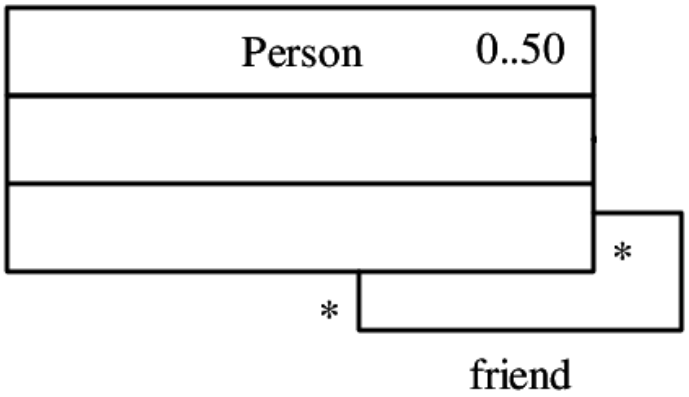
Class Diagram : ความสัมพันธ์

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



Class Diagram : ความสัมพันธ์

- ตัวอย่างของ Self Association เช่น ความสัมพันธ์ของเพื่อน



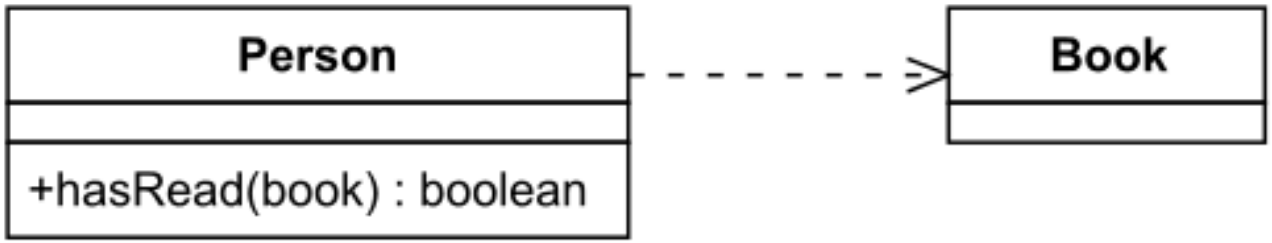
- สามารถเขียน Code ได้เป็น

```
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
```

