



01076105, 01076106

Object Oriented Programming

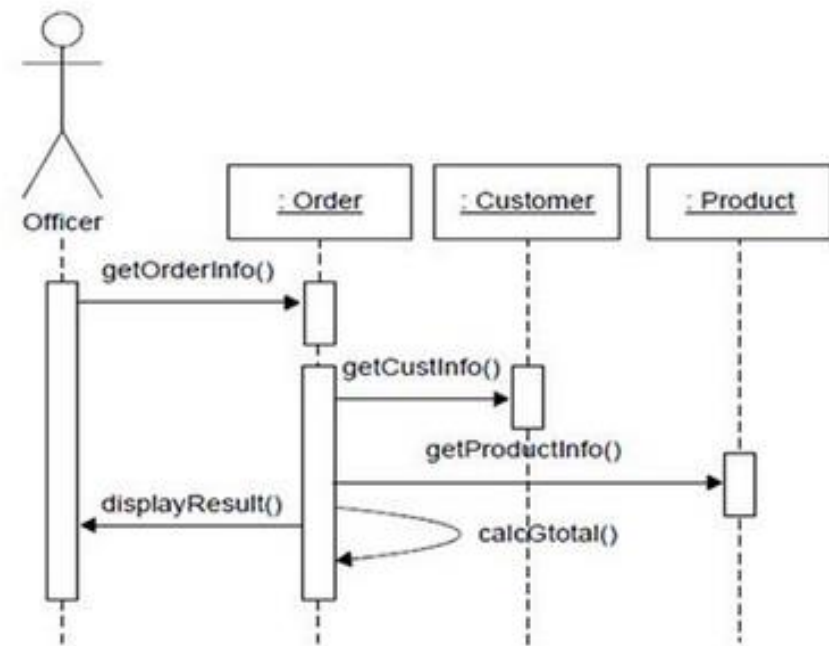
Object Oriented Programming Project

Sequence Diagram, Multiple Inheritance



Sequence Diagram


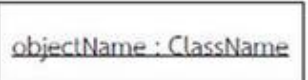


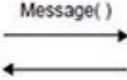

- เป็น diagram ที่สำคัญอีก diagram หนึ่ง
ทำหน้าที่แสดงลำดับการทำงานของโปรแกรม
โดยแสดงปฏิสัมพันธ์ (interaction)
ระหว่าง object ตามลำดับของเหตุการณ์ที่
เกิดขึ้น คล้ายกับ flowchart
- การทำงานใน sequence diagram จะเป็น
ระดับ method แสดงการเรียกใช้กันระหว่าง
method ที่อยู่ใน class ต่างๆ
- สำหรับภายใน method หากต้องการแสดง
การทำงาน ให้ใช้ flowchart เหมือนเดิม





Sequence Diagram

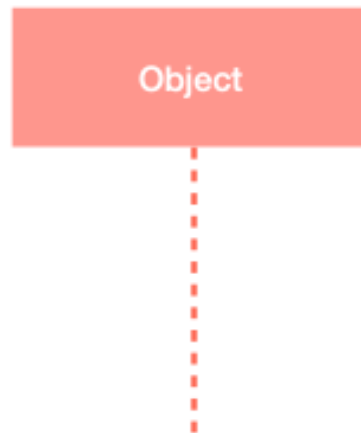
- Actor หมายถึงผู้ใช้แต่ละประเภทของซอฟต์แวร์นั้น
- Object ต้องเป็น class ที่อยู่ใน class diagram
- Lifeline แทนเส้นเวลาเหตุการณ์
- Activation bar แสดงขอบเขตหรืออายุการทำงานของ event นั้นๆ
- Message แสดงชื่อ method ที่เรียกใช้ และผลลัพธ์ (ถ้ามี)
- Call back สำหรับการคืนมาใน object เดียวกัน

สัญลักษณ์	ชื่อ	ความหมาย
	Actor	ผู้ที่เกี่ยวข้องกับระบบ
	Object	อ็อบเจกต์ที่ต้องทำหน้าที่ตอบสนองต่อ Actor
	Lifeline	เส้นแสดงชีวิตของอ็อบเจกต์หรือคลาส
	Focus of Control / Activation	จุดเริ่มต้นและจุดสิ้นสุดของแต่ละกิจกรรมในระหว่างที่มีชีวิตอยู่
	Message	คำสั่งหรือฟังก์ชันที่อ็อบเจกต์หนึ่งส่งให้อ็อบเจกต์หนึ่ง ซึ่งสามารถส่งกลับได้ด้วย
	Callback / Self Delegation	การประมวลผลและคืนค่าที่ได้ภายในอ็อบเจกต์เดียวกัน



Sequence Diagram

- Lifeline Notation (เส้นชีวิต)
 - คือเส้นชีวิตของ object หรือ class เป็นตัวแทนของ object หรือส่วนประกอบต่างๆ ที่มีปฏิสัมพันธ์ซึ่งกันและกันในระบบในลำดับต่างๆ
 - format การเขียนชื่อเส้นชีวิตคือ Instance Name:Class Name
 - เส้นชีวิตกับสัญลักษณ์ actor หรือเส้นชีวิตกับ object จะใช้เมื่ออย่างใดอย่างหนึ่งเป็นส่วนหนึ่งของ sequence diagram นั้น





Sequence Diagram

- Activation Bars

- Activation bars จะวางอยู่บนเส้นชีวิตเพื่อแสดงการโต้ตอบระหว่าง object, method หรือ module ความยาวของกล่องสี่เหลี่ยมจะแสดงระยะเวลาการโต้ตอบของ object หรือ แสดงจุดเริ่มต้นและจุดสิ้นสุดของแต่ละกิจกรรม
- การโต้ตอบระหว่าง 2 object เกิดเมื่อวัตถุหนึ่งส่ง message (หรือเรียกใช้) ไปยัง object อื่น โดย object ที่ส่งข้อความ (เรียกใช้) เรียกว่า message caller และ object ที่รับข้อความ (ถูกเรียกใช้) เรียกว่า message receiver เมื่อมีแถบ activation บนเส้นชีวิตของวัตถุ นั้นหมายความว่า object นั้นมีการทำงานในขณะที่โต้ตอบกัน






Sequence Diagram

- Activation Bars

- **Message Arrows** ลูกศรจาก message caller จะชี้ไปที่ message receiver และระบุทิศทางของ message ว่าไหลไปในทางใด โดยสามารถไหลไปในทิศทางใดก็ได้ จากซ้ายไปขวา ขวาไปซ้าย หรือส่ง message กลับไปที่ตัวมันเองก็ได้

- รูปแบบของ message มี 2 แบบ ได้แก่

- Synchronous message จะถูกใช้เมื่อ object ที่ส่งข้อความรอให้ object ที่รับ message ประมวลผลและส่ง return กลับมา ก่อนที่จะส่ง message อันต่อไป หัวลูกศรที่ใช้จะเป็นลูกศรแบบทึบ 

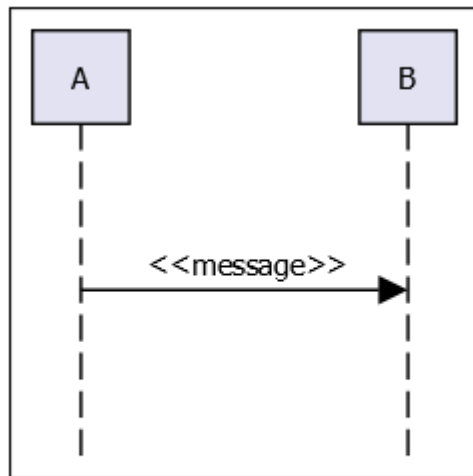
- Asynchronous message จะถูกใช้เมื่อ object ที่ส่ง message ไม่รอให้ object ที่รับ message ประมวลผลข้อความและส่งค่า return กลับมา แต่จะส่งข้อความต่อไปให้แก่วัตถุอื่นในระบบเลย หัวลูกศรที่แสดงในข้อความประเภทนี้เป็นหัวลูกศรเส้น



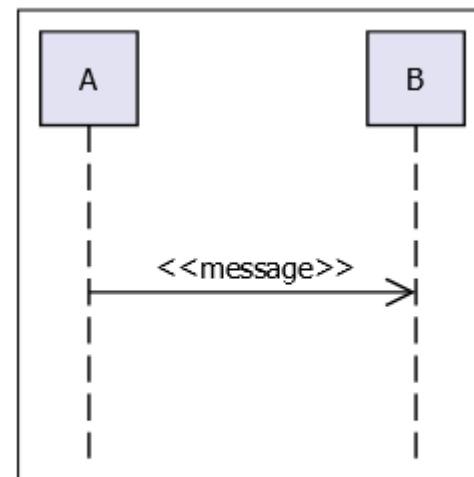


Sequence Diagram

- ตัวอย่างของ synchronous message และ asynchronous message
- การเรียกใช้ส่วนใหญ่มักจะเป็น synchronous message ยกเว้นบางกรณี เช่น งานที่ต้องใช้เวลานาน เมื่อส่งคำสั่งก็กลับไปทำงานอื่นได้เลย เมื่อฝั่งรับคำสั่งทำงานเสร็จแล้วค่อยแจ้งกลับผ่านทาง callback function หรือ ระบบ message queue สำหรับชำระเงินใน e-commerce