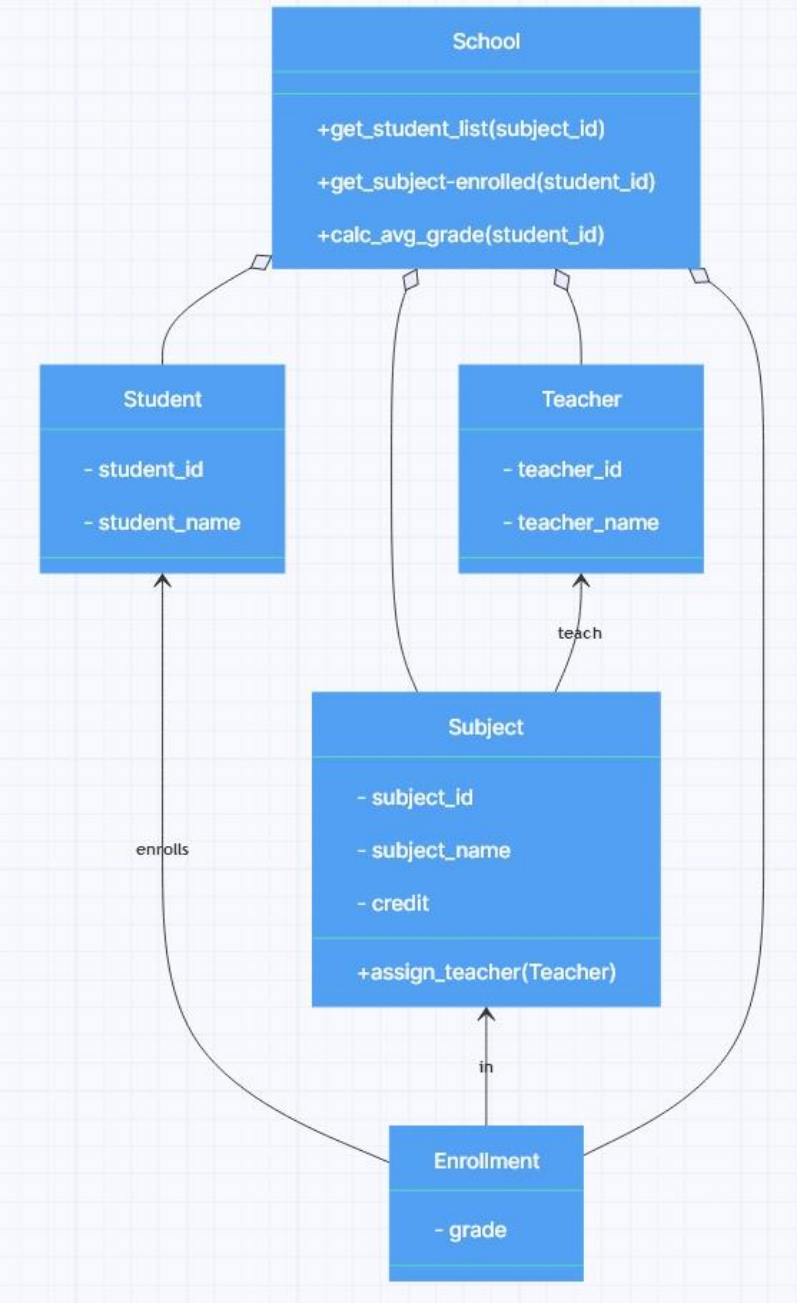
Class Diagram : ความสัมพันธ์

- **Dependency** : เป็นความสัมพันธ์ที่คล้ายกับ Association แต่เป็นความสัมพันธ์ที่อ่อนกว่า คือ ในขณะที่ ความสัมพันธ์แบบ Association จะมีการเก็บข้อมูลความสัมพันธ์ลงใน attribute ของ object แต่ใน Dependency จะไม่มีการเก็บเป็น attribute แต่อาจมีการใช้ object เป็นพารามิเตอร์ของ method ใน object เท่านั้น
- ความสัมพันธ์แบบ Dependency จะแสดงโดยใช้เส้นประลูกศร
- ตัวอย่างจะเห็นว่า Book จะเป็นเพียงพารามิเตอร์ของฟังก์ชันเท่านั้น
- ความหมายของ Dependency คือบางการทำงานของ object จะขึ้นกับ object อื่น โดยปลายลูกศรจะชี้ไปยัง object ที่ขึ้นด้วย (หรือต้องใช้)



Class Diagram : ความสัมพันธ์

- **Aggregation** แปลว่า **รวบรวม** มักใช้ใน 2 รูปแบบ โดยในรูปแบบแรกจะใช้ในกรณีที่ object เป็นที่เก็บรวบรวม object ของคลาสอื่น เช่น คลาส School เป็นที่เก็บของ object Student, Subject, Teacher และ Enrollment ตามรูป
- จากรูป ใน object School จะเก็บทุก object ที่สร้างขึ้นของ Student, Subject, Teacher และ Enrollment นอกจากนั้นยังเก็บ method ที่มีหน้าที่เกี่ยวข้องกับหลาย object
- ความสัมพันธ์ **Aggregation** จะใช้สัญลักษณ์เป็นสี่เหลี่ยมขนมเปียกปูนแบบโปร่ง ดังตัวอย่าง (อาจมีลูกศรหรือไม่ก็ได้)



Class Diagram : ความสัมพันธ์

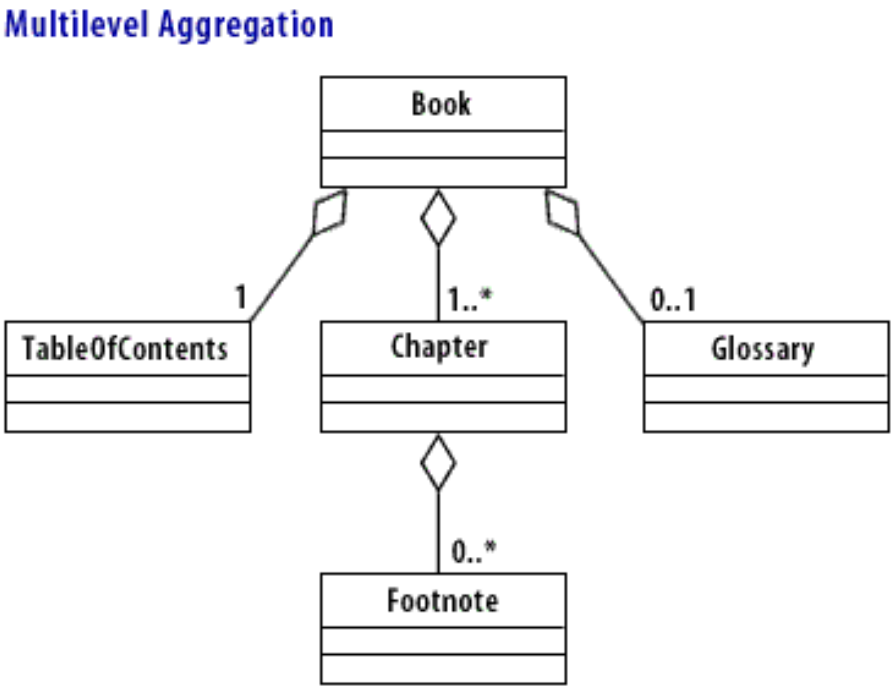
- คลาส School เมื่อเขียนเป็นโปรแกรม จะมีดังนี้

```python
class School:
    def __init__(self):
        self.students = []
        self.subjects = []
        self.teachers = []
        self.enrolls = []

    def add_student(self, student):
        self.students.append(student)

    def add_subject(self, subject):
        self.subjects.append(subject)

    def add_teacher(self, teacher):
        self.teachers.append(teacher)

    def enroll_student_in_subject(self, student, subject):
        if student in self.students and subject in self.subjects:
            self.enrolls.append(Enrollment(student, subject))
```

- จะเห็นว่ามีการสร้าง List ที่เก็บ student, subject, teacher, enrollment

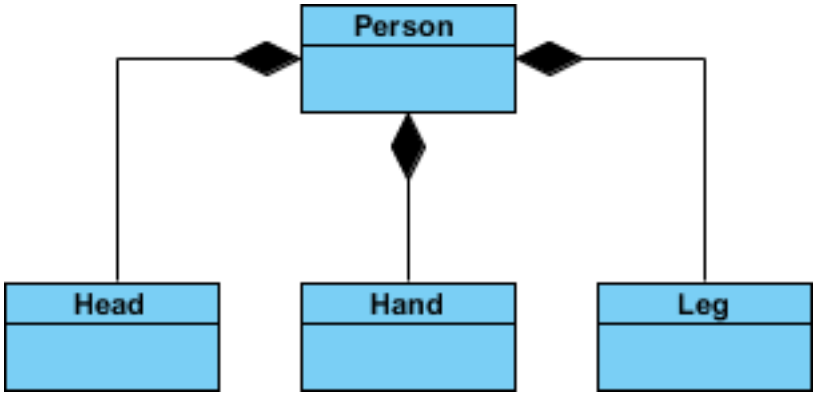
Class Diagram : ความสัมพันธ์

- อีกรูปแบบหนึ่งของความสัมพันธ์แบบ Aggregation จะใช้ในความหมายของ “การเป็นส่วนประกอบ” เช่น จากรูปจะเห็นว่า “หนังสือ” ประกอบด้วย ToC, Chapter และ Glossary



Class Diagram : ความสัมพันธ์

- **Composition** : เป็นความสัมพันธ์ที่มีความหมายว่า “การเป็นส่วนประกอบ” เช่นกัน แต่จะใกล้ชิดมากขึ้นไปอีก คือ object ที่เป็นส่วนประกอบนั้น จะไม่สามารถคงอยู่ได้ หากไม่มี object หลัก



- จะเห็นว่า หากไม่มี Person (คนนั้น) การคงอยู่ของ แขน มือ และ ขา จะอยู่ไม่ได้
- ความสัมพันธ์แบบ Composition จะใช้สี่เหลี่ยมขนมเปียกปูนแบบทึบ
- Aggregation กับ Composition ในแง่ของการเป็นส่วนประกอบจะคล้ายกัน ขึ้นกับการตีความของผู้ออกแบบเอง