A synchronous message

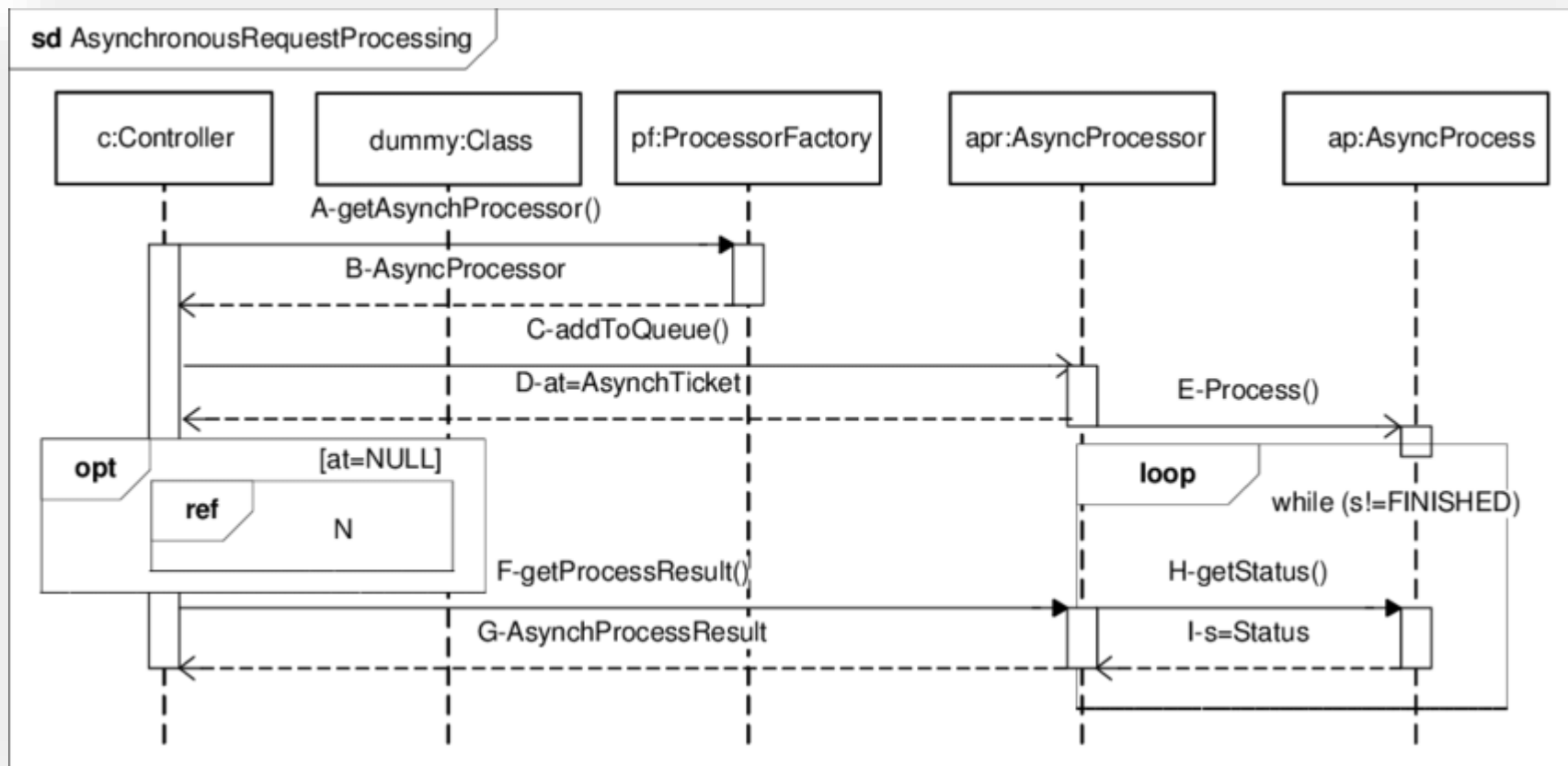


An asynchronous message



Sequence Diagram

- ตัวอย่างของ asynchronous message

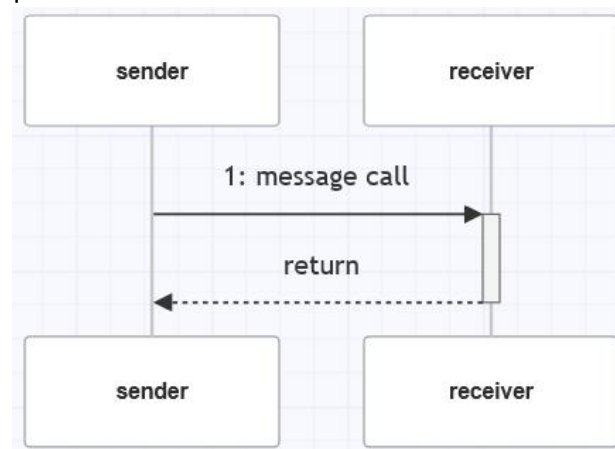




Sequence Diagram

- Activation Bars

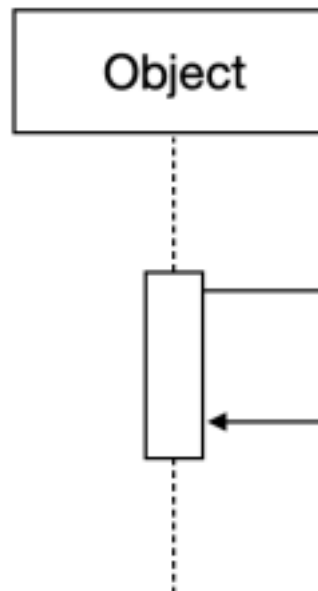
- return message ใช้เพื่อระบุว่า object รับ message และประมวลผล message เสร็จสิ้นแล้ว และกำลังส่งคืนการควบคุมไปยัง object ที่ส่ง message
- return จะใช้เส้นประในการแสดง สำหรับการส่ง message บนแถบ activation ด้วย synchronous message จะให้ความหมายโดยนัยว่ามี return message อยู่แล้ว แม้จะไม่ได้วาดเส้น return message ก็ตาม ดังนั้นเขียนหรือไม่ก็ได้
- ดังนั้น หากแผนภาพดูยุ่งเหยิงมาก ก็อาจไม่ใช้ return message ก็ได้





Sequence Diagram

- Activation Bars
 - Reflexive message ในบางครั้ง object อาจส่งข้อความหาตัวเอง (เรียกใช้ method ใน object เดียวกัน) จะเรียกว่า reflexive message จะแสดง message ประเภทนี้โดยการใช้ message arrow ที่เริ่มจากจบที่เส้นชีวิตเดียวกันอย่างตัวอย่างด้านล่างนี้



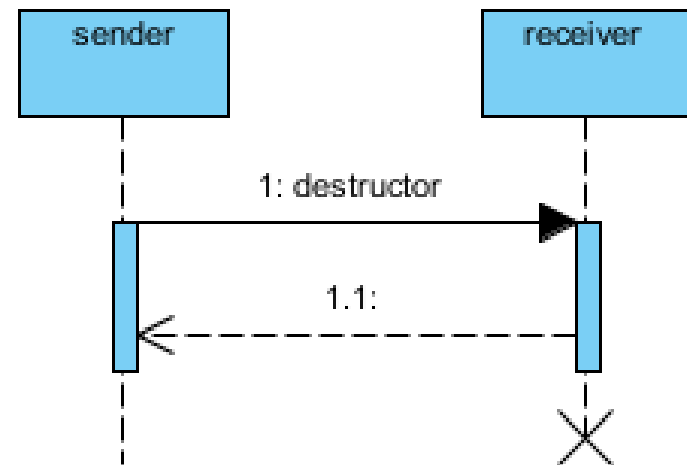
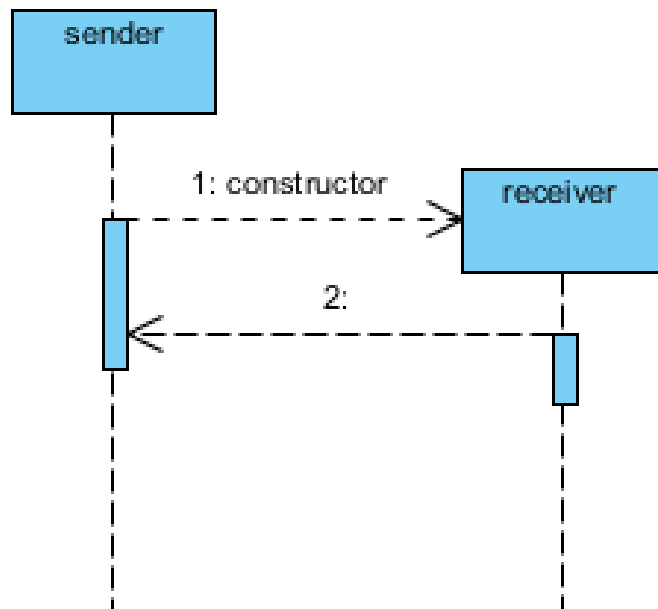


Sequence Diagram

- **Constructor Message**

เป็น message หรือการเรียกใช้ method ของ object อีกแบบหนึ่ง แต่เป็นการเรียกใช้ constructor ซึ่งเป็นผลให้มีการสร้าง instance เช่น สร้างวิชาเรียน

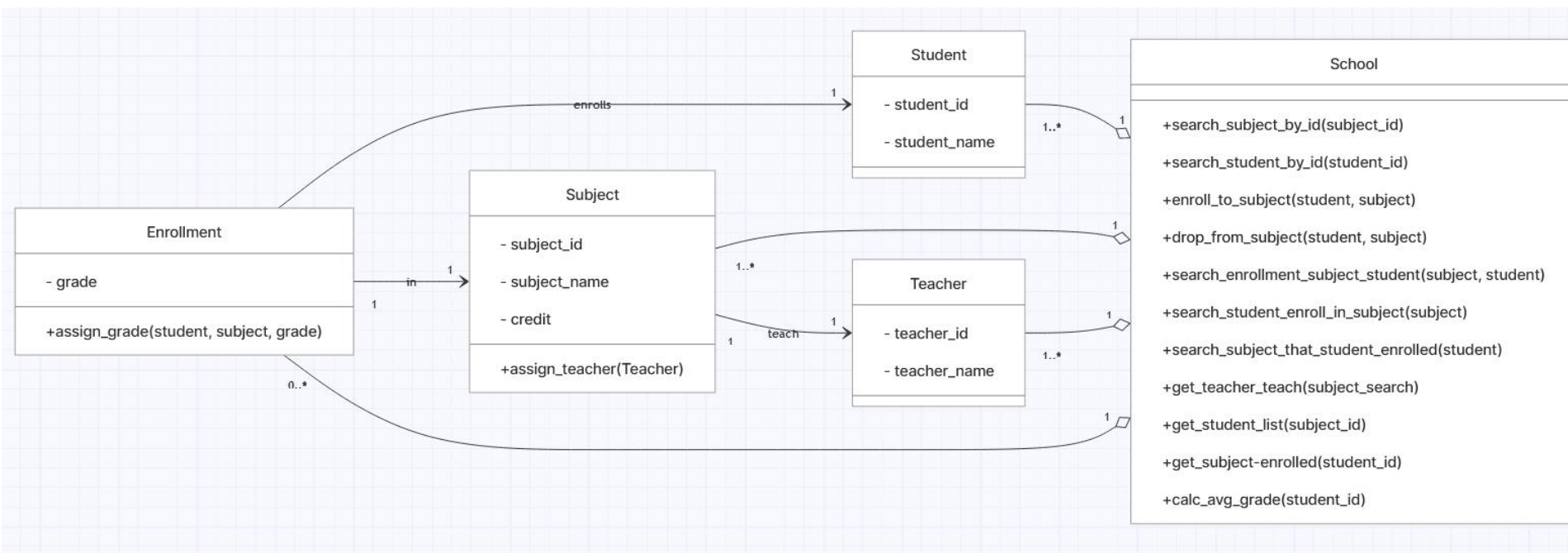
- **Destructor Message** จะเป็นการทำลาย instance (ใน python ไม่มีการใช้เพราะมี garbage collection ช่วยทำให้)