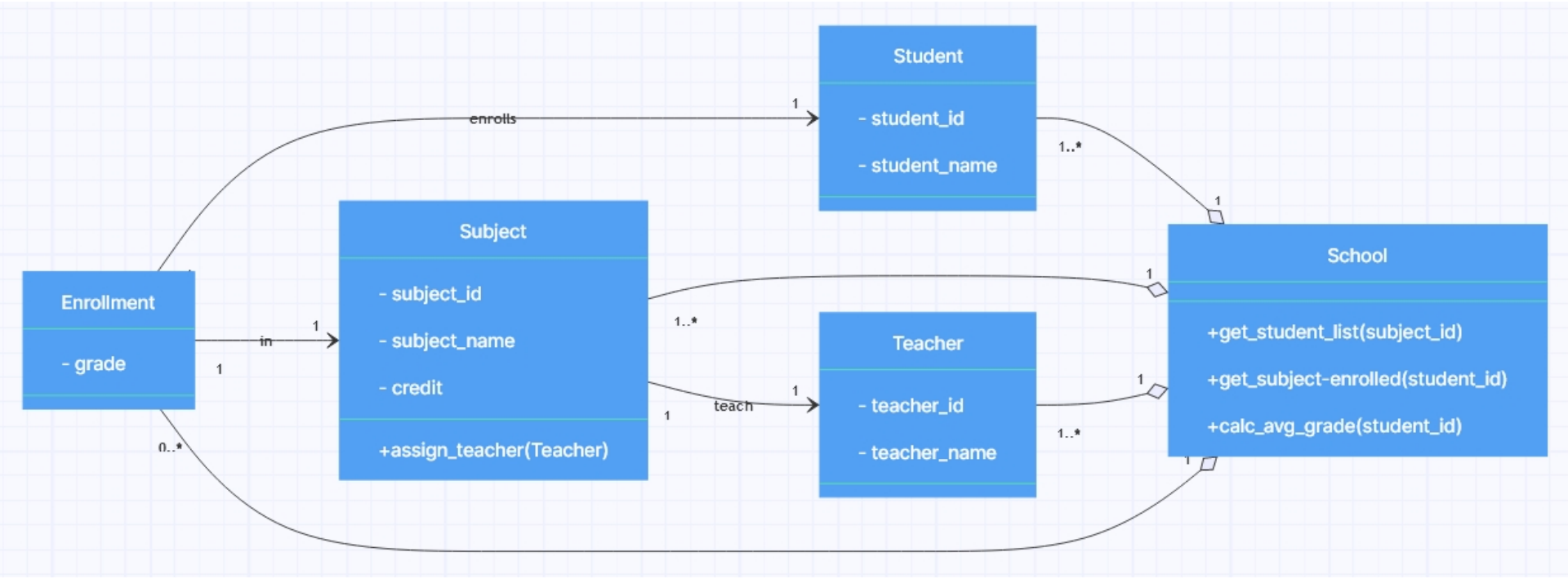
Class Diagram : Multiplicities

- 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

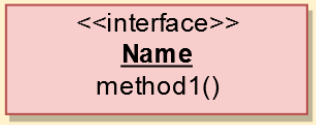
- จาก Class Diagram ก่อนหน้าเมื่อใส่ Multiplicities จะได้ดังนี้



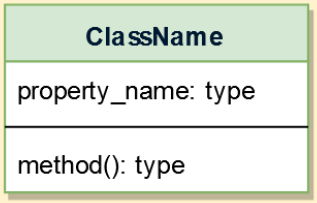
- จะช่วยบอกในการเขียนโปรแกรมว่าแต่ละ object มีจำนวนได้มากแค่ไหน เช่น ถ้าเป็น {1} ก็เพียงสร้างตัวแปรไว้รับ แต่ถ้าเป็น {0..*} ก็จะต้องสร้าง list ไว้รับ

Class Diagram : สรุป

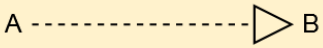
UML conventions



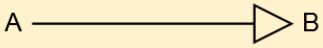
Interface: Classes implement interfaces, denoted by Generalization.



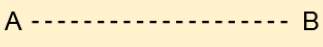
Class: Every class can have properties and methods.
Abstract classes are identified by their *Italic* names.



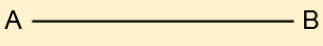
Generalization: A implements B.



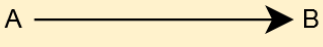
Inheritance: A inherits from B. A "is-a" B.



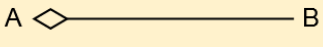
Use Interface: A uses interface B.



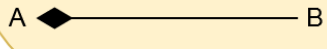
Association: A and B call each other.



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

- ได้กล่าวถึง attribute ประเภทแรกไปแล้ว คือ instance attribute
- ยังมี attribute อีกประเภทหนึ่ง คือ Class Attribute ซึ่งใช้เก็บข้อมูลส่วนกลางที่ใช้งานร่วมกันของทุก Instance ในคลาสนั้น
- Class Attribute เป็น attribute ของ class ไม่ใช่ attribute ของ instance ใด instance หนึ่ง ปกติจะกำหนด Class Attribute ไว้เหนือ __init__()

```
class ClassName:

    # Class Attributes

    # __init__()

    # Methods
```

Class Attribute

- การกำหนด Class Attribute เหมือนกับกำหนดตัวแปรทั่วไป

```
<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

Class Attribute

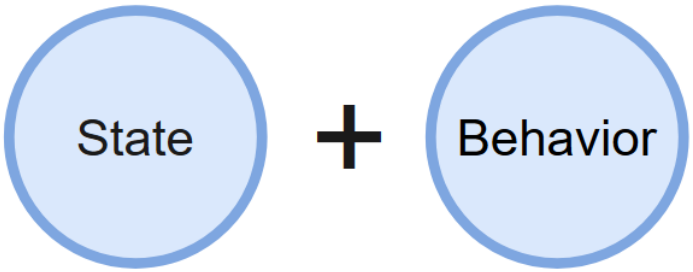
- เราสามารถใช้ Class Attribute ในการนับจำนวน Object ได้ เช่น

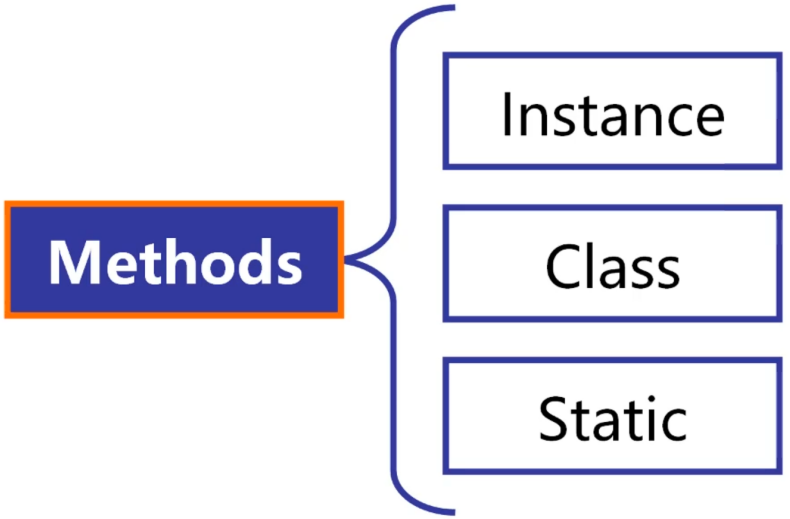
```
class Movie:
    id_counter = 1

    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
```

Methods

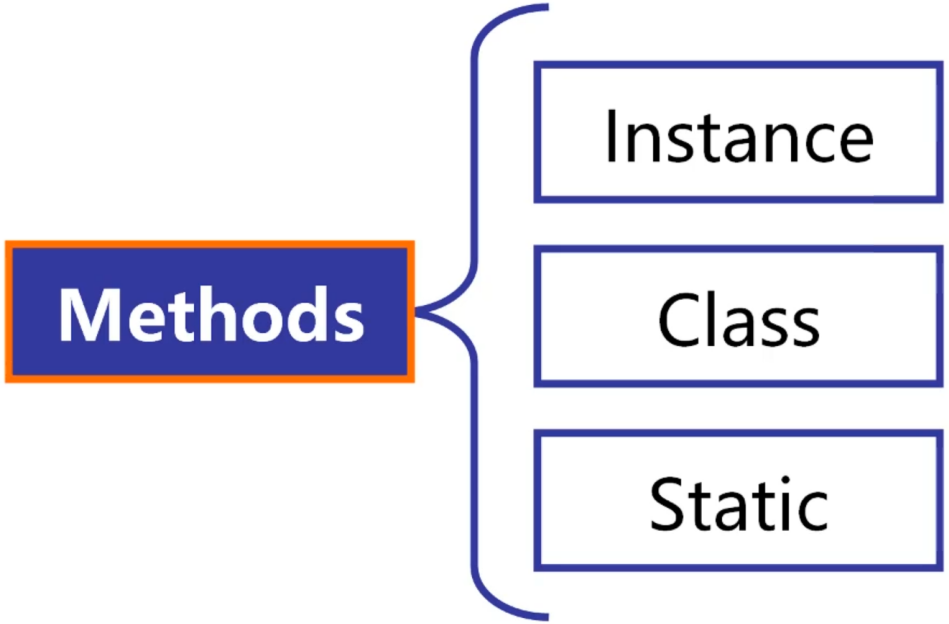
- เนื่องจาก คลาส ประกอบด้วย attribute และ method โดย attribute ทำหน้าที่เก็บสถานะของ object และ method ทำหน้าที่กำหนด พฤติกรรมของ object
- Method จะแบ่งออกเป็น 3 ประเภท คือ instance method, class method และ static method





Methods

- นอกเหนือจาก instance method แล้ว ยังมี method อีก 2 ประเภทได้แก่
  - Static method คือ method ที่ไม่มีการใช้ attribute ในคลาส ดังนั้นจึงสามารถเรียกใช้ได้โดยไม่ต้องสร้าง instance ก่อน
  - มักจะเป็นฟังก์ชันทั่วไป ที่เอาไปฝากไว้ที่คลาส เนื่องจากมีการทำงานที่ใกล้กัน



Static method

- เป็น method ที่สามารถเรียกใช้ได้โดยไม่จำเป็นต้องสร้าง instant

```
class Student:
    def __init__(self, name, height):
        self._name = name
        self._weight = weight
        self._height = height