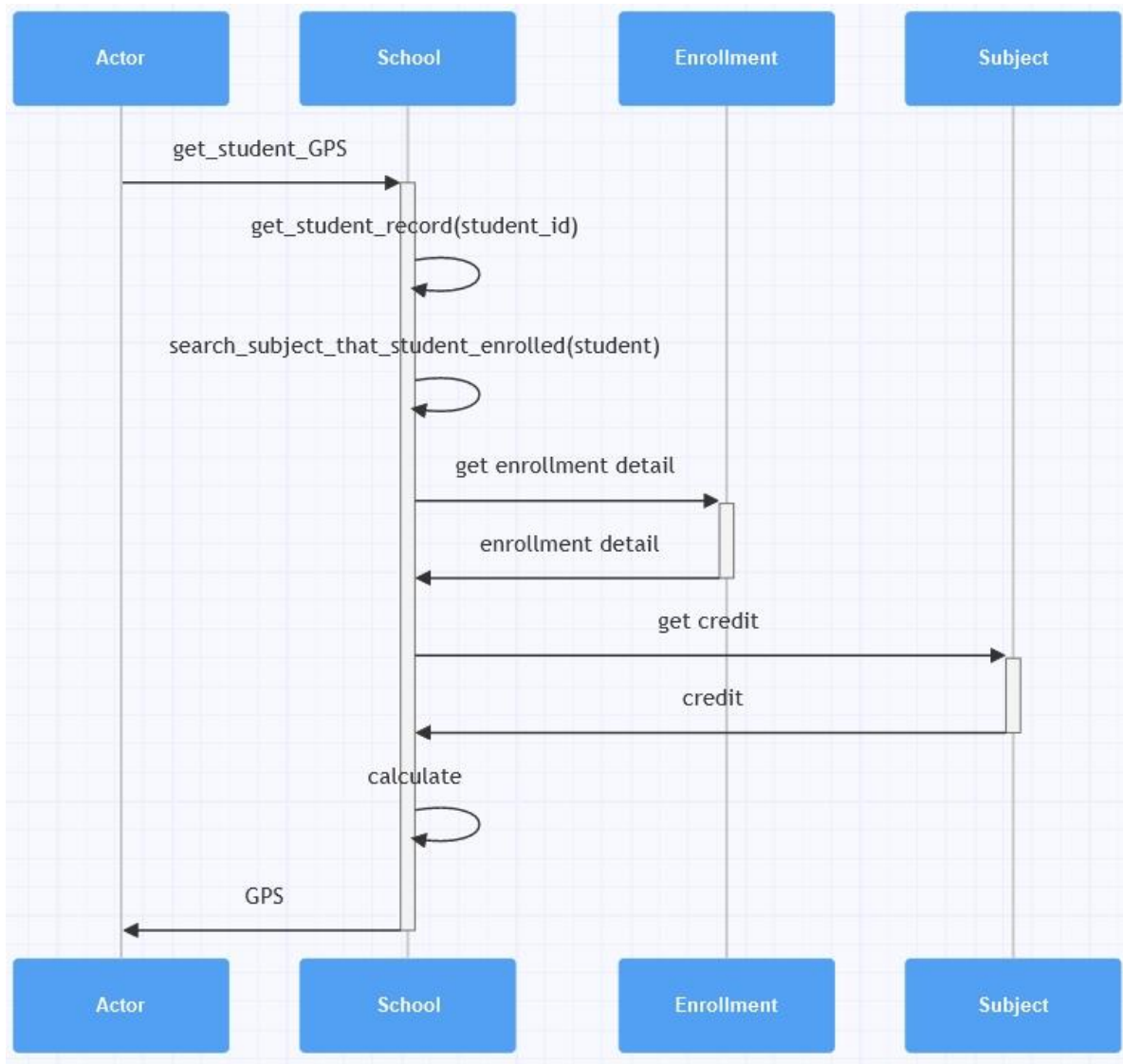


Sequence Diagram

- **Activity** : จาก class diagram ให้เขียน sequence diagram ของการคำนวณเกรดเฉลี่ยของนักศึกษา



Sequence Diagram



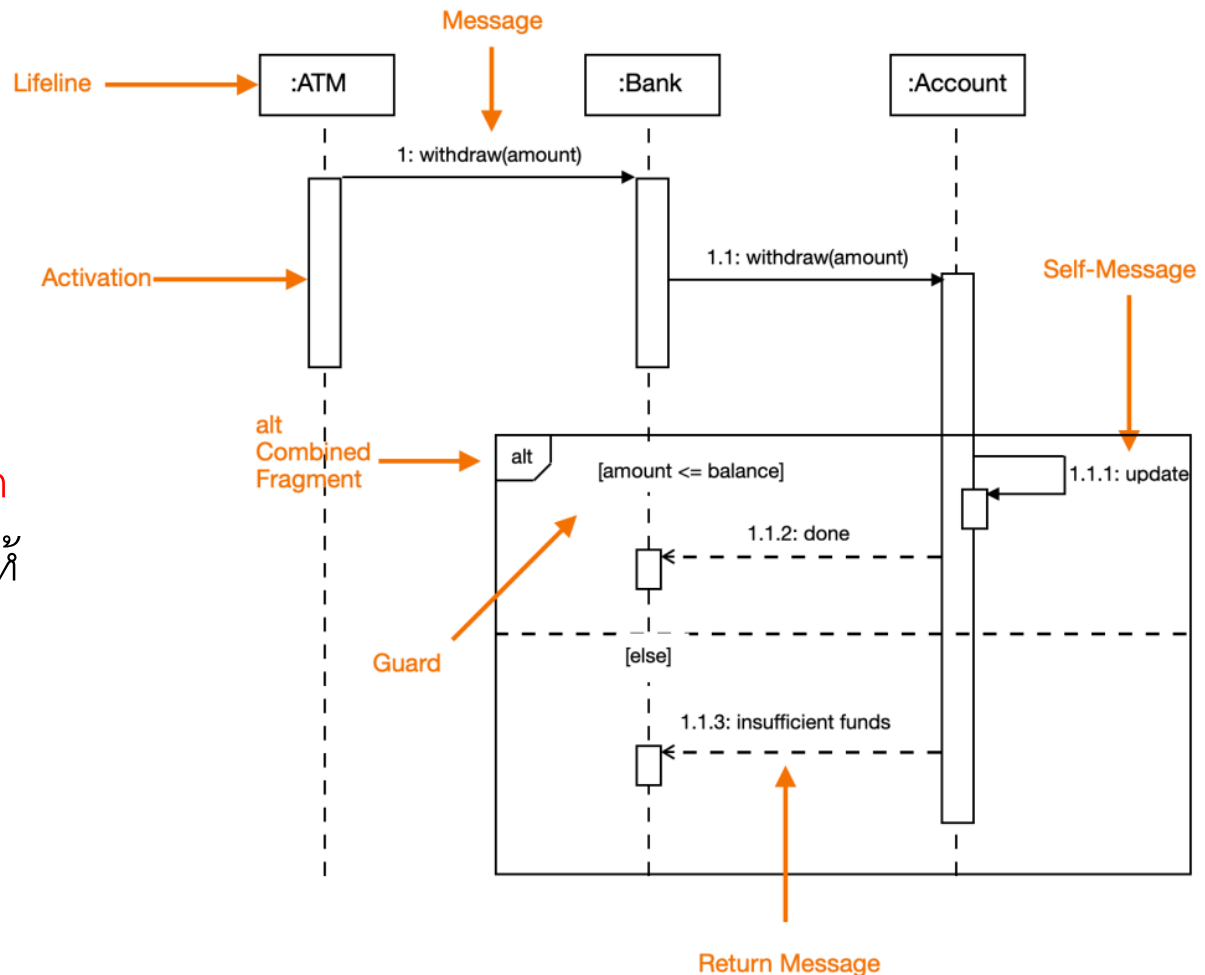


Sequence Diagram

- Sequence Fragment

- คือกล่องที่มีเครื่องหมายแสดง section การโต้ตอบระหว่างวัตถุใน sequence diagram

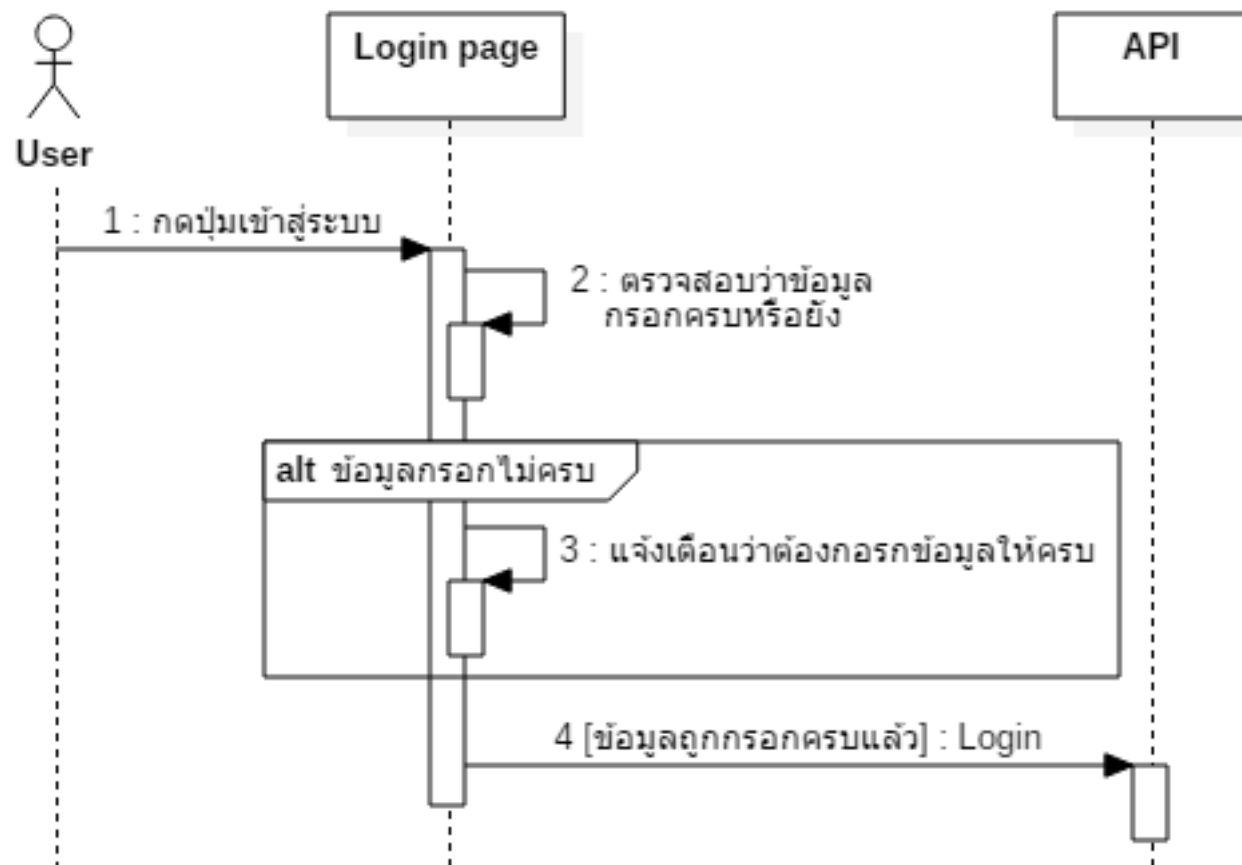
- Alternative combination fragment** ใช้เมื่อมีตัวเลือกให้เลือกตั้งแต่ 2 ตัวเลือกขึ้นไป ใช้ตรรกะแบบ “if then else”





Sequence Diagram

- ตัวอย่าง Alternative Fragment



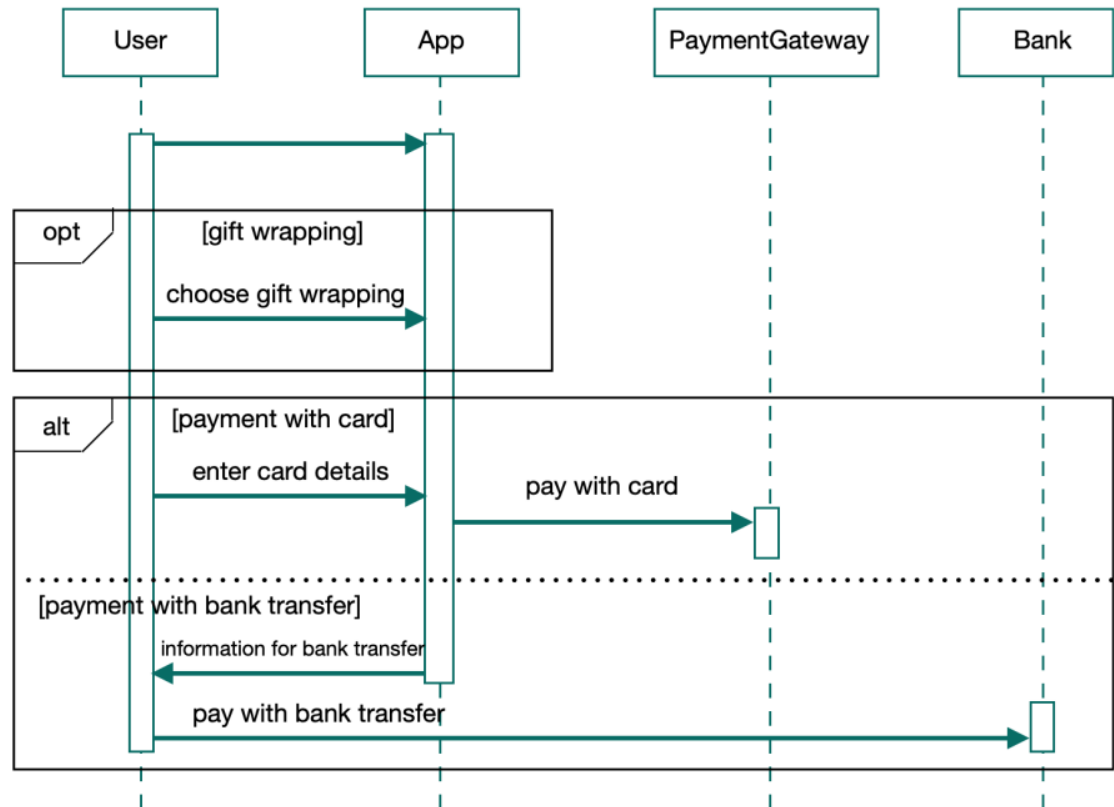


Sequence Diagram

- Options

Combination
fragment

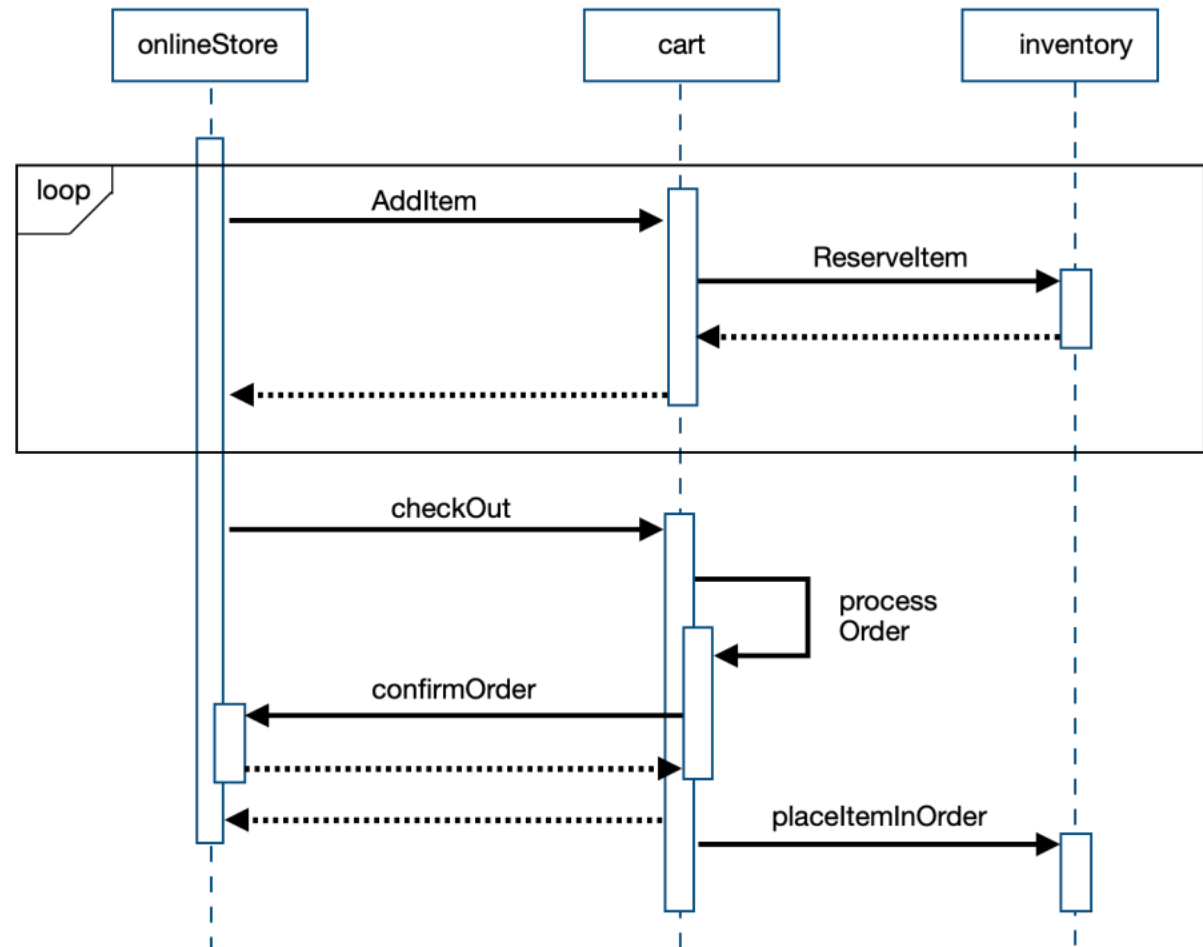
- ใช้เพื่อแสดงถึงลำดับเหตุการณ์ที่เกิดขึ้นภายใต้เงื่อนไขใดเงื่อนไขหนึ่งเท่านั้น ไม่เช่นนั้นเหตุการณ์นั้นจะไม่สามารถเกิดขึ้นได้ ใช้ตรรกะแบบ 'if then'





Sequence Diagram

- **Loop fragment**
- ใช้เพื่อแสดงลำดับเหตุการณ์ที่เกิดขึ้นซ้ำ โดยมี 'loop' เป็น fragment operation และ guard condition ระบุที่มุมด้านซ้ายของกล่อง





Sequence Diagram

- Fragment ต่างๆ ที่มีการใช้งานใน Sequence Diagram
 - Alt = ทางเลือกการทำงาน เมื่อมี 2 ทางเลือกขึ้นไป
 - Opt = เป็นเงื่อนไขว่าจะทำหรือไม่
 - Par = ทั้งสองทางทำงานขนานกัน
 - Loop = มีการทำงานแบบวนซ้ำ

Operator	Meaning
alt	Alternative multiple fragments: only the one whose condition is true will execute.
opt	Optional: the fragment executes only if the supplied condition is true. Equivalent to an alt only with one trace.
par	Parallel: each fragment is run in parallel.
loop	Loop: the fragment may execute multiple times, and the guard indicates the basis of iteration.
critical	Critical region: the fragment can have only one thread executing it at once.
neg	Negative: the fragment shows an invalid interaction.
ref	Reference: refers to an interaction defined on another diagram. The frame is drawn to cover the lifelines involved in the interaction. You can define parameters and a return value.
sd	Sequence diagram: used to surround an entire sequence diagram.



-
- ```

sequenceDiagram
 participant Customer as :Customer
 participant Order as :Order
 participant Stock as :Stock

 Customer->>Order: 1: <<Create>>
 activate Order
 Order-->>Customer: 2: Return()
 deactivate Order
 Customer->>Order: 3: * [for each product] addItem(product, qty)
 activate Order
 Order->>Stock: 4: checkAvailable(product, qty)
 activate Stock
 Stock-->>Order: 5: return done
 deactivate Stock
 Order->>Order: 6: addProduct(product)
 activate Order
 Order->>Stock: 7: reduceStock(product, qty)
 activate Stock
 Stock-->>Order: 8: return done
 deactivate Stock
 Order-->>Customer: 9: return done
 deactivate Order
 Customer->>Order: 10: save()
 activate Order
 Order-->>Customer: 11: return done
 deactivate Order
 destroy Order

```
- sd Place Order**
- Message**
- Anonymous Object**
- Sequence number**
- Iteration**
- Self-reference**
- Focus of control**
- Return**
- Object lifeline**