    @staticmethod
    def kg_to_pound(kg):
        return kg * 2.20462

    def cm_to_inch(cm):
        return cm * 0.393701

print(Student.kg_to_pound(50))
```

Class method

- เป็นคลาสที่ใช้ในการทำ Constructor แบบอื่นๆ ได้ (cls คือ constructor)

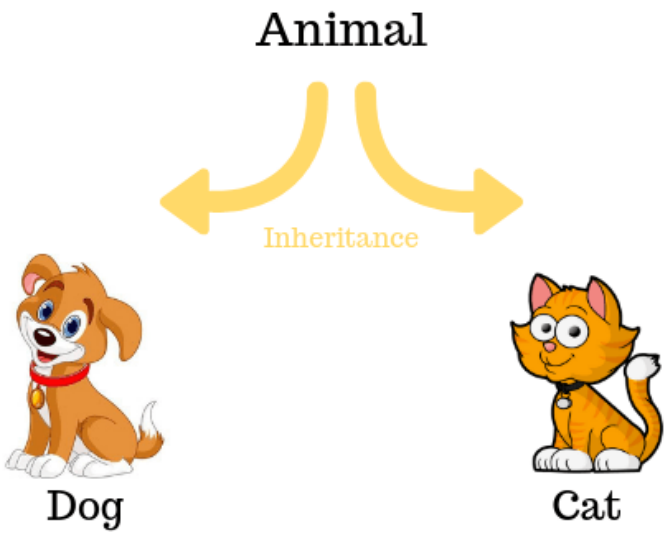
main.py

```python
class Point:
    def __init__(self, x, y):
        self._x = x
        self._y = y

    @classmethod
    def of(cls, point_string):
        s = point_string.split("-")
        return cls(int(s[0]),int(s[1]))

p1 = Point(5, 5)
p2 = Point.of("10-10")
print(p2._x)
```

```
10
```

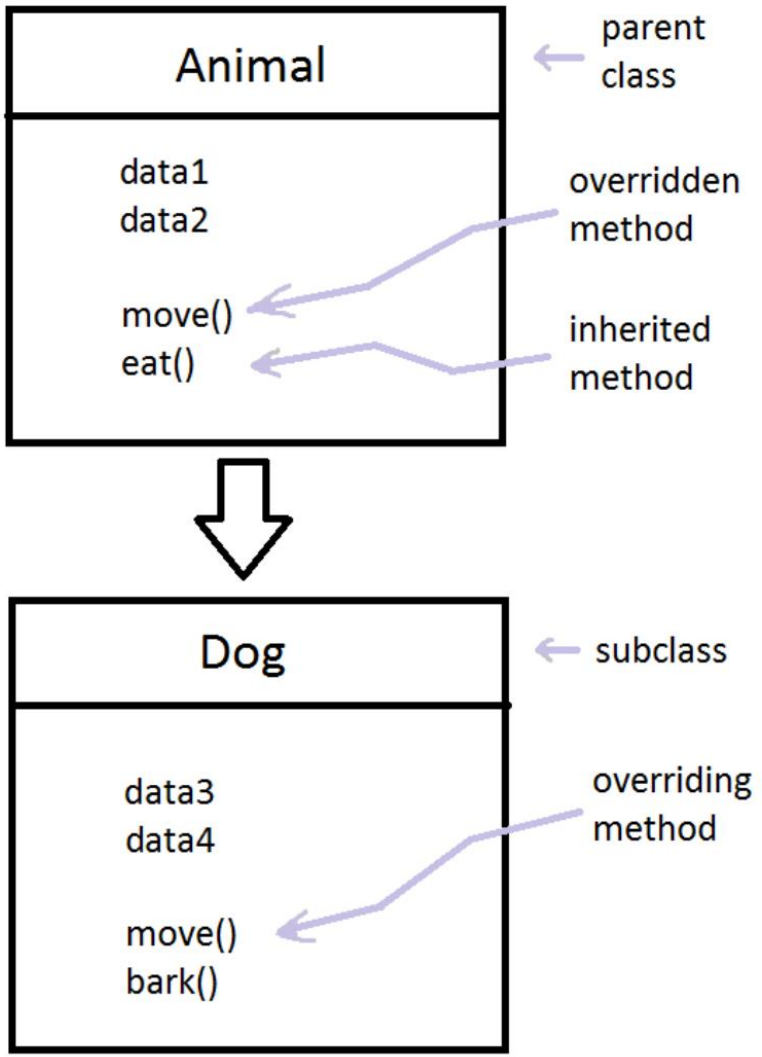
Inheritance

- 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 ก็จะทำได้ง่าย 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 ไม่ใช่ จึงเป็น 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 เวอร์ชั่นเก่า จะวงเล็บ Object ต่อท้ายหมดทุกคลาสแต่ในเวอร์ชั่นหลังๆ 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

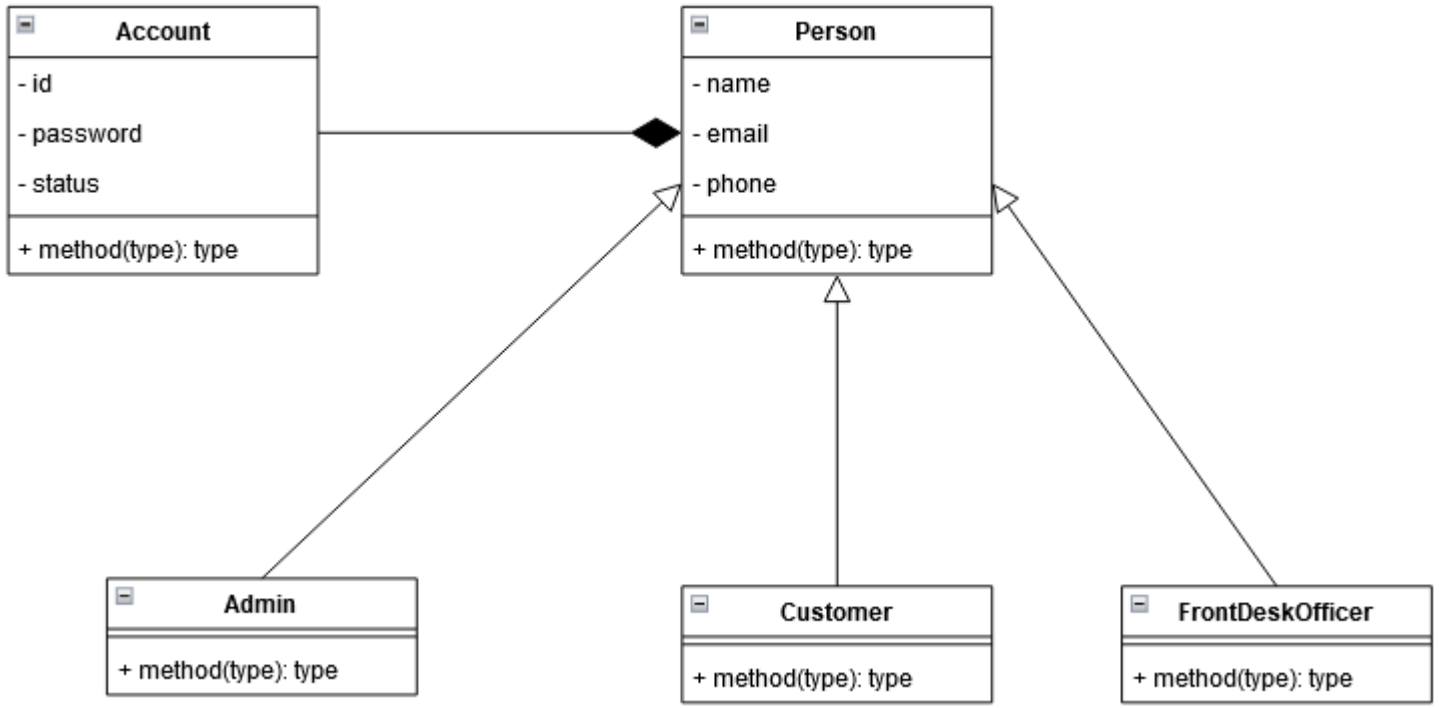
```
class Ractangle(Polygon):
    pass
```

Inheritance

- มีหลักการออกแบบคลาสข้อหนึ่งมีชื่อว่า Open-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** สมมติว่ามี Class Person ซึ่งทุกคนจะมี Account เนื่องจากหน้าที่ของ Person มีหลายประเภท จึงใช้ Inheritance คือ Admin, Customer, Front Desk Officer ซึ่ง Inherit มาจาก Person



Inheritance

- การแยกคลาส Person ออกมาจะทำให้ความซ้ำซ้อนของข้อมูลลดลง โดยข้อมูลที่เหมือนกันจะอยู่ในคลาส Person และข้อมูลที่ต่างกันจะอยู่ใน Subclass ของแต่ละประเภทย่อย
- การทำเช่นนี้ มีข้อดี ที่ทำให้การปรับเปลี่ยนในอนาคตสามารถทำได้โดยมีการแก้ไข Code เดิมน้อยลง หากมีการเปลี่ยนแปลงโดยรวมก็แก้ไขเพียงคลาส Person คลาสเดียว หรือ หากมีการเปลี่ยนแปลงย่อย ก็เพียงแต่สร้าง Subclass ใหม่ขึ้นมา
- ขอยกตัวอย่าง หากในอนาคตมีการเพิ่มผู้ใช้ประเภทใหม่ขึ้นมา หากใช้วิธี Inherit จะทำให้ไม่ต้องไปแก้ไข Code เดิมในคลาส Person โดย Code สำหรับกลุ่มผู้ใช้ที่เพิ่มเข้ามาใหม่ ก็จะอยู่ในคลาสที่สร้างเพิ่มเติมขึ้นมาใหม่

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 ก็จะไม่ใช่ 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)
```