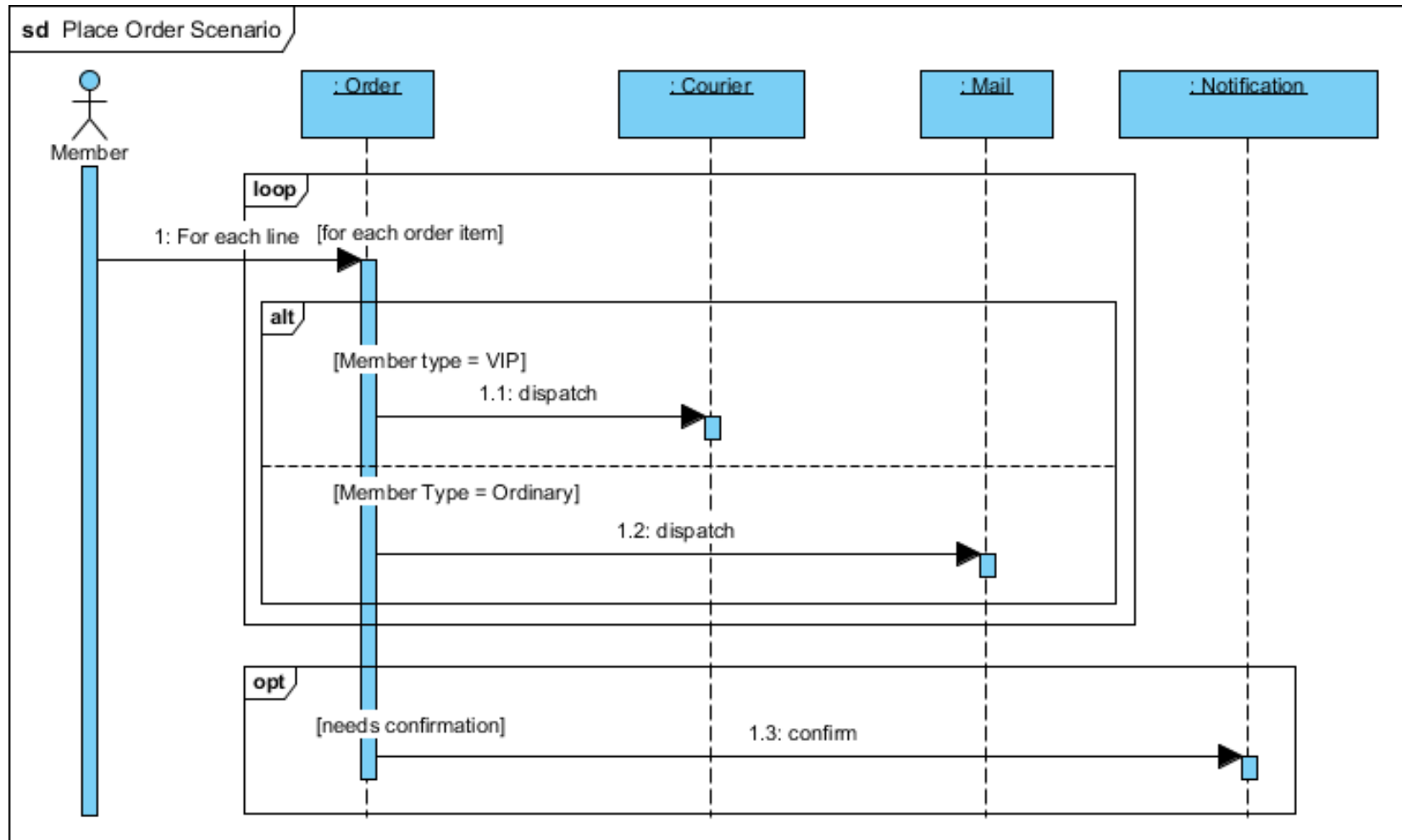
# Sequence Diagram

- จาก Slide ก่อนหน้านี้ จะเห็นว่า sequence diagram ไม่ได้มีรูปแบบตายตัวนัก เช่น
  - ในการทำซ้ำ แทนที่จะใช้ fragment loop ก็เขียนลงใน message line
  - สามารถเขียนตัวเลขลำดับการทำงานกำกับใน sequence diagram ได้
- เมื่ออ่าน sequence diagram จะพบว่าสามารถนำไปเขียนเป็นโปรแกรมได้ คล้ายกับ flowchart ดังนั้น sequence diagram จึงทำหน้าที่ในการอธิบายการทำงานการทำงาน เพื่อให้สามารถนำการออกแบบไปสู่การเขียนโปรแกรมได้
- sequence diagram ทำหน้าที่บอก interface ระหว่าง object ดังนั้นหากออกแบบ class diagram และ sequence diagram ได้ละเอียดพอแล้ว ทีมงานสามารถจะแบ่งงานกันทำได้
- การเขียน sequence diagram ที่ดีต้องเน้นที่การสร้างความเข้าใจ



# Sequence Diagram

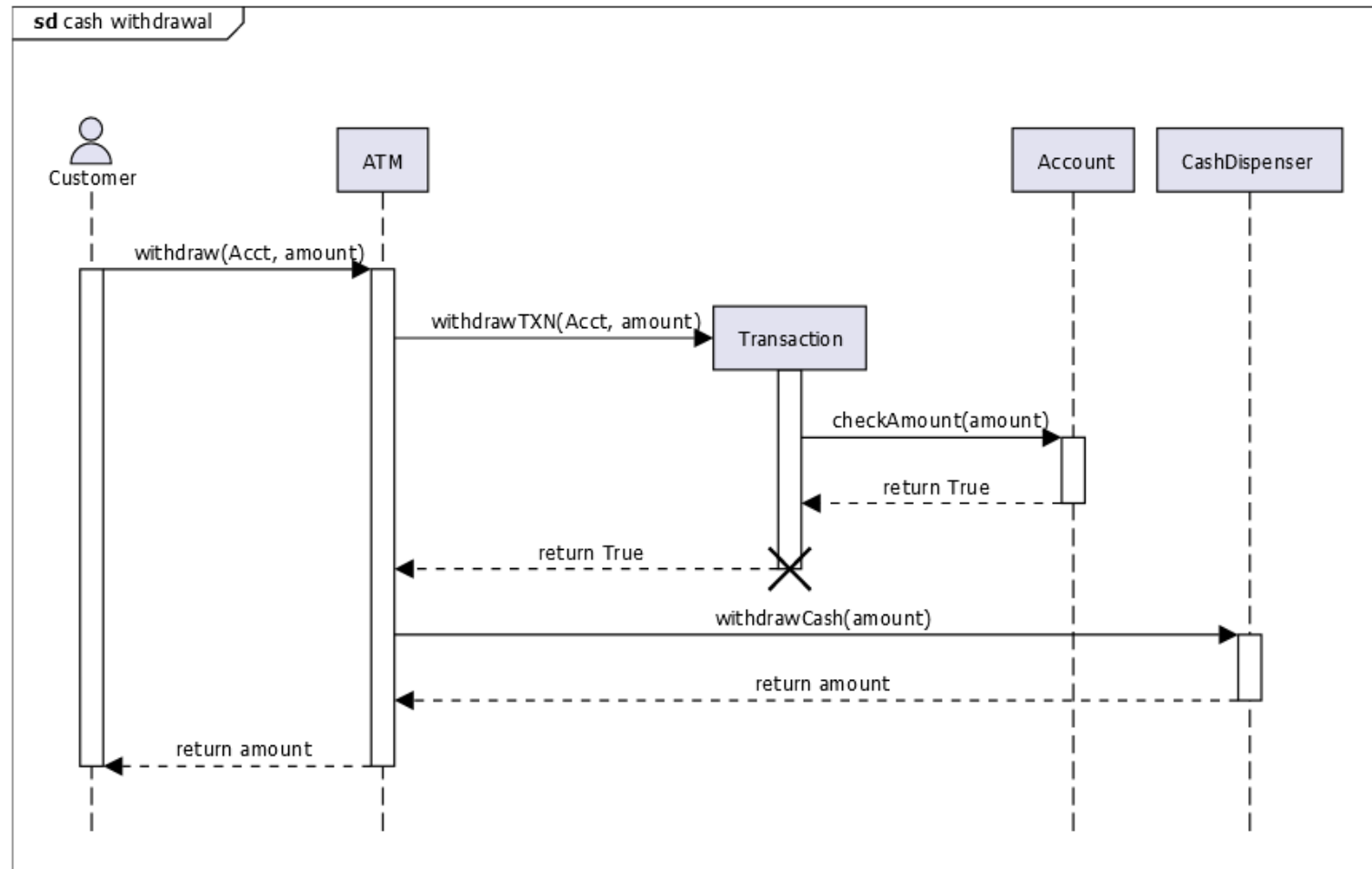
- ตัวอย่างการเขียน Sequence Diagram



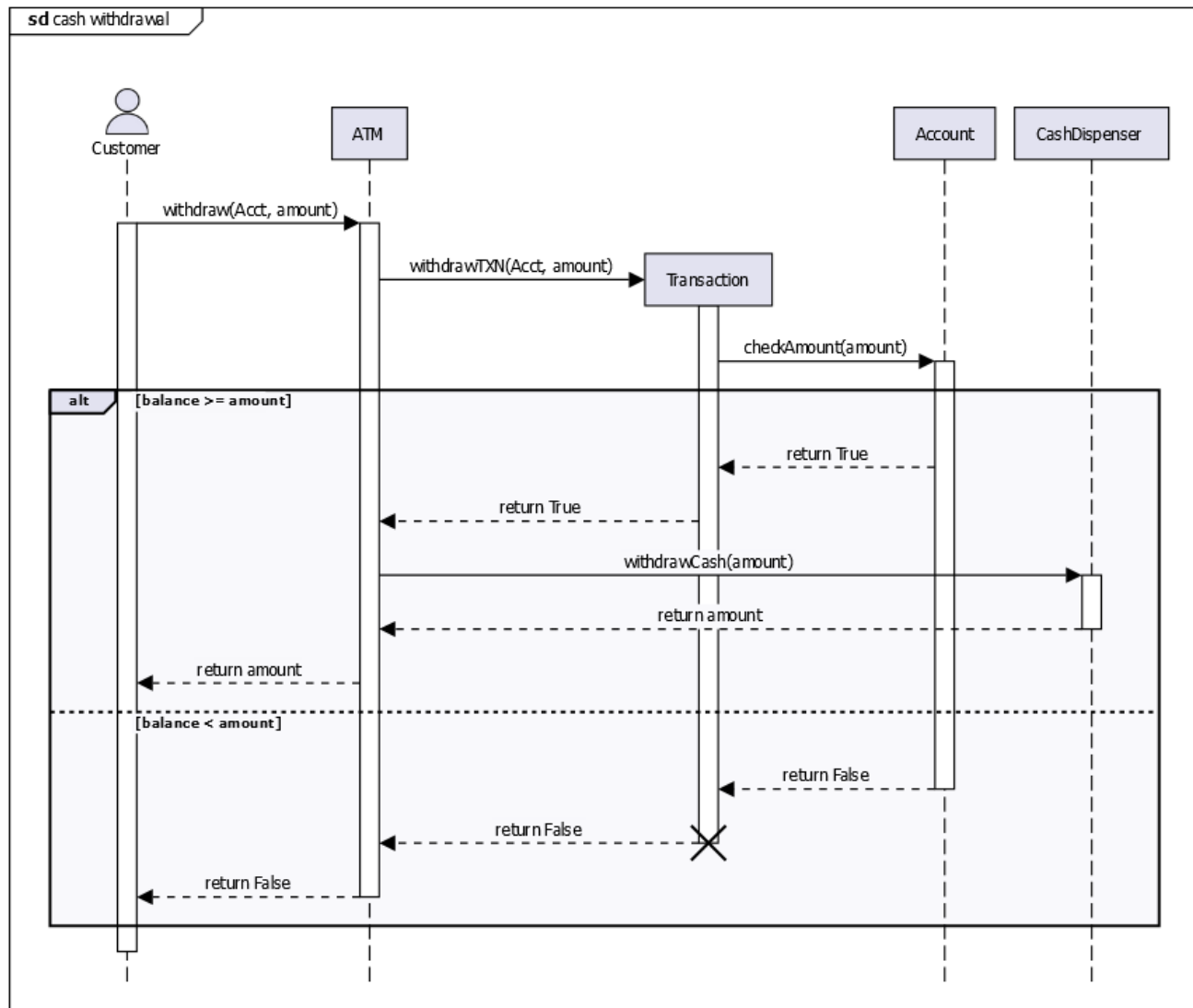


# Sequence Diagram

- ตัวอย่างการเขียน Sequence Diagram



# Sequence Diagram





# Sequence Diagram

- แนวทางการเขียน sequence diagram
  - ให้เขียน 1 use case ต่อ 1 sequence diagram
  - ให้ประเมินว่าใน use case นั้นมี actor ใดเกี่ยวข้องบ้าง
  - ให้ประเมินว่าใน use case นั้นมี class ใดเกี่ยวข้องบ้าง
  - นำลำดับการทำงานมาเขียนเป็น sequence diagram โดยพิจารณาว่าในแต่ละขั้นตอนนั้น ต้องสั่งให้ class ใด ทำหน้าที่อะไร จากนั้นจึงกำหนด method ของ class ที่เกี่ยวข้อง
  - ให้นำ method ที่กำหนดให้ class นั้นไปใส่ใน class diagram ด้วย
  - ให้คำนึงถึงการสร้าง object ด้วยว่าเกิดขึ้นในขั้นตอนใด