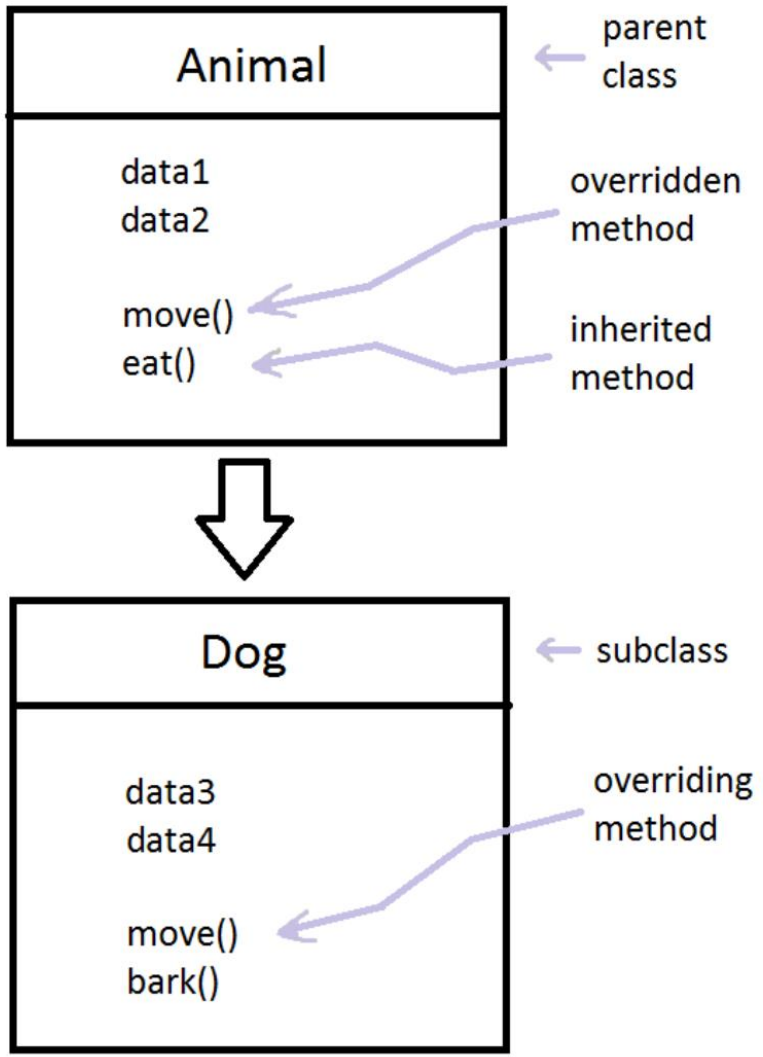
Inheritance

- 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)
```

Method Overriding

- คือการกำหนด การทำงานของ method ของ superclass ใหม่โดย subclass
- จากตัวอย่าง ใน class Animal มี method Move() อยู่ก่อนแล้ว
- สมมติว่าใน class Dog มีการเคลื่อนที่ ซึ่งจากไปจาก class Animal ก็สามารถกำหนดการเคลื่อนที่ของ Dog เสียใหม่ โดยการสร้าง method Move() ทับซ้ำ
- จะเรียกการทำงานแบบนี้ว่า Method overriding



Method Overriding

- จาก code ตัวอย่างมี 2 คลาส คือ class Parent และ class Child
- ใน class Parent มีการกำหนด method show( ) เอาไว้ โดยเป็นการแสดงคำว่า “Inside Parent”
- แต่ใน class Child มีการกำหนด method show( ) ทับไป เมื่อมีการเรียก show ของ child ก็จะมีการทำงานต่างออกไป

```
Inside Parent
Inside Child
```

```
class Parent():
    def __init__(self):
        self.value = "Inside Parent"

    # Parent's show method
    def show(self):
        print(self.value)

class Child(Parent):
    def __init__(self):
        self.value = "Inside Child"

    # Child's show method
    def show(self):
        print(self.value)

# Driver's code
obj1 = Parent()
obj2 = Child()

obj1.show()
obj2.show()
```

For your attention