# Multiple Inheritance

- การเรียนที่ผ่านมามีได้กล่าวถึง Inheritance ไปบ้างแล้ว สำหรับภาษา Python จะมีความสามารถในการ “สืบทอด” จากหลายคลาส ซึ่งในบางภาษาไม่มี
- คลาส Button inherit มาจาก 2 คลาส จึงมีความสามารถของทั้ง 2 คลาส

```
class Rectangle:
 def __init__(self, length, width, color):
 self.length = length
 self.width = width
 self.color = color

class GUIElement:
 def click(self):
 print("The object was clicked...")

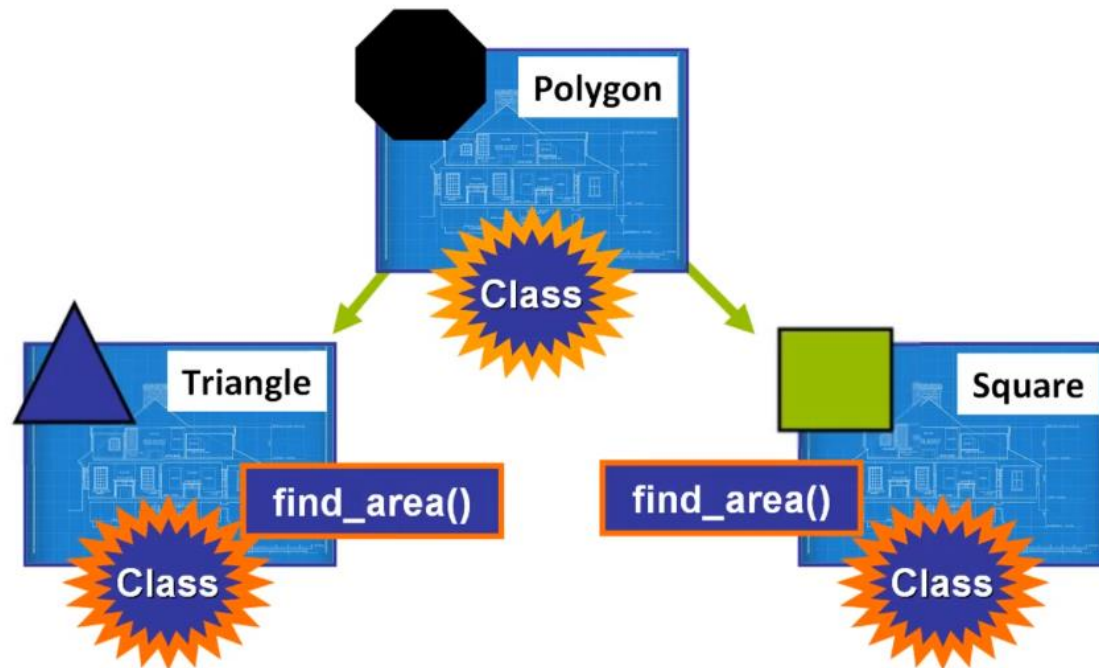
class Button(Rectangle, GUIElement):
 def __init__(self, length, width, color, text):
 Rectangle.__init__(self, length, width, color)
 self.text = text

bt = Button(10, 20, "RED", "Confirm")
bt.click()
```



# Multiple Inheritance

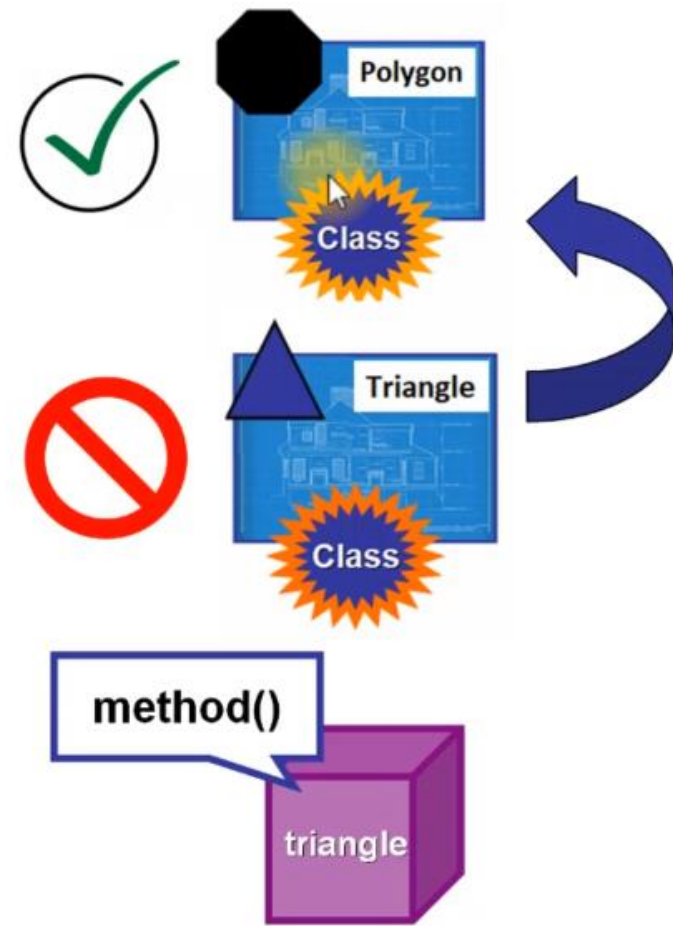
- ในเรื่อง polymorphism ได้กล่าวถึง class ที่มี interface เหมือนกัน ซึ่งถือได้ว่าเป็นจุดที่สำคัญของ OOP เนื่องจากทำให้โปรแกรมที่เขียนขึ้น สามารถเรียกใช้แบบเดียวกันกับข้อมูลที่แตกต่างกัน (เช่น การจ่ายเงินแบบต่างๆ)
- ตัวอย่างนี้ แสดง method `find_area()` ซึ่งเป็น interface ชื่อเดียว แต่ทำงานต่างกัน





# Multiple Inheritance

- ดังนั้นในแต่ละคลาส ก็อาจจะมี method ชื่อเดียวกันได้ คำถาม คือ หากมีการเรียกใช้ method จะเรียก method นั้นจากคลาสใด
- การค้นหา Method จะใช้หลักการตามรูป คือ จะค้นจากจากคลาaslลำดับขึ้นไป 1 ชั้น หากไม่พบ จึงจะหาใน Class ลำดับเหนือขึ้นไปเรื่อยๆ





# Multiple Inheritance

- นอกจากนั้น กรณีที่ superclass กำหนด method เอาไว้ ใน subclass สามารถกำหนด method ชื่อเดียวกันซ้ำได้ โดยเรียกว่า method overloading

```
class Teacher:

 def __init__(self, full_name, teacher_id):
 self.full_name = full_name
 self.teacher_id = teacher_id

 def welcome_students(self):
 print(f"Welcome to class!, I'm your teacher. My name is {self.full_name}")

class ScienceTeacher(Teacher):

 def welcome_students(self):
 print(f"Science is amazing.")
 print(f"Welcome to class. I'm your teacher: {self.full_name}")
```



# Multiple Inheritance

- จากตัวอย่างก่อนหน้านี้จะเห็นว่า มี code ที่ซ้ำกันบางส่วน ดังนั้นหากจะไม่ให้ซ้ำจะต้องให้ method `welcome_students` ในคลาส `ScienceTeacher` ไปเรียกใช้ method `welcome_students` ในคลาส `Teacher` ตามรูป (จะใช้ชื่อคลาสก็ได้)

```
class Teacher:

 def __init__(self, full_name, teacher_id):
 self.full_name = full_name
 self.teacher_id = teacher_id

 def welcome_students(self):
 print(f"Welcome to class!, I'm your teacher. My name is {self.full_name}")

class ScienceTeacher(Teacher):

 def welcome_students(self):
 print("Science is amazing.")
 super().welcome_students()
```



# Multiple Inheritance

- สามารถ Inherit จากคลาสมাত্রฐานก็ได้ เช่น เพิ่มการ search ให้ list

```
class ContactList(list):
 def search(self, name):
 matching_contacts = []
 for contact in self:
 if name in contact.name:
 matching_contacts.append(contact)
 return matching_contacts

class Contact:
 all_contacts = ContactList()

 def __init__(self, name, email):
 self.name = name
 self.email = email
 self.all_contacts.append(self)
```



# Multiple Inheritance

```
1 | class A:
2 |
3 | def x(self):
4 | print("Class A")
5 |
6 | class B(A):
7 |
8 | def x(self):
9 | print("Class B")
10 |
11 | a = A()
12 | b = B()
13 |
14 | # Output?
15 | a.x()
16 | b.x()
```

- Code นี้จะแสดงอะไร



1  
2

Class A  
Class A



1  
2

Class A  
Class B



1  
2

Class B  
Class B



# Multiple Inheritance

- ในกรณีที่ method ชื่อซ้ำกันระหว่าง superclass กับ subclass
- คำถาม คือ จะเรียกใช้ method จากคลาสใด ปัญหานี้เรียกว่า method resolution order (MRO) ซึ่งหากมีโครงสร้างคลาสไม่ซับซ้อน ก็ไม่มีปัญหาอะไร

```
1 # Python program showing
2 # how MRO works
3
4 class A:
5 def rk(self):
6 print(" In class A")
7
8 class B(A):
9 def rk(self):
10 print(" In class B")
11
12 r = B()
13 r.rk()
```





# Multiple Inheritance

- แต่ในกรณีที่โครงสร้างคลาสซับซ้อนขึ้น เช่น จากรูปจะมีวิธีการหาอย่างไร
- กรณีนี้ python จะใช้วิธีที่เรียกว่า depth-first search คือ หาทางลึกก่อน กรณีนี้มีการอ้างถึงคลาส B ก่อน ดังนั้นจะค้นหาจาก D -> B -> A -> C

*depth-first or breadth-first?*

```
class A(object):
 def dothis(self):
```

```
class B(A):
 pass
```

```
class C(object):
 def dothis(self):
```

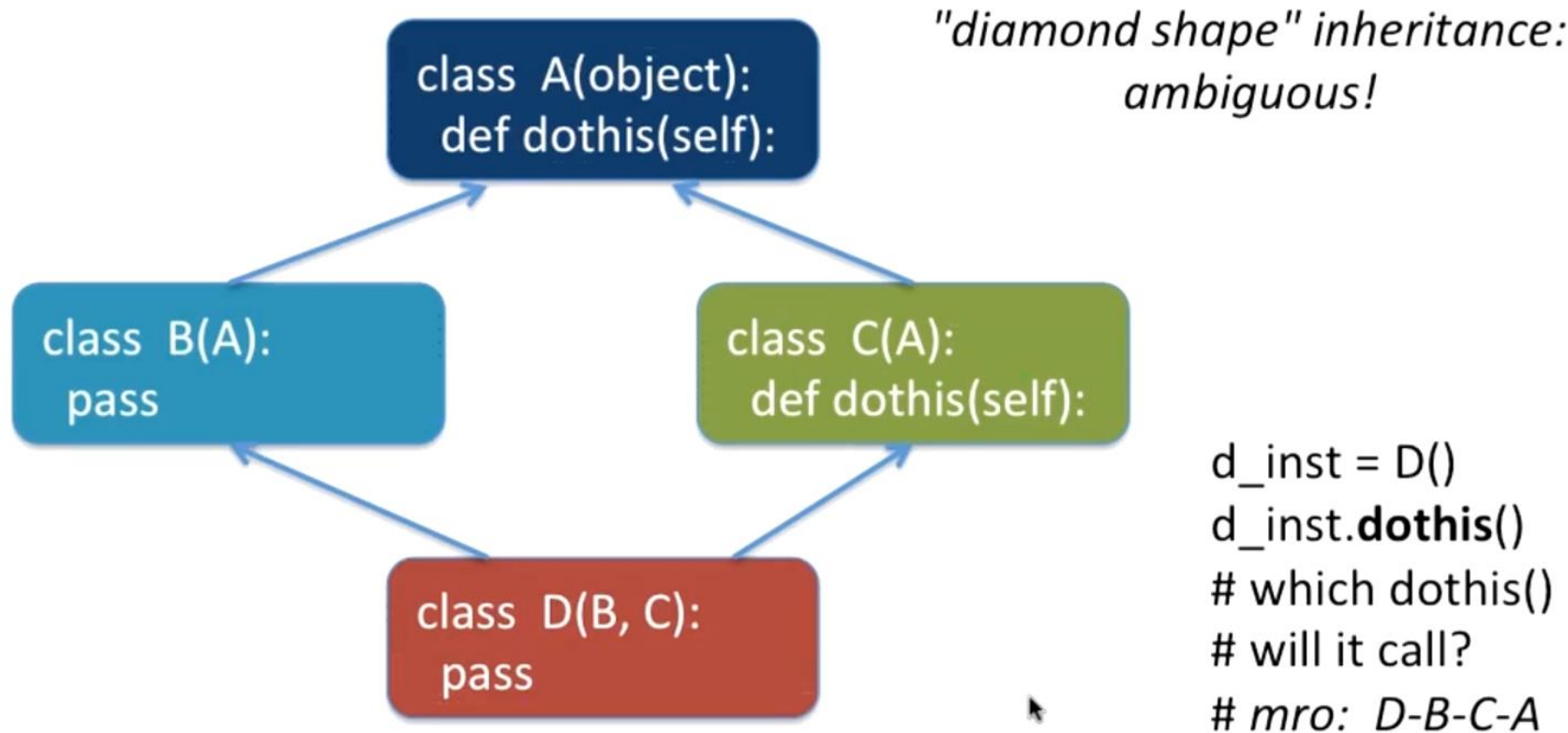
```
class D(B, C):
 pass
```

```
d_inst = D()
d_inst.dothis()
which dothis()
will it call?
mro: D-B-A-C
```



# Multiple Inheritance

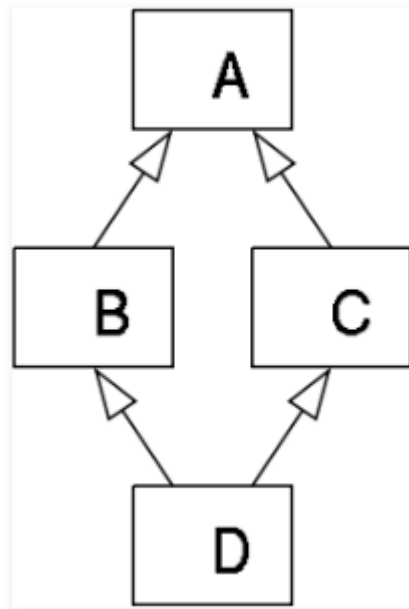
- มีปัญหาหนึ่งในการทำ MRO ของ python ปัญหานี้เรียกว่า diamond problem โดยได้ชื่อมาจากโครงสร้างคลาสที่เป็น diamond shape โดยกรณีที่คลาส inherit มาจาก class เดียวกัน จะไม่ใช้วิธี depth-first search แต่จะค้นหาตามลำดับในการอ้างคลาส





# Multiple Inheritance

- เช่นจาก code นี้ จะแสดงผลอะไร
- ถ้าลบ rk ในคลาส b ออก จะเป็นอย่างไร



```
class A:
 def rk(self):
 print("In class A")

class B:
 def rk(self):
 print("In class B")

class C:
 def rk(self):
 print("In class C")

class D(B, C):
 pass

r = D()
r.rk()
```



# Multiple Inheritance

- เราสามารถตรวจสอบ MRO ได้

```
Python program to show the order
in which methods are resolved
```

```
class A:
 def rk(self):
 print(" In class A")
class B:
 def rk(self):
 print(" In class B")

classes ordering
class C(A, B):
 def __init__(self):
 print("Constructor C")

r = C()
```

```
it prints the lookup order
print(C.__mro__)
print(C.mro())
```

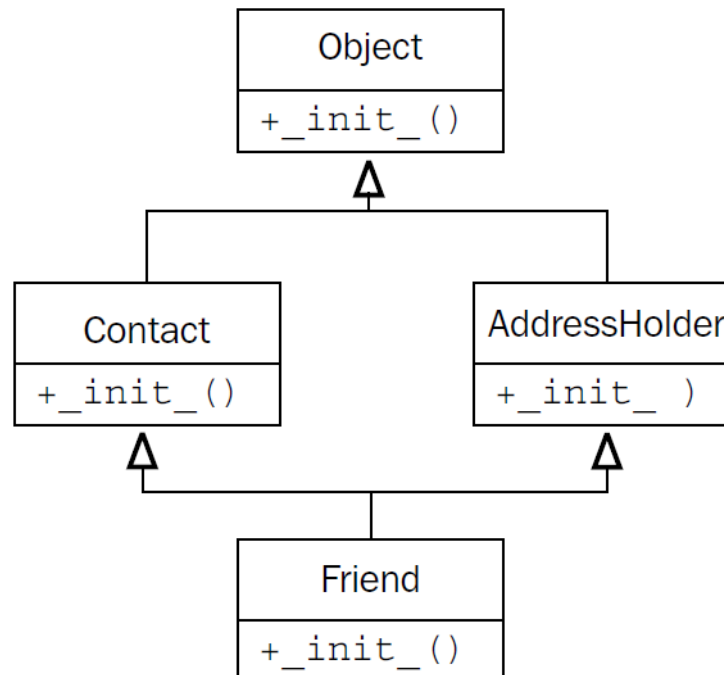
Constructor C

```
(<class '__main__.C'>, <class '__main__.A'>, <class '__main__.B'>, <class 'object'>)
[<class '__main__.C'>, <class '__main__.A'>, <class '__main__.B'>, <class 'object'>]
```



## Multiple Inheritance

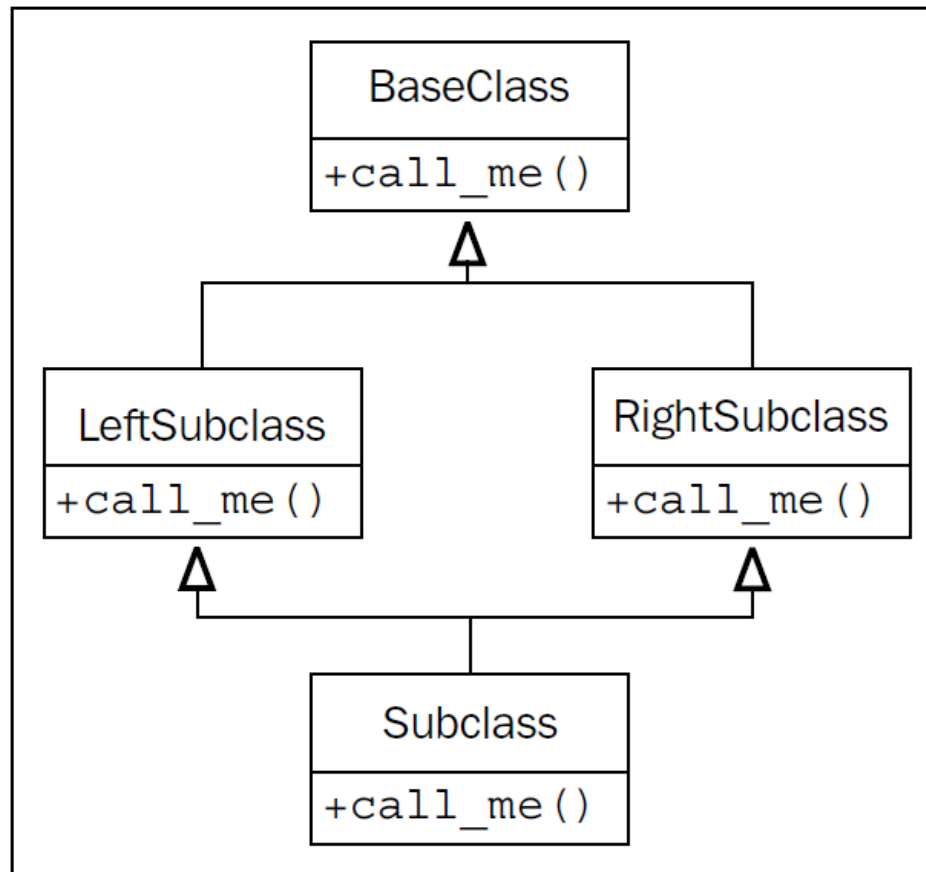
- ลองดูอีกตัวอย่าง จากรูปทุกคลาส inherit มาจากคลาส Object ซึ่งแปลว่าในการทำงาน top superclass อาจจะรัน method `__init__` 2 ครั้งได้ หากใน subclass มีการเรียก constructor ของ superclass โดยใช้ชื่อคลาส





# Multiple Inheritance

- จะจำลองเหตุการณ์โดยสร้าง method ชื่อ `call_me` ในทุก class ตามรูป





# Multiple Inheritance

- จากนั้นให้มีการเรียก call\_me จากระดับล่างขึ้นมาทุกระดับ

```
class BaseClass:
 num_base_calls = 0

 def call_me(self):
 print("Calling method on Base Class")
 self.num_base_calls += 1

class LeftSubclass(BaseClass):
 num_left_calls = 0

 def call_me(self):
 BaseClass.call_me(self)
 print("Calling method on Left Subclass")
 self.num_left_calls += 1
```



# Multiple Inheritance

```
class RightSubclass(BaseClass):
 num_right_calls = 0

 def call_me(self):
 BaseClass.call_me(self)
 print("Calling method on Right Subclass")
 self.num_right_calls += 1

class Subclass(LeftSubclass, RightSubclass):
 num_sub_calls = 0

 def call_me(self):
 LeftSubclass.call_me(self)
 RightSubclass.call_me(self)
 print("Calling method on Subclass")
 self.num_sub_calls += 1
```





# Multiple Inheritance

- จากนั้นเรียกใช้ดังนี้

```
s = Subclass()
s.call_me()
print(s.num_sub_calls, s.num_left_calls,
 s.num_right_calls, s.num_base_calls)
```

- จะมีการทำงานลำดับอย่างไร

```
Calling method on Base Class
Calling method on Left Subclass
Calling method on Base Class
Calling method on Right Subclass
Calling method on Subclass
1 1 1 2
```

- จะเห็นว่าการเรียก base class 2 ครั้ง ซึ่งต้องระวังในกรณีนี้



# Multiple Inheritance

- แต่หากใช้ `super()` แทน

```
class BaseClass:
 num_base_calls = 0

 def call_me(self):
 print("Calling method on Base Class")
 self.num_base_calls += 1

class LeftSubclass(BaseClass):
 num_left_calls = 0

 def call_me(self):
 super().call_me()
 print("Calling method on Left Subclass")
 self.num_left_calls += 1
```



# Multiple Inheritance

```
class RightSubclass(BaseClass):
 num_right_calls = 0

 def call_me(self):
 super().call_me()
 print("Calling method on Right Subclass")
 self.num_right_calls += 1

class Subclass(LeftSubclass, RightSubclass):
 num_sub_calls = 0

 def call_me(self):
 super().call_me()
 print("Calling method on Subclass")
 self.num_sub_calls += 1
```



# Multiple Inheritance

- เมื่อเรียกทำงาน จะเห็นว่ามีการใช้ base class เพียงครั้งเดียว
- ลำดับการเรียกใช้จะเห็นว่า มีการเรียกใช้จาก Subclass ไปยัง Left Subclass ไปยัง Right Subclass แล้วจึงไปยัง Base Class ทั้งนี้ Base Class เป็น super() ของ Left Subclass
- นี่เป็นความแตกต่างระหว่างการใช้ super() กับการเรียกโดยใช้ชื่อคลาส

```
Calling method on Base Class
Calling method on Right Subclass
Calling method on Left Subclass
Calling method on Subclass
1 1 1 1
```



## Inheritance : Difference set of argument

- ปัญหาหนึ่งที่สามารถเกิดขึ้นกับการ inherit หลายๆ ชั้น คือ พารามิเตอร์ที่ส่งไปที่ instance ของคลาสลำดับสุดท้าย บางส่วนอาจใช้โดยคลาสลำดับที่สูงกว่า เช่น มีคลาสดังนี้

```
class Contact:
 all_contacts = []

 def __init__(self, name, email):
 self.name = name
 self.email = email
 Contact.all_contacts.append(self)

class AddressHolder:
 def __init__(self, street, city, state, code):
 self.street = street
 self.city = city
 self.state = state
 self.code = code
```



## Inheritance : Difference set of argument

- สมมติว่าสร้างคลาส

```
class Friend(Contact, AddressHolder):
 def __init__(self, name, email, phone, street, city, state, code):
 Contact.__init__(self, name, email)
 AddressHolder.__init__(self, street, city, state, code)
 self.phone = phone
```

- จะเห็นว่าคลาสที่ inherit มา มีพารามิเตอร์ไม่ซ้ำกันเลย เมื่อเรียก constructor ของ superclass แบบระบุชื่อ จึงไม่มีปัญหา แต่ถ้าเปลี่ยนเป็น super() จะมีปัญหาหรือไม่

```
class Friend(Contact, AddressHolder):
 def __init__(self, name, email, phone, street, city, state, code):
 super().__init__(self, name, email)
 super().__init__(self, street, city, state, code)
 self.phone = phone

f = Friend("Terry", "terry@abc.co", "081-234-5678", \
 "sukumvit", "bkk", "bkk", 10520)
```



## Inheritance : Difference set of argument

- โปรแกรมจาก slide ที่แล้ว เมื่อรันจะพบ Error เนื่องจากจะฟ้องว่า

```
Traceback (most recent call last):
 File "main.py", line 22, in <module>
 f = Friend("Terry", "terry@abc.co", "081-234-5678", \
 File "main.py", line 18, in __init__
 super().__init__(self, name, email)
TypeError: __init__() takes 3 positional arguments but 4 were given
```

- วิธีการแก้ไขในกรณีนี้ ต้องใช้วิธีการส่ง argument เป็น List หรือ Dictionary แล้วให้แต่ละคลาสดึงข้อมูลที่ต้องการใช้งานไปใช้เอง
- ถ้าส่งเป็น argument list ให้ใช้เป็น \*args โดยข้อมูลจะต้องเรียงตามลำดับ
- ถ้าส่งเป็น argument dictionary จะใช้เป็น \*\*kwargs โดย key จะเป็นชื่อของ argument และ value เป็นค่าของ argument จึงควรใช้แบบนี้จะดีกว่า



## Inheritance : Difference set of argument

- ดังนั้นจะปรับปรุงโปรแกรมเป็นดังนี้ โดยให้ระบุเฉพาะ argument ที่ใช้ในแต่ละคลาส  
ดังนั้น argument อื่นๆ ก็จะอยู่ใน `**kwargs`

```
class Contact:
 all_contacts = []

 def __init__(self, name='', email='', **kwargs):
 self.name = name
 self.email = email
 Contact.all_contacts.append(self)

class AddressHolder:
 def __init__(self, street='', city='', state='', code='', **kwargs):
 self.street = street
 self.city = city
 self.state = state
 self.code = code
```





## Inheritance : Difference set of argument

- ให้สังเกตว่าจะใช้ `super().__init__(**kwargs)` เพียงครั้งเดียว โดยจะเกิดผลเป็นการเรียกใช้ superclass ทั้ง 2 คลาส

```
class Friend(Contact, AddressHolder):
 def __init__(self, phone='', **kwargs):
 super().__init__(**kwargs)
 self.phone = phone

f = Friend(name="Terry", email="terry@abc.co", phone="081-234-5678", \
 street="sukumvit", city="bkk", state="bkk", code=10520)
```



## Inheritance : Difference set of argument

- วิธีการข้างต้นจะมีปัญหา 1 จุด คือ ใน argument list จะไม่มี phone ดังนั้น หากใน superclass ใดมีการใช้ phone จะมีปัญหา
- วิธีแก้ 1 : ให้รวม phone ใน \*\*kwargs และให้ class Friend ค้นหาใน dictionary : kwargs['phone']
- วิธีแก้ 2 : เพิ่ม phone เข้าไปภายหลัง kwargs['phone'] = phone
- วิธีแก้ 3 : เพิ่ม phone เข้าไปโดยใช้ kwargs.update
- วิธีแก้ 4 : ส่ง phone ใน \_\_init\_\_(phone=phone, \*\*kwargs)



# Mixin classes in Python

- ใน OOP จะมีวิธีการในการเพิ่มหรือเปลี่ยนแปลงพฤติกรรมของคลาสอื่น โดยไม่ใช้วิธี inherit เรียกว่า mixin สมมติว่ามีคลาส Vehicle และมี subclass Car และ Boat

```
class Vehicle:
 """A generic vehicle class."""

 def __init__(self, position):
 self.position = position

 def calculate_route(depart, to):
 pass

 def travel(self, destination):
 route = calculate_route(self.position, to)
 self.move_along(route)

class Car(Vehicle):
 pass

class Boat(Vehicle):
 pass
```



# Mixin classes in Python

- หากต้องการให้รถยนต์มีคุณสมบัติ คือ มีวิทยุ แต่ไม่ต้องการให้เรือมีวิทยุด้วย สามารถจะสร้างคลาส Mixin ที่เป็นวิทยุ เพื่อนำมาให้คลาส Car inherit แล้วมีคุณสมบัติเพิ่มได้

```
class RadioUserMixin(object):
 def __init__(self):
 self.radio = Radio()

 def play_song_on_station(self, station):
 self.radio.set_station(station)
 self.radio.play_song()

class Car(Vehicle, RadioUserMixin):
 pass
```



# Mixin classes in Python

- ดูอีกตัวอย่างหนึ่ง สมมติว่ามีคลาส Person และ Employee

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

class Employee(Person):
 def __init__(self, name, skills, dependents):
 super().__init__(name)
 self.skills = skills
 self.dependents = dependents

if __name__ == '__main__':
 e = Employee(
 name='John',
 skills=['Python Programming', 'Project Management'],
 dependents={'wife': 'Jane', 'children': ['Alice', 'Bob']}
)
```



# Mixin classes in Python

- หากมีความต้องการให้สามารถ Export ข้อมูลใน object ออกมาเป็น dictionary ได้  
ด้วย สามารถสร้างเป็นคลาส DictMixin

```
class DictMixin:
 def to_dict(self):
 return self._traverse_dict(self.__dict__)

 def _traverse_dict(self, attributes: dict) -> dict:
 result = {}
 for key, value in attributes.items():
 result[key] = self._traverse(key, value)

 return result

 def _traverse(self, key, value):
 if isinstance(value, DictMixin):
 return value.to_dict()
 elif isinstance(value, dict):
 return self._traverse_dict(value)
 elif isinstance(value, list):
 return [self._traverse(key, v) for v in value]
 elif hasattr(value, '__dict__'):
 return self._traverse_dict(value.__dict__)
 else:
 return value
```



# Mixin classes in Python

- จากนั้นให้คลาส Customer inherit ก็จะสามารถพิมพ์เป็น dict ออกมาได้

```
class Employee(DictMixin, Person):
 def __init__(self, name, skills, dependents):
 super().__init__(name)
 self.skills = skills
 self.dependents = dependents

e = Employee(
 name='John',
 skills=['Python Programming', 'Project Management'],
 dependents={'wife': 'Jane', 'children': ['Alice', 'Bob']}
)

print(e.to_dict())
```

```
{'name': 'John', 'skills': ['Python Programming', 'Project Management'],
 'dependents': {'wife': 'Jane', 'children': ['Alice', 'Bob']}}
> □
```



# Exception Handling

- กรณีที่มีความผิดพลาดในคลาส จะไม่นิยมให้แสดง Error โดยการ print
- แต่จะใช้กลไกของ Exception โดยกำหนดให้คลาส Raise exception ขึ้นมา

```
class BankAccount:
 def __init__(self, balance):
 self.balance = balance

 def withdraw(self, amount):
 if amount > self.balance:
 raise ValueError("Insufficient balance")
 else:
 self.balance -= amount
```





# Exception Handling

- และในส่วนที่เรียกใช้ ให้ดำเนินการตามนี้

```
account = BankAccount(1000)
try:
 account.withdraw(1500)
except ValueError as e:
 print("Failed to withdraw:", str(e))
```



*For your attention*